THE UNIVERSITY OF CHICAGO

SINGLE PROGRAM TASK PARALLELISM

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
ERNESTO GOMEZ

CHICAGO, ILLINOIS
MARCH 2005
To Sue
ABSTRACT

This work is concerned with providing software support for procedural parallelism in parallel execution; specifically with the parallel programming and execution paradigm in which multiple copies of a single program execute concurrently, but in which each specific copy of the program may execute a different sequence of statements and routines than other copies.

We propose a framework for the support of process sets that subdivide and recombine dynamically at runtime. We call this MIPS, which stands for Merging Implicit Process Sets. We extend the concept of communication at global barriers to support communications between subsets of processes at barriers affecting only the communicating group. This, in combination with MIPS, allows us to guarantee a deterministic and deadlock-free execution. MIPS provide a theoretical basis for compiler analysis that is also required by our second innovation, which is a form of overlapping that requires semantic information. We propose the use of a finite-state machine to optimize the communication in the overlapping intervals between variable definition and use at different processes. This has the effect of a synchronization extended over time and allows program execution to tolerate and sometimes hide synchronization and communication costs. Finally we propose support for a short-cutting technique which uses information available in parallel to guide execution, save work, and in some special cases may lead to supra-linear speedup.
ACKNOWLEDGEMENTS

I dedicate this work to Sue, my love and inspiration.

I gratefully acknowledge the support of the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago.

Many people have made it possible for me to complete this work. Firstly, thanks to my aunt Gloria Diaz, who insisted I should return to graduate school for my PhD. I would like to thank my professors at University of Houston for an interesting and fun graduate experience. I thank Dr. Ernst Leiss for his help, encouragement, and many an interesting class and assignment. Amanda Vaughn at the Computer Science Department at University of Houston, and Susan Owen at TCAMP, helped me navigate the paperwork, requirements and general pitfalls of graduate studies at University of Houston.

At the University of Chicago I am extremely grateful to Margery Ishmael and Margaret Jaffey for all the help with registration, travel, requirements, housing, and many details of life in Chicago. I thank Drs. Todd Dupont and Bob Rosner of the Flash project for their help and support during my years in Chicago. Thanks to my colleagues in the Planguages project, Babak Bagheri and Terry Clark, for all they have taught me about parallel processing, plus discussions, inspiration and friendship. I thank the members of my committee, Dr. Ian Foster and Dr. Mike Odonnell for their help and advice. In particular, thanks to my advisor L. Ridgway Scott, for advice, encouragement, support and friendship, and to Jan Scott for making me feel at home in Chicago.

Finally, my deepest thanks and appreciation to my daughter Elisa and my wife Sue, who put up with me, encouraged and supported me, with humor, love and understanding, through all these years.
CONTENTS

ABSTRACT ................................................................. iii

ACKNOWLEDGEMENTS .................................................... iv

LIST OF FIGURES ........................................................ viii

LIST OF TABLES ............................................................ ix

1 INTRODUCTION: PARALLEL EXECUTION AND SPMD .............. 1
   1.1 Single program parallelism ........................................ 4
   1.2 Irregular and asymmetric problems ................................ 11
   1.3 Data transfer between parallel processes ......................... 13
       1.3.1 Communications .............................................. 17
       1.3.2 Keys in a distributed, asynchronous environment .......... 20
   1.4 Short-cutting, Overlapping and MIPS ............................ 24
   1.5 Planguages and the P model .................................... 27
   1.6 Synchronization and implicit process sets ...................... 31

2 SERIAL AND PARALLEL EXECUTION ................................. 33
   2.1 Serial execution ................................................ 33
   2.2 Parallel State and Execution .................................. 37
   2.3 Sets of executions .............................................. 43

3 COMMUNICATION ..................................................... 48
   3.1 Information transfer ............................................. 48
   3.2 The membership function ....................................... 52
   3.3 Synchronization ................................................ 55
   3.4 Fusion statements ............................................... 59
   3.5 Non-fusion message passing ................................... 63

4 EXAMPLES OF PARALLEL EXECUTION ............................... 65
   4.1 Trivial parallelism .............................................. 65
   4.2 Lockstep parallelism ............................................ 65
   4.3 Loosely synchronous execution .................................. 67
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>MIPS</td>
<td>73</td>
</tr>
<tr>
<td>5.1</td>
<td>MIPS and Synchrons</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Operations on MIPS</td>
<td>75</td>
</tr>
<tr>
<td>5.3</td>
<td>Parallel time</td>
<td>78</td>
</tr>
<tr>
<td>6</td>
<td>MIPS ON THE CFG</td>
<td>79</td>
</tr>
<tr>
<td>6.1</td>
<td>MIPS identification on the CFG</td>
<td>79</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Merge nodes and the Merge Tree</td>
<td>85</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Simplifying split-merge identification</td>
<td>97</td>
</tr>
<tr>
<td>6.2</td>
<td>MIPS level</td>
<td>99</td>
</tr>
<tr>
<td>6.3</td>
<td>Loops</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>MIPS EXECUTION</td>
<td>107</td>
</tr>
<tr>
<td>7.1</td>
<td>Code support</td>
<td>107</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Algorithms for code analysis and transformation:</td>
<td>110</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Runtime algorithms</td>
<td>114</td>
</tr>
<tr>
<td>7.2</td>
<td>Communications in MIPS</td>
<td>118</td>
</tr>
<tr>
<td>7.3</td>
<td>Deadlock-freedom and determinism</td>
<td>121</td>
</tr>
<tr>
<td>8</td>
<td>OVERLAPPING</td>
<td>124</td>
</tr>
<tr>
<td>8.1</td>
<td>Overlapping concepts</td>
<td>124</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Benefits with/without special hardware</td>
<td>130</td>
</tr>
<tr>
<td>8.1.2</td>
<td>The benefit from multiple processors:</td>
<td>131</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Out-of-order, asynchronous messaging</td>
<td>132</td>
</tr>
<tr>
<td>8.2</td>
<td>Overlap protocol</td>
<td>134</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Collective communications</td>
<td>138</td>
</tr>
<tr>
<td>8.3</td>
<td>The Overlap FSM</td>
<td>139</td>
</tr>
<tr>
<td>8.4</td>
<td>Correctness</td>
<td>144</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Point-to-Point communications</td>
<td>146</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Collective data transfers</td>
<td>148</td>
</tr>
<tr>
<td>8.5</td>
<td>Coding requirements</td>
<td>149</td>
</tr>
<tr>
<td>9</td>
<td>SHORT-CUTTING</td>
<td>152</td>
</tr>
<tr>
<td>9.1</td>
<td>Definition</td>
<td>152</td>
</tr>
<tr>
<td>9.2</td>
<td>Short-cut implementation</td>
<td>157</td>
</tr>
<tr>
<td>9.3</td>
<td>Interruptible Communications</td>
<td>159</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Correctness and safety</td>
<td>164</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>10</td>
<td>SOS PERFORMANCE</td>
<td>166</td>
</tr>
<tr>
<td>10.1</td>
<td>Hardware</td>
<td>166</td>
</tr>
<tr>
<td>10.2</td>
<td>Experiments</td>
<td>167</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Broadcast</td>
<td>167</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Scientific application</td>
<td>169</td>
</tr>
<tr>
<td>10.3</td>
<td>Results</td>
<td>170</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Broadcast comparison</td>
<td>170</td>
</tr>
<tr>
<td>10.3.2</td>
<td>Scientific application:</td>
<td>173</td>
</tr>
<tr>
<td>10.3.3</td>
<td>Overall results:</td>
<td>178</td>
</tr>
<tr>
<td>10.4</td>
<td>Analysis</td>
<td>178</td>
</tr>
<tr>
<td>10.4.1</td>
<td>Broadcast</td>
<td>179</td>
</tr>
<tr>
<td>10.4.2</td>
<td>More processes than processors</td>
<td>179</td>
</tr>
<tr>
<td>10.4.3</td>
<td>Scientific application</td>
<td>180</td>
</tr>
<tr>
<td>11</td>
<td>CONCLUSIONS AND CONTINUING WORK</td>
<td>182</td>
</tr>
<tr>
<td>11.1</td>
<td>Continuing work</td>
<td>183</td>
</tr>
<tr>
<td>11.1.1</td>
<td>Compiler support</td>
<td>183</td>
</tr>
<tr>
<td>11.1.2</td>
<td>Hierarchical sets of processes</td>
<td>184</td>
</tr>
<tr>
<td>11.1.3</td>
<td>Error handling</td>
<td>185</td>
</tr>
<tr>
<td>11.1.4</td>
<td>SOS development</td>
<td>185</td>
</tr>
<tr>
<td>11.1.5</td>
<td>Communication algorithms</td>
<td>186</td>
</tr>
<tr>
<td>11.1.6</td>
<td>Algorithms and computation</td>
<td>186</td>
</tr>
<tr>
<td>11.1.7</td>
<td>Implications for Computer Architecture</td>
<td>187</td>
</tr>
<tr>
<td>12</td>
<td>APPENDICES</td>
<td>188</td>
</tr>
<tr>
<td>12.1</td>
<td>SOS API</td>
<td>188</td>
</tr>
<tr>
<td>12.2</td>
<td>Lexical analysis of communication statements</td>
<td>205</td>
</tr>
<tr>
<td>INDEX</td>
<td></td>
<td>211</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>214</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Genealogy of SOS system: ............................................. 25

4.1 Parallel execution follows the CFG; transitions indicated here are de-
terministic. .............................................................. 70

6.1 Control structures and CFG ............................................. 82
6.2 Complete first merge node identification .............................. 90
6.3 Complete first merge nodes for split node 2 .......................... 91
6.4 Identification of partial merge nodes ................................. 94
6.5 Merge tree .............................................................. 95
6.6 Loop unrolling .......................................................... 105

8.1 BSP example: ............................................................ 126
8.2 Wait conditions .......................................................... 128
8.3 BSP example, showing speedup by overlapping ...................... 129
8.4 Overlapping, out of order communications .......................... 133
8.5 Broadcast, based on hypercube algorithm ......................... 140
8.6 Finite State Machine to implement overlapping protocol without buffer-
ing. This machine also supports short-cutting .................... 141

9.1 Comparison of serial and parallel short-cutting ..................... 156

10.1 Example simplex in three dimensions ............................... 171
10.2 Comparison of SOS and MPI performance .......................... 172
10.3 Timing data for scientific application .............................. 174
LIST OF TABLES

8.1 Sender(P1)-receiver(P2) timing ........................................ 135
8.2 States of a sender-receiver pair ........................................ 137

10.1 Scientific program: 8 processes ........................................ 175
10.2 Scientific program: 13 processes ........................................ 176
CHAPTER 1
INTRODUCTION: PARALLEL EXECUTION AND SPMD

In this work we are concerned with providing software support for procedural parallelism in parallel execution; specifically with the parallel programming and execution paradigm in which multiple copies of a single program execute concurrently, but in which each specific copy of the program may execute a different sequence of statements and routines than other copies. We are particularly interested in the applications to irregular scientific problems. A key characteristic of such problems is that static load balance is not possible because computation is strongly dependent on the data, and perhaps also because there are asymmetric features in the algorithm. Many irregular problems also require different computations to be performed on different portions of the data; which particular processes need to execute which specific routines may not be known until runtime, and may change for different instances of the same problem (the FLASH [FLASH] problem appears to be of this nature, as are some molecular dynamics problems). In such problems, process sets that execute particular sections of code are formed at runtime following data-dependent logic.

We here propose the following three innovations: Firstly, we introduce a framework for the support of process sets that subdivide and recombine dynamically at runtime. We call this MIPS, which stands for Merging Implicit Process Sets (for simplicity, we will take MIPS to be either singular or plural depending on context). We extend the concept of communication at global barriers from BSP [BSP 96] to support communications between subsets of processes at barriers affecting only the communicating group. This, in combination with MIPS, allows us to guarantee a deterministic and deadlock-free execution. MIPS provide a theoretical basis for compiler analysis that is also required by our second innovation, which is a form of overlapping that requires semantic information. We here propose the use of a finite-state machine to
optimize the communication in the overlapping intervals between variable definition and use at different processes. This has the effect of a synchronization extended over time and allows program execution to tolerate and sometimes hide synchronization and communication costs. As observed by developers of parallel computing systems such as Shrimp [ALPERT 96], this is a key requirement for efficient execution of code that includes barriers in an asynchronous environment. Finally we propose support for a short-cutting technique which uses information available in parallel to guide execution, save work, and in some special cases may lead to supra-linear speedup.

We introduce a model of parallel execution which we call MIPS, based on a notion of parallel state which we believe to be new. This model includes a notion of parallel time, although this is not explicitly used in this work. We also have a demonstration (based on induction and a posteriori computation) of the existence of a parallel execution. MIPS analysis starts from the initial set of all processes in SPMD execution and identifies points in the program control flow graph where sets of processes split into subsets and the closest point in the program at which sets proceeding from a split can merge. MIPS analysis uses a static analysis of a program which specifies where code must be inserted to identify the actual process sets that form at runtime, and also the data that each process must maintain to keep track of the set. In an MIPS execution we prove determinism and freedom from deadlock under certain restrictions on the type of communication allowed. We make no assertion that these restrictions are necessary, just that they are sufficient. MIPS specifically support the need for asymmetric execution of code in irregular scientific problems. Irregular scientific problems have the common characteristic that it is not possible to statically predict work distribution. This leads to differences in workload between processes which need to be ameliorated for efficient parallel execution. Our approach to overlapping is a technique that uses semantic information from the program code to determine intervals between the updating of a communicated variable and its use. For each communication, a finite state machine is generated as a gate-keeper, and communication is scheduled in the background and can occur synchronously, and without buffering, between processes whenever a pair of processes is ready to carry it out. Communica-
tion is not necessarily carried out in the order in which it is specified in the program, but its correctness is guaranteed by the runtime system using semantic information. Computation is not halted at any process unless there is a pending communication with respect to a variable that needs to be read or rewritten at that process. Additionally, our approach to overlapping can reduce delays for synchronizations which do not involve communication.

Short-cutting can be applied in some situations to use information available in parallel execution to reduce the amount of work a program needs to do. This may be particularly advantageous in irregular problems due to their inherently unbalanced or asymmetric workload. The method is demonstrated in a minimization problem in which different alternatives are tried on the same data by different processes. A process which finds a particularly promising partial result can then interrupt the computation and communications of other processes and propose its partial result as the base for a new iteration. Short-cutting exploits asynchronous execution because it favors the process that reaches a good solution in least time, which may be different on repeated execution. Short-cutting execution follows a non-deterministic, timing dependent path to a deterministic result. It is possible that more than one process will assert a short-cut, which may lead to subsets of processes analyzing different partial solutions. Here, MIPS runtime support can be used to identify the sets that are formed, and MIPS analysis can be applied to identify merge points following the short-cut. Short-cutting also requires some of the technology developed in our approach to overlapping: a process may be interrupted while communications are pending, and these must be gracefully terminated.

To support MIPS, overlapping and short-cutting, we have developed a runtime library called SOS (MIPS, Overlapping and Short-cutting), and we give some experimental results of its use. (The SOS acronym was selected before the MIPS terminology was adopted, and has been retained).
1.1 Single program parallelism

In this chapter we will give an overview of previous work, and of several current parallel programming languages. We will pay special attention to the Planguages, which supply the framework in which most of the ideas presented in this work were developed and tested. We will give some conceptual background on communications and message passing that underlies the rest of this work.

Given the large amount of work in this field, we limit ourselves to providing a representative sample. In many cases there are many different viewpoints on particular concepts and terminology. This is in particular true on what is included in the SPMD programming paradigm. We further describe some of the conceptual background that underlies the rest of this work.

There are many possible styles of execution of parallel programs, which depend on a large variety of programming models and architectures. It is worth noting that there is no universal agreement on which categories are most useful, and where different styles of execution actually fit in various classifications.

The best known classification scheme is the taxonomy proposed in the nineteen sixties by Michael Flynn [FLYNN 72]. Properly this is a taxonomy of information flow models, none of which logically depend on any particular hardware. However, this classification is most frequently used to classify computer architectures ([LEISS 95], [KOELBEL 94], [ROOSTA 00]). The taxonomy distinguishes the following cases:

SISD single instruction stream, single data stream,
SIMD single instruction stream, multiple data stream,
MISD multiple instruction stream, single data stream, and
MIMD multiple instruction stream, multiple data stream.

An execution model describes how a program (or programs) actually runs on a computational system; as such it includes both information and control flow. We note
that Flynn's taxonomy also implies control flow at least by specifying single or multiple instruction streams. It is therefore possible to map Flynn's classifications to a classification of execution models. Conversely, we may classify execution models in the same terms as we use for information flow. So, for example, a single program running serially can be described as having a SIMD model of execution as well as data flow. A pure data parallel program has different data at each process as its only source of parallelism; this data is analyzed by the same code and there is a single stream of instructions (this is the programming model of C* [ROOSTA 00]). Such an execution may be described as having a single instruction stream acting on multiple data MIPS, or SIMD. There is a distinction between data flow and execution model in that we may obtain SIMD data flow on a MIMD execution by replicating the same control stream and using barriers where necessary to preserve dependences, or even by time slicing multiple SISD processes. However, in general we can say that there is a two-way mapping between data flow and execution models, which allows us (and other authors) to use Flynn's classification almost interchangeably in describing both.

Flynn's classification has also been interpreted as an architectural taxonomy. Here, we believe the mapping is less clear. For example, both [LEISS 95] and [ROOSTA 00] see the need to subdivide MIMD into loosely and tightly coupled machines, and make further detail distinctions depending on memory systems and interconnection networks. There is some disagreement as to what MISD is. Many authors disparage or ignore MISD and are not sure what if anything fits architecturally in that category. [LEISS 95] puts pipelined processors in MISD, but reluctantly; he considers it a somewhat forced fit. [ROOSTA 00] is not sure where to put vector computers in Flynn's classification and in fact places them both in SIMD and in MISD. [ROOSTA 00] also feels the need to add a hybrid category for SIMD-MIMD and MIMD-SIMD machines. [KOELBEL 94] only uses SISD, SIMD and MIMD to categorize architectures, altogether ignoring MISD. However, if we consider Flynn's categories as execution models, describing the flow of data and instructions during execution, there are algorithms that appear to fall into a MISD style, notably oracle and backtracking algorithms [HOROWITZ 78] in which the same data can be analyzed in a number of different
ways. Of Flynn's taxonomy, the MIMD classification is the most general, and can
describe parallel computing platforms based on multiple independent processors. We
will not go into detail on these issues because our main concern is with programming
models; however we note that information flow does not seem to map well to the
parallel architectures that actually exist or have been built, and that it is difficult to
map architectures back to a pure information flow taxonomy.

There are a number of other ways of characterizing parallel processing, such as pro-
gramming models which do not necessarily depend on underlying hardware. Koelbel
et. al. [KOELBEL 94] identifies data parallelism, functional (task) parallelism and
master-slave parallelism. To these we could add client-server computing, which we
distinguish from master-slave in that there is no subordinate relationship. A client
process requests something from a server when it is required by its own computa-
tion, whereas a master process assigns tasks to a slave process. Although such
programming models are independent of hardware and could be executed in fact on
a uniprocessor, some models are better fits to particular architectures than to others.
For example, client-server computing falls within MIMD; data parallel computing
in which the same code (a single instruction stream ) acts in lock step parallel on
different data (multiple data streams) most naturally maps to SIMD.

A different broad classification of programming models depends on the style of
memory access; broadly we may talk about a message passing or a shared mem-
ory model. The key distinction between these is whether a process P can access a
value produced by another process Q independently of actions by Q (shared, one-
sided access) or whether both P and Q must act to transfer a value (message pass-
ing, two-sided access). We find this particular characterization of memory access in
[JORDAN 87], and it seems a natural way to make sense of a large variety of syntactic
structures and library calls.

Theory of parallel computation takes a different approach, attempting to come
up with a general computational model distinct from the underlying hardware. The
attempt here is to produce a general model analogous to the Turing machine or the
Von Neumann architecture [MCCOLL 92]. One approach is to generalize the model of
random access memory to parallel computation, which is known as the PRAM model [GREENLAW 95],[MCCOLL 92]. In this model there is a single shared memory, with unit access cost to all locations at all processors. Each processor has no local memory other than its register set, execution is in lockstep at all processors, and all communication between processors is by reads from, and writes to, memory. All instructions also have unit cost. There is a single process per processor, but no restriction on number of processors, size of memory or size of values that may be stored.

Due to doubts about the correspondence to reality of PRAM models, Greenlaw [GREENLAW 95] prefers to use a Boolean circuit model for analysis of parallel algorithms and complexity. Such a model has the advantage of corresponding directly to real devices such as logical gates, although it still ignores signal delays. It is, however, too low level a model to be of use in real program design; programming at the level of logic gates is below even machine language. Models such as this are very useful in exploring the limits of parallel computation and the properties of parallel algorithms. In particular, Greenlaw shows how the model can be used in analyzing algorithms to determine if they are inherently serial, and in determining the possible existence of different algorithms that might be efficiently parallelized. However we are here concerned with programming models for parallel computation. For our purposes the theoretical models do not seem appropriate, any more than we would use a Turing machine for practical scientific computation.

We are here concerned with software support for a specific parallel programming paradigm called SPMD (Single Program, Multiple Data), which is characterized by the execution of multiple copies of the same program text, each acting on data which may be different. We will explicitly develop software for message passing systems, but our theory is language independent and can be equally applied to message passing and shared memory systems. We will call the execution of each separate copy of the code a process, and will refer to the execution of an entire parallel program, consisting of multiple processes, as a parallel execution. SPMD may include task parallelism, in which different operations may be performed at different processes at the same time.
Foster [FOSTER 95] points out that task parallelism appears necessary to deal with programs that require adaptive or irregular data structures.

Bulk Sequential Parallelism, or BSP [BSP 96], [VALIANT 90] deserves special mention as a model that allows both theoretical analysis of performance and practical implementation as a real model for programming. BSP is a loosely synchronous model in which processes progress independently in long computational stages, alternating with barriers at which all communication takes place. This allows a clean separation of communication and computational costs, and avoids all problems of data dependences during computation. We will see in chapter 8 that forcing programs to conform to the BSP model at run time can produce unnecessary delays; and the model appears in any case overly restrictive to us both in its rigid separation of communication and computation and in its insistence that all processes in a program execution be in equivalent computational phases at the same time.

Various authors ([FOSTER 95], [KOELBEL 94]) propose that SPMD describes the programming style of data parallel languages like High Performance Fortran (HPF); in fact Chandy and Foster ([CHANDY 95], [FOSTER 95]) consider extensions to HPF to add task parallelism to extend the HPF model beyond SPMD. We consider that, within SPMD, the same program acting on different data may take different execution paths as a consequence of logic statements that depend on the data. This implies that, for full generality, an SPMD execution model needs to include task parallelism. We consider that SPMD, in its full generality, is a looser and more general style than SIMD, and believe that it properly fits inside the MIMD classification. Even when processes all execute the same program, the instruction stream at any given process may be different from that at other processes.

We therefore need to pay some attention to the (MIMD) hardware on which we will assume that execution takes place. We are assuming that each individual process runs on a single processor at any given time, although it may migrate to other single processors during execution. We have physical processors each of which is capable of running at least one process, but which typically are capable of time sharing. Therefore we will not assume a one-to-one match between number of processes and
number of processors. We would not be able to exploit more processors than processes, but we must consider that our parallel execution may require more processors than are available. Indeed, we may have a case in which our parallel execution runs on a single physical processor, and the illusion of processes running at the same time is provided by a multitasking operating system. This means in particular that we are not able to talk about events happening “at the same time” in different processes. Depending on the underlying hardware it may be impossible for such a thing to happen. The best we will be able to do is to talk about a two-way interaction between processes at some specific point of execution, so that it appears that an event at one process is simultaneous with an event at another. The underlying reality, however, may involve process waits and perhaps data buffering by the system.

If we do have multiple processors, they may be loosely coupled and each have their own clocks. In such a case, there is inevitable skew between times at each processor (see [LAMPORT 78]). Even in a tightly coupled system with a single clock we will have time delays in the wires between the clock and the individual processors which prevent the establishment of an exact common time. We may talk about an external time, or clock time, which we can establish by, for example, a stop watch which is started when we start the parallel execution, and which can be stopped to measure the time of any particular externally observable event (such as a printout or a program halt). Such a time bears only an approximate relation to whatever is happening at the level of individual processes. When we talk about parallelism, therefore, we are referring to processes which logically act as if they are executing at the same time. Similarly, when we define synchronization below, we will have to define what we logically mean by such a thing, given the inability to enforce “at the same time” in physical terms.

The SPMD programming model may be used to express different styles of execution, ranging from SIMD to MIMD, and even MISP. It is more general than SIMD because control flow that depends on data that is not the same at different copies of the program can lead to execution of different sequences of instructions at each process (as in MIMD). A fully general MIMD programming style implies the possi-
bility that totally different programs may execute cooperatively. It is easier in SPMD to coordinate different processes than in such an MIMD model by using the added information available in the program text common to all processes.

Within SPMD, we are going to consider the static process model, in which we have a fixed number of processes in a parallel execution, all of which start together. We first find the SPMD model explicitly described by Harry L Jordan, in the implementation in 1984 of The Force, an extension to Fortran [JORDAN 87]. Other early SPMD languages include Dino [MEHROTRA 87] and BLAZE [ROISING 87]. Where The Force was targeted at shared memory, Dino was designed for distributed memory. Dino hid the details of data transfer by defining environments of data and procedures (equivalent to processes), which could each include a local copy of a given variable (one per environment). Communication details were hidden by allowing references to the value of a variable in a different process environment to appear in assignment statements. BLAZE included the forall instruction which supports parallelizing loop iterations and also appears in Fortran 90 [METCALF 98] and HPF [KOELBEL 94]; it gave explicit support to task parallelism by a functional procedure call style (without side effects), which allowed such calls to be made in parallel whenever there are no dependencies between input and output values of different procedure calls.

Communicating Sequential Processes (CSP) [HOARE 78], [HOARE 85] may be considered to be conceptually SPMD, in that it defines a programming model and language based on composition of serial processes. CSP is, however, more general than SPMD in that it does not specify or require a single program. The Occam programming language [JONES 88], [OCCAM 84], which closely follows CSP concepts, runs as a single parallel program which includes multiple processes on different parallel processors, in an SPMD mode.

SPMD processes are conceptually independent programs which can each have local memory which is private to each process and shared memory which may be accessed by all processes. Each process can progress independently except as specified by the programmer with synchronization or communication statements. The execution model maps well to a computing environment with multiple processors which each
have their own memory.

SPMD has become a commonly used style of parallel program, implemented for example by the Planguages [PLANG 99] and by a host of message passing systems, including Message Passing Interface (MPI) [MPI 95], Parallel Virtual Machine (PVM) [DONGARRA 93] and many vendor-specific implementations. It has the advantage, for purposes of analysis, of providing a known starting point for the parallel execution. It is also a programming style that appears particularly suitable for application to irregular problems.

1.2 Irregular and asymmetric problems

There are different kinds of irregularity in parallel codes. Data structure irregularity is one common case that has been addressed previously [PONNU SAMY 93]. In other applications, data distributed among parallel tasks can be regular, but access unpredictable [HONG 97] or asymmetric [GEIST 88]. Different authors appear to have different concepts of what is meant by irregular or non-uniform problems. Fink and Baden [FINK 97], for example, characterize irregular problems as those which use irregular data representations to resolve fine structure in complicated geometries, which gives rise to an unpredictable communication pattern. On the other hand, [MUKHERJEE 95] considers that the defining characteristic of irregular problems is the use of sparse matrices with data access patterns that are not known until run time. Different views were presented at the Fifth International Symposium on Solving Irregular Problems in Parallel, held in Berkeley, California in the summer of 1998, papers such as [MARTINS 98] on Steiner graphs, and [GOMEZ 98], which included a parallel downhill simplex solver for multidimensional equations, were both considered to deal with irregular problems; neither used irregular data representation or sparse matrices.

We believe that a common thread in all these varied definitions of irregularity is that the work to be done in solving the problem, in which we include both computation, and data movement, is not known before the computation is actually started;
and in fact the distribution of work may change as the computation progresses. The implication for a parallel program is that we are not going to be able to assign the same amount of work to each processor in advance, since we are unable to anticipate the workload. This in turn implies that we will not be able to fully utilize the computing power of multiple processors, since lightly loaded processors will finish first and wait for heavily loaded processors to finish. We get the same effect from communication or data movement imbalances, where some processes may end up waiting extra time due to network congestion.

The workload of regular problems can be anticipated when writing a program, and this work can be carefully balanced, often fine-tuned using knowledge of the computer architecture on which the program is to run. In this way, problems such as matrix multiplication [EVANS 92] can be carefully analyzed and optimized for parallel execution. Regular problems lend themselves to data parallel approaches in which we perform the same operations on a uniform distribution of data.

This distribution of work can at best only be approximated for irregular problems. To solve such problems efficiently it is necessary to use methods which can tolerate or compensate for an irregular distribution of work, or balance the work dynamically at run time. It may be necessary to perform different computations on the data at each process, particularly in physics or chemistry simulations, for example the ASCI FLASH project at University of Chicago [FLASH]. This type of execution in which different processes may need to perform different actions from other processes depending on information available only at run time is well suited to an SPMD execution style which allows task parallelism.

Some problems are not irregular in the sense we have described, in that the computation and communication is totally predictable ahead of time. Many of these still present challenges to parallel computation because they are asymmetric; that is, for algorithmic reasons, the workload cannot be evenly distributed between processes. This applies to some standard and frequently used algorithms such as Gaussian elimination [ROBERT 90]. Although the methods we will study are developed with irregular problems in mind, some of them may also be fruitfully applied to asymmetric
problems.

1.3 Data transfer between parallel processes

In SPMD, the processes are distinct, but this is not necessarily the case for the data when we consider shared memory. A shared variable could be considered to be an entity separate from the process, and equally accessible to all. Reasons of efficiency may force us, depending on the underlying hardware, to modify this viewpoint. On common parallel computing platforms in which each processor has local memory and there is some communication network to transfer information from one processor to others, it will certainly be the case that the access time to a particular memory location will depend on whether that memory has to be accessed across the communication network or is attached to the same processor where the data is needed. Even if access to any memory location can be done in uniform time, it may still be convenient to assign particular areas of memory to particular processes, in order to divide the work, or to simplify memory allocation by distributing it.

These considerations may lead us to assign particular areas of shared memory to particular processes. In this sense we can talk about shared memory owned by some process, even though other processes theoretically have equal access to that shared memory. This concept of processes owning parts of shared memory fits well with the SPMD concept in which multiple equal processes run at the same time. It is supported by many real systems, including Co-array Fortran [NUMRICH 97], Titanium [TITANIUM 00] and Split-C [CULLER ]. Such considerations may allow us to impose different conditions on access to shared memory by the owner process than on others. The concept of guarded memory, in [BAGHERI 94], is such a condition; each process guards its local memory by controlling access to it by all other processes. In a sense, guarded memory builds a fence around shared data objects. This allows the guarded memory paradigm to guarantee determinism and freedom from deadlock.

We will consider that data transfer between processes occurs when information computed by one process is made available to a different process, either by an explicit
communication from private memory in one process to private memory at a different process, or when a process accesses (read or write) information in some shared area (where we may distinguish whether the process does or does not own the location accessed).

We first consider two-sided communication, that is, a message passing paradigm, and show that this leads to deterministic programs. Many existing SPMD systems (for example [TITANIUM 00], [NUMRICH 97]) use a shared memory paradigm in which one-sided access in the style of get or put to memory is permitted. However, all of these force a synchronization to enforce determinism (except for read-only data); once we include this synchronization the communication is two-sided.

A key fact of communication between processes is that it takes time. Previous theoretical work tends to ignore or hide the physical nature of communication by requiring synchronization at every communication statement (as in [HOARE 78]) which considers the communication act to happen when a sender places a communication on a channel and a receiver simultaneously removes it from the channel. This reduces the channel to an abstraction that merely serves to link processes. Another possibility is considering the communication medium or channel as an active communicating agent and thus ignoring communication time between agents, as [MILNER 89] does. Some work simply ignores the possibility of delays by specifying that the system is synchronous and proceeding from there, as is done in the PRAM model [FORTUNE 78], or abstracts computation to Boolean circuits [GREENLAW 95], with a loss of all concept of computation time.

Alternatively, work that explicitly considers communication delays concentrates on minimizing the total communications load and balancing the work load ([HU96], [SMITH 96] among many others), moving the data to match a varying workload (as in Kelp, [FINK 96]), balancing load to compensate for communication delays [ROBERT 90], [LEISS 95]. Some of this work is dependent on matching algorithms to specific network geometries [JOHNSSON 96]. Synchronization is dealt with by load balancing, under the assumption that all nodes will complete stages of computation at approximately the same time and so will be ready to communicate. But static load
balancing may not be possible for irregular or unstructured problems, and in any case static load balancing may not be effective on a heterogeneous set of processors or a parallel computer with varying workload. Dynamic load balancing at run time is always possible, but must always lag behind detection of the imbalance and may require significant work and delay to accomplish.

While it is undoubtedly simpler to deal with synchronous communications and processes, it is not real for any system in which computing elements are separated by physical distance. In a computer, our signals are carried by currents whose speed is less than that of light. If our distributed computer is a network of workstations, or even a tightly coupled parallel computer which is not dedicated to running only a single task, then the speed of a signal between any two separated processes is not constant (due to network congestion or possibly path variation), and the time delay is not constant. Therefore the signal time, rather than being a constant synchronization delay is in fact a lower bound on the time it takes to synchronize processes.

If processes that must communicate are carrying out different computations, either because they are working on different data sets or even executing different threads of control, then there is an added synchronization cost, which is the difference in the time it takes for the slowest process in a set and the fastest to reach the synchronization point. Suppose two processes, P1 and P2 are initially synchronized and must communicate after performing some computation. Suppose this computation is irregular, such that P1 executes N1 instructions and P2 executes N2 instructions, and that each instruction takes a time \( \tau \) to execute. Then there will be an added synchronization cost \( T_s = \tau | N1 - N2 |. \) Note that this cost is some fraction of \( N \), which is the number of instructions executed between synchronization points.

If P1 and P2 are not the only processes running on their respective processors, then either one could be delayed with respect to the others by arbitrarily large amounts, of the order of \( \tau N \). Suppose for example that a single other process of equal priority runs on the same node as P1, but P2 has the processor it runs on to itself - then P1 is effectively running on a processor that is half as fast as P2. These differences in time between communications at different processes are a form of load
imbalance, a measure of which is the fraction $\beta = \frac{|N_1 - N_2|}{N}$.

Now consider the problems for which we are likely to use a parallel machine. In
general, we would not choose to run in parallel an algorithm that required more
communication than computation time, since such algorithms would run much more
efficiently on a serial machine. We are instead likely to solve problems expressible as
fairly long runs of computation with some communication, as in [BSP 96]. In such
problems, the computation time exceeds the communication time. Let $T_n = \tau N$,
where $N$ is the average number of instructions between two communications, and $\tau$ is
the time it takes to execute a single instruction. Let $T_c$ be the actual communication
time; that is, the time it takes for signals to travel between processes. In the typical
case, we would compute in parallel in situations where $T_n \gg T_c$; in fact one way of
gaining parallel efficiency is by applying parallel processing to more computationally
intensive problems in which $T_n$ increases faster than $T_c$.

We can relate the synchronization cost $T_s$ to $T_n$ by observing that $T_s = \beta T_n$.
Merely taking a larger problem will increase $T_n$, but may not affect $\beta$ without doing
extra work to accomplish this. If $\beta$ remains constant, then we will tend to have
the synchronization cost $T_s > T_c$. That is, in just those problems in which we
are likely to employ parallel processing, it is very possible that the most significant
communication related cost is actually the synchronization cost. Therefore we must
explicitly deal with it; any attempts to simply ignore synchronization costs, including
theories that do not consider synchronization explicitly, are likely to lead to low
efficiency on networks of workstations, on parallel multiprocessors that may run more
than a single task, or on irregular problems.

The synchronization cost due to either irregularity or asymmetry in the algorithm,
or to random factors in the run time environment, or both, may not be eliminated,
since it is inherent in the problem and the computing system. We conclude that
a practical technique to deal with synchronization problems involves some form of
overlapping communications with computation.
1.3.1 Communications

We must consider what we mean by communication. In the following we will be
talking explicitly about messages, but communication between one entity or process
to another could be done by writing to shared memory. The following discussion is
couched in the terminology of messages for convenience, this is not meant to exclude
other methods of information transfer. While the ideas described in this section are
not the focus of this work, they serve as background concepts that underlie much of
the subsequent development.

Milner [MILNER 89], Hoare [HOARE 78],[HOARE 85] and others consider com-
munications to be an indivisible act between a sender and receiver. Hoare uses a
channel only to specify which processes may communicate with which others; the
action of a sender in placing a message on a channel and a receiver in taking a mes-
 sage from a channel is simultaneous. Milner goes so far as to consider the transfer
medium as a communicating agent; he therefore models all communications as in-
divisible handshakes performed simultaneously by two agents, and models what we
would call a channel or buffer as a set of agents which hand messages off to each
other [MILNER 89]. While this leads to an interesting algebraic theory, we believe
that such approaches miss relevant features of communication that must be dealt
with. They do, however, define limiting cases of instantaneous message send and
handshake that, although impossible on real hardware, do capture what we mean by
a message and a handshake. It is impossible to send a message instantaneously, as in
CSP. However, this is the ideal case which real message sending tries to achieve, and
can usefully serve as a definition of correctness. We will assume that it is correct for
a message to be instantaneously passed from sender to receiver, and require that our
message semantics be equivalent to this.

Similarly, a handshake can not physically be an atomic operation. In fact it
requires several distinct steps: some process must initiate the handshake by sending
a signal to another process. That other process must detect the signal, and then
respond to it with an acknowledgment. Each of these steps takes time, and the
interprocess communication is not symmetric. However, again we may say that the
ideal handshake would in fact be an instantaneous action as in Milner, and require that the semantics of a handshake be equivalent to this ideal.

Let us go back to what we mean by receiving a message. Suppose we receive an encrypted message. We can measure the amount of information in the message by counting bits (actually the ceiling of the information content, unless we know the entropy of the encoding). Suppose we do not have the encryption key; then we can not do anything with the message - we are unable to access the information it contains. We cannot really be said to have received the message until we receive the key.

We propose that the distinction between sender, receiver and channel is a meaningful one. Agents (eg. channels) that simply pass on a message to other agents (as in Milner) do not need the key, but the agent that is meant to receive the message does.

While our example uses an encrypted message, we believe the concept of a key is actually much broader. Consider receiving a message in plain text which happens to be in Basque. Unless we happen to know Basque, we are in the same situation as having received the encrypted message. We need a key (a Basque to English dictionary, perhaps). When we receive a message in plain text in a language we know, we still need a key, except that in this case we already have it.

Actually we may need additional information to properly interpret the message; assuming it is in a language we understand this usually consists of a shared context between the sender and receiver that may also require added information such as who is the sender. (Consider, for example, the difference in the meaning of the message "Your check bounced" if you know or do not know who the sender is, or what you would infer from it if you do not have a checking account).

We propose that the communication of interest is the transmission of information from a producer of information to a user of information, where the producer is a sender and the consumer is a receiver. We consider the channel, including intermediate senders and receivers, as a separate entity, and further require that the consumer or final receiver must always have additional information which we will call a key, distinct from the message. By a key we mean additional information that allows the
receiver to correctly interpret the information in the message. This may be different depending on the situation we are dealing with. Some examples are in order to clarify the concept.

It is instructive to consider the design of messages intended for radio transmission to unknown, hypothetical alien beings on planets of stars other than the Sun. This would appear at first to be a case where there is no key - the recipient knows nothing about the sender other than the message itself. Proposals for encoding and transmission of such messages depend on properties of the universe and mathematics, that are arguably as accessible to any other beings as they are to us, since they at least live in the same region of the same universe. So for example the message is constructed as a two-dimensional pictorial grid, with a total number of bits being the product of two primes (leaving only two ways to decode it into a grid), the transmission frequency chosen is the so-called water hole, because at least in this corner of the galaxy there is less interference there, and because a wavelength related to absorption frequencies of water molecules may suggest life. The message itself generally includes a decoding key based on physical properties such as the absorption wavelength of neutral hydrogen in interstellar space [SAGAN 83].

Consider messages in a client-server computing environment. The client sends messages to the server, and receives answering messages. Considering the server as receiver, the key is the sender identity and a shared context of message types and formats. Considering the client as receiver, the key is (at least) the ordering in time - that is, having sent a request, the next message from the server can be presumed to be the answer. Note that if the client has no memory of having sent a request (that is, it lacks the knowledge of time ordering), then the message from the server will make no sense to it. Note that this memory does not have to be explicit - it might simply be a state of the client that requires an update from the server.

Consider a system such as the current Pfortran buffered messaging. Sender and receiver have the shared context of executing common code. Senders execute non-blocking sends, which are kept in buffers at the receiver. The receiver blocks until the message is received, unless it finds the appropriate message in its buffer. The key
is the message tag, part of the shared context and attached to the message so that the receiver can pick out the correct one of multiple messages in its buffer. Suppose we had a system that guaranteed message ordering (MPI is one such). Ignore for the moment that the underlying system would need some kind of key to assure ordering - the Pfortran program might not need an explicit tag because it could then use message position in the buffer as the key - but this would still be a key. Suppose Pfortran were rewritten to use synchronized communications. The key would then be the handshake messaging required to do this - that is, information to the receiver that the next message is the one.

Consider also a shared memory system. A put to a shared memory area owned by another process may be considered a message without a key (although there is always at least shared context). In fact, such a put results in non-deterministic execution. A process reading memory that is subject to occasional writes by some other process may not know the meaning of what it is reading. For example, suppose the memory in question is the sum of some set of quantities. Has the sum been completed? If a key is not provided to tell processes how many writes have been made, or in fact if any writes have been made, this cannot be determined. In the case of shared memory programming, the key is usually in the form of a semaphore (or some construct such as a barrier which can be built on top of semaphores).

1.3.2 Keys in a distributed, asynchronous environment:

The problem: given multiple (>2) loosely coupled processors, with the same executable program running on multiple nodes (the SPMD model), and needing to perform multiple communications during the execution, what can we use as keys so receivers of messages can correctly interpret them?

Since the same executable is running on any pair of nodes involved in communication, we can presume that both processes know who the sender and receiver are, the type of message and its format. We still need a key to identify a particular message
out of the set of all messages that a process will receive from all other nodes during execution.

We cannot count on message timing because the processors are not synchronized. We cannot depend on message ordering without additional information, since even if messages from any given node may arrive at a receiver in order, messages from two different nodes may be arbitrarily interleaved.

Without attempting an exhaustive analysis of everything that could possibly serve as a key, we can consider two alternatives: first generation Fortran style message tags for buffered communication, or client-server style keys in which the receiver indicates to the sender that it is ready for a particular message. In the first case the key is presumed known to both processes, in the second case it is sent from the receiver to the transmitter and the transmitter responds. (We can allow out of order reception and buffering of multiple messages at a receiver by appending the key to the returned message so the receiver can determine which of the messages it is waiting for has arrived).

A common feature of both styles of keys is that there is a time lag between the start of communication and its end that depends on events at both sender and receiver. Consider the following:

Case 1: Buffered messaging with shared tag context.

The key in this case is a tag identifying the message. In SPMD, since the code is the same at both transmitter and receiver, we assume that we can match the code that performs a send with the code that performs the corresponding receive (In the following chapters we will consider how to do this in the general SPMD model, in our study of MIPS). The knowledge of which part of the program is performing the send at one process and which part is doing the receive at a different process allows us to set a local tag at each process to the same value. A key insight here is that every send in a correct, non-deadlocking program must be paired with a receive. A pair of communicating processes that perform multiple communications between themselves
can therefore match a sequence of sends with a sequence of matching receives. We
can use this to keep a count of how many communication statements each process has
encountered and use this count as the tag to match a particular send with a particular
receive.

The sender transmits the message together with its local tag value, and continues
execution. Having sent the message, it is free to reuse or update the memory which
contained it. The run time system is required to buffer the message until the receiver
process is ready to read it. The receiver process sets its local value of the tag and
looks in the incoming message buffer for a message with a matching tag. If it finds the
tag value, it reads in the message; otherwise it must wait for the message to arrive,
or perhaps continue computation and try again later.

The tag allows us to decouple the sender and receiver processes. The sender never
has to wait, and in fact there can be multiple messages in the receive buffer, which
the receiver can identify by their tags. The cost is the added memory required to
store multiple messages in system buffers until receiver processes are ready to read
them.

The time lag in this case is the difference between the earlier of the times at which
the sender and receiver set their local key values and the time at which the receiver
reads the message. This includes the actual data transfer time, but also may include
some computation, and possibly wait time if the receiver process sets its key and
looks for the message before it has been sent.

Case 2: Unbuffered, clear-to-send protocol

In this case the key is divided in two parts: the shared context of the SPMD source
program, in which we identify which particular send matches which particular receive,
and a clear-to-send (CS) message, which informs the sender process that it may send.
We will detail the protocol for this in a later chapter on overlapping.

As in the buffered, tag case, we use the idea of a sequence of matched sends and
receives. In this case, however, we will define a protocol where the sender cannot
send until the receiver is ready. It is possible that the receiver is able to receive some value and still can continue computing (if it does not need to immediately use the value), and possibly again signal that it is ready for a value. To allow for this, we keep a count of pending CS messages at the sender. The shared context of the same source program text allows the sender to assume that a positive count of CS messages means it can execute the next send statement it encounters.

The sender must be prepared to receive a CS message at any time, and must keep a count of CS messages from each other process. When the sender reaches the point in the code where it is ready to send, it checks to see if the count of CS messages is greater than zero. If it is, it may proceed to send its message. Then it can decrement the CS count and continue. The sender may have to wait for a CS if the count is zero, and it needs to reuse the message buffer.

When the receiver reaches a statement at which it is ready to receive some value from another process, it signals the other process with a CS. Depending on how quickly the receiver needs to use the value, it may be able to continue processing, or may have to wait for the message.

The time lag in this case is the difference between earliest of the sender checking for CS and the receiver sending a CS message, and the read of the message by the receiver.

In both cases, a lower bound on the time lag is the absolute value of the difference in clock time between the first step at the sender and the first step at the receiver, which is a measure of asynchrony between the processes. For irregular computation in general, and for any computation on non-dedicated processors, this asynchrony may be of the same order as the computation time, and is likely to be greater than the actual message transfer time.

Consider now collective communication. Even in the presence of logically atomic (they cannot be physically atomic) communications primitives, there remains the asynchrony component to the time lag. That is, if we apply the above analysis to a set of communicating processes, we see that the global time lag for a group communications is the time between the first process reaching its key definition, and the last
process reading its corresponding message. We see also that there is a lower bound on this time lag corresponding to the difference between start of communications between the fastest and the slowest process. Our argument that this lower bound is most probably the largest component of the total communications delay applies even more strongly to collective communications, since the maximum time spread between processes, if we assume random differences, increases with the number of processes.

1.4 Short-cutting, Overlapping and MIPS

The above discussion of time lag lower bounds can be considered an argument for load balanced algorithms on dedicated processors, preferably lock stepped with a common clock. What are we to do, however, if such processors are not available, or if we have an irregular computation that cannot be statically load balanced?

One solution is to dynamically load balance by shifting data (Kelp, for example, [FINK 97] and [FINK 96], among others). This is applicable to problems with data dependent irregularity, if we are running on dedicated processors. Even in these cases, we must balance the delays from asynchrony between processes with the delays caused by transferring massive amounts of data between nodes. If a dedicated system is not available, or in cases where there is algorithmic asymmetry or irregularity, data movement is not likely to help. Within the SPMD model we have studied a technique which we call short-cutting, which allows us to avoid unnecessary work in some cases by interrupting parallel processes at run time. This led to the need for a novel approach to the standard technique of overlapping computation with communication, in which we are more concerned with overlapping the execution of sections of code between definition and use of communicated variables at different processes.

In studying overlapping and short-cutting, we have found the need to consider process sets that are formed implicitly by program logic. In a fully general model of SPMD, sets of processes may be formed when control structures such as IF statements depend on predicates that may be different at each process. These implicit process
Figure 1.1: Genealogy of SOS system:

SOS – api/library
MIPS, Overlapping, Short−cutting

SOS compilers automatically insert Ipstream and Overlapping code. (SOS library incorporated as runtime system for PC and Pfortran)
sets are formed dynamically in a parallel execution, and present challenges to parallel programming systems which are not met by present technology.

For example, MPI [MPI 95] and Co-array Fortran [NUMRICH 97] support only process sets which must be explicitly defined by the programmer. It is then up to the programmer to ensure that the process set in fact matches the statements that use it for communication. In the case of MPI, the process set is local to each process and there is no assurance that the same set (communicator, in MPI terminology) at different processes will have the same membership. Titanium [YELICK 98] does not at present support communication between subsets of processes; it enforces a requirement that communication statements (statements with global effect) occur only in sections of the program that are executed by all processes. Other systems, specifically the data parallel HPF [KOELBEL 94], avoid the whole issue by a paradigm in which individual processes are not addressed (HPF allows expression of task parallelism through EXTRINSIC procedures, but these are “not properly part of HPF” - p. 29 in the reference).

First-generation Planguage [BAGHERI 92]) do not forbid communication statements that occur in sections which are not executed by all processes, but the present implementation also does not guarantee that such statements will not deadlock or cause later statements to be incorrect. At present, Planguage programmers wishing to write communication statements that are not seen by all processes may only do this with great care and extensive knowledge of how Planguage statements are compiled into an underlying communication system.

We have developed a theory of implicit process sets which we call MIPS, and implemented support for it in an API and library we call SOS (MIPS, Overlapping and Short-cutting). This theory and system will be described in the following chapters. Although historically short-cutting and overlapping were developed first [GOMEZ 98], we will begin describing the theory of MIPS, because we will need its results for analysis of overlapping and short-cutting. Implicit process sets in SPMD are formed when the same logic statement in each of multiple copies of the program depends on values or predicates that are not the same at all processes; we call these
Definition 1.4.1. Non-uniform expression: An instance \( \alpha \) of an expression in statement \( s \) in a SPMD program \( P \) is \textit{uniform} if and only if, for every initial state \( \Sigma \), when \( P \) executes starting in state \( \Sigma \), every process executes statement \( s \) the same number of times, and the sequence of values computed for \( \alpha \) when statement \( s \) executes is the same for every process.

A non-uniform expression is one which is not known to be uniform. Logic with uniform predicates cannot cause the formation of implicit process sets, but non-uniform predicates potentially can. For example, the process number is a non-uniform value, and any values that depend on it may also be non-uniform.

1.5 Planguages and the P model

The P model, which is the theoretical base for the Planguages, is based on the concept of guarded memory, described by Babak Bagheri in his dissertation [BAGHERI 94]. In the P model, memory is guarded by owner processes; all access to a memory location is through the guard. All updates to a guarded value are performed by the guard, and increment a tag value. Processes request particular versions of the contents of memory from a guard by including the tag value specifying a version number in the request. The semantics of memory access by multiple processes are completely defined by this mechanism. The guard enforces consistency of memory, since all references are through it. The tag mechanism allows processes to run at different rates and still get the correct version of a particular variable.

Essentially tags define a logical time value based on a count of references to the variable in question. A given process can keep track of the number of references it has seen to a variable \( X \); say this count is \( N \). When the process requests the value of \( X \) from its guard, it uses the value \( N \) as a tag and gets a value consistent with the changes the process has seen to occur in \( X \). Bagheri also proved that the restrictions of guarded access provide freedom from deadlock and determinism for a
parallel execution of loosely synchronous communicating processes. Although the P model is not limited to an SPMD programming style, the Planguages are implemented in SPMD.

In the first generation Planguages [PLANG 99], the guard is one of a fixed number of processes executing together. Each process guards its local memory, and each has a local copy of all variables. Through references to special variables numNode and myProc it is possible for each process to have unique non-uniform local values in particular variables. In an SPMD program all processes see the same program text, and therefore it is not necessary for a receiver process to explicitly request data from its guard, since the guard is assumed to execute the same code and see the data transfer. First generation Planguages use tags in all communication statements to allow buffering at the receiver. A sender process reaches a particular communication statement and sends its local value of the requested memory with a tag, then continues executing. A receiver that is running more slowly than the sender could have several tagged values of the same variable in its input buffer, and the tag allows it to pick out the right version.

The tag value of a Planguage variable is incremented at each communication statement: for example the first communication statement in a program, for example the transmission of a variable $X$ from process $p$ to process $q$, would have a tag of 1. The second communication statement would have a tag of 2, and so on (a single tag is currently used for all communication statements of every variable, monotonically increasing with each execution of a communication statement up to the limit of the memory type used to store tags on a particular computer).

Problems with tags can occur if some processes execute a communication statement and others do not. Suppose we have processes 1, 2 and 3 executing the following code (In the code fragment we use the Planguage notation $X@n$ to indicate the value of variable $X$ at process $n$; an assignment statement with @ on both sides is therefore a data transfer between processes):

1. IF( myProc.EQ.2 ) THEN
2. X=1
3. ELSE
4. X=5
5. ENDIF
6. IF( (myProc.EQ.1).OR.(myProc.EQ.2) ) X@1 = X@2
7. X@3-X@1

Suppose our initial tag value is 1. The communication statement at 6 is executed only by processes 1 and 2, but not by process 3. Therefore when statement 7 is executed, process 3 is requesting the value of X with a tag value of 1. At process 1, the value of X at statement 7 is 5, but this value has a tag of 2. X with tag 1 at 1 has a value of 1, so this is the value that statement 7 should transfer from process 1 to process 3. Although this result is semantically clear, it is unexpected to the typical programmer. Additionally, the current Planguage run time system will not support this result, because process 1 does not keep old versions of X.

Currently Planguages deal with this problem by requiring all processes to see every communication statement. In the above code fragment, this is accomplished by eliminating the IF in statement 6. Even though process 3 does not transfer data at statement 6, if it sees the communication it can increment its local tag value. Then, when statement 7 is executed, all processes have a tag of 2, and X at 3 is set to a value of 5, as expected. This approach, however, limits execution to a SIMD (Single Instruction, Multiple Data) style, at least for program sections where communication takes place. Part of the problem is that the semantics of the standard P model lead to results that are hard for a programmer to anticipate in cases where communications are not seen by all processes.

Note that the problem with communications inside code that is not executed by all processes is not merely a problem in the run time system of Planguages. Consider
a somewhat more complex example, this time using MPI, again with processes 1, 2 and 3:

1. TAG = 0
2. Y = myProc
3. IF( D.GT.0 ) THEN
4. X = 1
5. TAG = TAG + 1
6. IF( myProc.EQ.1 ) call MPI_SEND(Y, 1, type, 3, TAG, comm, status, error)
7. ELSE
8. X=0
9. IF( myProc.EQ.2 ) call MPI_SEND(Y, 1, type, 3, TAG, comm, status, error)
10. ENDIF
11. if( myProc.EQ.3 )
12. IF( Flag ) call MPI_RECV(X, 1 , type, MPI_ANY_SOURCE, TAG, comm, status, error)
13. ENDIF
14. ...

If D is the same at all processes, then the receive at 12 matches either the send at statement 6 from process 1 or the send at statement 9 from process 2; there is no problem in this case. If, however, D is not the same at all processes, then this code could lead to deadlock if the tag at statement 12 does not match, or process 1 executes statement 6 and process 2 executes statement 9, in which case one of the sends
would have no matching receive. Even if this code executes without deadlocking, it is possible that the tag value will not agree at all processes, which could cause problems later in the program. We could reduce the chance of deadlock by replacing TAG with MPI_ANY TAG in statement 12. In this case, the receive would match either of the two sends, but if both sends execute then the value of 12 is non-deterministic; the receive would match whichever message happened to arrive first.

We will develop theory and algorithms that will allow us to identify sets of processes and analyze communications both inside sets and between sets. The analysis will be extended to consider data transfer that does not fit into the P model.

1.6 Synchronization and implicit process sets

Individual processes in a parallel execution may progress at different rates. If a given process must interact with another during execution, and we wish that interaction to have the same results on repeated execution, then we will need to coordinate both processes so that, for example, each will have executed the same sequence of instructions before the interaction on each repeated execution. We call this a synchronization. Except at such synchronizations, each process has no information about what other processes are doing. Similarly, except at synchronizations we have no guarantee that any set of statements is being executed at any set of processes at any given external time, or indeed, that any such set of statements occurs together at all during a parallel execution. We are therefore here proposing an execution model for parallel programs in terms of synchronizations and transitions between them by sets of processes.

Synchronization by a set of processes requires action by all processes in the set, and generally requires information of set membership at each process. For example, a barrier instruction acting at a set of processes requires that each process wait until all processes in the set have reached the barrier. This requires knowledge somewhere of what processes are in the set. While the placement of the synchronization instruction must be done statically, when the program is written or at least at compile time, the
set of processes that participate may not be known until run time; it may for example be data dependent. In order to identify process sets that naturally form at run time, we will develop the concept of the implicit process set. We define:

**Definition 1.6.1. implicit process set (IPS)** *In SPMD execution, a set of processes that execute the same code at the same stage of execution. The set of all processes is an IPS that exists when the execution starts. Other process sets are formed at runtime by program logic that evaluates a non-uniform (def: 1.4.1) predicate which results in processes following different execution paths, and by merges between sets of processes formed by a previous split.*

We will develop the theory of MIPS to formalize the concept of IPS. We will identify these MIPS with the path being executed by the IPS in the Control Flow Graph, abbreviated CFG (see [AHO 85]). The formation of an IPS requires interaction between processes and is a synchronization; and synchronizations in an IPS logically involve the member processes. An MIPS exists where we can identify some set of processes executing as a group on the same code, and continues to exist while the set remains the same. The IPS is such a set of processes, which is known at program start. An MIPS splits into multiple MIPS when it encounters a conditional statement which causes different subsets of processes to follow different paths in the CFG; we call the node at which this occurs a split node. If we know the entering MIPS at a split, we can identify the implicit process sets for the departing MIPS.

The key problem in MIPS is identifying where implicit process sets should merge into a single process set. Corresponding to each split node, we wish to find a merge node at which the processes that split will merge again, re-constituting the original IPS. We will deal with this problem in the following chapters.


CHAPTER 2
SERIAL AND PARALLEL EXECUTION

In this chapter we describe deterministic serial execution and use it to define a parallel execution as a collection of serial executions. We use this concept to define a parallel state as a set of serial states with certain properties, and a non-deterministic transition between parallel states.

We here give general definitions. In the remainder of this work we will impose added restrictions and explore a restricted type of execution (which we will call an MIPS execution) that will allow us to guarantee determinism and freedom from deadlock.

We state a concept of progress on a serial execution and use it to define a partial ordering on parallel states that is similar to Lamport’s concept of time for distributed processes [LAMPORT 78]. We use relations between states to establish the concepts of valid state for a particular parallel execution. We also introduce the serial data view trace, which we will later use in defining an analogous concept for parallel execution, and in showing determinism. Finally we give a formal definition of parallel execution and obtain some properties of it.

2.1 Serial execution

We will define execution on the Control Flow Graph (CFG). We will use the standard definition of CFG from [AHO 85] as a directed graph \(< A, V, s, E >\) with vertices \(V\) which represent basic blocks, arcs \(A\) which represent transfers of execution from one block to another, a start node \(s \in V\) and a set of end nodes \(E \subseteq V\).

We will modify the standard definition of basic block, a set of statements which must be executed in sequence. For purposes of our analysis we will add communication
statements to the standard block leaders. A consequence of this is that there can be
at most one communication statement in each basic block.

Let $F = \langle V, A, s, E \rangle$ be a CFG. A serial execution of a program is modeled as
a path in the CFG from the initial vertex $s$ to a vertex in the set $E$ of end vertices.
We assume that program execution progresses from a start node through a series of
steps which may be numbered starting from 0, and that execution continues until the
program reaches an end state or enters an infinite loop.

We consider that a step involves execution of a single node in the CFG, and we
speak of the execution being at a particular state when it executes a particular state.
We define:

**Definition 2.1.1. serial state:** The $j$th state $\sigma_j$ of a serial execution is a tuple:

$$\sigma_j = (x_j, D_j) ,$$

where $x_j$ denotes execution of the basic block $x$ at step $j$, and $D_j$ is the set of data
values stored in memory by the process when execution reaches the entry point of the
basic block after performing $j$ transitions from the start of execution. An immediate
consequence of this definition is that the successor of $\sigma_j$, at step $j + 1$, is $\sigma_{j+1} =
(x_{j+1}, D_{j+1}).$

$D_j$ depends on the entire execution history. Every state implies execution of one
basic block, and every transition is a change from one basic block to another along an
arc of the CFG. The basic blocks appear in an order determined by the CFG, however
the state number $j$ is a simple count of steps or transitions; it always increases in a
serial execution.

We define a transition from one state to a next state:

**Definition 2.1.2.** Given a CFG, $F = \langle V, A, s, E \rangle$, the transition $\sigma_j \rightarrow \sigma_{j+1}$ is a
relation between serial states such that $x_j, y_{j+1} \in V$ are nodes in the CFG and there
is an arc $x \rightarrow y \in A$.

We now define a serial execution as a sequence of states connected by serial trans-
itions. Apt and Olderog [APT 97] show that this form of execution is deterministic.
([APT 97] uses the assumption that a single step may include `skip` statements, assignment or boolean expressions, but not input; and that such statements are not divided between two steps. This is consistent with our identification of a step as the execution of a basic block). A deterministic execution may not terminate, for example due to an infinite cycle. Execution could still be deterministic if it always enters such a cycle given the same initial conditions. We handle such a case (as do Apt and Olderog) by appending a divergent state \( \bot \) to the set of states \( E \).

**Definition 2.1.3.** Given \( F = \langle V, A, s, E \rangle \), let \( i \) be a step number, and let \( x_i \in V \) denote the particular node of the CFG executed at step \( i \). Let \( s \) be the start node and \( e \in E \) be an end node. Then a **serial execution** is a series of states that correspond to a path from \( s \) to either an end node \( e \) or divergence (non-terminating execution) designated by \( \bot \). We denote a path \( P \) of an execution with steps from 0 to \( n \) by:

\[
P = s \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_{n-1} \rightarrow e \land \bot
\]

and an execution \( X \) as:

\[
X = \sigma_s \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_e \land \bot
\]

The **serial steps**, denote successive execution of basic blocks in the CFG starting from the initial node \( s \) and ending in one of the final nodes \( e \in E \); the step number at a particular node indicates the number of blocks that have been executed from the start to reach that node in \( X \).

Note that the order of execution of the nodes \( x_i \) is only restricted by the CFG. *There is no requirement that the nodes appear in any order.* In particular, a given node in a loop may appear multiple times in \( X \); however the step number is different each time a particular node appears in \( X \).

We call such an execution **time-like**, in that the sequence of increasing step numbers gives us a notion of time and a concept of **progress** for the serial execution. Since we always transition to a higher-numbered step, we can say that such a step is always *after*, or *later* than, any lower-numbered step. Progress includes the assumption that the process does not fail to advance to a next step if one exists (the process is not at an end node).
Another way of putting this is to observe that, in order to reach a particular step number, we must transition through all the lower-numbered steps, and we must not have reached any of the higher-numbered steps in the sequence. This gives us a serial execution.

We define:

**Definition 2.1.4.** A *deterministic* serial transition is a state change \( \sigma_j \rightarrow \sigma_{j+1} \) such that \( \sigma_{j+1} \) is the unique successor of \( \sigma_j \). A serial execution of transitions of this type is a sequence of states each of which has an unique successor and is therefore a *deterministic execution*. The execution of a basic block is deterministic if there are no explicitly random instructions. In a deterministic execution, we have that the basic block \( x_j \) reached at step \( j \) is a function of \( j \), since each state has a unique successor.

The serial transition gives us a relation between serial states which defines one state as later than, or occurring after another. Another useful way of considering a serial execution is by considering its view of memory. Although \( D_j \) defines all data that the process could possibly have access to at step \( j \), the process usually does not in fact read or write all this data. When considering only a single serial process, we may also wish to consider the changes that a process makes to \( D_j \) in a transition \( \delta_j \rightarrow \delta_{j+1} \). Note that in considering parallel processes, there is also the possibility that some of the data at one process will change due to the action of some other process.

We define:

**Definition 2.1.5.** At state \( \sigma_j \) of a process, the *data view*, \( \nu_j \subseteq D_j \) is the data that is read or modified by the process at that step. The *data view trace* of a process is the list of the data view at each step. Given an execution that reaches an end state:

\[
X = \sigma_s \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_e
\]

the data view trace is:

\[
V = (\nu_s, \nu_1, \nu_2, \ldots, \nu_e)
\]
The data view at a given basic block can be determined by standard data flow analysis as described in [AHO 85].

Remark 2.1.6. It is evident that a deterministic program executing on the same data will access the same memory in the same steps, and perform the same computation at each step on repeated execution. Therefore it will generate the same data view trace. We can turn this statement around: if the same deterministic serial program executes the same path in the CFG and sees the same data trace in two different executions, then it computes the same results. This is because the data view trace includes all the values that the program sees in its computation, and the steps at which it sees them. Since the program reads no data outside the data view trace, and its computation is deterministic, the results are the same.

2.2 Parallel State and Execution

For motivation, let us think of a “parallel SPMD run” as a collection of serial executions of the same program text (we will formalize this concept below, as the basic idea for our definition of parallel execution). A natural way of defining a parallel state is as a collection of serial states \( \sigma_j = (x_j, D_j) \).

**Definition 2.2.1.** A **parallel state** is a pair \( S_{G,J} = (x_J, D_J^G) \), indexed by a step counter \( J \) which is a multi-index \( J = J_G = \{(j_p) \mid p \in G\} \). Here, \( J \) is an index formed from the indices of CFG nodes \( x_J = \{x_{jp} \mid p \in G\} \) and \( D_J^G \) is a list of mappings from the set of variables indexed by \( J \) to their values at the corresponding step \( j_p \) for each process.

We denote the complete set of processes that comprise a parallel run by a capital \( \Gamma \). The sets \( G \) are just subsets of \( \Gamma \). If \( G \) is a proper subset of \( \Gamma \) we will call the state a ‘partial state’, and if \( G = \Gamma \) we will refer to the state as a ‘total state’. The multi-index notation used here is somewhat non-standard because we need to specify multi-indices that match particular sets of processes. For a given set \( G \), \( J_G \) is a multi-index (a list of non-negative integers) indexed by the members of \( G \). When the
context is clear, we drop the subscript $G$ on $J_G$. We can think of a multi-index $J_G$ as a partially defined object where we could write

**Equation 2.2.2.** $J_G = (i, j, *, \ldots, *, k, *, \ldots)$

where $*$ indicates locations $p$ corresponding to $p \notin G$, and $i, j, k$ etc. are integers. The range of entries corresponds to the cardinality of $\Gamma$.

Since all processes in $\Gamma$ are in some state at all times during execution, there must always be a total state $S_{\Gamma,I}$. We say that a partial state with process set $G$ is **consistent with** a total state $S_{\Gamma,I}$ if every $p \in G$ is individually in the same serial state in both the partial parallel state and the total parallel state; that is, if the index of every $p \in G$ is the same in $S_{\Gamma,I}$ and in $S_{G,J}$. We denote this by a relation:

**Equation 2.2.3.** $S_{G,J} \preceq S_{\Gamma,I}$

Note that the total state fully specifies the individual states of all processes, whereas the partial state does not. As a result, there is a set $\{ S_{\Gamma,I} \mid S_{G,J} \preceq S_{\Gamma,I} \}$ of total states which are consistent with any specific partial state.

Applying $\preceq$ to partial states gives us a relation of increasing specificity. Given $H \subseteq G$ we can say

**Equation 2.2.4.** $\Phi \preceq S_{H,K} \preceq S_{G,J} \preceq S_{\Gamma,I}$

where $\Phi$ is the empty state (a state with no processes which is consistent with all states), and each $p \in H$ has the same index in $K$ and $J$ (and also $I$).

Any total state consistent with $S_{G,J}$ must also be consistent with $S_{H,K}$ because $S_{H,K} \preceq S_{G,J}$; but since $H \subseteq G$, state $S_{H,K}$ is less specific. That is, there are more total states consistent with $S_{H,K}$ than with $S_{G,J}$. We have:

**Equation 2.2.5.** $\{ S_{\Gamma,I} \mid S_{G,J} \preceq S_{\Gamma,I} \} \subseteq \{ S_{\Gamma,I} \mid S_{H,K} \preceq S_{\Gamma,I} \}$

In the sequential case, a computation is just a sequence of transitions $\sigma_j = (x_j, D_j) \rightarrow \sigma_{j+1} = (x_{j+1}, D_{j+1})$. Although a precise notion of time is not relevant for us, we can think of $\sigma_{j+1}$ occurring after $\sigma_j$. In the parallel case, the set of possibilities is much richer.
Definition 2.2.6. A parallel transition is a tuple \( S_{G,J} \rightarrow S_{H,K} \) of parallel states such that

\[ G \cap H \neq \emptyset \]

for each \( p \in G \cap H \), we have \( 0 \leq k_p - j_p \),

\[ \exists \text{ total states } S_{G,J} \preceq S_{\Gamma,J} \text{ and } S_{H,K} \preceq S_{\Gamma,L} \text{ such that for each } p \in \Gamma, \text{ we have } 0 \leq l_p - i_p \leq 1, \text{ and } \]

for at least one \( p \in G \cap H \) we have \( k_p - j_p > 0 \)

We will use the notation \( S_{G,J} \rightarrow_p S_{H,K} \) to denote a parallel transition such that \( p \in G \cap H \) and \( k_p = j_p + 1 \). Note that the \( p \) subscript to a transition \( \rightarrow_p \) does not exclude the possibility that for some \( p \neq q \in G \cap H \) we may also be able to say \( S_{G,J} \rightarrow_q S_{H,K} \).

Note that the parallel transition between partial states exists only where we have a pair of total states consistent with the partial states and that satisfy the same condition on process indices that the partial states do.

Now we can define parallel execution. (We will here deal only with executions that terminate, although they may be arbitrarily long.) First we will define an execution in total states:

Definition 2.2.7. An SPMD total parallel execution \( E_\Gamma \) is a sequence of transitions between total states, starting at \( S_{\Gamma,0} \) and ending at \( S_{\Gamma,\text{End}} \), in which for every process there is a series of transitions \( E_\Gamma = S_{\Gamma,0} \rightarrow^* S_{\Gamma,\text{End}} \).

Note 2.2.8. Here and in the remainder of this work we will use superscripts * applied to a transition to denote 0 or more transitions (Kleene's star operator), and superscript + to denote one or more transitions.

Given SPMD, if each individual process present at the start state reaches an end state, we have a total SPMD execution, and the absence of a total execution implies that at least one process does not reach an end state. We therefore consider that the existence of a total execution is a necessary condition for the existence of any kind of SPMD execution. In particular, we require a total execution to establish the existence of an execution in terms of partial states:
Definition 2.2.9. An *SPMD parallel execution* $E$ is a directed graph of transitions starting at $S_{T,0}$ such that there is at least one total execution $E_T$ that fulfills the condition that for every partial state $S_{G,J} \in E$ there is a total state $S_{G,J} \preceq S_{T,J} \in E_T$.

The above definition makes it evident that we can specify sets of partial states for which no execution exists. A fully general SPMD model, however, needs to include the possibility that specific program statements may only be executed by a subset of all processes. This implies the need for the definition of partial states to describe such an execution. Consequently, when we refer to SPMD parallel execution without other qualification we must include the possibility of partial states, as in the above definition. We will impose additional conditions on such an execution that will enable us to guarantee its existence in chapter 5.

We note that the existence of a parallel transition $S_{G,J} \rightarrow S_{H,K}$ does not require that every $p \in G$ have a serial transition. This reflects the reality of asynchronous processes, which may be progressing at different rates. However, if a parallel execution $E = S_{T,0} \rightarrow^* S_{T,End}$ exists and reaches an end state, it must be the case that every process in it progresses from the start state to the end state. We require therefore the existence of a total execution $E_T = S_{T,0} \rightarrow^*_p S_{T,End}$ that is consistent with $E$. Note that we are not here in a position to say that any execution does in fact exist; we make the more limited point that if it does exist, then every process in it must have a set of transitions from beginning to end.

Theorem 2.2.10. An *SPMD parallel execution* is a Directed Acyclic Graph (DAG).

Proof. If there were a cycle $T = S_{G,J} \rightarrow_p S_{H,K} \rightarrow^* S_{G,J}$, then for some process in $T$ we have $j_p, j_{p+1}, \ldots, j_p$. This requires that there be some total transition $S_{T,J} \rightarrow_p S_{T,K}$ in which the index of $p$ is decremented: $j_{p+1} \rightarrow k_p = j_p$. This is not possible in our definition of parallel transition. So $T$ cannot be part of a SPMD parallel execution.

There is a natural relation $J_G \leq K_J$ defined by $j_p \leq k_p$ for all $p \in G \cap H$. 

\qed
We will say that a state is valid if it could occur in an execution; we may further specify that the state is valid for a particular execution.

**Definition 2.2.11.** A state \( S_{G,J} = \{(i_p, D_j) \mid p \in G\} \), \( S_{G_{tot}} \neq S_{G,J} \) is **valid** if it appears in some parallel execution; it is valid for a parallel execution \( E \) (def: 2.2.9) if it appears in that particular execution.

Because a parallel state can have multiple successor states (def: 2.2.1,2.2.6), we cannot in general show that a valid state does occur; only that it is not prevented from occurring by conflict with a state we know is valid. States which are observed (for example through data output) or can be guaranteed to occur in that execution (such as the start state) are always valid. Other states may be inferred to occur, for example by computed results. However, in some cases it may be that we are unable to rule out the occurrence of some states. For example, if we somehow know we are in a state \((2,2,2)\), we also know that \((0,0,0)\) occurred (the start state). If we do not know how we reached \((2,2,2)\), we can not rule out any state \( S \) such that \((0,0,0) \rightarrow S \rightarrow (2,2,2)\).

**Example 2.2.12.** We can have situations where we can see that only one of a set of states could occur in an execution: for example, we could have \((0,1)\) or \((1,0)\) in the execution of a pair of processes, but once we get \((0,1)\) we know that \((1,0)\) can not happen because it would require process 2 to progress backward.

Here we will introduce definitions and notation for some relations between parallel states without, however, the ability to always construct states that satisfy these relations or always determining if a pair of states does satisfy them.

**Definition 2.2.13.** We define the following relations between parallel states that are valid for an execution \( E \).

i. **After:** \( S_{G,J} > S_{H,K} \) if all paths in the graph of execution of \( E \) must pass through \( S_{H,K} \) before reaching \( S_{G,J} \)

ii. **Before:** \( S_{G,J} < S_{H,K} \), if \( S_{H,K} > S_{G,J} \) (this is just the inverse of After).
iii. **Conflicting**: \( S_{G,J} \nsucccurlyeq \nsucccurlyeq S_{H,K} \), if \( S_{G,J} \) is valid for \( E \), then \( S_{H,K} \) is not valid, and vice versa (see example 2.2.12).

iv. **Independent**: \( S_{G,J} \nsucccurlyeq S_{H,K} \) if both are valid for \( E \) and they could appear in \( E \) in any order.

The above relations are exhaustive, but not exclusive. For example, two states that actually occur in an execution have to occur either before, after, or at the same time as each other; but any of these is possible for two independent states, and two states that do occur at the same time are not necessarily equivalent.

Note also that we may not be able in general to determine that these relations exist between an arbitrary pair of states without first establishing that an execution exists, and that the states we are interested in could be valid for that execution.

Given these relations and the parallel transition, we have:

**Theorem 2.2.14.** Two conflicting states \( S_{G,J} \nsucccurlyeq \nsucccurlyeq S_{H,K} \) can not both appear in the same execution \( E \).

**Proof.** From definition 2.2.6 we know that we have \( H \cap G \neq 0 \) and that for some process \( p \in G \cap H \) we have \( j_p > k_p \) and for some other process \( q \in G \cap H \) we have \( j_q < k_q \).

But then we have \( S_{G,J} \rightarrow_q S_{H,K} \rightarrow_q S_{G,J} \) (a cycle) and if there are a pair of total states such that \( S_{T,J} \rightarrow S_{T,K} \), then there can be no total states such that \( S_{T,K} \rightarrow S_{T,J} \) and one of the transitions in the cycle is invalid. \( \square \)

**Example 2.2.15.** Note that the absence of conflict still does not give us a way to construct an execution only with partial states. We still can specify transitions that imply that some process would have to return to a previous state, which would not be consistent with any total execution, even without conflict in partial states. For example, we can have \((1,0,1,*,*) \rightarrow (1,1,*,0,*) \rightarrow (*,*,0,1,0) \rightarrow (1,0,1,*,1)\). Note that no pair of states conflicts, although given the first transition there is no total state that would permit the second transition in the sequence since the third index would have to be decremented.
It is possible to have states that are valid for every execution of a program. For example, the start state is always valid. Depending on the style of execution, we can show that some states are always valid, and some are never valid. In a lockstep execution (section 4.2), states in which the step numbers at each process are the same as at all other processes are valid, and any state in which two or more processes are at different step numbers are not valid.

We could also state that the start state is always valid, and in a terminating execution so is the end state. A problem that we will address in the following chapters is ensuring that states in which communication or synchronization occurs are valid.

2.3 Sets of executions

Recall that a parallel state does not necessarily have a unique successor state. As a result of this, repeated execution of the same program, on the same data, with the same number of processes does not necessarily imply that the same states will occur in every execution. However, it may be possible to show that even executions that do not repeat the same states do repeatedly compute the same results from the same input. If we are mainly interested in the final results of a computation, we would consider this level of determinism to be sufficient.

In general there needs to be some interaction between processes so that we can say that a state occurs, or is forced to occur. This is because, absent such interaction and permitting asynchronous processes, serial states in individual processes may occur in any order. In such a case there is no way even of knowing that any particular parallel state did occur. We may be able to demonstrate that a particular state is valid in more than one execution.

For example, the start state is valid for all executions. A state in which one process sends data and a different process executes a matching receive must be a valid state in all executions that actually execute the data transfer; note that if the send and receive actually take place both processes must reach specific nodes in the CFG at specific stages of their respective execution.
**Theorem 2.3.1.** The start state $S_{Γ,0}$ is valid state for any set $\{E\}$ of executions of the same program.

*Proof.* Because all processes start at step 0, every execution is consistent with a state where all processes are in the start node. \qed

We would like to show that there are other valid states in a parallel execution, besides the start state. A major result of this work will be to show conditions under which the end state is always valid.

**Remark 2.3.2.** We note that correct execution requires that variable declarations be in valid states. This is immediately true for global variables which are declared at the start of a program, but it is also requisite for any variables local to specific routines. This is because processes that interact with each other make references to variables defined at other processes, and such variables must exist for the reference to work.

The existence of a particular state limits what other states can appear in an execution. For example, if an execution of three processes includes the state $(1,1,\ast)$, then the state $(2,0,\ast)$ is not valid since no total state could be consistent with both the given states. If we know that some states are always valid for some set of executions, this limits the set of all valid states that could appear.

**Definition 2.3.3.** Given a set of executions $\{E\}$, the set of all valid states $U_{\{E\}}$ contains those parallel states that are either present in all the executions, or at least are not a priori prevented from appearing in any of the executions.

We now define a parallel data view trace:

**Definition 2.3.4.** Given a set of serial processes each having its own serial data view trace (def: 2.1.5), the parallel data view trace of a parallel execution is:

$Δ_{E_{\text{Gtot}}} = \{VP\}$,

where $VP$ is the data view trace of process $p$. 
Two instances of a serial execution can be said to be the same if they exhibit the same sequence of states; this is the case for execution of the same deterministic program text on the same data. If the parallel data view traces of two different executions $E_1$ and $E_2$ are the same, then we would like to claim that they are two instances of the same execution, even though the sequence of parallel states in $E_1$ is not the same as that of $E_2$. We judge two serial executions to be the same based on their states. We would like to be able to make a similar statement about parallel executions:

**Definition 2.3.5.** We will say that two executions $E_1 \approx E_2$ are equivalent if their respective sets of valid states (def: 2.3.3) are equal: $U_1 = U_2$.

To observe a state it must be possible to see or deduce the effects of the state from the parallel data view trace (def: 2.3.4); such states limit what valid states can appear (theorem 2.2.14). This leads to the following:

**Theorem 2.3.6.** Given an execution $E_1$ and an execution $E_2$ on the same data distribution $D$, if $E_1 \approx E_2$ (def: 2.3.5) and any information transfer is deterministic, then their parallel data view traces (def: 2.3.4) are equal: $\Delta_1 = \Delta_2$; therefore both executions compute the same results.

**Proof.** The process set $\Gamma$ must be the same for both executions, or we could not have $E_1 \approx E_2$.

Since the data distribution is the same, if there is no interaction between processes then each deterministic serial process in the execution has the same data view trace (def: 2.1.5) and must compute the same result, therefore so will the parallel executions.

If there is interaction between processes that modifies the data view trace, then the state in which this happens is a valid state (this will be proved in lemma 3.1.2), which must occur in both $E_1$ and $E_2$ because their respective sets of valid states are equal: $U_{1,D} = U_{2,D}$ by definition 2.3.3. Therefore information transfers in $E_1$ and $E_2$ occur in the same states. By assumption the information transfer is deterministic, therefore its results at each state are the same.
Since the data view trace at each process was the same except possibly for interaction between processes, and the interaction between processes must occur at the same states with the same data in both executions, then the changes to the data view trace at each process in both executions $E_1$ and $E_2$ are the same. Therefore the parallel data view traces are the same $\Delta_1 = \Delta_2$, and the computed results of both executions are the same.

**Definition 2.3.7.** We will say that a set $\{E\}$ of parallel executions with the same input data set $D$ is results-deterministic if every execution in the set has the same data view trace $\Delta$. We will say in this work that a parallel program is **deterministic** if it is results-deterministic.

This level of determinism is sufficient when we are interested only in the results of computation. There are cases, such as real time control applications, in which a stricter definition of determinism is required, but we will not address them in this work.

The description of a parallel execution, unlike that of a serial process, is not unique. We can for example describe the same execution as a total execution (def: 2.2.7) or as an execution with partial states (def: 2.2.9), and we may be able to find multiple total executions consistent with any given (partial state) execution. Repeated (asynchronous) execution of the program may result in a different series of transitions, even if we have the same data set and the same set of start and end states at each process, since only states with communication or synchronization statements are forced to appear.

Even for a deterministic execution under a stronger definition than we are using we can have different descriptions. We can see this by considering that we could select different sets of processes in describing the same execution. For example, the description of trivial parallelism could also fit lockstep execution if we simply do not group processes. However, such a description would ignore information about the execution which would otherwise be available. Our previous characterization of a parallel execution (def: 2.2.9) is too loose to describe particular types of execution.
We note that, if there are interactions between the serial states during execution, these lead to valid parallel states (lemma 3.1.2), which in turn limit the possible valid states in an execution.

We can, for some executions, define total states which are valid for every execution. For example the global barrier synchronizations of BSP [BSP 96] are such states; by limiting communications to occur at these global barriers BSP ensures that all processes are present at all communication sections, and that if each communication section succeeds, the next section can be reached. In general, all other SPMD systems with which we are familiar (see the first chapter) enforce the same kind of limitation on communication, though not usually as formally as BSP.

We will later (Chapter 5) show that it is possible to extend a BSP-like model to cover partial states.
CHAPTER 3
COMMUNICATION

Although some useful forms of parallel processing involve simply running multiple programs independently on the same or different data sets (e.g. trivial parallelism), the most interesting cases involve processes that in some way interact with each other while they are running. This requires information to pass from one process to another, which may be as simple as an indication that some other process is present or has reached some specific point in its code or as complicated as a computation performed by all processes on some shared or transmitted data structure, or anything in between.

In this chapter we first give an overview of information transfer. We classify information transfer as two-sided or one-sided, depending on the actions required at processes involved in the transfer. We examine synchronization as a form of information transfer.

We here define the communication membership function, which may be computed (under some restrictions) to determine if a particular communication will succeed. We describe fusion communications as a general form of two-sided information transfer that can be shown to be deterministic and free from deadlock.

In the following discussion, we assume that communication is reliable, and that each process has an unique identity known to the all processes participating in communication.

3.1 Information transfer

Typically information transfer involves passing some value from one process to another, but it may also be something on the order of a handshake (as used for example by [MILNER 89]) in which the information is simply that both processes are at a
certain point in execution. Information transfer between a set of serial states implies a parallel state that includes those serial states; if this were not the case then we would have to require some third entity in addition to processes that could hold the information being transferred for the time between the action of $p$ and the action of $q$. If in fact there is something like a channel which can hold information after $p$ produces it, so that $q$ can obtain it at some other time, then we will require that this channel cannot change the meaning of the communication. In other words, the effects of the information transfer must be such that the data view trace (def 2.1.5) at both processes is the same as that in which information transfer occurred instantaneously, and it must at least appear as if there were a parallel state that included the serial states between which information is transferred.

**Definition 3.1.1.** We will say that information transfer occurs between a process $p$ and a process $q$ if process $p$ performs some action (for example writing a data value $v$ to memory or sending that data value) when $p$ is in state $(j_p, D_j^p)$, and process $q$ performs some action which depends on the action of $p$ (such as reading or receiving that data value $v$) when $q$ is in some state $\delta_q \geq (j_q, D_j^q)$, where we further assume that $j_q$ occurs at an external time no earlier than $j_p$, so the information is not available before $q$ reaches state $(j_q, D_j^q)$ . We use the $\geq$ sign to indicate that the action of $q$ might not take place in the same state in which the information is logically available, it could take place at that state or at some later state.

The meaning of information transfer is frequently taken to be that the actions of $p$ and $q$ are simultaneous; that is, $\delta_q = (j_q, D_j^q)$ (as in CSP [HOARE 85]). If there is in fact some underlying mechanism such as a channel or a buffer that makes it possible that the actions at $p$ and $q$ are not simultaneous, this may not affect the state either of process $p$ or of process $q$ (if there is such an effect at either process, then the system or channel must be explicitly included as a communicating agent, as in [MILNER 89]). Note that $p$ and $q$ are not necessarily in the same basic block or at the same execution step.
We call the particular statements at processes \( p \) and \( q \) that transfer information *communication statements*, and we modify the standard definition of basic block leaders (see [AHO 85]) to say that communication statements are leaders.

Note that, since information transfer statements are block leaders, we can only have one such statement in a basic block. By our definition of a program step in a serial process (in section 2.1), this means that two different information transfer statements must occur in two different steps, since they are in different nodes of the CFG. We in fact modified the basic block definition to this end, otherwise the CFG would not be sufficiently fine grained to analyze communications.

We will also consider collective communications which involve multiple processes. For all cases of interest we will be able to define these collective communications in terms of multiple information transfers between pairs of processes. However the syntax expressing such collective transfers frequently takes the form of single statements, and in such cases we will view them as if they were in fact single communication actions. For example, a barrier can be defined in terms of multiple messages between pairs of processes or it may use some collective mechanism such as a hardware synchronization network. Although this makes a difference to external timing, the effect on the execution is the same: no process can progress to another state until after the barrier, and the syntax for either underlying action may be the execution of a single barrier statement at each process.

Information transfer must happen in a valid state by definition 2.2.11.

**Lemma 3.1.2.** A state \( S_{G,J} \) in which communication succeeds is valid in the execution \( E \).

**Proof.** For communication to occur, execution must reach the state \( S_{G,J} \) and the process set in the state cannot be empty; therefore there is a transition \( S_{T,0} \rightarrow^* S_{G,J} \) in \( E \), and the state is valid by definition 2.2.11.

If information transfer happens, we can use its consequences to infer that a particular state actually occurs in an execution. This is closely related, but not identical to the above lemma. The difference is that a valid state is not prevented from occurring,
but does not necessarily occur in fact. If information transfer occurs between, for example, a pair of processes in \( G \), then we know that the state \( S_{G,J} \) is valid, but we do not actually know that it exists in the execution because processes in \( G \) that are not communicating may in fact be in steps that are different from those specified in \( J \); we have no way of knowing the actual state of these other processes without some interaction between processes or with the outside world.

A state is observable, and can be said to occur, if there is information transfer such that the effects of the information transfer appear in the data view trace of processes that participate in the state (def:2.1.5). Note that for us to be able to say that a state occurs we must be able to prove that it did occur through some property of the state or through results indicating information transfer. To say that a state is observed to occur, there has to be information transfer in that state, since without some interaction between processes there is no difference in the computed results or in the data view produced by serial processes executing independently.

We have seen that communication occurs in a valid state. This implies that to verify that a process \( q \) can communicate with some other process \( p \), we need to know that both are in some state \( S_{G,J} \). In the absence of this knowledge, we cannot show that communications will succeed.

At a state \( S_{G,J} \), we can identify a nodes \( x(j_p) \) at which each \( p \in G \) executes when in the state. By lexical analysis of code in that node (section 12.2), we may be able to determine the set of processes \( c_p \), to which \( p \) will send or from which \( p \) will receive, when in state \( S_{G,J} \), at least in terms of variables or data structures that will contain the actual process identifiers at run time. Our ability to do this may depend on the particular communication syntax; we will show that this analysis is always possible for fusion communication statements (see def: 3.4.1).

In a point-to-point communication it is a logical requirement that sender and receiver be specified, although part of this specification may be implied (and if sends and receives are separate instructions, it may not always be possible to match a particular send with a particular receive). For example, the sender or receiver may only be specified implicitly as the process that executes a send or receive statement, respectively.
We can assume, however, that we know the local process (for example, the special variable myProc in Plangages [PLANG 99] or the result of MPI_COMM_RANK in MPI [MPI 95]).

In the case of a collective communication such as a broadcast there is some data structure that stores the members of the set involved in the communication (for example, an MPI communicator). In particular, for collective communications there is some predefined communication pattern that allows each process to determine, given its own identity, what other processes it needs to send to or receive from - for example, a process may locate its identity in a spanning tree which it can compute at run time, and its position in the tree then tells it which other specific processes it needs to communicate with. Although the identities of the processes are not known statically, the algorithm to build the tree is known, which allows us to statically know (that is, at compilation) a function of $p$ which at run time will yield $c_p$. (It is necessary that we be able to do this statically, because if we could not, then we could not guarantee that all processes would compute the same communication pattern at run time, and communication would fail).

### 3.2 The membership function

The communication set at state $S_{G,J}$ is the union of the communication sets at each of the processes in $G$: that is, $C(S_{G,J}) = \bigcup_{c_G \in G} c_G$.

Given a state, we can define a function that lets us determine if the set of communicating processes is in the state:

**Definition 3.2.1.** We define the **membership** function as a boolean function of the set of processes $C(S_{G,J})$ involved in communication at a state:

$$\exists(C(S_{G,J})) = (C \subseteq G) = \bigwedge_{c_G \in G}(c_G \subseteq G).$$

$\exists$ is **TRUE** if every process in $C(S_{G,J})$ is in $G$ and **FALSE** otherwise. If no communication takes place in $S_{G,J}$, then the communication set is empty: $C = \emptyset$, and $\exists$ is defined as **TRUE**. We have communication failure if $c_g \not\subseteq G$ for some $g \in G$. 
It may be difficult or impossible to compute $\exists(C(S_{G,J}))$, for example because the necessary information may not be available at every process and at every state in a parallel execution. However, it is often sufficient for our purposes to know when $\exists$ is FALSE, since in such a case we know communication will not be correct.

Consider a set with a single process $p$: since $C(S_{G,J}) = \cup_{G \cap p}$, the restriction to the communication set for $p$ gives us $C = c_p$. Therefore we have $\exists(C(S_{G,J})) = \wedge_{p \in G} \exists(c_p)$, and communication would fail at $p$ if $\exists(c_p) = $FALSE. We can therefore conclude that if the local communication membership function $\exists(c_p) = $FALSE at any process $p \in G$, then $\exists(C(S_{G,J})) = (C \subseteq G)$ must be FALSE also.

The local communication function $\exists(c_p)$ may be computed from information available at $p$, since we know $c$ from the program text and $p$ and $G$ are known at run time. We use this property to obtain scalability in the implementation of the run time system, since for communication to succeed we only need that a process $p$ know the status of other processes in $c_p$ rather than that of all process in $G$.

Usually it will be enough, for practical purposes, if $\exists(C(S_{G,J}))$ may be computed at every $p \in G$ when $p \in H$ for $S_{H,K} \geq S_{G,J}$. For example, if $\exists$ was FALSE at some individual $p$, we will eventually know this at every process. We will later, in our discussion of MIPS (chapter 5) discuss restrictions on a parallel execution which will allow us to always compute $\exists$ in this way.

**Definition 3.2.2.** If the membership function $\exists(C(S_{G,J}))$ is TRUE, we will say that the state $S_{G,J}$ is **closed under communication**. That is, all processes in $G$ need to communicate only with other processes in $G$.

Information transfer can take several forms; we will particularly distinguish transfers between pairs of processes that require actions at both processes, which we will call two sided, and transfers which only require action at one process which we will call one sided.

**Definition 3.2.3.** A **two-sided** information transfer requires action at sender and receiver processes and links a particular state at (at least) one process to a particular state at a different process or processes.
We will say that the two sided information transfer is a message, if it requires p to perform an action (which we call a send) when in state \( (j_p, D^p_j) \), that matches an action (which we call a receive) performed by q in state \( \delta_q \geq (j_q, D^q_j) \). If the action of p is the sending of some data value \( v \), then process q cannot use this value until it has reached step j of its execution, and it must be able to use it at step j.

A handshake can be considered a message in which the only information conveyed is the fact that the message was sent and received, and this information is visible at both processes (i.e., the receipt of the message is acknowledged or guaranteed). If the action is some form of handshake, process q cannot see it or react to it until it reaches step j, but it must at least acknowledge the handshake when it reaches that step.

Note that a handshake must be two sided [MILNER 89]. A basic block specified in a state may include code to ensure two-sided information transfer (defs: 3.1.1, 3.2.3). If any processes execute one of the basic blocks specified in the state, that process must then perform information transfer. If the information transfer succeeds, we can infer that other processes in the state also performed their part of the transfer and we can conclude that the state was valid in the execution. (If the successful information transfer involved all processes in the state, then we can conclude the state was not only valid, but actually present).

We may also have one sided information transfer.

**Definition 3.2.4.** A one sided information transfer requires action at a single process \( p \) in a state \( (j_p, D^p_j) \), and links that state at \( p \) to some state \( \delta_q \) at a different process \( q \) without restricting that state. Information transfer implies that \( \delta_q \geq (j_q, D^q_j) \), but in a one sided transfer there are in general many states \( \delta_q \) that satisfy this condition.

We will say that an information transfer is a **put** of a value \( v \) from \( p \) to \( q \) if, when \( p \) reaches state \( (j_p, D^p_j) \), it places the value \( v \) in some location accessible by \( q \) (that is, in \( D^q_j \), the data set of \( q \)); and if the **put** action places no restriction on the state of \( q \). However, \( q \) is nevertheless at some execution step \( j_q \) when the **put** occurs, therefore
process $q$ does not have access to $v$ until it has reached step $j_q$ of its execution; and it does have access to $v$ after that step.

We will say that the data transfer is a get of $v$ at $p$ from $q$ if, when $p$ reaches state $(j_p, D^p_j)$, it obtains the value $v$ from the location where $q$ wrote it or will write it (that is, from the data set of $q$, $D^q_j$); and if the get action places no restriction on the state of $q$. Process $p$ does not have access to $v$ until it has reached step $j_p$ of its execution. In this case there is no restriction on the state of $q$, however process $q$ is in some state and executing some step $j_q$ when the get occurs. Therefore the value $v$ that $q$ gets is the value that $q$ had computed at step $j_q$. We note that a signal can be considered a put in which no added information is conveyed beyond the fact that the put occurred; we can use signals to construct a handshake.

In either case we can say, as we did in definition 3.2.3, that states $(j_p, D^p_j)$ and $(j_q, D^q_j)$ appear to be simultaneous in the parallel program run; however since one sided information transfer places no restriction on the state of $q$, it is possible that if the parallel run is repeated, the particular step $j_q$ and the data set $D^q_j$ may be different when the information transfer takes place.

## 3.3 Synchronization

The term synchronization is often used to cover mechanisms that allow for any kind of interprocess coordination. Tanenbaum [TANENBAUM 95] classifies, critical sections, mutual exclusion, election algorithms, atomic transactions and various forms of clock algorithms as synchronizations. The NYU Ultracomputer project [GOTTLIEB 87] adds coordinating operations based on their proposed fetch-and-add instruction that fetches a value in memory, adds an increment to it, and stores the modified value, and which allows multiple parallel fetch-and-add to operate simultaneously (this has the effect of accumulating all the adds in memory, and providing a different partial sum value to each process that executes fetch-and-add). Algorithms based on this operation are used to distinguish different parallel tasks, and to implement readers/writers protocols.
We see two common threads in the above. Firstly, the action of a synchronization is a two-sided information transfer (def: 3.2.3) in order to support interprocess coordination. It generally requires execution of code at all involved processes, and therefore occurs at a specific node or set of nodes in the CFG.

Secondly, the synchronization returns a value at each process. In the case of a barrier synchronization, we can consider its action to be defined by the following code:

\[
\text{until(\text{synch}(G))};
\]

such that this code is inserted at the nodes in the CFG on which the synchronization is defined, and \( G \) is the participating process set. The synchronization then can be considered a function on \( G \) that returns TRUE when all the processes in \( G \) are present, and otherwise diverges. It is possible that a synchronization will return a different value at each process. For example a semaphore returns TRUE at one process and FALSE at all others; the fetch-and-add Ultracomputer instruction mentioned above can be used to write a synchronization function that returns a different number at each process.

A key difference between a general parallel state and a state that includes a synchronization is that information transfer that occurs at a synchronization happens at a specific and repeatable point in the program execution. Information transfer at some parallel state that is not a synchronization may occur at a different state if the execution is repeated, and therefore may transfer different values each time, even when the input data to the program is the same.

Note also that a parallel state is defined on a specific set of processes, but the restrictions that force a synchronization are normally part of the program code; that is, they are attached to a particular basic block or set of blocks in the CFG (or else they are conditions that force such a block or set of blocks to occur due to the logic of the program or execution model). Therefore it is possible to refer to the same synchronization in two different executions of a parallel program even if the number of processes and input data are different, but it would not make sense to speak of the same state in such a case.
Definition 3.3.1. A synchronization is a state in which code is executed that performs two-sided information transfer (defs: 3.1.1, 3.2.3) The synchronization returns the value of a corresponding synchronization function at each process that executes the code:

\[ F_G = \text{synch}(S_{G,i}) \]

where \( F_G \) is a list of values of the synchronization function indexed on processes in the synchronization state (these values are TRUE \( \lor \bot \) for a barrier, TRUE \( \lor \) FALSE for a semaphore, and may be other things depending on the particular function; for example a list of participating processes). Note that \( F_G \) may be non-deterministic if its results depend on non-deterministic arrival order of processes at the synchronization. On the other hand, a single-valued function such as that used in a barrier must be deterministic.

In general we are interested in synchronization and two-sided communication in order to ensure determinism: If each serial process has the same input data on repeated execution, and a synchronization ensures that a particular parallel state appears, then the individual serial states \( \sigma_p \) of processes participating in the synchronization will have the same memory contents \( D^P_j \), and two-information transfer will contain the same data and change memory at each process in the same way on each execution. This is necessary for our definition of determinism (def: 2.3.7) to hold.

The restriction to the same data at the same step in each process is not true of one-sided communication since such communication does not restrict the state at all involved processes; however two-sided communication can be written in a way that leads to deadlock. Consider the following MPI code:

1. /* shift x value to process at right */
2. /* find out number of processes and local process number */
3. MPI_Comm_size(MPI_COMM_WORLD, pmax);
4. MPI_Comm_rank(MPI_COMM_WORLD,me);
5. /* set source, dest processes arranged as a ring */
6. if( me+1>pmax ) dest=0; else dest=me+1;
7. if(me-1<0 ) then source=pmax-1; else source=me-1;
8. /* send to process on right, get from left */
9. MPI_Recv(y, size, type, source, MPI_ANY_TAG, MPI_COMM_WORLD);
10. MPI_Send( x, size, type, dest, MPI_ANY_TAG, MPI_COMM_WORLD);
11. x=y;

This code is intended to arrange processes logically as a ring, connecting the highest
numbered process to process 0, and then to shift the value of x at each process to
the process on the right. It matches sends and receives at each pair of processes, so
is a two-sided information transfer. In fact this code deadlocks; no process can reach
the send statement, because every process needs to execute the receive first. We have
here a circular data dependence (see def: 3.4.3) in which all processes execute the
same code.

This particular example can be fixed by exchanging the order of send and receive
at one process; however it is possible to have more subtle errors that are harder to
detect and repair.

It is difficult to come up with a general definition of deadlock; usually we find dis-
sussions in the literature of conditions for its avoidance or detection, and examples of
when it occurs [TANENBAUM 95]. Tanenbaum, in [TANENBAUM 81], character-
izes deadlock in a protocol graph as a subset of states such that there is no transition
out of the subset and no transition that causes “forward progress”, but does not define
what is meant by such progress. We will take a more limited approach, and define
an absence of deadlock:

**Definition 3.3.2.** The state \( S_G,J \) **does not deadlock** if, for \( S_G,J \neq S_I,End \) there is
some consistent total state \( S_G,J \preceq S_I,J \) and there is no \( p \in G \) which is not in a final
state $\sigma^p_{End}$ and has no transition $S_{G,J} \rightarrow_p S_{H,K}$ to any other state $S_{H,K} \neq S_{G,J}$. That is, if a state is not deadlocked, every serial process that has not already reached the end has to be able to transition out of the state.

### 3.4 Fusion statements

We have previously [PLANG] defined fusion statements, a form of interprocess transfer that includes synchronization. We now give a variant on our previous definition of fusion statements:

**Definition 3.4.1.** A fusion statement is a single statement that expresses a complete (two-sided) communication, identifying producers and consumers either explicitly (e.g. a send-receive) or implicitly (e.g. a broadcast), and which optionally specifies computation on communicated values.

We will take fusion statements to include not only those that transfer data from memory at some process or processes to memory at other processes, but also statements that transfer only synchronization information (e.g. barrier statements), as long as they meet the conditions of requiring interaction between processes which execute the same text.

Lexical analysis of a fusion statement allows us to identify the communication set, either as an explicit set of processes or as the set of all processes that execute the statement (the implicit process set, def: 1.6.1).

For example, a Planguage send of a variable $x$ from process $p$ to a variable $y$ at process $q$ is written as a single statement:

$$ y @ q = x @ p; $$

In MPI, an equivalent construct combines send and receive in a single statement:

$$ MPI\_Sendrecv(x, xsize, xtype, q, y, ysize, ytype, sendtag, p, recvtag, comm); $$

A Planguage send of $x$ from process $p$ to the variable $y$ at all processes that execute the statement (logically a broadcast) is written:

$$ y = x @ p; $$
In MPI, an equivalent broadcast is:

\[ y = x; \text{MPI\_Broadcast}(y, \text{ysize}, \text{ MessageType}, p, \text{comm}); \]

In the Planguage case, the receivers of the broadcast are the implicit process set executing the statement. In MPI, the set of processes receiving the broadcast is the set of processes in the communicator \text{comm}, which must be in the implicit process set executing the statement for correctness.

Note that fusions may include computation. For example, reductions may be expressed as fusion statements; in Planguage we have:

\[ y = \text{op}\{x\}; \]

where \text{op} is a binary Abelian operator. This statement denotes the pairwise application of \text{op} to \( x \) at each process executing the statement. For example, if \( \text{op} \) is +, then \( y \) at every process executing the reduction (the implicit process set) is set equal to the sum of the values of \( x \) at all processes. In Planguage, the order in which the operator is applied is unspecified and may vary upon repeated application. MPI has several statements that include computation, such as MPI\_Reduce and MPI\_Reduce\_Scatter (where MPI\_reduce sets the resulting value at a single process, while reduction followed by scatter is similar to the Planguage reduction). As in the Planguage, MPI does not specify the order in which \text{op} is applied, but it does allow the programmer to specify that the order be the same on repeated execution.

We may generalize the concept of reduction to allow arbitrary computation on values communicated by a fusion statement, and the delivery of different results to different consumer processes (A simple send-receive is an example of delivery of different results to different participating processes, but we also have more complicated fusions such as MPI\_Scan which implements a prefix operation in which the value received at each process depends on its identity).

We cannot guarantee that a complex fusion statement involving multiple processes is correctly implemented; we can, however, state the following:

\textbf{Theorem 3.4.2.} A fusion statement \( F \) can express any deterministic combination of communication and computation involving a set \( G \) of processes, and can be imple-
mented in a way that does not deadlock if the state in which $F$ appears is closed under communications (def: 3.2.2) and if the computation does not diverge.

Proof. Proof is by construction. We first note that the state executing $F$ is closed under communication, therefore all involved processes are present. Since communication is assumed reliable, a single send-receive statement between a pair of processes will succeed.

Given a set $G$ of processes each of which has a different process i.d., we designate one of them as leader (for example the process with the lowest i.d. value). Given $N$ processes in $G$, we create a list $L = (p_0, p_1, \ldots, p_{N-1})$ of processes, sorted by process i.d., where $p_i$ denotes the process i.d of the process ranked $i$ in the list.

We can gather any and all information available to processes executing $F$ by executing a sequence of send-receive communications:

$$for(i = 1; i < N; i = i + 1)\{ \text{send-receive } x_i \text{ (from } p_i \text{ to } p_0); \}$$

where $x_i$ is any data present at process $p_i$. Since we have a sequence of send-receive statements each of which succeeds, the sequence executes to completion. Since we specify participants of each send-receive and sequence of communications, the communication is deterministic.

We have data $(x_0, x_1, \ldots, x_{N-1})$ at process $p_0$ and perform any computable series of operations on the data. Since the $x_i$ potentially include any data present at each process when the statement $F$ is executed, the result may be any set computable from the data present at all processes when $F$ is executed. Computation at $p_0$ is standard serial computation as in definition 2.1.3; it is deterministic if each serial transition is deterministic 2.1.2.

If the computation does not diverge, we again execute a sequence of send-receive communications, communicating results from $p_0$ to the other processes:

$$for(i = 1; i < N; i = i + 1)\{ \text{send-receive } r_i \text{ (from } p_0 \text{ to } p_i); \}$$

where $p_i$ is a result of computation at $p_0$, and each $p_i$ may have a different value depending on $i$. Since we have a specified sequence of send-receive statements each of which succeeds, the sequence executes to completion and is deterministic. 
\qed
The theorem says nothing about efficiency of implementation; it will usually be possible to implement communication and computation more efficiently than expressed in the proof. We limit ourselves to the statement that any deterministic combination of computation and communication that can be performed from data present at any or all processes executing \( F \) can be written as a fusion statement.

A key point in all the above examples is that senders and receivers all execute the same statement text. This allows us to show that we cannot write a circular dependence with fusion statements, as long as all processes execute the same text.

**Definition 3.4.3.** To graph data dependences in a set of program statements we first let each statement be a node. We then draw arcs from statements at which a variable is written or updated to statements at which data is used. A **circular dependence** is a situation in which this graph contains a cycle.

Circular dependences produce deadlock or **race conditions**, because each statement in the cycle requires that some other statement complete first, in order to execute correctly. If the statements can execute in any order, at least one has to execute first and it will be incorrect because the statement on which it depends has not executed. We have a race condition in which what actually happens depends on which statement is executed first. If each statement is blocked from executing until the statement on which it depends has executed, then no statement in the cycle can execute first, so they all remain blocked and we have deadlock.

**Theorem 3.4.4.** The dependence graph of a sequence of unconditionally executable fusion statements (def: 3.4.1) is acyclic (that is, does not contain a circular dependence (def:3.4.3 ).

**Proof.** Since each fusion statement is unconditionally executable, all processes executing the code segment encounter every fusion statement in the same sequence.

Since each fusion statement expresses a complete communication, the only dependence a fusion statement can have on other statements is a data dependence, which must be satisfied by execution of some previous statement. Therefore a graph of
dependences can contain only arcs from a statement to later statements, and has no cycles.

\[ \square \]

3.5 Non-fusion message passing

Traditional message passing systems frequently distinguish between sends and receives, at least in some of their communication primitives. MPI [MPI 95], for example, has MPI_SENDRECV which has the form of a fusion point to point transfer, but the standard way of defining such communication is with separate sends and receives.

Unless there is some guarantee in the program that a particular send and its corresponding receive will in fact be executed by the correct processes, this mode of communication can lead to deadlock. Many present parallel programming systems (Co-array Fortran [NUMRICH 97], Titanium [TITANIUM 00], first generation Plang 99) restrict communication to sections of code that will be executed by all processes to avoid this problem. (Plang do not prevent communications in code sections that are not global, but current versions do not support such code and require an understanding of the run time system by the programmer to make the communications work in a predictable fashion).

In order to extend non-fusion message passing to a model which supports communication in or between implicit process sets, it is necessary to perform analysis equivalent to what we will describe for MIPS (chapter 5). Without guarantees of which implicit process set will include a particular process at run time, it would be necessary to place both send and receive statements on every path in the CFG that may be executed by the process in question for each particular communication. It would also be necessary to surround sends and receives with logic to prevent them from being executed by incorrect processes.

Such analysis also must be performed for fusion objects; if a data transfer is to be performed between a pair of processes which may be executing one of several possible paths in the CFG, it is necessary to place the fusion object in all possible paths. However, the code is simpler because only a single statement needs to be inserted,
and the analysis is easier to do because statements in different parts of the code that are part of the same fusion object (e.g. if the fusion statement implies a state which includes several nodes in the CFG) are textually the same.
CHAPTER 4
EXAMPLES OF PARALLEL EXECUTION

In this chapter we give examples of several types of parallel execution and show how they may be described in our framework. We describe the particular form of SPMD parallel execution to which the rest of this work applies, which we term *loosely synchronous* execution.

4.1 Trivial parallelism

If there is no interaction between parallel processes, or all interactions are one-sided we have a trivially parallel execution. Such an execution must exist if each serial execution exists, since there are no restrictions on allowed parallel states. In such an execution any transition from a parallel state is allowed; processes may execute in any order. In fact, processes do not even need to start together; one way of executing a set of trivially parallel processes is simply to execute them serially, one after another.

This style of execution is not deterministic if there is any communication between processes. Suppose a process $p$ calculates a value and a different process $q$ requires that value; without some link between the execution of processes $p$ and $q$ we cannot guarantee that the correct value will exist when $q$ needs it. For that matter, given the possibility that processes could be executed serially, there is no guarantee that that process $p$ is even running when process $q$ requires the calculated value.

4.2 Lockstep parallelism

A way to prevent communication from introducing nondeterminism is to run processes in lockstep; that is, to force all serial processes in a parallel execution to transition at the same time. This usually leads to the SIMD style of parallel execution; intuitively
the simplest way to insure that communication statements will match up at different processes is to ensure that all processes run the same code at the same time. SIMD also guarantees that all processes will execute the same number of statements and thus can start and end together. We can describe such an execution as follows:

\[ S_{G,0} \rightarrow S_{G,1} \rightarrow S_{G,2} \rightarrow \ldots \rightarrow S_{G,End}, \text{ where } G = G_{total}. \]

In this case the only parallel successor state we are allowing is the unique state in which every process advances by one step; we could rewrite this execution using the deterministic transition we will define in 5.0.1. In this case we also have that the process set at each state is the set of all processes. All processes transition together between states, so all will have the same index; we indicate this with the numbers 0, 1, 2, \ldots instead of a multi index.

A problem with lockstep parallelism is non-uniform predicates which may cause processes to execute different blocks of code. Consider for example code of the form:

IF ( x ) THEN
A1
A2
...
ELSE
B1
B2
...
ENDIF

The standard technique is to set a flag depending on the value of x at each process, and transform the code into:

IF (flag==TRUE) A1
IF (flag==TRUE) A2
..  
IF(flag==FALSE) B1
IF(flag==FALSE) B2
Every statement is executed at every process in lockstep, but processes at which
the flag is false simply wait for the others. This has the disadvantage that the total
number of steps executed at each process is the sum of the number of steps in all
the alternative code blocks, but it allows definition of a valid state at each step with
\( G = G_{\text{total}} \).

In this case, the set of all valid states is:

\[ U = \{ S_{G_{\text{tot}}J} | i = j, \forall i, k \in J \} \]

which is the same for every execution. Therefore, by theorem 2.3.6, the results are
the same for computations with the same data and process set.

SIMD execution has been productive where the machine architecture is designed
for this style (e.g. the CM-2 Connection Machine [JOHNSON 93]), and to some de-
gree where hardware support is at least available (e.g. Purdue PAPERS [DIETZ 94]).
SIMD is seriously limiting where, for example, data dependences or algorithmic asym-
metry require application of different instructions to data. Even when two different
calculations are logically independent and could be performed in parallel, SIMD re-
quires that they be executed serially, even going to the extreme of executing null
operations on some processes that could otherwise be performing different useful
work.

4.3 Loosely synchronous execution

We will here investigate the properties of an intermediate style of parallel execution
in which, without going to the extreme of lockstep execution, we can nevertheless
show that some states appear in a parallel execution, or can be forced to occur. We
say that we have a \textit{forced state} if we can identify (or cause to appear) some point in
the parallel execution in which two or more processes execute specific serial states in
such a way that these states appear again if we repeat the parallel execution.

In an SIMD execution, every state is required to be present; in a trivially parallel
execution no state is required to appear. What we are looking at is an execution
mode in which some states are forced to appear, but not all. We will use the term
loosely synchronous to describe an execution style in which processes advance independently of each other much of the time, but sometimes synchronize. For example, we could have all processes always start together, so there would be a repeatable initial state. We could then have all processes perform computation without communicating, and periodically synchronize and transfer data: this is the BSP processing model [VALIANT 90]. However, BSP is still unnecessarily limiting. Processing is carried out in supersteps consisting of a pure computational section without communication followed by a global barrier synchronization after which all communication is done at once. This delays communications that could have taken place during computation, enforces a coarse-grained parallelism that is not always desirable, and maximizes network congestion during the communication phase.

We would like a less restricted style that allows us to mix communication with computation and does not require periodic synchronization of all processes at every step.

Consider the following pseudo-code:

1. start( P processes )
2. .... do computation
3. if( my_process is odd )
4. ............ send x to my_process+2 mod P
5. ............ receive x from my_process-2 mod P
6. else if( my_process is even )
7. ............ send y to my_process+2 mod P
8. ............ receive y from my_process-2 mod P
9. endif
10. .... continue computation
11. ....

Statement 1 is a (repeatable) state where we start all processes. Line 2 indicates a computation section where all processes can progress independently of each other, without communication. Then lines 4-9 have the effect of synchronizing separately at all odd numbered processes and at all even numbered processes. After the data transfer, processing continues independently at all processes. This execution cannot be represented on one line, as we have done in our previous examples. To show how it would be represented, we first need to analyze the above pseudo-code into basic blocks, which we may enumerate:

1. (statements 1, 2 and 3)
2. (statements 4, 5 and 6)
3. (statements 7, 8 and 9)
4. (statement 10 and following)

A process executing this code fragment would execute three steps, since depending on its process number it would execute either block 2 or block 3, but not both. We show this as a graph in figure 4.1

We can also have forced states that affect only some of the processes, for example by allowing processes to communicate at any time: consider the following pseudo-code:

1. start( processes p,q,r,...)
2. .... do computation
3. if( my_process == p ) send x to q else if (my_process==q) receive x
4. .... do computation: z = function( x )
5. if( my_process==q ) send z to r else if (my_process==r) receive z
Figure 4.1: Parallel execution follows the CFG; transitions indicated here are deterministic.
At line 3, $p$ sends the value of $x$ to $q$, which receives it. Suppose we have atomic messages, as in CSP. Then for the message transfer to take place, $p$ and $q$ must synchronize so that $p$ executes a send instruction for $x$ at the same time at which $q$ executes the matching receive instruction. If $p$ reaches the send instruction before $q$ reaches the receive instruction, $p$ must wait for $q$; likewise $q$ must wait if it reaches the receive instruction before $p$ reaches the matching send. Therefore the transmission of $x$ from $p$ to $q$ synchronizes those two processes at a specific pair of matching serial states for $p$ and $q$. (With buffered message transfer the semantics remains the same; therefore the program must act as if $p$ and $q$ had synchronized at the message transfer).

In this example the message transfer is the first forced state after the program start. Processes $p$ and $q$, each of which is a deterministic serial process, would each execute its own set of states until each reaches the point of the message transfer, where they synchronize. Note that although each process is deterministic, and executes the same set of states in the same order each time, there is no particular restriction on whether a particular state of $p$ will occur before or after a particular state of $q$, until the message state is reached. So repeating the parallel execution could result in many different sets of parallel transitions, but all would include the particular state where $p$ is sending and $q$ is receiving.

The forced state between $p$ and $q$ does not include other processes; some process $r$ that is not involved in the message transfer between $p$ and $q$ is not restricted as to what state it can be in when the message is transmitted. However, if, for example, $q$ calculates something with the value it received from $p$ and then must send that result to $r$, there will be another forced state between $q$ and $r$, and this state must occur after the first such state. That is, if $r$ reaches the point where it must receive the message from $q$ first, it must wait for $q$ to first receive its message from $p$ and then perform its corresponding calculation before sending to $r$.

We are interested in dealing with this last case, where processes can proceed independently of each other except at synchrons (see def: 5.1.3), which may include some subset of all processes. Characterizing the set of valid states for a particular
execution is more complex than for the previous two models, because we have a set of states that are forced to appear and this limits the valid states, but there may not be a simple relation between these forced states. Additionally, we may have one-sided information transfer (def: 3.2.4) that may allow us to observe the effects of a state that is not a synchronization and does not necessarily occur on repeated execution of the same program on the same data.
CHAPTER 5
MIPS

We now introduce new concepts we call synchrons and MIPS. Synchrons are parallel states that will allow us to speak of sequences of state in parallel execution, but without necessarily requiring code to implement actual barriers. MIPS are constructs that will allow us to describe sets of processes in SPMD execution, and restrict changes in the process set. We show which MIPS may be concurrent with each other and use this property to express rules for information transfer between processes in different MIPS.

We are going to work with a more restricted transition between states, which we will call the double-arrow. This is a restricted type of parallel transition in which all processes participate; we will be able to show determinism for this transition in some cases.

Definition 5.0.1. **Double-arrow**: Given two states $S_{G,J}$ and $S_{H,K}$ then $S_{G,J} \Rightarrow S_{H,K}$ if $G \cap H \neq \emptyset$ and $\forall p \in G \cap H$ there is a serial transition $(i_{jp}, D_{jp}) \rightarrow_p (i_{kp}, D_{kp})$.

Note that double-arrow applies only to the processes in common between a pair of states $p \in G \cap H$. Therefore we can have several such transitions entering or leaving a given state. Specifically:

Remark 5.0.2. If $S_G \Rightarrow S_H$ and $G \supseteq H$, then there can exist transitions $S_G \Rightarrow S_{\{H_i\}}$ such that $H \cup \{H_i\} = G$, $H \cap H_i = \emptyset$ and $\forall i \neq j, H_j \cap H_i = \emptyset$. That is, we can have multiple $\Rightarrow$ transitions out of a state with set $G$, to multiple states which together include all the processes in $G$ and which are disjoint in the sense of having no processes in common.
If $G \supset H$, then there may exist transitions $S_{\{G_i\}} \Rightarrow S_H$ such that $G \cup \{G_i\} = H$, $G \cap G_i = \emptyset$ and $\forall i \neq j, G_j \cap G_i = \emptyset$. This is the reverse of the above situation; we can have multiple disjoint states which together include all the processes in $H$, and we can have multiple $\Rightarrow$ transitions into $S_H$.

Combining the two possibilities, we see that we can have multiple $\Rightarrow$ transitions into and out of some particular state $S_G$, as long as every $p \in G$ is in one entering $\Rightarrow$ transition and in one departing $\Rightarrow$ transition, the process sets of all states from which we have $\Rightarrow S_G$ transitions are disjoint, and so are the process sets of states to which we have $S_G \Rightarrow$ transitions.

### 5.1 MIPS and Synchrons

We now use the double arrow transition to formalize the concept of a set of processes that execute together.

**Definition 5.1.1.** An **MIPS** $\Phi_G$ is a set of states that are closed under communication (def.3.2.2) with the same process state $G$ connected by $\Rightarrow$:

**Equation 5.1.2.** $\Phi_G = (S_{G,I} \Rightarrow S_{G,J} \Rightarrow S_{G,K} \Rightarrow S_{G,L} \ldots)$

Since $G$ is constant and the relations depend only on the indices, we may instead write: $\Phi_G = (G, (I \Rightarrow J \Rightarrow K \Rightarrow L \ldots))$.

The states in an MIPS have additional properties. We will call these states **synchrons** because they have some properties in common with synchronizations, and also because we will be able to demonstrate that a barrier synchronization at a synchron involving all processes present at a synchron would not change the semantics of the program.

**Definition 5.1.3.** A **synchron** is a state $S_{G,J}$, inductively defined as follows:

We define the start state $S_{G,0}$ as a synchron. An arbitrary state $S_{G,J}$ is a synchron iff it is closed under communications and there is a transition $S_{H,K} \Rightarrow^+ S_{G,J}$, with $H \cap G \neq \emptyset$, where $S_{H,K}$ is a synchron (as described in note 2.2.8, $^+$ is used here to denote one or more transitions).
We will call a synchron a **point synchron** if all processes in the state are at a single node in the CFG.

If there is no communication in a state, then the membership function $\exists(C(S_G,J))$ is TRUE by definition 3.2.1; if a state includes communication, then we must be able to compute $\exists(C(S_G,J))$ to verify that the state is a synchron.

Note that the communication membership function $\exists$ must be computed on a state at runtime. Strictly, then, all we can say statically is that a state could be a synchron; to verify that a state in a particular execution is in fact a synchron we need to verify $\exists$ at runtime by means of additional code.

**Definition 5.1.4.** The first synchron in a MIPS is called the **leader**, which we write $L(\Phi_G)$ and the last is called the **trailer**, which we write $T(\Phi_G)$

### 5.2 Operations on MIPS

The start of a program is a MIPS leader, and the end of a program is a MIPS trailer, but it is possible for a MIPS to split into multiple MIPS at a trailer, or for several MIPS to merge into a single MIPS at a MIPS leader.

We can use $\Rightarrow$ to define a relation on MIPS:

**Definition 5.2.1.** A MIPS $\Phi_H$ is the **successor** of a MIPS $\Phi_G$ if there is a transition $T(\Phi_G) \Rightarrow L(\Phi_H)$. We will write this $\Phi_G \Rightarrow \Phi_H$.

We also have:

**Definition 5.2.2.** Suppose that there are multiple $\Rightarrow$ transitions from a trailer 5.1.4):  

**Equation 5.2.3.** $T(\Phi_G) \Rightarrow L(\Phi_{G_1}), T(\Phi_G) \Rightarrow L(\Phi_{G_2}), \ldots, T(\Phi_G) \Rightarrow L(\Phi_{G_M})$

such that $G = \bigcup_{i=1,\ldots,M} G_i$ and $G_i \cap G_j = \emptyset$ if $i \neq j$. Then we say that a **MIPS split** occurs and define

**Equation 5.2.4.** $\Phi_G \Rightarrow \Phi_{G_1}, \Phi_G \Rightarrow \Phi_{G_2}, \ldots, \Phi_G \Rightarrow \Phi_{G_M}$
Since a parallel transition (def: 2.2.6) is only defined for non-empty sets of processes, it is possible that the number of sets $G_i$ may vary between different particular executions of the same program; we may need to consider a limiting case in which there is a single set $G_i$ on the right-hand side.

We will be identifying MIPS splits as occurring at nodes of the CFG with multiple outgoing edges; that is, nodes where program logic determines which outgoing path each process will take. The identification of MIPS with paths on the CFG limits the possible number of MIPS that may result from a split. Note that it is possible, however, that some outgoing path will not be taken by any process, or that all processes will take the same path; this would result in a 'split' with a single resulting process set in some executions.

Corresponding to a split, we also define a merge.

**Definition 5.2.5.** Suppose that there are multiple $\Rightarrow$ transitions to a leader 5.1.4):

**Equation 5.2.6.** $T(\Phi_{G_1}) \Rightarrow L(\Phi_G), T(\Phi_{G_2}) \Rightarrow L(\Phi_G), \ldots, T(\Phi_{G_M}) \Rightarrow L(\Phi_G)$

such that $G = \cup_{i=1,\ldots,M} G_i$; and $G_i \cap G_j = \emptyset$ if $i \neq j$. Then we say that a **MIPS merge** occurs and define

**Equation 5.2.7.** $\Phi_{G_1} \rightarrow \Phi_G, \Phi_{G_2} \rightarrow \Phi_G, \ldots, \Phi_{G_M} \rightarrow \Phi_G$.

Since the relation $\rightarrow$ is defined in terms of $\Rightarrow$ which is in turn defined in terms of $\rightarrow$, Theorem 2.2.10 implies the following.

**Theorem 5.2.8.** The graph of the set of MIPS with relations $\rightarrow$, given an initial MIPS $\Phi_G$, is a DAG (Directed Acyclic Graph), and the underlying set of $\rightarrow$ transitions forms an execution.

**Proof.** From definition 5.0.1 and theorem 2.2.10 we have that the graph of all MIPS is a DAG.

From definitions 5.2.2 and 5.2.5 of split and merge, we see that every process is always part of one and only one MIPS, and the total number of processes does not change.
By definition 5.2.2, the same processes exit in disjoint departing MIPS as entered the split. Since the process sets in departing MIPS are disjoint, they are consistent with each other. Therefore if the state before the split is consistent with a total state, then a state can be constructed by taking one state from each of the departing MIPS, and this state will be consistent with a total state (since the new state has the same processes as the consistent state before the split).

By definition 5.2.5 the same processes exit in the merged departing MIPS as entered the merge. The process sets in entering MIPS are disjoint because they must have been formed in splits, so they are consistent with each other. Therefore if the states before the merge are consistent with a total state, then a state after the merge will be also consistent with a total state (since the new state has the same processes as the consistent states before the split).

The initial MIPS $\Phi_G$ is made up of total states, which are by definition consistent with total states. Since the only operations that produce partial states are splits and merges, and both of these operations preserve consistency with total states, therefore all states in the MIPS DAG are consistent with total states, and we have an execution by definition 2.2.9.

From the theorem we see that the successor relation (def: 5.2.1) is transitive and induces a partial order on MIPS. We can now also define:

**Definition 5.2.9.** A MIPS $\Phi_H$ is **concurrent** with a MIPS $\Phi_G$ if there is no path $T(\Phi_G) \rightarrow^* L(\Phi_H)$ in the MIPS DAG of theorem 5.2.8.

It is important to observe that the underlying element in the definition of MIPS is the synchron, which is a state satisfying a particular communication condition 3.2.1 which may require verification at run time for a parallel execution. While it may be possible in special cases to prove the existence of an execution based on MIPS via static program analysis, in general this may only be done **a posteriori** based on an actual computation.
5.3 Parallel time

A time-like relation $>$ can be established between synchrons in MIPS as follows:

Given two states $S_{G,J} \in \Phi_G$ and $S_{H,K} \in \Phi_H$, both appearing in an execution $\mathcal{E}$, and the MIPS in which they appear: We have $S_{G,J} > S_{H,K}$ (that is $S_{G,J}$ appears in execution after $S_{H,K}$), if $H = G$ and $J_G > K_H$, or if there is a path trailer($\Phi_H$) $\Rightarrow^+$ leader($\Phi_G$) in the synchron DAG. Therefore $>$ is a partial order on synchrons in MIPS.

We show this by observing that if $G = H$ then by definition 2.2.6 we have transitions from $S_{H,K}$ to $S_{G,J}$ in every process in both states, so $S_{G,J}$ is unambiguously later than $S_{H,K}$. If $H \neq G$ then the states are in different MIPS. The trailer of $\Phi_H$ is the last state in the MIPS and must occur after every other state in the MIPS. Therefore $S_{H,K}$ is either the trailer or before the trailer. Similarly, the leader of $\Phi_G$ is the first state in the MIPS and must occur before every other state in the Pstream. Therefore $S_{G,J}$ is either the leader or after the leader of the MIPS.

Since the graph of all synchrons in MIPS is a DAG by theorem 5.2.8, the synchrons must be a partially ordered set; since the synchrons are connected by transitions, the ordering means that synchrons are before or after other synchrons depending on the direction of the transitions. However, since we have a partial order, we are not always able to establish that one synchron is later than another.

$S_{G,J}$ is equal to or later than a synchron which is reachable by $\Rightarrow$ transitions from another synchron which is equal to or later than $S_{H,K}$. Therefore $S_{G,J} > S_{H,K}$ and this means after in the sense of a temporal relation between states. If the two states were in MIPS that are not connected by $\Rightarrow$, then one is not reachable from the other and we cannot order them. We will say that such states are concurrent.
CHAPTER 6
MIPS ON THE CFG

In this chapter we describe how to identify MIPS split and merge nodes on the Control Flow Graph (CFG) of a program. Our selection of split and merge points in the program has the consequence that an MIPS set is implicitly formed by program logic, and therefore corresponds to the implicit process set we previously defined (def: 1.6.1). We show that there is unique closest complete merge node to every split node, at which all processes departing the split node must merge.

We consider the possibility of a multi-way split, produced by a node of outdegree more than two. Such a split leads to the possibility that some of the process sets exiting the split will merge before the others do, and that multiple merges will correspond to a single split. We show that the number of partial merges is linear in terms of the number of departing edges from the split node. We introduce a data structure we call a merge tree to identify and support the process sets formed at partial merges.

We extend our analysis to reducible CFGs that contain loops.

6.1 MIPS identification on the CFG

From definition 5.1.1 we see that we need to compute the membership function \( \mathcal{S} \) (def: 3.2.1) which can in general not be done until runtime. However, we can set up conditions which will allow us either to ensure that states are closed under communications or at least to easily verify (at runtime) that they are or are not closed. To accomplish this, we analyze the CFG, which we also used to define parallel execution in general (def: 2.2.9). In particular we want to identify sections of code in which all processes must execute the same statement sequence, sections in which it is possible for processes to take more than one alternative, and statements in which
processes following different statement sequences will continue execution on the same statement.

Nodes of the CFG are basic blocks: sets of statements which must be executed in sequence. The CFG (Control Flow Graph) [AHO 85] is a representation of the basic blocks in a program and the relation between them. We represent a specific program execution as a path through the CFG.

Here we present one way of determining points in a CFG where we need to insert code to change process set membership. Split nodes are easily identified on the CFG as nodes of outdegree greater than one with non-uniform predicates to select the outgoing path. The major challenge is identifying nodes at which merges occur. We want to find the first node or nodes following a split node (defined below) such that processes that took different paths after the split must meet and thereafter follow the same path.

**Note 6.1.1. Deriving the CFG from program code.**

The algorithms in [AHO 85] for identifying basic blocks and generating the CFG are designed to be applied to intermediate code rather than to high level source code. However the intermediate code used in [AHO 85] is not optimal for MIPS analysis, because logic statements (such as case or switch) that may transfer control to multiple targets may be translated as a series of two-way choices. If a single statement with \(N\) possible outcomes is translated into \(N - 1\) two-way choices, this may result in the identification of \(N - 1\) split nodes instead of a single split node.

We will see in section 7.1.2 that split nodes require communications at runtime. Replacing a single split with \(N - 1\) splits could multiply communications cost by a factor up to \(N - 1\). While the analysis of an \(N\)-way split is more complicated than that of a two-way split, the saving in communication costs should easily outweigh the extra complexity. Therefore we recommend either deriving the CFG from high level code directly, or using an intermediate representation that preserves multiple choice logic.

Syntactically a subroutine call is a single statement. If multiple processes in an MIPS call a subroutine from the same code block, all of these processes return to the
same position in the calling code and resume execution at the statement following the call. Therefore we regard a subroutine call as a single statement executed by a fixed process set, from the viewpoint of the calling program.

A subroutine has a CFG of the same form as a full program. It has a single entry point and possibly multiple exit points. Execution of a subroutine is analogous to that of an entire SPMD program in that we have a fixed set of processes that enter it at one point and complete their execution at a return statement. We therefore treat each subroutine as if it were a separate program, with a start node at its entry point and end nodes at return statements. We will perform the same analysis and transformations to subroutines that we do to the main program code, and perform the same code insertion. In particular, each subroutine will have its own copies of data structures used to support MIPS, which will be initialized on entry to the subroutine, maintained on the system stack if or when a subroutine calls another, and deleted upon return from the subroutine. (An exception to a single point of entry is Fortran subroutines that use the entry statement. This allows defining different entry points, call parameters and return types within a single subroutine. The implications of multiple entry points in a subroutine are the same as for a program; i.e. the CFG for the subroutine will in general be different, have different connectivity and data dependences. Therefore a subroutine with multiple entries will be treated as a different subroutine for each entry statement).

In the following sections we will describe code applied to a single CFG; subroutines are single statements in this CFG and each routine will be individually analyzed and transformed in the same way as the main program.

We show an example of a program CFG in figure 6.1.

At the start node we initialize a data structure identifying the total set $\Gamma$ of processes, as well as code capable of keeping track at each process of subsequent information about process sets (described below). In a successful execution all processes reach the end node, therefore we again have the set of all processes at the end node. At the end node we may need to do some system-specific clean up to exit gracefully
Figure 6.1: Control structures and CFG

Node S: an IF-THEN-ELSE statement, with the left portion of the graph being executed if the condition is true, and the right side of the graph being the ELSE case.

Loop L9: a FOR or WHILE loop, with exit from the header node.

Loop L1: a DO-WHILE loop in which the exit node is at the tail of the back edge instead of at the header.

Node 1: a simple IF statement, which either permits the execution of the code in nodes 2, 3, 4, 5, 6 and 7 or skips directly to node 8.

Node 2: a SWITCH statement (in C/C++) or a computed GOTO (in Fortran).
and terminate the parallel computation. (End nodes might also be merge nodes, and these would get additional treatment as indicated subsequently).

Any nodes with communication must be identified; at these, we add code to verify that the communication condition (that all processes in the communication set are present) holds, as well as code to handle irregular termination when the communication condition does not hold. We will simply assume that termination is signaled to all processes in this case. Code generation for such a case can be more complicated, allowing generation and output of information to help in debugging, but this is beyond the scope of the present work.

We now describe split and merge nodes. An SPMD computation splits into multiple paths in the CFG at a conditional statement with a non-uniform predicate. We call the node at which this occurs a split node (defined below, in 6.1.4). Merge points in the CFG are nodes where different processes in parallel computation which have previously split into different paths could (or, more precisely, will have) come together. Corresponding to each split node, we wish to find a merge node (def: 6.1.5) at which the processes that split previously can be guaranteed to re-join. We now describe these points more precisely.

**Remark 6.1.2.** If it is uncertain whether any particular node is a split node it is not incorrect to treat it as a split node; such a node would act as a split in which all processes took the same outgoing arc.

The case of a uniform predicate, which sends all processes on the same exiting edge from a node, can be included in the case of a split. Determining uniformity can only be done approximately (conservatively), so we leave for subsequent work the discussion of methods for determining uniformity by code analysis. (In the general case, interprocedural analysis would be needed.) The advantage to limiting the identification of split nodes is only in terms of efficiency, since the knowledge at compile time that a process set will not split in passing through a node keeps us from having to do work to determine the resulting set at runtime.

**Remark 6.1.3.** We follow two guiding principles in deciding where we will have MIPS splits and MIPS merges. Firstly, we require all processes in the same MIPS to be
executing the same code (def: 5.1.1). This greatly simplifies communication between processes in the same MIPS set, because all such processes agree on the set and can execute a single communication statement (specifically a fusion statement 3.4.1). Secondly, since we are communicating inside an MIPS set, we would therefore like to include as many processes as possible in our MIPS. This guiding principle leads us to merge MIPS at the earliest identifiable point in execution where we can carry out a merge.

Split nodes are easily identifiable, because they contain statements that may cause the flow of control to take alternate paths in the CFG.

**Definition 6.1.4.** A split node is any node in the CFG of outdegree greater than one with a non-uniform (def: 1.4.1) predicate. At such nodes, we insert code to identify the sets taking each branch. (A node of in-degree greater than one and outdegree greater than one may be both a split and a merge node (def: 6.1.5). It will usually be convenient to split such a node into two nodes, one of in-degree greater than one and outdegree one followed by a node of in-degree one and greater outdegree, thereby separating the merge from the split.)

We informally identify a merge node as the first node following a split node such that paths that leave the split must enter the merge node.

A complicating factor in a general CFG is that there may be loops between a split and its matching merge. Such loops could potentially lead to a process remaining in the loop and never reaching a merge node. We will initially limit our analysis to the case an acyclic CFG, and later extend our treatment to programs with loops, in section 6.3.

**Definition 6.1.5.** A node $m$ in a CFG $F = \langle V, A, s, E \rangle$ is a merge node corresponding to a split node $l$, a set of $D > 2$ departing edges $\{l \rightarrow b_i\}, 0 < i \leq d$, and any node $e \in E$ if all paths $l \rightarrow b_i \rightarrow^* e$ are of the form $l \rightarrow b_i \rightarrow^* m \rightarrow^* e$ for $i = 1, \ldots, d$. $m$ is a first merge node if there is no node $n \neq m$ which is on all paths $l \rightarrow b_i \rightarrow^* n \rightarrow^* m$. $m$ is a complete merge node if it is a merge node for all edges departing from $l$. 
That is, every path to an end node $e$ of the CFG that exits $l$ on any one of a set of departing edges must also pass through $m$. The merge node $m$ depends on $l$ and the set of departing edges $D$; it could be different for different departing edges.

Remark 6.1.6. If the CFG takes the form: $F = < V, A, s, e >$ with a single end node, then we can identify a complete first merge node for all split nodes, since at least $e$ is a merge for everything. If there are multiple end nodes, we transform $F$ to single-ended form by adding a single node $e$ and jumps to $e$ from all the original end nodes. This transformation does not change the meaning of the program since it only introduces code after the original program end, and this code only stops.

We have that an MIPS executes a specific path on the CFG:

Lemma 6.1.7. Two processes $p \in G$ and $q \in G$ which are both members of the same MIPS $\Phi_G$ must follow the same path on the CFG.

Proof. Immediately follows from definitions of MIPS (def: 5.1.1), split node and merge node (defs: 6.1.4, 6.1.5).

\[ \square \]

6.1.1 Merge nodes and the Merge Tree

We now need some standard definitions about graphs:

Definition 6.1.8. Let $(a, b)$ be nodes in a directed graph such that there is a path $a \rightarrow^* b$. Let $P_{\text{min}} = a \rightarrow^* b$ be the path with the least number of edges connecting $a$ and $b$. Then the distance $d(a, b)$ from $a$ to $b$ is the number of edges in $P_{\text{min}}$. Note that distance on a directed graph depends on the order of nodes in the node pair; there may not be a path $b \rightarrow^* a$, and even if there is such a path we may have $d(a, b) \neq d(b, a)$.

There is a standard ordering relation on a directed graph with single entry, called the Dominator, or DOM, which allows the identification of pairs of nodes such that, for a path from the start node of a graph to reach the second node, it must pass through the first. Our strategy to identify merge nodes is to invert the CFG and define this DOM relation on the inverted graph. If a split node is dominated by
another node in the inverted graph, this means that all paths from the end on the
inverted graph must pass through this other node before reaching the split node. In
the original graph, this means that all paths to the ends of the program must pass
through the node identified by DOM on the inverse, so this is a merge node. Since
DOM is an ordering relation, it will also allow us to identify a closest merge. Standard
algorithms for finding DOM are described in [AHO 85].

We now introduce some notation and properties of DOM.

**Definition 6.1.9.** A node $a$ in a CFG $F = \langle V, A, s, E \rangle$ dominates a node $b$ in $F$
if every path from $s$ to $b$ passes through $a$. We write: $a$ DOM $b$. DOM depends on
the choice of the start node for all paths. If not otherwise specified, DOM on a CFG
is computed from the start node $s$. We will use the notation $\text{DOM}_n$ to denote DOM
calculated from some node $n$ in a directed graph, other than the given start node.

Some properties of DOM (for a fixed start node $s$) are [AHO 85]:

DOM is asymmetric: if $x$ DOM $y$, all paths from $s$ to $y$ pass through $x$. Even if
there is a path from $y$ to $x$ (a cycle), we need to pass through $x$ to reach that path,
so we do not have $y$ DOM $x$.

The DOM relation is transitive: if $x$ DOM $y$ and $y$ DOM $z$, since we must pass
through $x$ to reach $y$, it must also be true that $x$ DOM $z$.

Given nodes $x, y, z \in V$, if $x$ DOM $z$ and $y$ DOM $z$, then either $y$ DOM $x$ or $x$
DOM $y$. By definition, for $x$ and $y$ to dominate $z$, every path $s \rightarrow^* z$ must include
both $x$ and $y$; but if there is no DOM relation connecting $x$ and $y$, then it would be
possible to reach either $x$ or $y$ without passing through the other. It must also be
possible to reach $z$ from either $x$ or $y$ without passing through the other. But then
we could have a path $s \rightarrow^* z$ that only passed through one of $x$ or $y$, and both could
not dominate $z$.

From the above properties, a graph of the DOM relation is a directed acyclic
graph in which there is a single path from the root to any node; such a graph is a
tree. It is convenient to represent DOM computed from a start node $n$ as a DOM
tree rooted in $n$. We will use the notation $T_{G,n}$ to denote a DOM tree on a directed
graph $G$, rooted in node $n$. Where $G$ is a CFG, and the DOM tree is rooted in the start node $s$, we may sometimes omit the node designation and use the notation $T_G$, or simply $T$ where the context is clear. See [AHO 85] for a description and theorems about DOM trees.

Aho, Sethi and Ullman [AHO 85] establish that the DOM relation applies between any two nodes that can reach each other in a reducible CFG. In the following proofs, we make use of DOM relations both on the original CFG $F$ and on a CFG $F'$ in which all arcs are reversed; therefore we will require that both $F$ and $F'$ be reducible. We will make the restriction to loops with single entry and single exit in section 6.3, where we also show that we can always transform loops to this form; this restriction will assure us that both $F$ and $F'$ are reducible and allow us to apply all the results of this section to a CFG with loops. For simplicity we at present restrict ourselves to an acyclic CFG.

We now can state the following theorem:

**Theorem 6.1.10.** Let $F = < V, A, s, e >$, be an acyclic CFG, and let $F' = < V, A', e, s >$ be the CFG obtained by reversing the direction of all the edges in $A$, and interchanging the start and end nodes of $F$. Let $l \in V$ be a split node with a set $D$ of departing edges.

$m$ DOM $l$ in $F'$ iff $m$ is a complete merge node for $l$ and all edges $D$ in $F$. In addition, if $m$ is a first merge node (def: 6.1.5) then the distance between $m$ and $l$ in the DOM trees $T_{F_l}$ and $T_{F'E_l}$ is 1.

**Proof.** Let $m$ be a complete merge node for $l$ and all of its departing edges in $F$. By definition 6.1.5 all paths $l \rightarrow^* e$ are $l \rightarrow^* m \rightarrow^* e$. But then all paths $e \rightarrow^* l$ in $F'$ must be $e \rightarrow^* m \rightarrow^* l$ and $m$ DOM $l$ in $F'$.

Let $m$ DOM $l$ in $F'$. Then it must be the case that all paths $e \rightarrow^* l$ in $F'$ pass through $m$ by definition of DOM. Inverting the direction of the arcs, all paths $l \rightarrow^* e$ in $F$ must be $l \rightarrow^* m \rightarrow^* e$, and $m$ is a complete merge node for $l$.

Let $m$ be a 'first' merge node. Then by definition there is no node $n \neq m$ such that $l \rightarrow^* n \rightarrow^* m$ for all paths $l \rightarrow^* m$. Therefore there is no node that is dominated
by $l$ that in turn dominates $m$, by definition of DOM. If we compute the DOM tree $T_{F,l}$, rooted in $l$, the only node that dominates $m$ must be $l$ itself, which is therefore at a distance of 1.

If we invert all the edges to obtain $F'$ we have that there is no node $n \neq m$ such that $e \rightarrow^* m \rightarrow^* n \rightarrow^* l$ for all paths $e \rightarrow^* m \rightarrow^* l$. Since $m$ is a complete merge, all paths that depart $l$ in $F$ pass through $m$ and therefore all paths from $e$ to $l$ in $F'$ are $e \rightarrow^* m \rightarrow^* l$. Therefore $e$ DOM $m$ and $m$ DOM $l$, and there is no node $n \neq m$ that dominates $l$ in $G'$ and is closer to $l$. Therefore the distance between $m$ and $l$ in $T_{F',e}$ is 1.

Matching nodes in DOM trees for $F$ and $F'$ is shown in figures 6.26.3, based on the control flow graph of figure 6.1.

**Corollary 6.1.11.** In an acyclic CFG there is an unique first complete merge node $m$ to a split node $l$.

**Proof.** By properties of DOM tree, if $l$ DOM $m$ and $k$ DOM $m$, if $l \neq k$ they must be at different distances from $m$ in the tree, but a first merge node must be at a distance of 1.

**Corollary 6.1.12.** In an acyclic CFG all merge nodes for $l$ and proper subsets of $D$ are interior nodes of the DOM tree $T_{F',e}$ and appear in the subtree rooted in $m$.

**Proof.** Let $k \neq m$ be a merge node for edges $K \subseteq D$ departing $l$. We know $m$ DOM $k$ in $F'$ because $m$ is a complete merge and $k$ is not complete. If there were any path $k \rightarrow^* m \rightarrow^* l$, then all paths from $k$ to $l$ in $F'$ must pass through $m$ (since $m$ is a complete merge) and therefore $k$ would also be a complete merge. Therefore $m$ DOM $k$ in $F'$ and $k$ appears in the subtree of $T_{G',e}$ rooted in $m$.

Suppose $k$ is a leaf node of $T_{F',e}$. We know $k$ is in a subtree rooted in $m$, and also that $l$ is in that same subtree, and at a distance of 1 from $m$ (by theorem 6.1.10). Since $k$ is a leaf node, we know that there is no node $p$ such that all paths $m \rightarrow^* p \rightarrow^* l$ in $F'$ must pass through through $k$ after passing $p$, since $k$ would dominate $p$ and not be a leaf node. However, we distinguish different paths out of $l$ because they go to
different nodes; if $k$ were a merge for some subset $K$ of departing edges, then there must be some node $p$ on at least one path in $K$ that is encountered before reaching $k$ because $k$ must be a merge of at least two different paths. Since $k$ is a merge node, then all paths through $p$ must also pass through $k$ before reaching the complete merge at $m$. If such a node $p$ exists, then $k$ is not a leaf node of $T_{F',e}$ and must be an interior node.

The theorem refers to complete merges, however it is possible that paths leaving a split node on a subset of all the departing edges will merge before reaching a complete merge. We expect that the normal situation would be a multi-way choice statement structured along the lines of a Pascal case statement, which would lead to separate paths all merging at the end of the case structure. However, use of multi-way choice statements like the Fortran computed goto could easily produce partial merges before the first complete merge. Multiple merges to a single split could arise even with structured programming constructs such as the C or C++ switch statement, if break statements are omitted at the end of any of the branches.

Note that, since all possible departing paths for a particular split meet at a complete merge, all merges that appear after such a first complete merge must also be complete merges for that split.

We will need to identify all merge nodes in order to insert code to support merges. In the case of merge nodes that involve subsets of edges departing a particular split node, we will also need to know which edges merge at each node. We now present an algorithm to identify merge nodes.

Algorithm 6.1.13. Given: a single exit subgraph $F_l = (V_l, A_l, l, e)$ of a single exit CFG, $F = (V, A, s, e)$ where $l$ is a split node of outdegree $N$, $V_l$ is the set of nodes in $V$ reachable from $l$ and $A_l$ is the set of arcs connecting nodes in $V_l$.

To find: $m$, the unique first complete merge node (def. 6.1.5 and theorem 6.1.10). Compute the DOM tree $T_{F_l}$ rooted in the split node. Compute the DOM tree $T_{F',e}$ on the CFG $G'$ in which all edges are reversed. By theorem 6.1.10, there is a unique
Figure 6.2: Complete first merge node identification
Identification of complete first merge nodes for split nodes S and 1 from the CFG of figure 6.1.
Figure 6.3: Complete first merge nodes for split node 2
Identification of complete first merge nodes for split node 2 from the CFG of figure 6.1.
node $m$ such that the distance between $l$ and $m$ is 1 in both $T_{F,l}$ and $T_{F',m}$. This is the first complete merge node for $l$.

**To find:** any first merge nodes corresponding to subsets of edges departing $l$. If $l$ is of outdegree $N > 2$ then it is possible that some subset of edges will merge before the first complete merge node.

Modify $F_l$ as follows: insert dummy nodes $b_i$, $0 \leq i < N$ on each of the departing edges of $l$, so they appear between $l$ and each node at a distance of 1 on a departing edge. Let $B$ be the modification of graph $F_l$ with these dummy nodes. Let $B'$ be the graph which results from reversing each edge in $B$ and switching the nodes sand e. Construct DOM trees $T_{B,l}$ and $T_{B',e}$. Extract the subtree $T_m$ of $T_{B',e}$ rooted in the first complete merge node, $m$. By theorem 6.1.12 all merge nodes for subsets of departing edges from large interior nodes of this subtree.

By construction of the $b_i$ nodes, each path departing $l$ must pass first through one such node; all merges of paths that depart $l$ must involve at least two such paths and therefore every merge node must dominate at least two nodes $b_i, b_j$ with $i \neq j$.

We find such nodes through a depth-first traverse of $T_m$, as follows: whenever we reach a leaf node $b_i$, label all nodes on the path from the root with the number $i$ (if the depth first traverse is controlled by a stack, these will be all nodes on the stack).

When the traverse is complete nodes in $T_m$ will be marked with path numbers corresponding to the departing edges they dominate. The node $m$ will have the labels $0$ to $N - 1$, since it is a complete merge. All merge nodes will be labeled with more than one number, and first merge nodes will be the farthest nodes from $m$ in $T_m$ that are labeled with a particular set of numbers.

We now compress $T_m$ into a structure we call a **merge tree** $M_l$, where $l$ denotes the split node for which $m$ is a complete first merge. First we remove all nodes from $T_m$ that are not marker nodes for departing edges and are not marked with one or more path numbers. Secondly, we replace every node that has a single child labeled with the same path numbers by the child node. The resulting structure is a tree rooted in $m$ in which all the leaves are path marker nodes and all the interior nodes, as well as the root, are merges.
An example of the application of the algorithm is shown in figures 6.2, 6.3 and 6.4. Generation of the merge tree is shown in figure 6.5. Theorem 6.1.15 and its corollary are a proof that we can always generate the merge tree. (Although the CFG in the examples is not acyclic, it follows the restriction that all loops be single entry and single exit. This allows us to extend our treatment to a CFG with loops in section 6.3.)

Note, in the figures, that node 8 is a merge for both nodes 1 and 2, and therefore it is not identifiable as a merge node using tree $T_{F_1}$ and requires us to compute $T_{F_2}$. The algorithm has us compute DOM trees rooted at each split node so that nodes such as 8 are revealed. Multiple merges should not arise if we are using nested statement blocks as in C or C++, since the ends of each logic statement would be marked as nested ends of blocks. However, the use of labeled statements as targets of jumps, common in Fortran, can easily lead to multiple merges at the same label.

Immediate consequence of the construction in algorithm 6.1.13 are:

**Theorem 6.1.14.** A split node of outdegree $N$ has at most $N - 1$ first merge nodes.

**Proof.** By algorithm 6.1.13 merge nodes can be arranged in a DOM tree in which the $N$ nodes $b_i$ are leaf nodes. Such a tree can have no more than $N-1$ nodes of outdegree $> 1$. \qed

We also have:

**Theorem 6.1.15.** Given a CFG $F = <V, A, s, e>$, let $l \in V$ be an $N$-way split node. Let $D$ be the set of all departing edges from $l$. Given two first merge nodes $m_i \neq m_k$ corresponding to some subset of $D$, let $D_i \subseteq D$ be the largest set of departing edges that have a first merge at $m_i$ and let $D_k \subseteq D$ be the largest set of departing edges that have a first merge at $m_k$. Then either $D_i \subseteq D_k$ or $D_k \subseteq D_i$, or $D_k \cap D_i = \emptyset$.

**Proof.** The theorem follows directly from construction in algorithm 6.1.13, because each first merge node includes all paths from the leaf nodes the subtree rooted in itself, where the leaf nodes are the dummy nodes $b_i$ that are used to mark the departing edges. Since all nodes appear on the same DOM tree, any pair of merge nodes either
Figure 6.4: Identification of partial merge nodes
Identification of all the first merges corresponding to the split at node 2, based on the CFG from figure 6.1.
Figure 6.5: Merge tree

Generation of merge tree from figure 6.4.

appear on the same path from the root (in which case the leaf nodes of one are a subset of the leaf nodes of the other) or they appear on different paths from the root, in which case the leaf nodes are disjoint.

\[ \square \]

**Corollary 6.1.16.** First merge nodes corresponding to a single split \( l \) can be arranged as a tree rooted in a single complete first merge node that is a merge for all departing edges from \( l \).

**Proof.** By the theorem, the relation \( \subset \) on the departing edges that merge is a partial ordering for first merge nodes. By [GRUNWALD 93], a partial order is transitive, antisymmetric and reflexive, represented as a DAG.

Since \( m_e \) is on all departing paths from \( l \), any merges of any subsets of departing paths that occur before \( m_e \) is reached are in \( H \) and are nodes in \( V_H \). We now construct the graph \( H' = \langle V_H, A'_H, m_e, l \rangle \) formed by reversing the direction of all edges in \( A_H \) and swapping \( l \) and \( m_e \).

If \( |D| > 2 \), then there can be 'first' merges for subsets of edges; let \( M \) be the set of such merges:
\[ M = \langle m_i | m_i \in V_H \rangle, \text{ where } m_i \text{ is a merge for some } D_i \subset D. \]

Assume there are nodes \( m_i, m_j \in M, m_i \neq m_j \). It must be the case that \( D_i \neq D_j \), otherwise \( m_i \) and \( m_j \) would be the same node. Assume there is some edge \( l \to b \in D_i \) which is also \( l \to b \in D_j \). Then every path \( l \to b \to m_e \) must pass through both \( m_i \) and \( m_j \). Since \( m_i \) and \( m_j \) are both 'first' merges (for some subsets \( D_i, D_j \subset D \) of the departing edges of \( l \)), it must be the case that either all paths through \( b \) are of the form \( l \to b \to m_j \to m_i \) or all paths are of the form \( l \to b \to m_i \to m_j \in D_j \); otherwise it would be possible to leave \( l \) along the edge \( l \to b \in D_j \) and reach either \( m_i \) or \( m_j \) without passing through the other, and at least one of those two nodes would not be a 'first' merge. Therefore if \( D_i \cap D_j \neq \emptyset \), then either \( D_i \subset D_j \) or \( D_j \subset D_i \). Therefore all merge nodes \( m_i \) for \( l \) and a subset \( D_i \subset D \) of the departing edges of \( l \) are partially ordered by the relation \( \subset \) between the \( D_i \). Therefore if \( i \neq k \), either \( D_i \subset D_k \) or \( D_k \subset D_i \), or \( D_k \cap D_i = \emptyset \).

We generate the tree of merge nodes from the corollary of theorem 6.1.15 by omitting all nodes that are not merge nodes from the inverse DOM tree (see figure 6.5). We also have:

**Corollary 6.1.17.** The **merge node tree** given by corollary 6.1.16 specifies the order in which first merge nodes are encountered between a first complete merge and its corresponding split.

**Proof.** The merge node tree is a subgraph of the inverted DOM tree rooted in the first complete merge. \( \square \)

We now show that pairs of split nodes and any of their corresponding 'first' merge nodes encountered on the same path on the CFG are nested.

**Theorem 6.1.18.** Let \( (l_1, m_1) \) be a split node and a matching first merge node. Let \( l_2 \) be a split node that is on a path \( l_1 \to m \). Then any first merge node \( m_2 \) matching \( l_2 \) must also appear on the path \( l_1 \to m_2 \to m_1 \) before reaching \( m_1 \). That is, split-merge node pairs that appear on the same path are nested.
Proof. (For readability, we will omit the notation $\in X$ from the paths departing the split node. In every case of paths departing $l_1$ and $l_2$ it is implied. Than is, a path $l_2 \to^* y$ is shorthand for $l_2 \to b \to^* y$, such that $(l_2 \to b) \in X_2$)

Let $l_1 \neq l_2$ be split nodes in a CFG $F$, where $D_1$ is the set of departing edges from $l_1$ and $D_2$ is the set of departing edges from $l_2$, Let $m_1$ be the first merge node for a subset $X_1 \subseteq D_1$ of departing edges from $l_1$, and let $m_2$ be the first merge node for a subset $X_2 \subseteq D_2$ of departing edges from $l_2$. Assume $m_2 \neq m_1$.

Assume the theorem is false. Then there is a path $l_1 \to^* l_2 \to^* m_1$ that does not include $m_2$. There must also be a path $l_1 \to^* l_2 \to^* m_1 \to^* m_2$, since $m_2$ is a merge for $l_2$. However, since $m_1$ is a first merge for $l_1$, all paths $l_2 \to^* m_2$ must be $l_2 \to^* m_1 \to^* m_2$, or else $m_1$ would not be a first merge node. But then $m_2$ is not a first merge node, and we have a contradiction.

Therefore all paths must be of the form $l_1 \to^* l_2 \to^* m_2 \to^* m_1$. \hfill $\square$

The fact that split-merge pairs along any path are nested simplifies the code to support them; it is not necessary to use a data structure more complicated than a stack to keep track of splits and merges, and in many cases simple counters will be sufficient.

### 6.1.2 Simplifying split-merge identification

Some of the analysis described above in section 6.1 is of non-linear complexity. We do not have simpler algorithms to perform the analysis, however we can in some cases simplify the graph on which the analysis is performed, if we can identify subgraphs that are single entry and single exit [JOHNSON 93a]. For such subgraphs, we have:

**Theorem 6.1.19.** Let $S = \langle L, B, h, x \rangle$ be a single entry, single exit subgraph of CFG $F = \langle V, A, s, e \rangle$. If $l \notin L$ is a split node, and $m$ is a first merge node matching $l$, then either $m \notin L$ or $m = h$, the entry node to $S$.

Proof. Assume false. Then $m \in L$, $m \neq h$ is a ‘first’ merge node for split $l \notin L$. But since $S$ is a single entry subgraph, $h$ dominates every node in $L$, including $m$. Since
l is not in L, all paths \( l \rightarrow^* m \) are \( l \rightarrow^* h \rightarrow^* m \) and \( m \) is not a first merge, which contradicts the assumption.

\[ \square \]

**Theorem 6.1.20.** Let \( S = \langle L, B, h, x \rangle \) be a single entry, single exit subgraph of \( CFG \) \( F = \langle V, A, s, e \rangle \). If \( l \in L \) is a split node, and \( m \) is a first merge node matching \( l \), then \( m \in L \).

**Proof.** All paths that leave \( S \) must pass through \( x \), therefore it satisfies definition 6.1.5 of a complete merge node for any split in \( S \). \[ \square \]

That is, all split-merge pairs are either completely inside a single entry, single exit subgraph, or they are entirely outside the subgraph except for the possibility of a merge at the entry node. Therefore we can perform the analysis to identify first merge nodes separately on single entry, single exit subgraphs rather than on the entire CFG.

Similarly, if a graph contains a single entry, single exit subgraph \( S \), we know that any split outside \( S \) will not have an interior node of \( S \) as a first merge node. Therefore each single entry, single exit subgraph \( S \) can be replaced, when analyzing the graph that contains \( S \), by a single pseudo-node.

**Theorem 6.1.21.** Let \( S = \langle L, B, h, x \rangle \) be a single entry, single exit subgraph of a graph \( F = \langle V, A, s, e \rangle \). Let \( F_S \) be a graph constructed from \( F \) by removing all nodes and arcs in \( S \) from \( F \), replacing them with a single node \( h' \), such that arcs entering \( h' \) are the same arcs that entered \( h \) and arcs leaving \( h' \) are the arcs that left from \( x \). If \( l \) is a split node in \( F \) and not in \( S \), then the same first merge nodes will be identified for \( l \) in \( F_S \) as would have been found in \( F \), except that if \( h' \) is itself a first merge in \( F_S \), then \( h \) is a first merge in \( F \).

**Proof.** Since \( l \notin L \), we know from 6.1.19 that it has no first merge nodes inside \( S \) except that \( h \) might be a first merge. Since \( S \) is single entry and single exit, and the same arcs that entered and departed \( S \) are connected to \( h' \), we know that if we had a path \( l \rightarrow h \rightarrow m \) we must have \( l \rightarrow h' \rightarrow m \) in \( F_S \). Since \( S \) is single entry and single exit, the same must hold for \( F' \) and \( F'_S \), the graphs with all arcs reversed: if
we have a path \( m \rightarrow h \rightarrow l \) in \( F' \) we must have \( m \rightarrow h' \rightarrow l \) in \( F'_S \). Since \( S \) is single entry and single exit, our construction adds no path between nodes in \( F_S \) that was not present in \( F \).

First merge nodes are found by considering paths from the split node (definition 6.1.5). For every path in \( F \) from a node not in \( S \) to a possible merge node we have a path in \( F_S \), and there are no paths in \( F_S \) that are not also in \( F \), therefore the same merge nodes identified for \( l \) in \( F \) must be identified in \( F_S \).

\[ \square \]

### 6.2 MIPS level

We have from theorem 6.1.18 that split-merge pairs along a specific path on the CFG are nested. This allows us to define an MIPS level corresponding to the number and type of splits and merges traversed along a given path through the CFG, analogous to a parenthesis nesting level. MIPS level is more complicated than parenthesis nesting level because a single split can have more than one first merge along a particular path, and because multiple splits can have merges at the same node.

The MIPS level depends on the exit path from a split, because multiple merges can correspond to a particular split along some paths, and because a single merge node can match multiple splits. We define:

**Definition 6.2.1.** MIPS level \( L \): defined inductively as follows:

i. At exit from a split node \( l \) along some edge \( d_i \), \( L \) is incremented by 1 plus the number of partial first merges encountered on a path starting on \( d_i \) and reaching the first complete merge node \( m \) matching \( l \).

ii. At entry to a first merge node \( m \) matching split \( l \), \( L \) is decremented by 1.

iii. If \( m \) also is a first merge for some split \( l_i \) such that there is a path \( l_i \rightarrow^* l \rightarrow^* m \), then decrement \( L \) by 1 on entry to \( m \), in addition to the decrement by rule ii. Repeat until there are no nodes \( l_i \) with first merge at \( m \).

iv. Base: \( L = 0 \) at node \( s \) in a CFG \( F =< A, V, s, e > \).
The rules insure that the increment at each split is balanced by the decrement at one or more first merges matching the split. Since \( c \) is a complete merge node for \( s \), it follows from theorem 6.1.18 that \( L \) is always \( \geq 0 \), and will be 0 at node \( c \).

The MIPS level alone is insufficient to identify the member of a particular MIPS; it simply reflects how many splits a process has encountered without matching merges. We define an MIPS level array \( IP[p] \), indexed on process number, with one entry for each process. (In section 7 we describe in detail how this array is constructed and used). A particular process \( p \) keeps track of which processes have taken the same path that it is taking at a split, and are therefore in the same MIPS, by incrementing their MIPS level in \( IP \) to match its own. At a merge, each process decrements its own level following rules ii and iii; since processes in the same MIPS encounter the same merges, \( p \) also decrements entries in \( IP \) that match its own MIPS level. Note that the contents of \( IP \) are the same only for processes that are in the same MIPS; if \( p \) and \( q \) are in two different MIPS, the contents of \( IP \) at \( p \) will be different from its contents at \( q \). This is true because, at splits and merges, processes only modify the \( IP \) entries of other processes that are in the same MIPS.

At a two-way split \( s \), there is only a single matching first merge, therefore rule i only requires that processes increment their own MIPS level by 1. When they later decrement their level and that of matching processes in \( IP \) following rule ii, the MIPS level of \( p \) matches the level in \( IP \) of the processes it is merging with, since its level is brought back to the value it had on entry to \( s \), and the level of processes that did not take the same branch at \( s \) was not changed in \( IP \) at \( p \). Since splits and merges along any path of execution are nested, we do not need any special treatment to allow for the possibility that processes departing split \( s \) may encounter some other split before they reach their first merge matching \( s \); if there is such another split, all of its first merges must be encountered before reaching any first merge for \( s \).

A splits \( s \) with multiple exits introduces the possibility of partial merges encountered by some processes before reaching the first complete merge. We can determine where these merges will occur and on which paths departing from \( s \) they will be found (algorithms 6.1.13 and 7.1.5). If a process \( p \) is departing \( s \) along a path which
includes a partial merge or merges, it must increment its own MIPS level (and that of other processes in its MIPS) by more than 1 following rule i. In order to properly identify the MIPS that forms at that partial merge, it must also anticipate which processes will merge with it at that partial merge, and increment their entries in IP by an appropriate amount. It can do this because code executed at a split allows each process to determine which departing path is taken by other processes, and therefore which processes depart along paths that will reach partial first merge nodes.

6.3 Loops

In the case of loops, we note that the CFG at time of execution is different from the static CFG, because loops repeat the same subgraph multiple times in sequence. An analysis of the CFG with loops unrolled reveals merge nodes that are not visible in the static CFG. We will see that we identify merges between nodes separated by loop iterations; that is nodes created by a separation in time but which are not in fact graphically different. This situation requires different code for loops, and a different analysis.

Up to this point our analysis applies to a CFG with no loops and a single exit node (Any CFG can be transformed to single exit; see remark 6.1.6). We now impose the additional restriction that the CFG be reducible (Following the standard definition of a reducible CFG in [AHO 85]). We find that the analysis of loops, particularly in the case where the loop bounds may be non-uniform (e.g. a while loop depending on data), is simplified by considering only natural loops which have a single entry point, which is a consequence of a CFG being reducible.

Most programs have a reducible CFG [ALLEN 70], and there are standard methods for transforming non-reducible CFGs to reducible form (see [AHO 85]) so we do not consider this a serious limitation. In a reducible CFG, an edge \( a \rightarrow b \) must be either a forward edge in which \( a \) DOM \( b \), or a back edge in which \( b \) DOM \( a \). In such a CFG, all loops are natural loops, arising from back edges [AHO 85]. Most of our definitions and proofs in section 6.1 use the DOM relation both on the original
CFG $F$ and on an inverted CFG $F'$ in which all the edges are reversed. To simplify our treatment of loops, we will impose a restriction to single exit loops, that will guarantee that both $F$ and $F'$ are reducible without changing the meaning of the program.

**Definition 6.3.1.** Aho, Sethi and Ullman (p. 604, [AHO 85]), define a natural loop as follows: “Given a back edge $a \rightarrow h$, we define the natural loop to be $h$ plus the set of nodes that can reach $a$ without going through $h$. Node $h$ is the header of the loop.” That is, the natural loop $L$ of a back edge $a \rightarrow h$ is the set of those nodes that have paths to $a$ without passing through $h$, and $h$ itself. ([AHO 85], among others, give algorithms for identifying natural loops). Note that $h$ dominates every node in the loop; that is, there are no arcs to any node in $L$ from any node outside $L$ that do not pass through the header $h$.

It is possible that we have another back edge or edges $b \rightarrow h$ such that $b \neq a$; we could then define a set of nodes $M$ as the natural loop of $b \rightarrow h$. We follow [AHO 85] in such cases by treating the union $M \cup L$ of natural loops with the same header as a single loop with multiple back edges. Such a loop could be transformed to a loop with a single back edge by adding a dummy node $c$ with no code, adding edges $a \rightarrow c$, $b \rightarrow c$ and $c \rightarrow h$. This enables us to limit our considerations to natural loops with a single back edge. If all loops are (or are transformed to) loops with a single back edge, then that back edge must be traversed at each iteration of the loop.

**Definition 6.3.2.** An exit node $a$ from a natural loop $L$ is a node in $L$ of outdegree greater than one in which at least one arc leads to a node outside $L$. (Note that if $a$ were of outdegree one, with an arc to a node outside $L$, then $a$ would not be in the loop by definition 6.3.1).

The header node $h$ is frequently an exit node (for example in while loops). However, loops written with an if … goto statement after the loop body, or with logic such as do{…}while would have an exit node at the tail $a$ of a back edge $a \rightarrow h$. Break statements and conditional goto statements in the loop body may result in exits from any node in the loop.
A loop with more than one exit and non-uniform exit conditions causes trouble because we cannot in general assume that multiple exits will be traversed in a fixed sequence - so it is possible for processes at exit$_1$, for example, not to know that some process has actually exited at exit$_2$ and will never reach exit$_1$. In such a situation, the split at exit$_1$ will deadlock. The easy way out is to transform all loops to single exit (a multi-exit loop then looks like a single exit loop immediately followed by a split), although it is by no means the only possibility.

We will here limit ourselves to single exit while and do{...}while loops (or equivalent for loops) in which there is a single back edge. In fact there is no loss of generality in restricting ourselves to such loops: Bohm and Jacopini [JACOPINI 62] show that all flow graphs can be transformed to graphs written in terms of while loops without changing their meaning, and give ways to perform the transformation. The code support we will describe applies only to single exit while loops and do{}while, and loops that are transformed to such loops.

A single entry, single exit loop has a single entry, single exit subgraph. Therefore by theorem 6.1.19 a split outside a loop always has a matching first merge outside the loop or at entry to the loop. By theorem 6.1.20 a split inside a loop also has a first merge inside the loop. An immediate consequence of single entry, single exit loops is that the same MIPS that enters the loop must exit the loop (since any splits inside the loop must be matched by complete merges also inside).

We will still need to distinguish between loops in which all processes perform the same number of iterations, and loops in which this might not be the case:

**Definition 6.3.3.** We will say that a loop is **uniform** if loop indices and bounds are uniform (that is, can be guaranteed to be the same at all processes), and if any split nodes (on non-uniform predicates) inside the loop have a matching complete first merge node also inside the loop (def: 6.1.5). Otherwise we call it **non-uniform**.

Uniform loops need no special treatment because every process in the MIPS that enters goes through the loop the same number of times and exits the same way, so it unrolls as the same straight line code everywhere. We need, however, to consider what
happens to non-uniform loops, since in such cases the exit node can be considered a split and the set of processes remaining in the loop can change at each iteration.

We will assume that the exit from a single exit loop is an arc to a single node outside the loop. If it is not, it can clearly be transformed to a single arc by adding a node \( x \) and an arc \( b \rightarrow x \) from the exit node \( b \), connecting \( x \) to all the nodes outside the loop at a distance of 1 from \( b \) and moving the test that determined which exit arc to take from \( b \) to node \( x \). Therefore there is no loss of generality in making the assumption.

**Theorem 6.3.4.** Given a single exit, non-uniform while or do{}while loop \( L \) with exit arc \( b \rightarrow x \), node \( x \) is a merge for all paths departing from the loop.

**Proof.** Exit from a while or do{}while loop is controlled by the while statement, therefore it must be located in node \( b \), which is either the header (while) or the tail of the back edge (do{}while). By theorem 6.1.20 and its corollary, \( b \) is a merge node for all paths inside the loop.

The execution of \( L \) by a process \( p \) that iterates the loop \( I \) times is graphed by unrolling the loop \( I \) times. Let \( U \) be the unrolled graph plus the arc to \( x \). The node \( x \) is outside the loop, so it only appears once in \( U \). Since \( b \) is replicated in the unrolling, there will be \( I \) arcs to \( x \). In \( U \), the first iteration through node \( b \) dominates all nodes. The only way of reaching \( x \) is \( b \rightarrow x \), so \( b \) dominates \( x \) at a distance of 1. In a non-uniform loop, departing processes may leave from any iteration of node \( b \), so \( b \) is a split for departing processes (see figure 6.6).

In the graph rooted in \( b \), \( b \) dominates \( x \) at a distance of 1, and in the graph formed by reversing all the arcs of \( U \), \( x \) dominates \( b \) at a distance of 1. Therefore by lemma 6.1.10, \( x \) is a first merge for a split at \( b \). \( \Box \)

A consequence of the theorem is that all processes that leave a non-uniform loop still merge into a single MIPS on exit. For convenience and to simplify code insertion, we can insert the merge code in node \( b \) at the exit to the loop instead of at \( x \).

We need additional code to manage the processes that remain in a non-uniform loop, because the exit node is a split, and processes remaining in the loop increment
Figure 6.6: Loop unrolling
An unrolling of loop L9 from figure 6.1.

Loop 9, CFG: $F=(V,A,s,e)$
(unrolled)
MIPS level by 1 every time they take a branch that leaves them inside the loop. Since the departing MIPS is the same as the entering MIPS, the MIPS level must be reduced to whatever the entry level was. We need extra data structures to keep track of the number of iterations at each process so that all processes may be restored to matching MIPS levels. These structures are detailed in section 7.1.1, in note 7.1.2, and their use of these data structures is described in 7.1.2.
CHAPTER 7
MIPS EXECUTION

In this chapter we will describe the analysis of the CFG to allow the insertion of MIPS code. We describe transformation of the CFG to the form we require for the rest of our analysis while preserving its meaning. Then we describe the identification of merge and split nodes.

We then describe the runtime code required to identify the membership of implicit process sets, and to compute and propagate the changes in sets that occur at split or merge nodes.

We consider the properties of an MIPS execution; that is, execution of a program that contains code describe here. We show that if we restrict communications to fusions including only processes that are members of a single MIPS, then execution is deterministic and free from deadlock.

7.1 Code support

Given an initial MIPS in SPMD, the membership of a MIPS is always known on entry to a split. At the split (at runtime) we must transfer information to all processes in the parent (entering) MIPS indicating which of the possible departing edges was taken by each process. Departing edges may be numbered statically while analyzing the CFG (for example they may be marked with an integer); the actual membership of departing MIPS is known at runtime and must be communicated to all processes involved in the split, for example by a vector combine data transfer.

Definition 7.1.1. A vector combine is an all-gather (see [NUMRICH 97], page 92) data transfer between a set $G$ of $P$ processes in which each process $p \in G$ places a data item in a vector $W$, at a position that corresponds to the order (index) of $p$ in

107
\( G \). \( W \) is exchanged between processes in \( G \), with each process updating its local copy of \( W \) with the data it receives from other processes. The result is \( P \) copies of \( W \), one at each process, with data from every process.

We see from lemma 6.1.18 that split-merge pairs identified on any path on the CFG must be nested. The present implementation uses this nesting, and the MIPS level (def. 6.2.1) to keep track of MIPS membership as follows:

At program start, each process creates a process array \( IP \): this an integer array with one entry for every process in the execution, including itself, and initializes this array to 0 everywhere. This array is used to keep track of the MIPS level (def. 6.2.1). At split nodes, each process \( p \) changes its own entry in the array as described in definition 6.2.1 and also changes the entries corresponding to other processes that are following the same path, determined from a vector combine operation.

At a merge node, process \( p \) decrements the its own entry and those of all matching processes as described in definition 6.2.1. Since split-merge pairs on the same path are nested, this decrements the level of \( p \) to match the level it had before the split, which is then the same as that of processes which split from the path of \( p \) at that time (this is modified for partial merges, where the increment at the matching split allows for multiple merges before reaching a complete merge). Since every process executes the same algorithm on its local process arrays, and all initialize the array the same way at program start, all merging processes have matching arrays at each merge.

Special handling is required for N-way splits, as described in definition 6.2.1. If there are partial merges before reaching a complete first merge, the paths that lead to such merges have their MIPS level incremented by the number of partial merges they will encounter, so that after undergoing them, the level reduction at the complete merge will match other processes that do not pass through partial merge nodes.

The exit node of a while loop is both a merge by theorem 6.1.20, and is a split for a non-uniform loop. By theorem 6.3.4 all departing processes from exit node \( b \) merge at the first node \( x \) on the exiting arc \( b \rightarrow x \), forming the same process set that entered the loop. Although the merge is actually at \( x \), since in fact all departing processes take the same arc \( b \rightarrow x \), it is preferable to insert any extra code at \( b \). We are only
treating single exit loops, but by [JACOPINI 62] we have that all other loops can be transformed to our limited case, so the code we present here is sufficient for the more general case.

Even though the set that departs the loop is the same as process set inside the loop, we need to distinguish processes inside the loop from processes exiting the loop. If we treat loop exit as a normal split, then the MIPS level of processes leaving on arc \( b \rightarrow x \) will have their MIPS level incremented at \( b \) and then decremented again at \( x \) (actually, since we are placing the merge code at \( b \), we get the same effect by not incrementing the departing MIPS level). This means that processes at \( x \) will include processes that remain inside the loop in the loop in their process set; this is correct since all such processes will eventually merge at \( x \). However, processes that are inside the loop need to exclude processes at \( x \) from their membership set, so they can correctly communicate inside the loop. This can most easily be accomplished by incrementing the MIPS level of a process each time the process passes through the split at \( b \) and remains inside the loop, keeping an iteration count, and then decrementing by the iteration count on loop exit.

It is possible that a single node may be a merge for several splits. As a result, some paths entering such a merge node do not participate in all the merges. It may be possible to resolve such cases by modification of the merge code, or by lifting the merge code to the exit point of nodes leading to the merge (as is done for the exit from a loop). For the present, we choose instead to use the result of theorem 6.1.18; at a split node, a process examines the merge tree for its particular departing path and pushes the identifiers for the merge nodes it will see on a stack. At a merge node, a process follows rule iii of definition 6.2.1, pops nodes off its stack and decrements the MIPS level until the node on stack top does not match the current merge node.

The following are data structures and algorithms sufficient to support the operations described above.

Note 7.1.2. **Data structures**: Let \( F \) be a CFG of a program or subroutine. Let \( Nproc = |\Gamma| \), the number of processes in an SPMD execution. \( IP[p] \), the MIPS level
array, is an integer array of size at least $N_{proc}$, indexed on process number $p$. $W[p]$ is an integer array of the same size as $IP$, also indexed on process number.

To each split node $l$, assign a pointer $M$ to a tree data structure. If algorithm 6.1.13 finds more than one merge node, and the merge tree $M_I$ computed in that algorithm has height $> 1$, let $M$ point to $M_I$. If there is a single first merge node (when $M_I$ has height 1 or when $l$ has only two departing edges), construct a tree that has only the root node, set this root node equal to the first complete merge node $m$ from the algorithm, let $M$ point to this node.

Define a stack $MSTACK$ with operations $TOP$, $PUSH(x)$ and $POP$, to keep track of splits and corresponding merges. Define a stack $LSTACK$ to track loop indices, and allow for nested loops.

To each split node and merge node assign an integer node which is set equal to the node number in the CFG.

Code requires the following:

**7.1.1 Algorithms for code analysis and transformation:**

We describe here the algorithms executed on the code at compile time. Split nodes are known as nodes of outdegree $> 1$ that depend on a non-uniform predicate. If it is not known that a particular node depends on a non-uniform predicate, we can correctly assume that it is a split node and treat it as such. The CFG is generated and loops are identified by standard techniques described in [AHO 85]. Loop transformations follow [JACOPINI 62]. We assume the CFG is reducible. If it is not reducible, loop analysis rejects the program; in such a case the program may be re-written so that the CFG is reducible as described in [AHO 85].

Transformations to the CFG that add nodes and arcs are accomplished in the program through labels and goto statements.

Code transformations are carried out in the order given below. Each algorithm here describes assumes all previous transformations have already been applied.

We first transform the program to single exit form:
Algorithm 7.1.3. Program exit:

**Input:** program text and corresponding CFG.

If $F$ is not single exit, apply the transformation of remark 6.1.6 to the program code.

**Output:** CFG with single exit.

Next we identify and transform loops.

Algorithm 7.1.4. Loop identification and transformation:

**Input:** program text and corresponding single-exit reducible CFG.

**Purpose:** To transform all loops so that: there is a single back edge, there is a single exit arc, the exit node is either the header or the tail of the back edge, the exit node is of outdegree 2, and the position of runtime code to support the loop is found.

Natural loops are identified by finding all back edges, as described in [AHO 85].

All header nodes are marked in the CFG. If there are any edges that are not identifiable as forward or back edges, the CFG is not reducible and the algorithm fails. (It is always possible to transform the program code to convert multiple-exit loops and single exit loops with an exit node that is not the header to single exit while loops [JACOPINI 62].)

The following transformations, if applicable, are performed in order on each natural loop.

i. Given a loop with header $h$ and multiple back edges $b_0 \to h \ldots b_n \to h$, to transform to a loop with a single back edge: Add a single node $b$ to the loop, with code that transfers control to $h$. Replace code at the end of each node $b_i$ that transfers control to $h$ with code that transfers control to $b$. The result is that $b \to h$ becomes the single back edge, preserving the property that a process that takes a path through the loop to $b_i$ returns to $h$ for the next iteration. Mark $h$ in the CFG as the header node.

ii. Given a loop with multiple exit arcs $b_0 \to x_0 \ldots b_n \to x_n$, to transform to a single exit loop: Add a node $b$ inside the loop, a node $x$ outside the loop, and an arc $b \to x$. Replace the statement at the end of each node $b_i$ that transfers control to $x_i$ with code that sets a selector $sel = i$, sets a flag $ex = \{true, false\}$ depending
on the condition in \( b \) that causes a process to exit the loop, and transfers control to \( b \). Insert code in \( b \) that, depending on flag \( \text{ex} \), transfers control to node \( x \) outside the loop. Insert in \( x \) code that performs an \( n+1 \) way split depending on the value of \( \text{sel} \), transferring control to nodes \( x_0 \ldots x_n \). The result is a single exit arc \( b \to x \), preserving the property that a process that exits from node \( b \) takes a path through node \( x_i \). Mark \( b \) in the CFG as the exit node.

iii. Given a loop with a single exit arc \( c \rightarrow x \), back edge \( b \rightarrow h \) and \( c \neq b, \ c \neq h \), to transform to a loop such that the exit arc is \( b \rightarrow x \). Replace code at the end of node \( c \) that evaluates a loop exit predicate and conditionally jumps to \( x \) with code that sets a flag \( \text{ex} = \{ \text{true, false} \} \) depending on the exit predicate and conditionally transfers control to \( b \). Replace logic in \( b \) that transfers control to \( h \) with logic that evaluates \( \text{ex} \) and transfers control to either \( h \) or \( x \). Result is an exit arc \( b \rightarrow x \) that is taken iff \( c \rightarrow x \) would have been taken in the original code. Mark \( b \) in the CFG as the exit node.

iv. Given: a loop with a single exit arc \( b \rightarrow x \) such that \( b \) is a node of outdegree \( \geq 2 \), to transform to a loop in which the exit node is of outdegree 2. Split \( b \) into two nodes \( b_1 \) and \( b_2 \), such that the first is a split of outdegree 2 that has the exit arc \( b_1 \rightarrow x \) as one alternative and \( b_1 \rightarrow b_2 \) as the other. \( b_2 \) is a split of outdegree \( N - 1 \) entirely inside the loop. Both \( b_1 \) and \( b_2 \) evaluate the same predicate that \( b \) originally evaluated. If \( \text{b} \) would have selected the path \( b \rightarrow x \), then \( b_1 \) selects \( b_1 \rightarrow x \). Otherwise \( b_1 \) transfers control to \( b_2 \). The result is a loop exit node of outdegree 2. Mark \( b_1 \) in the CFG as the exit node.

**Loop entry code:** Identify the first statement in the loop. Immediately before this statement, outside the loop, add the code: \( \text{PUSH(0) on STACK} \); this initializes the loop iteration counter. If the loop entry statement is the target of a jump from outside the loop, add a new label as target for logic external to the loop, before the counter initialization code.

**Result:** Program text with reducible CFG, transformed so that all loops are single exit, CFG annotated to identify loop headers and exit nodes and with loop entry support code added to program text.

Split node identification and data structure insertion:
Algorithm 7.1.5. Split nodes:

**Input:** Program source code and CFG with single exit node, loops identified and transformed to single exit,

**Purpose:** identify and mark all split nodes, insert split support code, identify matching merge nodes and construct merge tree for each split.

Identify all split nodes following definition 6.1.4. For each split node \( l \), label the exit paths from 0 to \( N - 1 \). Allocate a pointer \( M \) to each the split node.

Apply algorithm 6.1.13 to identify the (up to \( N - 1 \)) 'first' merge nodes that match each split node. Construct the merge tree \( M_1 \); note that the leaves of \( T \) are dummy nodes \( b \) corresponding to each path. Set the pointer \( M \) to the root of the tree \( M_j \). If the height of \( M_1 \) is equal to 1, remove all child nodes from \( M_1 \), leaving only the root.

A split node \( l \) is characterized by a statement \( S \) depending on some predicate \( X \), causing control flow to jump elsewhere in the program. \( S \) is the last statement of \( l \). Split support code is inserted just before \( S \). Predicate \( X \) is evaluated by the split support code to determine which branch of the split will be taken. To avoid evaluating \( X \) twice, it is replaced by its value in statement \( S \). The tree \( M_1 \) is written explicitly into the code as a data description that can regenerate the actual tree at runtime.

**Modification for loops:** Given a non-uniform loop with a single back edge, and single exit arc \( b \rightarrow x \) from a node of outdegree 2 which is either the header or the tail of the back edge: To identify loop exit split and merge for all paths departing the loop from any iteration.

Unroll the loop once forming a graph \( U = \langle V, A, b, b' \rangle \) in which \( b' \) is a copy of node \( b \) that appears now as the end node of the loop graph, and perform the split node analysis on \( U \). All exit paths from \( U \) are seen to merge at \( x \). Node \( b \) is a split matched by the merge at node \( x \), therefore insert split code in node \( b \) as described above.

**Result:** Program code annotated with data structures required to carry out splits at runtime, and including split runtime support code. Merge tree \( M_1 \) for each split \( l \)

Merge nodes code insertion:
Algorithm 7.1.6. Merge nodes:

**Input:** Program source code and CFG with single exit node, loops identified and transformed to single exit, splits identified and marked, merge tree for each split.

**Purpose:** to insert merge node support code.

Merge nodes require insertion of runtime code to decrement the MIPS level. If \( m \) is a merge node, merge support code is inserted at the start of \( m \). If \( S \) is the first statement in block \( m \), \( S \) is an executable statement, and \( S \) has a label such that it may be the target of a jump, then the label is removed from \( S \) and inserted just before the first statement of the merge support code.

**Modification for loops:** Given a non-uniform loop with a single back edge, and single exit arc \( b \rightarrow x \) from a node of outdegree 2 which is either the header or the tail of the back edge: All exit paths start from node \( b \) (on different iterations), so we insert the merge code at the end of \( b \), conditioned on the exit predicate being true.

**Result** is program source code including merge runtime support code.

### 7.1.2 Runtime algorithms

We now describe algorithms executed at runtime. Code for these algorithms has been inserted as described in 7.1.1. First we show how each process keeps track of the MIPS process set.

Algorithm 7.1.7. **process set:** let \( G \) be a list, initially empty. Function \( G = \text{myGroup}(P) \) returns, at a process \( p \), the process set \( p \in G \subseteq \Gamma \) which includes \( p \).

myGroup executes the following algorithm:

\[
\begin{align*}
c &= \text{IP}[p]; \quad G = \emptyset; \\
\text{for} (i = 0; i < N; i + 1) &\{ \text{if}(c == \text{IP}[i]) \text{ add } i \text{ to } G; \}
\end{align*}
\]

\( G \) is an array of the same size as \( P \) in the present implementation, but it could be replaced by a dynamic data structure. Array \( P \) contains the MIPS level, such that processes with the same MIPS level are in the same MIPS.

Now we describe the code inserted at split nodes.
Algorithm 7.1.8. Split:

Algorithm to compute increase of MIPS level at split nodes.

Insert this code at all split nodes, as determined in algorithm 7.1.5. Let l be a split node of outdegree N. Let X be the predicate evaluated to select the departing path, as described in algorithm 7.1.5. First evaluate X. Trace the path in the merge tree M to the departing edge selected by X.

For each merge node m on that path, PUSH(m) on stack MSTACK. Do this in the order in which nodes are found traversing M from m to l. Note that if there is only one complete merge (the typical case), the traverse is not needed because M will then just be the merge node.

At each process p, let outcome_p be the number of the departing edge. Set W[p] = outcome_p.

Perform a vector—combine exchange of W on all processes in G (def: 7.1.1), Then, at each process, do the following:

Increment the MIPS level of all matching processes:

while(i ∈ G&&W[i] == W[p]) { IP[i] = IP[i] + 1; }

If the split is an N-way split and M is not a single node, and m is the root of M, add the following (note that by construction of M, each edge k is represented by a dummy node b_k in M, see algorithm 6.1.13):

If outcome_p selects a node b_i that is not at a distance 1 from m, then for every interior node n on a path between b_i and m, find the leaf nodes b_j that are descendants of n (including b_i itself). For a particular n, let O = {outcome_i, outcome_j, ...} be the set of outcomes corresponding to those descendant nodes. for each such outcome, do:

while(i ∈ G&&W[i] == outcome_j ∈ O)) { IP[i] = IP[i] + 1; }, repeat this for every node n.

Reset the set G = myGroup(IP);

Result: the entries in G of processes that selected the same outgoing arc as p will be incremented by the same amount.
Having done this, running $myGroup(IP)$ at each process will return the new sets at each process.

In the event of an $N$-way split, processes will increment their own entries by the number of merges they will encounter on the way to the final merge from that split, and the entries of all other processes in the set by a number corresponding to the merge at which they will rejoin $p$ (see the definition of MIPS level, 6.2.1).

The following processing is required at merge nodes.

**Algorithm 7.1.9. Merge:**

*Algorithm to reduce MIPS level at merge nodes.*

*Insert at nodes as identified in algorithm 7.1.6.*

*Code applies to stack MSTACK*

$mergeLevel = IP[p];$

$decrement = 1;$

$\text{while( TOP == node) do:}\{ POP; decrement = decrement + 1 \}$

$\text{for(} j = 1 \ldots M \text{)while(} i \in G \}\{ if(IP[i] == mergeLevel) IP[i] = IP[i] - decrement; \}$

$set \ G = myGroup(IP);$  

*Result: the entries in G of all merging processes are reduced to the same MIPS level.*

Having done this, running $myGroup(IP)$ at each process will return the new sets at each process.

**Example 7.1.10.** A simple example of split and merge may be in order: take a two-way split, entered by an MIPS at level 0. Assume processes 0,1,2 take the first branch and processes 3,4,5 take the second branch. Every process going through the split increments its own MIPS level by 1, so all are at an identical MIPS level of 1 and global MIPS info tells us nothing. However, the first 3 processes will have an MIPS table with entries 1,1,1,0,0,0 and the second three will have 0,0,0,1,1,1 because in addition to incrementing their own levels, they also increment the table entries of processes that took the same path. So by comparing the local MIPS level with the table, each process can determine the membership of its MIPS. If the MIPS of
processes 0,1,2 for example now undergoes another split, levels at those processes will be changed, but the tables at processes 3,4,5 are unaffected. Because of the nesting property of split-merge pairs, by the time all processes merge again, any splits within MIPS will have been resolved and the level after a merge including processes 0-5 will set the level everywhere back to 0.

The present code does not provide a way of tracking every individual MIPS after a split. For example, if we have a 3-way split of an MIPS Φ₀ into Φ₁, Φ₂, Φ₃, each process will be able to determine its own MIPS, but a process in Φ₁ will in general not be able to calculate the membership of Φ₃. Communications between processes in different MIPS will therefore require a more complicated data structure.

Algorithm 7.1.11. Loops:

Algorithm to support MIPS level increment for processes inside loops, and decrement for processes departing loops.

At the loop exit node h, insert the following (this assumes that 7.1.4 has been performed, so the actual loop exit is a split with only 2 alternatives):

Set iteration = TOP for stack LSTACK.

Insert code as in algorithm 7.1.8, with the following modifications: If the selected path leaves the loop, then before executing the statement that causes the process to depart, execute algorithm 7.1.9 with decrement set to iteration. Do POPLSTACK to remove the loop counter at the departing process.

If process remains inside the loop, do the following on LSTACK:

POPLSTACK; PUSH(iteration + 1);

Result: processes departing the loop have their MIPS level decremented to the level they had on loop entry, and the set of processes with that MIPS level set to the same set that entered the loop. Processes that remain inside the loop have their MIPS level incremented once per iteration and their process set reset to those processes that remain in the loop at the same iteration.
An advantage of the present system is that merges do not require any communication, since the information on membership of the parent MIPS (and possibly of the process set involved in all partial first merges) is kept locally at each each process.

The code as described in this section could be further optimized; in particular the code relating to the merge tree $M$ in algorithm 7.1.5 could be replaced by an $N \times N$ array containing the results of the repeated traverses of $M$. The current code is described instead for reasons of clarity, since it more directly relates the increments to MIPS levels to the structure of merges. Since split nodes require communications, which are likely to be much more time consuming than any local processing, time spent executing traverses of $M$ may not be significant.

A non-uniform split controlling loop exit requires a vector combine (def: 7.1.1) at each loop iteration. This is an expensive operation, particularly significant in loop execution. A uniform loop, in which we can guarantee that all processes iterate the same number of times requires no special support code and is greatly to be preferred.

## 7.2 Communications in MIPS

In order to establish conditions for non-deadlocking and deterministic execution, we need to establish conditions on communications that preserve these goals. In this work we will restrict our considerations to message passing communications, and we will further assume these communications are reliable. In SPMD execution, each process normally has its own local copy of memory, which makes message passing a good match to the execution paradigm.

Claim 7.2.1. We consider the following sufficient conditions for success of a particular set of messages: Given a communication pattern $R = \{V, A\}$ and a set of processes $G$ that execute communication statements implementing $R$,

1. $V \subseteq G$,
2. $R$ is the result of a uniform computation (def: 1.4.1), and
3. $R$ is acyclic.
If condition **i.** fails, then some process \( p \) required by the communication pattern is not present and communication will fail because other processes wait forever for \( p \).

If condition **ii.** is not true, then it is possible that some process \( p \) expects a message exchange with some process \( q \neq p \), and process \( q \) does not expect that exchange, causing \( p \) to wait forever.

If condition **iii.** does not hold all processes in the cycle will wait for each other and the communication will never finish.

We do not claim that these conditions are required; in particular it is possible that condition **iii.** is violated and still results in the same pattern \( R \) being computed at all nodes. Assume for instance that \( R \) is a ring in which each process sends to the next highest process number, modulo the number of processes. Such a ring can be the result of computation on purely uniform values, or it can be the result of arithmetic on the process number which is non-uniform. We claim merely that if we restrict the computation of \( R \) to uniform expressions, then we can guarantee that \( R \) is the same at all participating processes.

Claim 7.2.1 describes restrictions on a single communication. Deadlock can still result if two communication patterns \( R_1 \) and \( R_2 \) are defined by multiple statements, and execution of those statements is interleaved. It may then be possible that \( R_1 \) and \( R_2 \) contain a cycle when taken together, even though each is individually acyclic. In general this may force us to consider all communications in a program at the same time, which would not be practical.

To resolve this dilemma we further restrict our communication statements to **fusion statements** (def: 3.4.1), characterized by a requirement that all participating processes execute the same statement text. These are defined in the Planguages [BAGHERI 92], [BAGHERI 94] and are a syntactic construct in which each communication is expressed as a single statement in the language. Mpi **send-receive** and collective communication statements can also be considered fusion statements, but to our knowledge only the Planguages restrict all communication statements in this way.
We modified the standard basic block definition in [AHO 85] to include communication statements as leaders. If we restrict communications to fusion statements, this means that each basic block can contain one, and only one, complete communications action. Since we have defined states in terms of basic blocks, we also have only one communication action occurring inside a single state.

Lemma 7.2.2. Communications expressed as fusion statements and restricted to a set $C \subseteq G$ of processes, such that $G$ is the MIPS process set that executes the communication statement, and which meet the conditions of claim 7.2.1, are free from deadlock.

Proof. Conditions of claim 7.2.1 are met:

Condition i. is guaranteed by the restriction $C \subseteq G$; by lemma 6.1.7 all processes in $G$ execute the same code and therefore each executes the communication statement. The restriction to fusion communications assures us, in conjunction with the restriction to processes in a single MIPS, that each communication statement completes before the next statement involving any process in $C$; therefore no other communication statements affect the pattern $R$. Computation of $R$ is generated by the compiler and can be guaranteed to be uniform and free from cycles. Therefore ii. and iii. are met.

Since there are no cycles, all involved processes are present, $R$ is uniform and each message in $R$ is delivered (since communication is reliable), the execution of $R$ reaches completion and the communication does not deadlock. \( \Box \)

We also need to consider determinism. We observe that the only possible source of non-determinism in the conditions of claim 7.2.1 is in the computation of $R$. We note that this computation may include the set of participating processes. If we add the requirement that the computation of $R$ must not depend on random factors, then communication is deterministic as well as free from deadlock.

Lemma 7.2.3. If, to the conditions of claim 7.2.1, we add the condition that the computation of $R$ may not depend on any random factors, then communications that meet all these conditions are deterministic.
Proof. If \( R \) does not depend on random factors, it can only depend on data present at each of the communicating processes. If we have the same set of processes and the same data, then the computation of \( R \) yields the same result. Since we have the same communication code executing the same pattern, communication is deterministic. \( \square \)

### 7.3 Deadlock-freedom and determinism

We will now show that, given communications as described in section 7.2, an MIPS execution is free from deadlock and deterministic. In all proofs in this section we make the following assumptions:

**Condition 7.3.1.** i. SPMD execution with fixed number of processes.

ii. No individual process fails or diverges for reasons unrelated to interprocess communication or interaction.

iii. The transformations of section 7.1.1 have been applied.

iv. Runtime code has been inserted to carry out the algorithms of section 7.1.2.

v. Communications are expressed as fusion statements and restricted to a set \( C \subseteq G \) of processes, such that \( G \) is the MIPS set, as described in section 7.2 and lemma 7.2.2.

First we show conditions for deadlock freedom.

**Lemma 7.3.2.** Given a program that satisfies conditions 7.3.1. Let \( p \in G \) be a process in the process set of an MIPS \( \Phi_G \).

The first statement executed by \( p \) after formation of the MIPS is either: i. the first statement in the program, ii. the first statement executed after a split statement, or iii. a merge statement.

The last statement executed by \( p \) while in \( \Phi_G \) is: i. a split statement, ii. the statement just before a merge statement, or iii. the end of the program.

We refer to these statements as "the first statement executed by the MIPS" and "the last statement executed by the MIPS".
Proof. Identification of first and last statements follows directly from definition 5.1.1. Only split and merge statements change the MIPS set, there is an initial set formed at the first statement and no process continues beyond the last statement. From the definition and lemma 6.1.7 we have that all processes in an MIPS set follow the same path in the CFG. Therefore they execute the same code, so the reference to a “statement executed by the MIPS” holds in the sense that all processes in the MIPS set execute the same statements in the same order. □

We can now state:

**Theorem 7.3.3.** Given a program that satisfies conditions 7.3.1. Let \( \Phi_G \) be an MIPS; let \( s_1 \) be the first statement executed by \( \Phi_G \) the and let \( s_2 \) be the last statement executed by \( \Phi_G \). Then every process \( p \) that executes \( s_1 \) reaches \( s_2 \).

Proof. From lemma 6.1.7 we have that every \( p \in G \) follows the same path on the CFG and therefore executes the same sequence of statements, starting with \( s_1 \). In particular, every \( p \) executes the same communication statements in the same sequence. By lemma 7.2.2, the first individual communication statement does not deadlock. By induction, the following communication statement does not deadlock, and every process continues through all communication statements until \( s_2 \) is reached. □

Since MIPS split-merge pairs are nested, we have:

**Theorem 7.3.4.** Given a program that satisfies conditions 7.3.1. Let \( l \) be a split node and \( m \) be a complete first merge node matching \( l \). Let \( \Phi_G \) be the MIPS that executes \( l \) and splits. Then every \( p \in G \) reaches \( m \).

Proof. Let \( \Phi_K \) be one of the MIPS formed at \( l \). If \( \Phi_K \) does not encounter any split statements, then by theorem 6.1.18 it does not encounter any merges, and by theorem 7.3.3 all processes in \( K \) reach the merge at \( m \).

If \( \Phi_K \) does encounter a split \( l_1 \), by theorem 6.1.18 all merges corresponding to \( l_1 \) must be performed no later than \( m \). Since every process leaving a split is in an MIPS, by induction the same applies to MIPS formed at \( l_1 \) and all later splits, therefore all processes in \( k \) still reach \( m \).
The same argument applies to every MIPS leaving split \( l \), so all processes in \( G \) reach merge \( m \).

\( \square \)

**Corollary 7.3.5.** Let \( \Phi_\Gamma \) be the initial MIPS. Then every process in \( \Gamma \) reaches the end node \( e \).

*Proof.\* Since \( \Phi_\Gamma \) is a single MIPS that includes all processes, the first statement it can encounter that changes the MIPS must be a split statement, by definition 5.1.1. By construction in algorithm 7.1.3, there is a single end node \( e \). By lemma 6.1.10, \( e \) is a complete merge for all processes in \( \Gamma \). By the theorem, every process at the first split reaches \( e \), and since all processes are initially in an MIPS, they all reach the first split by theorem 7.3.3. \( \square \)

We conclude that an SPMD MIPS execution does not deadlock, since all processes present at the start reach the end. If we further restrict communications to be deterministic, as in lemma 7.2.3 then an MIPS execution does not introduce any non-determinism.


CHAPTER 8

OVERLAPPING

We introduced in [GOMEZ 98] a novel form of overlapping, in which not only does data transfer happen concurrently with computation, but in which we also overlap intervals between variable definition and use at different processes to absorb differences between workload or computation speed at different processes.

Our concept of overlapping leads to dynamic, out of order communications scheduling and zero-copy communications. To take full advantage of overlap intervals, a protocol is defined that depends on information about read and write access to variables being communicated. In order to implement the fine grained control required to implement this protocol, a finite state machine is attached to each communicated variable; this machine is described. Although overlapping may be applied to more general communications, we prove its correctness only for fusion communications.

Finally we describe the requirements for inserting support code manually or by compiler.

8.1 Overlapping concepts

We will show that synchronization costs in highly irregular problems can be of the same order as the computation time. We therefore need to consider how they may be minimized or absorbed, to allow efficient solution of such problems. Synchronization cost is due to the properties of the code, or to random factors in the run time environment, and therefore may not be eliminated.

A standard way to hide communications cost is to overlap computations with communications. However, this will not be beneficial to processes that finish their execution in less time than others, and in any case the benefits of such overlap are only possible if special hardware allows communications to be executed in parallel.
For any non-trivial computation carried out in parallel there is some communication cost added to the actual computational cost. This is at least the cost of moving data between processes. In asymmetric or irregular computation (described in the introduction to this work), however, we have an added cost due to time or control asymmetry. Note that this time is in addition to the communications required to synchronize processes. In the following we will refer to “synchronization time” as this waiting time, and include any message passing required to synchronize in the communications time. Suppose two processes, p1 and p2 are initially synchronized and must communicate after performing some computation. Suppose that p1 executes $n_1$ instructions and p2 executes $n_2$ instructions, and that each instruction takes a time $t$ to execute. Then there will be a synchronization cost $T_s = t(n_2 - n_1)$ added to the actual message times. One measure of the irregularity or asymmetry of the problem is given by:

$$ t = \frac{|n_2 - n_1|}{\max(n_i)} $$

Let $T_c$ be the time it takes for signals to travel between processes (including any signals required for process coordination), and $T_n$ be the computation time equal to $t \times \max(n_i)$. Normally we would want to compute in parallel in situations where $T_n \gg T_c$. In highly skewed, irregular problems, the difference $|n_2 - n_1|$ can be of the same order as $\max(n_i)$; the synchronization time $T_s$ is therefore much greater than $T_c$ and the irregularity $T_s/T_n$ can be a large fraction of unity.

Consider an example of asymmetric load balance, based on a figure from the BSP research set at Oxford University (figure 8.1). We want to emphasize that in this, and other examples, we are not talking about badly load-balanced codes. We have in mind codes that are well balanced at a coarser scale but irregular at a fine scale. In the BSP model the data is not needed before the next barrier. Using our definition of overlapping, and starting to communicate data when it becomes available, we can avoid the barrier synchronization, which is replaced by a logical synchronization. Much of the time spent waiting at the barrier for other processes is also saved.

We here use the term “overlapping” to refer to a technique different from overlap-
Figure 8.1: BSP example:
(see http://web.comlab.ox.ac.uk/oucl/research/highlights/bsp_computing.html)
ping computation with communication at a single processor. We define:

**Definition 8.1.1. Overlapping** is a technique and protocol whereby we seek to overlap the time between definition and redefinition of a variable at a sender process with the time between use of one version and the need to use a different version of a variable at a different process. We first described it in [GOMEZ 98]

This is perhaps best understood in terms of a loop. Suppose a producer - sender of a value X and a receiver - consumer of that same value are both executing a loop with index i varying from 0 to M. At each iteration of the loop, the producer updates X and sends it to the consumer, which then uses the updated value of X. In general, a number of other things will be done in the loop, so there will be a time interval dT at the producer between repeated definitions, and a possibly different time interval dT’ at the consumer process between repeated uses. Depending on which process is running faster, either producer or consumer may have to wait (see figure 8.2.)

If, however, processes are loosely synchronous (that is, if they have to synchronize periodically) then it is likely that no process is too far ahead of the others, which makes it probable that there will be some overlap in time between the definition-redefinition interval dT at the sender and the use-reuse interval dT’ at the receiver.

If we can arrange it so that data transfer occurs during this overlap period, then the sender does not have to wait, because it has sent its data before needing to update the variable, and the receiver does not have to wait, because it gets the data before it needs to use it. No buffering is required at either sender or receiver since the sends takes place when both are ready for it; it is in fact a synchronous send, even though the actual time of its occurrence is not fixed at either the sender or the receiver. In figure 8.3 we show the same BSP example of figure 8.1, showing possible speedup from overlapping.

Of course, if there is no overlap between dT and dT’ then the first process to reach the end of its interval must wait for the other; or we could use lazy buffering, in which the message is copied into a buffer for later transmission only if it could not be transmitted during the overlap period. This sender is always allowed to continue
Figure 8.2: Wait conditions

- X needed
- Receiver waits
- X defined
- Sender waits
- Data in buffer
- Unbuffered send
- X needed

- Unbuffered send
- Data in buffer
- Sender waits
- X defined
- Receiver waits
- X needed
(The receiver cannot continue because it must use the data). Alpert et. al. propose a technique similar to lazy buffering for large messages in the NX message passing system for the Shrimp multiprocessor [ALPERT 96], in which a sender process simultaneously checks if the receiver is ready and starts copying the message into a buffer while waiting for a reply from the receiver. Copying to a the buffer is only completed if the receiver was not ready, and the buffer is only used in this case.

We should note that this concept of overlapping is not obvious even to researchers who are explicitly considering overlapping communication with computation. For example, [BSP 96] explicitly rejects overlapping as able to produce gains only of the order of the communication time, ignoring the synchronization wait time.

A conceptually similar technique has been proposed in [CHAMBERLAIN 97], and implemented as the Ironman library. However, this implementation focuses on hints to the compiler as to where communication may and must occur. The library then instantiates communication using the fastest mechanism available on the tar-
get machine. This reduces transfer latency and increases bandwidth compared to
an implementation built on standard portable libraries, but does not address the
synchronization wait problem.

Another related approach is that of Split-C [CULLER], in which one-sided com-
munication, in the form of gets and puts, followed by a synchronization which enforces
completion. This allows multiple communications to be initiated and pipelined, and
allows the communication to occur in parallel with computation if special hardware is
available to support this. It is more limited than our approach in that communication
still proceeds in the order in which it is declared, so if one request is stalled, the whole
pipeline must wait. Also, synchronization requires completion of all communication
and does not allow the possibility that some requests must complete at a given point
in execution, but others do not need to and could be allowed to continue.

8.1.1 Benefits with/without special hardware

As we have seen, overlapping does not depend on special hardware, there will be ben-
efits regardless. However the communications cost of overlapping may be somewhat
higher than that of other techniques, for two reasons: the first is that, in order to
use synchronized data transfers, some additional communication between processes is
required to determine that both processes are ready. Second, processes must be able
to respond to messages received at any time; for example, a sender process must be
able to receive a clear to send signal from a receiver in order to know that it can send
the data, and it must be able to respond to this signal at some arbitrary time in the
interval dT between definition and redefinition of the variable being sent.

In early SOS implementations this is done by polling the communications system;
in the present implementation it is done through timer signals periodically interrupt-
ing the computational process and checking the communications system. The code
that performs these interruptions is compiled together with the program and runs as
part of the same process without incurring the expense of context switches.
The overhead required by the overlap run time system is proportional to the amount of communications that must take place (both number of messages and size of messages). As the computational size of a problem increases, this communication cost is likely not to increase as quickly, and should therefore become less important than the synchronization delays that overlapping directly addresses.

However, if there is a communications processor capable of handling the actual messaging and of two way signaling to the main process, then all this checking of the communications system can be done by the communication processor, and it does not have to interrupt the computational process unless there is actual data to be received or sent. We still require a run time system to support overlapping, but the processing that must be done by the run time system, and the actual sending and receiving of data, could then be conventionally overlapped with computation, producing a greater saving.

### 8.1.2 The benefit from multiple processors:

The above discussion assumes that, regardless of the presence or absence of communication processors, each process in a parallel execution is running on an individual processor. Suppose, however, that we are instead running multiple processes on a single physical processing unit, through the facilities of an operating system.

This situation may appear to be equivalent to running on a multi-processor system which is running many other tasks on each CPU. A closer examination reveals that this is not the case. On a single processor, a process that is blocked because it is waiting for information from another process takes very little CPU time. The operating system does need to periodically check each process to see if it is ready to run, but does not need to dedicate significant time to an unready process; it can dedicate most CPU time to processes that can continue. Therefore no significant time is spent waiting for synchronization; assuming at least one process in the parallel execution can run (no deadlock), all wait time is spent in useful work on some other
process. This conclusion is not affected by the presence of other tasks on the system, since it applies to whatever time is available to our multi process execution.

Therefore, if running multiple parallel tasks on a single CPU, overlapping will not save synchronization wait time, because there is no such wait time in this case. Since overlapping takes extra work to achieve as compared to simple messaging, it is likely to be slower than standard methods when multiple processes run on a single CPU.

8.1.3 Out-of-order, asynchronous messaging

It is likely that more than one variable may need to be transferred, between processes, and also that a given process will be a sender of some values and a receiver for others. It is also possible that a given process will be both a sender and receiver as a participant in some collective communication.

The benefits of overlapping would be lost if, while waiting for a pair of processes to be able to transfer a variable X, communication involving one or both processes for a different variable Y were blocked. Therefore any run time system that manages overlapping communications must be, and SOS is, designed to check all pending communications at each process and carry out those that are able to complete. Communications therefore may occur in an order different from that specified in the program code (see figure 8.4), and the system and protocol ensure that it occurs in such a way that the semantics of the program is not changed.

Suppose a process is updating both X and Y, sending X to one process and Y to a different process, all processes executing a loop. It is possible that, in two different iterations of the loop, the sender process will in send X and Y in a different order, regardless of the order in which it actually updates them.

In two repeated runs, the order of sends and receives for the same iteration of the specified loop could be different. That is, the order of sends and receives between which there are no dependencies is nondeterministic. (Of course, if there is a real dependence involved, such that X must be sent before Y for some reason, then the overlap run time system will force the send of X to complete before that of Y begins.)
Figure 8.4: Overlapping, out of order communications
One consequence of this applies particularly to collective operations such as reductions. For example, a reduction that applies an operator OP to all pairs values of \( \{X\} \) in Planguages) over all processes is normally specified in such a way that it must be correct regardless of the order in which the operations are carried out. Particular systems generally implement this operation in some order, however. The overlapping run time system may in fact carry out these operations in a different order on two consecutive runs. It is possible for the system to enforce some fixed order of operations, but some of the benefit from overlapping will be lost. (The present API does not give the programmer access to the flag that enforces a fixed order in reduction operations; this will be corrected in a later version).

### 8.2 Overlap protocol

We proposed in [GOMEZ 98] to separate the execution of sends and receives and overlap the periods between generation and use of data in different processes. If this period at a producer process overlaps the period between references to data at a consumer process, and the data can be sent during this overlap, then neither process will have to wait for the other. Depending on which process is faster we generally only need to have either sender or receiver wait, but rarely both at once. It is difficult for the programmer to implement such overlap, so we propose here a protocol for overlapping communications that may be implemented automatically at the level of a compiler or preprocessor, or with a library with suitable run-time environment. We will implement this protocol in the context of the theory of MIPS (def: 5.1.1) and fusion communication statements (def: 3.4.1) developed above.

Consider a simple case of point to point communication between sender P1 and receiver P2 for data item X. We have the following possibilities:

X.1, X.2, ...X.n are the first, second and nth values in memory location X. In general we have statements such as:

\[ S1: X = (\text{expression}) \]
Table 8.1: Sender(P1)-receiver(P2) timing

<table>
<thead>
<tr>
<th>time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>case 1: sender is slightly faster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>write X.1</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>read X.1</td>
</tr>
<tr>
<td>3</td>
<td>write X.2</td>
<td>—</td>
</tr>
<tr>
<td>case 2: sender is much faster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>write X.1</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>wait X.2</td>
<td>read X.1</td>
</tr>
<tr>
<td>3</td>
<td>write X.2</td>
<td>—</td>
</tr>
<tr>
<td>case 3: receiver is faster</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>—</td>
<td>wait X.1</td>
</tr>
<tr>
<td>2</td>
<td>write X.1</td>
<td>wait X.1</td>
</tr>
<tr>
<td>3</td>
<td>write X.2</td>
<td>read X.1</td>
</tr>
</tbody>
</table>

... 

S2: \( Z = \text{(expression using X)} \) 

... 

(S3: \( X \at \ P2 = X \at \ P1 \); original communication statement) 

... 

S4: \( Y = \text{(expression using X)} \) 

... 

S5: \( X = \text{(expression)}; \) maybe next iteration of a loop 

... 

S6: \( Y = \text{(expression using X)} \)

All references to \( X \) are in S1 ... S6. Referring to Table 1; S1 is a write at \( P1 \) and S4 is a read at \( P2 \). Since the processes are not synchronized, later statements at one
process can occur before earlier statements at another; e.g. case 3, t=1 is the situation where P2 reaches S4 before P1 reaches S1; this forces P2 to wait before it can read. Recall that both P1 and P2 are executing the above sequence of statements, but in the given case we need the value of Y only in P2, and we are only using the value of X produced at P1.

P1 has data which it may send at S1, and it must send it before S4 when it needs to update the local value of X. P2 may read X any time after S2; it must read it at S4. Similarly, the value of X that P2 uses in statement S6 must be read after S5 (where X is once more updated - P2 executes this statement even though this value of X is not used). X must be read at S6 at the latest.

We now use this simple example to define the various states of our protocol.

P1 is in a state MAY-SEND (with respect to a given data item and communication statement) as soon as it calculates the value of X at S1; it is in a state MUST-SEND at S5, because here it cannot proceed to recalculate X until it sends the old value. P2 is in a state MAY-READ after S2 and before S4. It can not read X before S1, because here it executes code to recalculate X, even if we do not want this value. Once P2 reaches S4, it is in a state MUST-READ, and here it waits until P1 is ready to send.

In case 3, if we start at t=2, sender writes X.1 when receiver wants it, that is, synchronized. If sender is much faster than receiver, it starts waiting for X.1 earlier, the end result is same as case 3.

Processes in MAY states (MAY-SEND, MAY-READ) can continue processing until they reach a statement that puts them in a MUST state (MUST-SEND, MUST-READ). In a MUST state we either need to use a value from another process to calculate something, or we need to replace a value that some other process needs. In either case, we have to wait for the data transfer to take place. We consider a process to be in a QUIET state if it is neither ready to send nor to receive.

Table 2 enumerates the possible state interactions between the processes.

Data is transferred in states 11, 12, 21 and 22. In cases 12 and 21, one process must wait for one data copy, but the other (the one in the MAY state) can continue. In case 11, neither process has to wait, computation can completely overlap communications.
Table 8.2: States of a sender-receiver pair

<table>
<thead>
<tr>
<th></th>
<th>P1 state</th>
<th>P2 state</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>QUIET</td>
<td>QUIET</td>
<td>both processes continue</td>
</tr>
<tr>
<td>01</td>
<td>QUIET</td>
<td>MAY-READ</td>
<td>both processes continue</td>
</tr>
<tr>
<td>02</td>
<td>QUIET</td>
<td>MUST-READ</td>
<td>P2 waits, P1 continues</td>
</tr>
<tr>
<td>10</td>
<td>MAY-SEND</td>
<td>QUIET</td>
<td>both processes continue</td>
</tr>
<tr>
<td>11</td>
<td>MAY-SEND</td>
<td>MAY-READ</td>
<td>P1 sends to P2, both continue</td>
</tr>
<tr>
<td>12</td>
<td>MAY-SEND</td>
<td>MUST-READ</td>
<td>P1 sends to P2, P2 waits</td>
</tr>
<tr>
<td>20</td>
<td>MUST-SEND</td>
<td>QUIET</td>
<td>P1 waits, P2 continues</td>
</tr>
<tr>
<td>21</td>
<td>MUST-SEND</td>
<td>MAY-READ</td>
<td>P1 sends to P2, P1 waits</td>
</tr>
<tr>
<td>22</td>
<td>MUST-SEND</td>
<td>MUST-READ</td>
<td>P1 sends to P2, both wait</td>
</tr>
</tbody>
</table>

In case 22, both processes wait, but they wait at the same time so total wait is only for one copy.

Overlapping is a zero-copy protocol. The protocol establishes that both sender and receiver are ready, which allows a synchronous transfer to take place. Data is transferred directly from its original location at the sending process to its final location at the receiver, without the need for a system buffer (except for low level buffers that the transport layer may require). This forces a sender to wait before rewriting data that it needs to send. A simple extension would be to buffer data at the sender only when the sender process needs to block, thereby allowing the sender to continue, a technique we call lazy buffering. A similar strategy is used in NX message passing on the Shrimp multiprocessor [ALPERT 96], in which the sender starts to copy data to a buffer while waiting for a receiver to acknowledge that it is ready. The buffered copy is used to allow the sender to continue processing if the receiver is not ready.

All process states in table 2 refer to some specific data item. For example, a process may have calculated X and so be in MAY-SEND with respect to it, and need a value Z from someplace else in the current statement being executed, which puts it in MUST-READ with respect to Z.
8.2.1 Collective communications

Each communication is regarded as a set of point to point communications executed at each node. Note that, at a particular node, some of these point to point communications must occur in specific order, but others do not have to. For example, in a broadcast, a given process will in general receive a single data item and forward it to multiple other processes. The single receive must precede all the sends, but the sends may and occur in any order, and should occur in the order of processes that are ready to receive.

A reduction operation is more complicated, involving a receive phase from multiple processes in any order (with computation of the actual reduction operator), followed by a send to a single process, followed by the same steps of a broadcast. (In a later section we will describe the algorithm SOS actually uses for broadcasts and reductions). However, buffering must be taken into account here. Some systems (e.g. MPI asynchronous receives) require different buffers for each receive operation concurrently posted, to prevent a receive from interfering with computation following a previous receive. (If it is necessary to avoid multiple buffering while ensuring correctness, this receive section may be executed in order, to force each receive to a single buffer to complete before each other receive).

In general, a single collective communication may proceed at any given node in several phases, such that each phase must precede the next but individual sends or receives in each phase may be in any order.

Also, it is possible to preserve correctness while relaxing some of the restrictions on when a process must proceed. Specifically, a process participating in a collective operation enters into a MUST-ALL state just before the next use of the variable in question, the same as a receiving process. This is because a collective operation in general includes receives, and the data must be received before it may be read. When, however, the collective operation has reached a phase in which only sends are left, there should be no further restriction on reading the data. We could here transition from MUST-ALL to MAY-SEND and follow thereafter the restrictions for a sending
process (the root process for a broadcast would always be in this situation, since it
does not receive).

8.3 The Overlap FSM

Table 2 gives the states of a Finite State Machine (FSM) that implements a protocol
that is sufficient to handle simple point to point communications. However, we also
wish to handle more complex operations, such as reductions and limited broadcasts,
including reductions among a subset of processors. We refer to this general class of
operations as group communications. Figure 1 defines a Finite State Machine (FSM)
that implements our protocol, with added states to permit short-cutting and group
communications. We assign a separate FSM at each node to each variable being
communicated. Group communications, to be efficient, require cooperative action,
in which some nodes will pass information on, both sending and receiving. We use
a message count to determine completion, implemented by adding a counter to our
FSM, which emits a “zero” message when there are no pending operations. Messages
define the transitions in the FSM: each message corresponds to a particular event,
and is a letter in the language accepted.

States in Fig. 1 are divided in 3 general paths corresponding to sends, receives
and group communications. Any particular communication is defined as a set of
point to point sends between nodes, such that the receiver sends a CS message when
ready to receive, and the sender then sends the data message. One or more sends
from a node may start with Send, allow multiple RH and CS (each CS corresponds
to a message send and a decrement of the count, each RH is a use of the variable
without redefinition), terminating in Z or blocking at a redefinition (LH), followed
by at least one CS (send and decrement), terminating in Z. (That is, $S(RH|CS)^*Z$
or $S(RH|CS)^*LH \cdot CS \cdot CS^*Z$ is a word accepted by the FSM.)

One or more receives at a node allow multiple message (MSG) arrival (and decre-
ment count), terminating in Z or blocking on first use or definition (RH or LH), fol-
Figure 8.5: Broadcast, based on hypercube algorithm.
With overlap FSM, only data dependences have to be satisfied; for example node 1 must wait for the step 1 message from node 0. However, if nodes 0 and 4 arrives at the broadcast before node 1 does, the message from 0 to 4 could be sent first. Protocol restrictions can also change during the broadcast; for example an RH before node 0 sends to node 1 will cause node 1 to block; once the message is received only sends remain at node 1 so that process can be released, it can continue computing until it encounters an LH.
Figure 8.6: Finite State Machine to implement overlapping protocol without buffering. This machine also supports short-cutting.

Alphabet: (messages) { 
F – false 
T – true 
S – define send 
R – define receive 
A – define allgather 
RH – variable used 
LH – variable written 
CS – clear to send 
MSG – message received 
Z – pending count is 0 
C – shortcut 
DC – set shortcut 
SP – send phase}
lowed by at least one MSG (and decrement), terminating in Z. (That is, \( R \cdot MSG^*Z \) or \( R \cdot MSG^*(LH|RH)MSG \cdot MSG^*Z \))

A group communication allows multiple message (MSG, decrement) arrival, multiple CS (message sends, decrement), terminating in Z or blocking on first use or definition (RH or LH), followed by at least one MSG (and decrement) or CS (send and decrement), terminating in Z. (That is, \( A(MSG|CS)^*Z \) or \( A(MSG|CS)^*(LH|RH)(MSG|CS)(MSG|CS)^*Z \))

We have defined protocols for overlap between variable definition and next definition at the sender; and between variable use and next use at the receiver. Processes are in a may send/receive state during the overlap period, and enter a must send/receive state just before the next use/definition. Note, however, even if we are in an overlap section, communication can not occur until the senders and receivers are identified. Therefore we consider the ‘may communicate’ interval to begin when senders and receivers are defined (that is, at the SOS communication statement), and to end (at sender) before the next definition and (at receiver) before the next use.

Suppose we have a set of processes \( \{p \in G\} \) indexed by a set G with \(|G|\) elements. For example, if \( G = \{1..n\} \) then \(|G| = n\). We are presently able to handle the following cases (others are treated as compositions of these):

1. Point to point: send \( x \) at \( j \), receive \( y \) at \( i \); a single send matched by a single receive. (In this case, we define the count to be 1.)

2. Send to a set \( G \): (count of \( n = |G| \) at sender, count of 1 at each receiver). It is possible to define it as a collaborative broadcast, using e.g. a spanning tree rooted in \( j \). Doing this costs implicit synchronization and ordering

3. Reduction: \( Y \) at \( i \) is the result of applying some operator to pairs of values \( x \) at all nodes in \( G \). (count of \( n = |G| \) at receiver, count of 1 at each sender). Order is arbitrary, and communications are dynamically scheduled. Therefore correctness requires operations to be commutative and associative (it may be sufficient that operations be only approximately commutative and associative, as is the case for floating point operations on most computers).
The requirement for associativity and commutativity may be removed at the cost of forcing reduction operations to occur in a specific and repeatable order. This would increase waits for synchronization, since it would restrict some opportunities for dynamic scheduling.

All communications are resolved into sets of point to point sends and receives (note that this does not preclude collaborative communications, since a particular node may receive a message from another and forward it to yet another node, optionally adding information of its own).

We assume a correct program in which communication statements express a transfer of information and include senders and receivers; this is the style of MPI [MPI 95] group communications and of Pfortran [BAGHERI 92], to name two examples. Processes each see the communications in which they participate in the same order. As a result, we know the communication pattern before communicating (that is, we know the set of processes involved, and what sends and receives are required at each process). We can therefore resolve any communication into a pattern of point to point sends ahead of time. Therefore for every send of a variable at a given process, there is a matching receive (of possibly a different variable) at the destination process, and vice-versa. Assume the message includes an identifier that indicates what variable is being sent (this is necessary because, if several messages are received by a given node, it must know which buffers to use for the received data).

Note that communications proceeds by alternating CS (receiver→sender) and MSG (sender→receiver). A sender state may not accept without receiving a correct count of CS messages; a receiver state may not accept without receiving a correct count of MSG, and an all-reduce state may not accept without receiving a correct count of both CS and MSG.

Therefore communications are ordered with respect to each variable, each communication completes before the next communication of the given variable is required. It can be shown that the correct value is transmitted. For collaborative communication, we must also prove the correctness of the group algorithm. This issue will be addressed in subsequent publications.
8.4 Correctness

Overlapping takes communication which is logically a synchronization and spreads it out over time. This leads us to introduce the concept of logical synchronization.

If we restrict the memory access (or show that the access is restricted) so that the data available to each process is the same as what it would be if we had placed barriers at each synchronization, then we still have a synchronization even if processes are not physically present at the same time. We will call this a logical synchronization.

A logical synchronization for a set of parallel executions is a restriction on each execution such that the parallel data view (def: 2.3.4) is the same as if a barrier synchronization or blocking synchronous communication had taken place at the logical synchronization. We have:

**Definition 8.4.1.** A logical synchronization $S_{G,J}$ is a state such that there is some state $S_{G,K}$ with $J \leq K$ at which the information transfer can be expressed as a fusion statement (def: 3.4.1), and such that the parallel data view at every state $S_{G,L}$ with $S_{G,J} \leq S_{G,L} \leq S_{G,K}$ does not include data communicated at $S_{G,J}$. (There is always at least $S_{G,J}$ itself with this property, and if there is no other such state, then the logical synchronization is an actual synchronization).

Note that a logical synchronization is not as restrictive as a real barrier. A barrier requires information exchange between all processes as they encounter the barrier. A logical synchronization only requires that a process that views information after the barrier see results that are consistent with whatever was supposed to be exchanged at the barrier. A process that does not view any information exchanged at a barrier may pass through a logical synchronization without interacting, and any interaction between processes can be delayed until information is actually going to be viewed.

A logical synchronization occurs, for example in buffered message passing. Suppose there is a send instruction in a process $p$ and a matching receive instruction in process $q$ for a variable $x$. Process $q$ receives the value of $x$ that $p$ has when it executes the send, so logically the paired send and receive are a synchronization; data is transferred as if a state had occurred with $(p,send)$ and $(q,receive)$. Since data
transfer in fact takes time to accomplish, and in fact the send may be buffered, it is not necessary that the two serial states in fact occur together; in general the receive can occur after the send. If we required an exchange of information instead, where for example \( p \) and \( q \) exchange their respective values of \( x \), then we really would need \((p, \text{exchange})\) and \((q, \text{exchange})\) to execute together, and if one process was running faster it would have to wait for the other so that both can execute the exchange instruction simultaneously.

The concept of logical synchronization is related to that of synchron (def: 5.1.3). An actual barrier synchronization inserted in a synchron, or in the same basic block as a logical synchronization would execute correctly and make no difference to the data view (def: 2.3.4) of an execution. A logical synchronization is a restriction on the execution such that the parallel data view is the same as if a barrier synchronization had occurred, but the number of possible states in the execution is larger.

We also require that variable declarations be logical synchronizations. As we noted in remark 2.3.2, it is necessary that all variables be declared before they may be used in communications. Absent any kind of synchronization on variable declarations, it would be possible for some processes to reach a communication statement involving some variable before other participating processes have declared that variable. Status messages sent by the overlap protocol with respect to a particular variable may then fail. Possible solutions to this are to perform an actual synchronization after variable declarations, or to store status messages with respect to variables not known to the run time system, and re-examine those messages as variables are declared and entered in SOS. Since SOS communications are synchrons, we know that we must encounter a suitable declaration at every process before the actual communication statement is reached.

The use of SOS requires a static analysis of the program and insertion of calls to run time support as described in section 7, to identify and keep track of implicit process sets. Knowledge of the process set allows us to determine at run time if the membership function \( \exists = \text{FALSE} \). It is possible to have correct point-to-point communications between processes in different MIPS if the programmer can ensure both
MIPS will encounter matching communication statements. However, the analysis we have described allows us to support communications between processes in the same MIPS, and the present implementation of the SOS library restricts collective operations to a single MIPS. This limitation may be removed in later work.

SOS communication statements are fusion statements (def: 3.4.1). Since a fusion statement expresses a complete communication, all statements in a program following an SOS communication statement in the text must execute as if the data transfer and any computation specified in the fusion (such as a reduction operation) has actually taken place at the SOS statement.

The finite state machine (FSM, given by fig: 8.6) which implements the overlapping protocol requires information about variable usage, given in the figure by RH (variable read) and LH (variable update) messages. These messages are, in the current implementation, produced by $SOSdoRH(x)$ and $SOSdoLH(x)$ calls inserted in the code ahead of each appearance of a variable $x$ that appears in an SOS communication statement. RH and LH messages are used by the FSM to determine if program execution needs to be suspended until the communication specified by an SOS statement is complete.

Given MIPS support we have:

\textbf{Lemma 8.4.2.} \textit{SOS communication statements executed by processes in an MIPS, for point-to-point, broadcast and reduction operations, given identification of MIPS as described in 7, are non deadlocking and deterministic.}

\textit{Proof}. The lemma follows from lemmas 7.2.2 and 7.2.3, since SOS communication statements are fusion statements. \hfill \Box

\subsection{8.4.1 Point-to-Point communications}

We first consider correctness of point-to-point communications:

\textbf{Lemma 8.4.3.} \textit{Point-to-point data transfer of the contents of a variable X at process p to a variable Y at process q is seen to occur (that is, the data view trace def: 2.1.5
is the same as if the SOS statement had been a synchronous fusion statement def: 3.4.1) at the SOS point-to-point statement.

Proof. Data transfer does not occur before both p and q execute the SOS statement: because it is a two sided statement with a sender and a receiver and requires participation by both processes. The SOS data transfer statement is a fusion 3.4.4.

The overlapping protocol requires that an RH message be generated for Y before the next appearance of Y on the right hand side of an expression (a read of the variable). If the communication has not completed at this time, the receiver process q will be blocked until it completes. Since this is the first appearance of Y, the data view trace (def: 2.3.4) is no different from what it would be if the communication had happened at the data transfer statement, therefore the set of executions at which communications occurs between execution of the SOS communication statement and the RH message have the same data view trace and are equivalent (theorem 2.3.6).

If an RH message is generated for variable X ahead of the next read of X, it will have no effect, because X is being read by the communication statement and not redefined. Neither p nor q needs to block, and neither will do so (no blocking operations are required in this case).

An LH message is required for X and an LH message is required for Y before the appearance of, respectively, X or Y on the left-hand side of an expression (a write to the variable). If the communication has not completed at this time, the FSM at whichever process (p or q) receives an LH statement will be blocked until the communication completes. Therefore the value transmitted by p will be the value of X at the time the data transfer statement was executed; and the value received at q will not overwrite a redefinition of Y at q.

Therefore neither process blocks unnecessarily, but the next references to variables X and Y will be consistent with Y having been set to the value of X at the SOS point-to-point statement.

Theorem 3.4.4 shows that a sequence of fusion statements has a non-cyclic dependence graph. Since SOS communication statements are fusions, the theorem applies to them. However, since actual execution of SOS data transfers may happen after the
program has executed the communication statement, in some cases execution of two
different SOS communications may be interleaved. We now show such interleaved
execution will not deadlock. □

**Lemma 8.4.4.** Two SOS point-to-point data transfer statements do not conflict (def:
2.2.13) with each other.

*Proof.* If the two statements are in different concurrent sections of the CFG, or if one
statement occurs in a section of the CFG that must complete before the other then
there can be no conflict because either the two statements are executed by different
processes (or not at all if the named processes do not execute the particular section
of the CFG), or one statement must complete before the other is begun.

If the two statements refer to variables that are independent of each other, then
they may be executed in any order and it does not matter in which order the SOS
system actually executes them.

If the two statements refer to variables between which dependences exist, then the
variables must appear in expressions that define the relations between them. If this
is the case, then LH and RH statements must be inserted ahead of such expressions,
and these will cause processes to block and force the data transfers to occur in the
correct order. □

### 8.4.2 Collective data transfers

Collective data transfers such as broadcasts and reductions are defined in terms of
point-to-point data transfers. Therefore we can say:

**Lemma 8.4.5.** Given a broadcast or reduction that acts between processes in an
implicit process set, the data transfer defined is seen to occur (that is, the data view
trace def: 2.1.5 is the same as if the SOS statement had been a synchronous fusion
(def: 3.4.1) at the SOS broadcast or reduction statement, and does not conflict def:
2.2.13) with other SOS statements.
Proof. The implicit process set is known through MIPS analysis and the SOS support for it.

The collective communication is defined as a set of point-to-point transfers between the nodes in the implicit process set. Since all nodes have the same process set and all use the same algorithm, all define the same set of point-to-point transfers and know the sends and receives (and possibly operations, in the case of a reduction) that they must execute.

RH and LH messages with respect to any variable in the collective communication will cause a process to block if there is any pending point-to-point communication with respect to the named variable that would cause the process to block. Therefore all the individual point-to-point communications in a collective communication will be correctly executed, and act as if they occurred at the point where the given process executed the SOS data transfer statement, by lemma 8.4.3. Also, by lemma 8.4.4, statements will not conflict with each other or with other communication statements.

Therefore the collective communication is seen to transfer data at the point in the code where the communication statement is executed, and does not conflict with other SOS communication statements. □

8.5 Coding requirements

MIPS and fusion statements (definition 3.4.1) define the semantics of communication, and we see in section 8.4 that overlapped communication will be correct, deterministic and free from deadlock as long as the LH and RH messages are correctly generated during program execution. Specifically, we see in lemmas 8.4.3, 8.4.4 and 8.4.5 that RH and LH messages must be received by the FSM controlling a communication involving X ahead of the next read and write access (respectively) to a variable X that is named in an SOS communication statement.

The current implementation generates RH and LH messages to the FSM by insertion of calls to the runtime system before the use of each variable X that appears in an SOS communication. These calls are SOSdoRH(X) and SOSdoLH(X), respectively
generating RH and LH messages to the FSM controlling communication involving variable X.

As we see from the description of the overlap protocol, insertion of these calls is required to generate the RH and LH messages and ensure correctness. We now show that, in cases where we are not sure if an SOSdoRH or SOSdoLH call is required, it is not an error to insert one.

**Lemma 8.5.1.** An SOSdoRH(X) or SOSdoLH(X) call with respect to a variable X may be inserted anywhere in a program without changing its meaning or causing it to deadlock.

*Proof.* We see from the SOS finite state machine definition that a SOSdoLH or SOSdoRH call with respect to a variable X does nothing unless there is a pending communication (either send or receive) with respect to that variable. Therefore there is no effect on the program if this is the case.

Suppose there is some communication pending with respect to X; then an SOSdoRH(X) or SOSdoLH(X) may cause a process to block until the communication finishes. But this will not change the meaning of the program because the communication, having been declared, may correctly take place. At most this will force a communication to finish before it needs to be completed.

That SOS communication will not deadlock is established by lemma 8.4.2. □

If a fusion includes several nodes in the CFG, it is possible that a process executing the fusion could execute any one of its nodes, and continue along any of several possible paths on the CFG. It is therefore necessary to identify the next read and write to X along every path that passes through any of the nodes in the fusion. This can be accomplished through standard data-flow analysis of the program, specifically by identifying reaching definitions in the program (see Aho and Ullman, or any standard compiler text).

SOSdoLH and SOSdoRH calls are extremely lightweight if no message is pending for the variable they reference, returning immediately after a single read. Therefore a practical alternative in the absence of full compiler analysis is to simply insert the
appropriate call ahead of every reference to X in the program. Superfluous calls will have no effect on correctness (lemma 8.5.1) and minimal effect on time. This is in fact what was done to insert SOS calls in the test problems described in chapter 10.
CHAPTER 9
SHORT-CUTTING

Some problems, in particular some types of search and optimization problems, are
or may be parallelized in ways which may lead to large differences in computational
workload at different processes. In [GOMEZ 98] we proposed a technique we call
short-cutting to reduce the execution time of such problems. These codes can have
parallel efficiency exceeding one in some cases. We now describe short-cutting.

9.1 Definition

In some problems, a parallel execution may give dramatic improvement over its serial
counterpart. For example, suppose the computation involves testing various cases
and terminates as soon as any one case passes a test. In a parallel execution, as soon
as one process finds a passing case, it can short-cut the other processes by telling
them to quit. The parallel algorithm may do less work if it finds the answer on one
node before finishing the computation on other nodes. We have so far applied this
technique to a minimization problem, but we believe its applicability can be extended
to more general cases.

Short-cutting gains are typically of the same order as the synchronization delays
in the problems to which they may be applied, since in order to assert a short-cut,
a process must have completed its share of computation. We proposed in chapter
8 a protocol to tolerate or hide synchronization delays by overlapping intervals be-
tween variable definition and use at each parallel process. Both techniques should be
employed together so as to prevent losses from synchronization offsetting gains from
short-cutting.

Consider a backtracking algorithm (See [HOROWITZ 78]). Conceptually we are
performing different computations on the same data, by following first one possible
computational branch, then another if the first fails, and so on. Suppose we parallelize this algorithm by going down multiple branches simultaneously. (This is a parallel emulation of a non-deterministic choice algorithm, except that, lacking unlimited parallelism, we may either have to sequence multiple branches on each node or limit the approach to problems with a fixed branching factor). Suppose further that we have some criterion that tells us when a solution has been reached. Since each branch follows a different computational path, the amount of work to be done along each branch is different.

Let us consider a simple case in which we have $P$ processors and $P$ branches must be explored. Let $W_i$ be the work required on branch $i$. If all branches are explored one after another, the total amount of work done serially is:

$$ W_{seq} = \sum_i W_i + O $$

(9.1)

where $O$ is the serial overhead. Henceforth we will assume the overhead is much smaller than the computational load and drop this term.

We may calculate the average work per process: $W_{avg} = \frac{\sum W_i}{P}$, which allows us to rewrite the serial work as

$$ W_{seq} = P \cdot W_{avg} $$

(9.2)

Assume that not all branches lead to solutions, but that out of $P$ branches there are $s < P$ solutions, and that we have some criterion that tells us when a solution has been reached along a particular branch without comparing to other branches (for example, this could be the case for a search with a fixed goal state where all we care about is the goal, not the path). Then we can halt as soon as a solution has been reached. When this will happen depends on the order in which we take the branches. Assume for the sake of simplicity that there is a single solution: $s=1$. Also assume that the units of work and time are the same, so that one unit of work executes in one unit of time. On the average we would expect to find a solution after trying about
half the branches, and we should have, in the average case, a serial time of:

\[ T_{seq} = \frac{1}{2}(P \cdot W_{avg}). \] (9.3)

(Here we are assuming that the algorithm halts on all branches; we can always force this by some artificial criterion like a counter).

Now consider the parallel case. In this situation, we can halt as soon as any one of our processes has found a solution. We would then have that parallel work done on \( P \) branches is:

\[ W_{par} = P \cdot (\min(W_i) + O). \] (9.4)

Take the case of small overhead. Then we have:

\[ T_{par} = .5P \cdot W_{avg} \] (9.5)

and the parallel execution time \( T_{par} \) is

\[ T_{par} = W_{par}/P = \min(W_i) = W_{\min}. \] (9.6)

One measure of parallel processing is the parallel efficiency. We make the assumption that an ideal parallel computer would achieve a speedup equal to the number of processors (when compared to a single processor). For example, a four processor machine would solve a problem in one-fourth the time of a single processor, using the same algorithm. We define the efficiency as the serial run time divided by the parallel run time multiplied by the number of processors; this would be equal to one for our ideal computer. There are discussions of speedup and efficiency in, [ROBERT 90], [LEISS 95] and [GREENLAW 95], among others. We note that Roberts, for example, concludes in his discussion that efficiency greater than one is not possible. This is the standard view on efficiency, based on the concept that the total work done by an algorithm can not be less when executed in parallel than when executed in series. We disagree with this reasoning because it ignores the fact that computational results
are available in parallel at different times and in a different sequence from their availability in a serial computation. The arguments and results we are here presenting, and also the work of Segre and Sturgill in the nagging system [SEGRE 97], provide counterexamples to this conventional view.

In the short-cutting case, the parallel efficiency would, on average, be:

\[ E_{\text{par}} = \frac{T_{\text{seq}}}{PT_{\text{par}}} = \frac{W_{\text{avg}}}{2W_{\text{min}}} \]  

(9.7)

where we know that \( W_{\text{min}} \leq W_{\text{avg}} \). Thus the parallel efficiency is always \( \geq 0.5 \). If it should happen that \( W_{\text{min}} < 0.5W_{\text{avg}} \), then we would have a parallel efficiency greater than 1 in the average case.

In the worst case for parallel efficiency, we would find the solution on the first serial trial. Then the total serial work would be \( W_{\text{min}} \), much less than the parallel work of \( P \times W_{\text{min}} \). But the parallel task would still complete at the same clock-time. In the best case (for parallel advantage), the serial algorithm finds the solution on the last branch; this would give us a best case parallel efficiency greater than 2. A comparison of parallel and serial short-cutting is shown in figure 9.1.

Short-cutting can have additional beneficial effects on performance in that it improves load balance, since processes that are interrupted by a short-cut are those that have extra work as compared to the short-cutting process. This will be seen in experimental results, below.

A conceptually similar system called nagging has been proposed by Segre and Sturgill [SEGRE 97], in which parallel processes suggest alternatives to a main search process and help save it from doing extra work. The method differs from short-cutting in that there is a main process and a hierarchy of nagging processes (nagging processes can themselves receive suggestions from other naggers). In short-cutting, all processes have equal standing and the solution may be reached by any one of them. Nagging applies where no, or minimal, data transfer is required between processes, whereas a short-cutting process may need to share substantial amounts of data with others.
Figure 9.1: Comparison of serial and parallel short-cutting. Figure represents a specific run in which only process P3 is able to shortcut.
9.2 Short-cut implementation

In short-cutting, each process has some local criterion to decide when the result it has calculated is good enough to interrupt the work of other processes. This requires, therefore, a call to the run time system placed in the appropriate point (or points) of the program by the programmer. A process that is asserting a short-cut would then continue to a section of the program where something is done with the calculated result; let us call this the result-process section. Normally, such a section would include code to select the best result to use from all processes, probably through a reduction operation.

A short-cutting process would not need to select the best result; it presumes that its local result is the best. It synchronizes with the processes that received the short-cut interrupt, and sends them the data it has computed. The short-cutted processes can also skip the determination of which process has the best results, and use data from the short-cutting process to replace their own unfinished computation results.

A process that receives a short-cut might have finished calculating its own local result and already be in the result-process section (it might not have a good enough result to assert a short-cut itself), or be in some computational section before reaching a result. In either case, it should accept the short-cut and use data from the short-cutting process. Even if it has finished calculating its own result, we can presume that the result from the short-cutting process is better; otherwise the local process would itself have asserted a short-cut. The short-cutted process should jump to the beginning of the result-process section, where it will receive data from the short-cutting process to replace the local data.

It can happen that more than one process may decide to assert a short-cut; and in an asynchronous computation, the order in which these short-cut messages are received at short-cutted processes is undefined. Suppose processes P1 and P2 both assert short-cuts. It is possible that some processes will receive the short-cut from P1 first, while other processes receive the short-cut from P2 first. We also can not assume that the short-cut from P1 will interrupt P2 before it asserts a short-cut, or
vice versa. So the question is, which short-cut do we accept at each process (recall that accepting a short-cut implies using data from the short-cutting process)?

The simplest possibility is not to skip the reduction that determines which process has the best result in the result-process section. A short-cut would simply cause an early jump to this section in short-cutted processes, and the comparison of results everywhere would select the best one from multiple short-cutting processes (it would have to be one of the results at a short-cutting process, because if any process had a better result, it too would have asserted a short-cut). The disadvantage is that in cases where we have a single short-cutting process we are performing an unnecessary and probably expensive reduction operation.

Another possibility is for multiple short-cuts to define multiple MIPS. For example, processes that receive a short-cut from P1 first for one MIPS and ignore the short-cut message from P2, and similarly for processes that receive the message from P2 first. Each MIPS would execute separately on the result-process section and merge at the end of that section.

It does not seem to us that we can choose in general between these options. Which is better may depend on properties of the particular computation being performed. There may also be other possibilities. For example, if we are solving a problem that has more than one solution, it may be desirable for MIPS that have been formed by short-cuts not to merge, but follow separate computational paths.

In any case, since a short-cut causes processes to exit an MIPS early, and may lead to formation of more than one new MIPS, we regard the code in the result-process section as defining a new MIPS in the CFG. The first statement of result-process would then be an MIPS leader and contain code to increment the MIPS level. The last statement of result-process would be an MIPS trailer and contain code to decrement the level (see chapter 6).

SOS presently provides support for both of the above options. This is done through the SOSshortCutTest function and the SOSshortCutStart function. The Test function returns the number of the short-cutting process and this number can be provided to the Start function to define an MIPS of processes that have been short-cutted by
a particular process. *SOSshortCutStart* also transmits data from the short-cutting process to the other processes in the set. A programmer can choose to not use *SOSshortCutStart*, and instead of a special short-cut code section, simply cause processes to jump to the end of the computation section on receiving a short-cut. If this is done, short-cutted processes would still do less work, but the program would then always perform the evaluation of which process has the best value, as if the short-cut had not occurred.

9.3 Interruptible Communications

Short-cutting codes are applicable to some types of irregular problems, and require interruption of computation as a result of events which may occur non-deterministically and asynchronously at a different process or processes. We have argued in the introduction and in chapter 8 that asynchrony in irregular computation can introduce delays that are of the same order as the computation time. If this is indeed the case, then the probability that a short-cut signal will arrive at a process while it is in a state of uncompleted communication with another process is of the same order as the probability that such a signal will interrupt a pure computation stage.

If we treat communications as indivisible, then we would have to wait until the communication completes before we can interrupt the communicating process, which could incur delays of the same order as the computation time. The possible gains from short-cutting are of the order of the computation time, since a short-cut is an interrupt that reduces computational work at some processes to a fraction of what it would have otherwise been (see section 9.1). Accepting these delays could negate advantages to be gained from short-cutting. Therefore we must be able to interrupt communications actions. The question then is, how and when to interrupt?

We may first consider where not to interrupt: specifically in the communication channel itself. The channel is likely to be a wire, a network connection, some physical medium that transmits a signal. Special hardware would in any case be required to interrupt a message in transit. Since multiple messages may be in transit, and it
may not be correct to interrupt all of them, this special hardware would also have
to be able to read and identify messages. But this would result in delays which
would impact the much more common case of messages that are not interrupted.
Furthermore, the actual transit delay is likely to be much less than the time spent
waiting in a buffer or waiting for a clear-to-send (CS) signal. We therefore choose to
interrupt communication only at the sender or receiver, and not in transit. This has
the advantage that no special hardware is required; interrupt signals can be carried
by messages.

How to interrupt would depend on whether we are using buffered or unbuffered
messaging. We will briefly note buffered messaging, but consider mainly the type
of unbuffered messaging we have implemented for overlapping communication. Note
that the interrupting short-cut signal is processed separately by sender and receiver.
To understand how to interrupt we refer to our discussion in the introduction (section
1.3.2) of communication in general. We will see that how and when to interrupt
depends on what we are using as the key to interpret the message.

Case 1: Buffered messaging with tags

Here the message key is a pair of tags, one at the sender and one at the receiver. The
receiver matches its tag with the sender's tag, which is attached to the message to
select the correct one out of possibly multiple messages in a buffer. Sender does the
following:

1. set tag-sender

2. send-message with tag-sender

There is very little waiting at the sender, since send-message could (and normally
would) occur immediately after set-tag. If the short-cut signal arrives before send-
message, then do not send. Otherwise, the sender cannot do anything about the
message, it simply proceeds to execute the short-cut code.

At the receiver, we must:
1. set tag-receiver

2. look for tag-sender == tag-receiver in message buffer:

3. repeat 2 if not found

4. read message

If the send was aborted, then the receiver will either not have set *tag-receiver*, or be looping in step 2 (in the overlap case it could be executing between checks for message arrival). In either case, the receiver, upon getting the short-cut, knows that it should stop waiting for the message and proceed to the short-cut code.

If the send was not aborted, and the short-cut arrives during or after step 3, then the communication completes, and the receiver executes the short-cut code (which may tell it to ignore the results of this message).

A problem can occur if the short-cut arrives at the sender too late to halt the message, but arrives at the receiver while it is still in step 2. In this case, the receiver will abort the communication and jump to the short-cut code, then the message will arrive and will be in the receiver’s buffer; but the receiver is no longer looking for that particular tag and so does not clear the buffer. This has the potential to cause a later error if the buffer fills and blocks communication because there are dead messages in it, or if the tag is reused later during processing, resulting in two messages in the buffer with the same tag and the possibility of getting the wrong one.

Simply flushing the receive buffer on getting a short-cut signal is not correct. Consider a process P1 which short-cuts processes P2 and P3, and suppose that after the short-cut P3 has to send to P2. It is possible in asynchronous processes that the short-cut arrives at P3 first, and then it transmits to P2, and that the message P3->P2 arrives at P2 before the short-cut arrives at P2. In such a case (perhaps not likely, but possible!) P2 will contain in its buffer, when it receives the short-cut, a message that corresponds to logic after the short-cut, that it should not ignore. Some method must be used that separates message tags before and after a short-cut, so that a short-cuted process (and for that matter, even the short-cutting process)
can flush messages that logically occurred before the short-cut and keep messages that occurred logically after it, even though they may arrive at arbitrarily varying physical times. (One possibility - use logical time stamps that match lines of code in the program source on messages. Then the short-cutted process could simply discard any messages with a time stamp later than the short-cut but not matching lines in the code executed after short-cut).

Case 2: Unbuffered messages with clear-to-send protocol

The key here is the CS (clear to send) signal from the receiver, together with the shared context of the source code at both processes which allows identification of which send matches which receive. At the sender:

1. look for key (CS) from receiver:
2. repeat 1 if not found
3. send message

If the short-cut arrives before or during step 1, do not send the message and continue execution at the short-cut code; otherwise the message gets sent. The likely case is that a short-cut would arrive while the sender is looping waiting for CS, since this is the only stage that takes significant time.

The receiver must:

1. send CS
2. look for message:
3. repeat 2 if not found
4. read message
If a short-cut arrives before step 1, do not send the key. If short-cut arrives during step 2, abort the communication, stop looking for the message and jump to the interrupt code. Otherwise read the message and then go to the short-cut code.

Two problems can occur. Consider process $P_1$ sending to $P_2$. Assuming that it is equally likely that $P_1$ or $P_2$ is faster, then it is equally likely that $P_1$ is waiting to receive a key or $P_2$ is waiting to receive a message when the interrupt arrives. In the first case, it is possible that $P_2$ sends $CS$, and that $P_1$ gets the short-cut before receiving $CS$. Then $P_1$ must know to ignore that particular $CS$, but it can not just flush the next $CS$ from $P_2$ because it is possible that $P_1$ needs to send to $P_2$ again after the short-cut, and that the $CS$ signal corresponds to the second send. In the second case, $P_2$ has already sent $CS$ and is waiting for a message from $P_1$. Since $P_1$ may have received a short-cut before receiving $CS$ and responding, $P_2$ can not stay in the loop, because it would deadlock in that case. Suppose $P_1$ has to send to $P_2$ again after the short-cut. $P_2$ can not simply ignore the next message from $P_1$ because it might be the correct message after the short-cut, but it can not simply accept it either, because it might be an old, pre-short-cut message if $P_1$ did not get short-cut before sending.

Essentially, the $CS$ - message protocol is an alternating sequence of $CS$ from one process setting a flag at the other, and the responding message resetting it. Short-cutting interrupts the alternation, and depending on when it happens some form of (at least logical) synchronization is needed to reset the sequence.

Case 2, the unbuffered protocol, appears harder to handle than the buffered case 1. Nevertheless, we prefer it for two reasons: Firstly, it minimizes the amount of communication, since if possible it blocks messages before they are sent. In the buffered case there may be multiple messages which will have to be flushed. handling those messages wastes time and resources. Of course, the unbuffered protocol has an additional message load, but these are very short messages which take little bandwidth and are consumed immediately on arrival. Secondly, we see in our consideration of short-cutting that some form of synchronization will in any case be necessary after a short-cut, since more than one process may short-cut, since multiple MIPS may
form which must be identified. It is possible to use this synchronization to also communicate whatever information is needed to use or ignore any pending messages; if this is the case then no additional communication would be required to clean up the messaging after a short-cut.

One possibility is to attach the MIPS level to every status message. As described in the section on run time support for MIPS 7.1.2, MIPS are maintained in a single static array with one entry per process. When an MIPS is formed, processes increment their own MIPS level and the levels of other processes in the same MIPS (as they become known). When a process reaches the end of an MIPS section, it decrements its own level, and also all matching MIPS levels in its array. Since a short-cut acts to form one or more MIPS, we increment the MIPS level on entry to the short-cut code and decrement it on exit.

9.3.1 Correctness and safety

Correctness of a short-cutting code will need to be demonstrated on a case by case basis. We here discuss the more limited goal of avoiding execution errors that will prevent the program from completing its execution.

In a messaging system (eg. MPI) it is safe to receive buffered messages even if not expecting them, but it is an error to receive an unexpected unbuffered message. So, in a send-receive situation we have the following: the sender process may wait for CS, then send upon receipt of the CS. If it receives a short-cut interrupt while it is waiting for CS, it does not send the message. It is not an error to receive a subsequent CS even so, and it will know that it is a CS prior to the interrupt if its MIPS number is less than the MIPS number after the interrupt. The problem is on the receiver side, since it might be waiting for a message. If a receiver is interrupted before sending CS or after receiving the message, there is no problem - either communication does not happen at all (and if the sender was waiting for CS this will certainly be the case), or communication is completed.
However, it is not safe for the receiver to wait for a message even though it is interrupted, because it might deadlock - but it is equally unsafe for a receiver having sent CS to not wait for a message - because if the CS reached the sender before it was interrupted, then an unbuffered message is on the way and must be received or else we will have an error condition. We can solve this problem by having the receiver always wait for the message, even if it receives an interrupt, before continuing to the interrupt section.

The sender, will always jump to the interrupt section upon receiving the short-cut. If it later finds a CS in its buffer, it can tell by the MIPS level if the CS was before or after the interrupt. If after, it knows that this corresponds to a new send it must execute in the short-cut code section. If the CS is from before the interrupt, then it can send a null message back; this will take little time and not affect data at the receiver, but it will get the receiver out of its wait loop. Since, in the absence of a short-cut, CS and message are alternating pairs, it is not necessary to keep any additional information.

Nested short-cut interrupts, if they occur, can be dealt with similarly. A short-cut interrupt is a status message, so we also attach the MIPS number to it. If we get two short-cuts from different nodes, we can distinguish if they correspond to the same short-cut logic (in which case we respond by joining one MIPS and declining the other) or to an interrupt that follows the one we are responding to, in which case we jump to a different interrupt code section. (Recall that short-cut interrupts and interrupt checks are preprogrammed - that is, if we could have a nested interrupt it is because that is how we programmed it, and we will have both a second short-cut code section and the appropriate call that will check for it and send control there).
CHAPTER 10
SOS PERFORMANCE

We describe two experiments; one is a test of broadcast times in a synthetic benchmark, the second is a test based on a standard function solution algorithm. A comparison is made between time to completion using raw MPI calls, and using the SOS library calls implemented on top of MPI. Although SOS calls incur more overhead than direct calls to MPI, they successfully reduce synchronization delays. Programs using SOS calls exhibit substantial improvement in time to completion, as compared to raw MPI, in cases where the computation time is greater than the communication costs and when there are not more processes than physical processors. Overlapping and short-cutting were introduced in [GOMEZ 98], where we showed results from a precursor of the present SOS system. The results described here use the current system, and are more recent than [GOMEZ 98].

10.1 Hardware

All programs were run on the Avalon cluster at the Computer Science department in the University of Chicago. The cluster has 12 nodes, each of which has dual Pentium II processors of 400 or 450 MHz or dual Pentium III 500 MHz processors, 512 MB or 1 GB of RAM, and runs a Linux 2.2 kernel. Communications in these tests were over 100MB/s switched Ethernet, with a single Ethernet card on each dual processor node. The Ethernet cards do not have separate communication processors, so overlap gains are due solely to the protocol taking advantage of difference in run times between processes running in parallel. The SOS run time system was set to interrupt the computation and check if it needed to perform communication every 2000 microseconds.
If communication activity was necessary, the SOS system would interrupt the computation and carry out all communications it could perform according to its protocols before returning.

10.2 Experiments

10.2.1 Broadcast

Broadcast experiments were done with an artificial program constructed to use overlapping to advantage, although loosely modeled on a common situation in scientific processing. Our test program was written in Pfortran for convenience in initialization; actual communication was using direct calls to MPICH broadcast instructions, or to the SOS function library. SOS implements the overlapping protocol on top of calls to MPICH point to point communication instructions.

Program outline:

```plaintext
RTB.PF:

loop 0

- set work for each node
- select root node for broadcast
- start timer

loop 1

- compute X array (long computation)

end loop 1

- broadcast X from root processor

loop 2
```
• compute Y array (short computation, 1/50 of work in loop 1)

end loop 2

• broadcast Y array from root processor

• get time

• print results: X, Y, time

end loop 0

X and Y are both 100x100 arrays of double precision real numbers, each requiring 80Kb of storage.

Timing was done for ten iterations of loop 0, adjusting the root node and the work per node according to a control file input previously. The work per node was adjusted so that the total amount of work done on each iteration of loop 0 was the same, but the distribution between nodes varied so that some nodes had 1.5 or 2 times as much work as others. The largest amount of work done by any one node was also held constant; if broadcast were a synchronizing operation this should have the effect of forcing the iteration time to be at least equal to the time taken to do this maximum work.

On some iterations the root node was one of the nodes that did least work, on others the node that did most work, and on others it was a node that did an intermediate amount of work.

There is no dependence between X and Y; the overlapping protocol was set to force all communications to complete before the print results statement, but otherwise allowed the time between the broadcast statement and the print statement for communication to happen.

The program simulates a situation in which some large computation must be done and results from one node shared with others, and then some other much shorter computation must be done to determine which node is going to share its results in the next iteration.
10.2.2 Scientific application

The test program was written in Fortran77 for serial processing. It was parallelized using Pfortran and MPI, and a version without communications was used for speed benchmarks running on a single node. MPI collective broadcast and reduce calls were used in the MPI version. All MPI and Pfortran specific communication in the computation loop of the program was then replaced with SOS calls, including SOS broadcast and reduction calls.

We chose to implement a real algorithm in the belief that it should be more indicative of real world performance than an artificial benchmark program, which we could write to respond arbitrarily well to our overlap synchronization strategy. We picked a multidimensional function minimization technique called the Downhill Simplex Method in Multi-dimensions for use in our test (fig: 10.1).

We parallelized by projecting through all faces of the simplex at the same time rather than just through a single face: speedup was obtained through convergence in fewer steps. The parallelism and branching factor in this case is limited to the number of dimensions in the problem. The problem is irregular because the evaluation at each node is different.

We modified the algorithm by short-cutting as follows: If at any stage S1, S2 or S3 we have a better simplex than the original (possibly by some adjustable criterion of enough better) then go directly to C, the convergence test, interrupting all the other parallel tasks. Evaluations of “good” or “very good” are based on heuristic criteria.

We then have:

- Input: simplex; Si: parallel stages at each vertex

1. Project through opposite base; if good goto C

2. S1: If good project farther; if very good goto C

3. S2: If nothing worked contract simplex

4. S3: Compare all nodes and pick best simplex
5. C: halt parallel tasks

6. If simplex has converged stop; else go back to Input.

Potentially at each stage some process could short-cut the computation after doing only 1/3 the work, at P1. Therefore it is possible that the parallel program will end up doing less work than a serial equivalent.

We have described previous version of this program in [GOMEZ 98]. In those tests, the combination of short-cutting and overlapping yielded an advantage on the order of 5%. That system used a first implementation in Fortran of only the broadcast function of SOS, used polling instead of interrupts, and did not implement the out of order communication features of SOS.

10.3 Results

10.3.1 Broadcast comparison

The graph shows the results of runs using 10 processors on 5 computers connected through 100 MB/s Ethernet, runs using 8 processors on 4 of those computers, and runs using 16 processors on 8 computers, in which three of the computers were connected through 10 MB/s Ethernet.

We display runs on 8, 10 and 16 processors in figure 10.2. The first 5 computers, containing processors 1-10, are connected via 100Mb/s Ethernet; four of them are dual 450MHz machines and one is a dual 400 MHz. The last three computers (used only on the 16 processor runs), containing the highest numbered 6 processors, are connected via 10 Mb/s Ethernet, and are also 400 MHz machines; this makes communication and average run times on these machines slower than on the first four or five computers. The 20 processor runs used two additional computers with dual 400 MHz PII processors which were outside the Avalon cluster and connected through a separate switch.

The times for MPI and SOS are average run times per process for ten runs, varying load and with the root node cycling through processes 0-4. The “none-max” times
Figure 10.1: Example simplex in three dimensions.

Algorithm

- Evaluate function $F(\text{vertex})$ at all vertices, find worst vertex ($\text{max}$) and best vertex ($\text{min}$).
- Project a line from $\text{max}$ through the center of the opposite face.
- Find points $A$, $B$, $C$ on that line, at distances $\alpha H$, $\beta H$ and $.5H$, distance measured from $\text{max}$, $H$ is the distance from $\text{max}$ to the opposite face.

If $F(A) < F(\text{min})$ then

- if $F(B) < F(A)$ replace $\text{max}$ with $A$
- else replace with $B$

else

- If $F(C) < F(\text{min})$ then replace $\text{max}$ with $C$
- else contract whole simplex around $C$

Repeat until $F(\text{max}) - F(\text{min}) <$ convergence criterion
Figure 10.2: Comparison of SOS and MPI performance.
Labels (sos-N, mpi-N) indicate line that corresponds to runs using sos broadcast
and mpi broadcast on N physical processors. Times for the equivalent run without
communication are marked non-N.
are the maximum of the average process times without communication at any of the nodes for runs on 8, 10 and 16 physical processors and the given number of processes. Note that, since number of processes on both 8 and 10 processors range up to 20, some nodes will have 2 or three processes timesharing on them. However, the work will not increase by a simple factor because it is not evenly distributed across processes. The “none-max” times should be indicative of the lowest possible time for completion of a broadcast at all nodes. Broadcast time at a specific node may be less than this depending on that node’s position in the collective communication; to the extent that communication costs are exposed, broadcast times at specific nodes may also be more than computation times.

10.3.2 Scientific application:

Program was run twice for each table entry, results averaged and rounded. Time variation between runs was of the order of 1%. Flops is number of floating point operations per function call. Time in seconds is time to solution (using the MPI time function), and "funct." calls are total number of function calls to solution at all processes. They were adjusted by enclosing the function in a loop and repeating the same computation on each call. Total amount of data transferred between processes is the same for all runs.

MPI times are with all communications in the computation loop performed with direct calls to MPI functions. SOS times replace all the MPI calls with equivalent SOS calls. short-cut times add short-cutting logic and short-cut calls to the SOS version of the program.

Times in the first row are an upper bound on raw communication cost and general program overhead; the computation time was not significant (tests not tabulated with a lower number of floating point operations yield essentially the same times). SOS, MPI and SOS with short-cutting run times are plotted in figure 10.3 up to the time at which SOS with short-cutting overtakes the performance of SOS without short-cutting.
Figure 10.3: Timing data for scientific application. Shows greater cost of Overlapping and Short-cutting with low computational load, and better performance as synchronization costs are absorbed when computational load is greater. Data is displayed up to the point where Short-cutting overtakes pure Overlapping.
Table 10.1: Scientific program: 8 processes

<table>
<thead>
<tr>
<th>Flops</th>
<th>MPI sec.</th>
<th>SOS sec.</th>
<th>Calls</th>
<th>Short-cut sec.</th>
<th>Calls</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500</td>
<td>2</td>
<td>5.3</td>
<td>10196</td>
<td>6.4</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>3\times10^4</td>
<td>9.8</td>
<td>7.9</td>
<td>10196</td>
<td>8.9</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>3\times10^5</td>
<td>17.7</td>
<td>10.9</td>
<td>10196</td>
<td>11.9</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>7.5\times10^5</td>
<td>41.2</td>
<td>20.1</td>
<td>10196</td>
<td>20.5</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>1.5\times10^6</td>
<td>80.4</td>
<td>35.1</td>
<td>10196</td>
<td>34.0</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>3\times10^6</td>
<td>157</td>
<td>65.3</td>
<td>10196</td>
<td>63.1</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>4.5\times10^6</td>
<td>227</td>
<td>95</td>
<td>10196</td>
<td>91</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>7.5\times10^6</td>
<td>386</td>
<td>153</td>
<td>10196</td>
<td>145</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>1.5\times10^7</td>
<td>770</td>
<td>298</td>
<td>10196</td>
<td>281</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>1.5\times10^7</td>
<td>744</td>
<td>295</td>
<td>10196</td>
<td>281</td>
<td>10096</td>
<td>4</td>
</tr>
<tr>
<td>3\times10^7</td>
<td>1537</td>
<td>588</td>
<td>10196</td>
<td>555</td>
<td>10096</td>
<td>8</td>
</tr>
<tr>
<td>3\times10^7</td>
<td>1486</td>
<td>582</td>
<td>10196</td>
<td>558</td>
<td>10096</td>
<td>4</td>
</tr>
</tbody>
</table>

The same program was run on 13 processors (this was the number of available dual processor Pentium II and III computers, connected on 100 MBit/s Ethernet). As in the previous case, only a single process was run on each dual processor machine. This left one processor free for other jobs, which continued to run on our cluster while these tests were performed. Timings are not directly comparable to the 8 processor case, because we were solving a twelve dimensional function in this case. One set of calibration runs were done with 230 floating point operations per function call, to get an idea of raw communications time. A second set of runs was done with 9,200,000 floating point operations per function call, accomplished as before by enclosing the function in a loop and evaluating it repeatedly.

In this set of runs, the MPI code was instrumented to find the average time spent on each of the communication statements and on the function evaluation. Timing communication statements is not meaningful for SOS code (with or without short-cutting), because communication statements return almost immediately, but the actual data transfer is mixed with program execution. For this reason only the function evaluation was instrumented in the SOS codes.
Table 10.2: Scientific program: 13 processes

<table>
<thead>
<tr>
<th></th>
<th>MPI 230</th>
<th>MPI 9.2e6</th>
<th>SOS 230</th>
<th>SOS 9.2e6</th>
<th>Shrt 230</th>
<th>Shrt 9.2e6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls</td>
<td>54688</td>
<td>54688</td>
<td>54688</td>
<td>54688</td>
<td>46798</td>
<td>46798</td>
</tr>
<tr>
<td>Time</td>
<td>57</td>
<td>743</td>
<td>89</td>
<td>728</td>
<td>80</td>
<td>625</td>
</tr>
<tr>
<td>Bcast 1</td>
<td>15</td>
<td></td>
<td>99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bcast 2</td>
<td>8.3</td>
<td></td>
<td>4.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bcast 3</td>
<td>.7</td>
<td></td>
<td>.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bcast 4</td>
<td>5.5</td>
<td></td>
<td>4.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduct</td>
<td>27</td>
<td></td>
<td>254</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eval.</td>
<td>.004</td>
<td>378</td>
<td>.004</td>
<td>389</td>
<td>.005</td>
<td>331</td>
</tr>
<tr>
<td>Subtot</td>
<td>56.5</td>
<td>740</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The program includes the following communication operations:

1. Broadcast: P, 104 bytes
2. Broadcast: P, 1248 bytes
3. Broadcast: NODE, 4 bytes
4. Broadcast: RTOL, 8 bytes
5. Reduction: Find minimum and node at which it occurs, uses arrays A and B, 16 bytes (user defined function).

Results are shown in the following table. All times are in seconds. The numbers on the column labels refer to number of floating point evaluations per function call. Times are for single runs, but are typical of repeated test runs for similar number of floating point operations. The runs with 230 floating point operations per function call give an idea of the basic communication cost of the program.

In the code, broadcast 1 immediately precedes broadcast 2, and the reduction operation closely precedes broadcasts 3 and 4. There are significant amounts of code executed before the reduction in particular, and also before broadcast 1. We see
the effect of synchronization waits in the significantly longer times for the reduction 
operation and for broadcast 1 than for all other communications in the program. 
Particularly note that broadcast 2 takes less time than broadcast 1, even though it 
transfers twelve times more data. The faster times for broadcasts 2 and 4 in the 
second and longer MPI run are probably due to varying computational loads on our 
cluster between these runs. Note that the subtotal for communications and function 
evaluation is almost equal to the total run time, indicating that this in fact the only 
part of the code that is taking significant time.

The SOS code in this case is only slightly faster than the MPI code. This is unlike 
the eight process case; however in the first experiment it was feasible to take runs 
for times 100 to 1000 times greater than the basic communication cost, whereas here 
we are exploring a case where the longest run is only about 8 to 10 times the basic 
communication cost.

The SOS code with overlap only is about twenty seconds faster than the MPI code, 
but notice that the function evaluation, for the same number of function calls, actually 
takes eleven more seconds. This is consistent with the idea that SOS is interleaving 
communication with computation; since there is no special communication processor, 
this makes the computation take somewhat more time. That is, we are not actually 
overlapping computation with communication. The time to completion is still less, 
because the overlap protocol and run time system allows us to avoid some of the 
synchronization waits.

The short-cutting code is faster that either of the other two alternatives, mainly 
because it does some fifteen percent fewer function evaluations. However some of the 
speedup is because short-cutting also provides improved load balance, since slower 
processes tend to be short-cutted and so do less work. There may be some added 
speedup from more efficient overlapping, because processes that are more closely 
synchronized are less likely to force either senders or receivers to wait for the other to 
catch up. We see that something of the sort must be happening because the short-
cutting code is overall 103 seconds faster than the code that only uses overlapping, 
but the function evaluation portion is only 58 seconds faster.
10.3.3 Overall results:

In all cases where computation time is greater than communication time, the SOS versions of the program ran at least as fast as the MPI version. Where computation is substantially greater than communication, or where the number of nodes increased beyond 8 (for the broadcast comparison), the SOS program outperformed the MPI program, reaching completion up to 3 times faster.

Where computation times are not significant, MPI shows substantially less communication overhead. MPI also has the advantage in cases where there are more processes than physical processors.

10.4 Analysis

A design goal of overlapping is to hide synchronization delays in a parallel execution. These delays can be due to algorithmic reasons, or due to data dependent computation, or to differences in processor speeds or other programs running on some processors. Synchronization delays are due to differences in finish times of stages of different processes; these may be of the same order as the computation time, and may be the major cause of communication delay in a parallel execution.

Running the overlap protocol is not free. There are time costs in communicating status information and keeping track of and executing multiple pending communications actions. There are space costs in the data structures needed to support the protocol finite state machines and the pending communications. However, these costs are of the order of the communications costs in the program; these costs may become small compared to computation and synchronization as the problem size is scaled up. Therefore even without special communication hardware or a separate communication processor (that is, even where the communication and overlapping work takes cycles away from computation), overlapping can give an overall advantage (since the time it can save is greater than its direct cost).
10.4.1 Broadcast

We note that SOS total execution times are very close to the computation times for all cases in which the number of processes is less than or equal to the number of physical processors, even in the case of 16 processes and processors that includes part of the slow network. We conclude that SOS is effectively hiding the communication costs in every case in which we are running each process on a separate physical processor. One way to see this is to note that the SOS curves are essentially flat until the number of physical processors is reached, and only then do they turn up.

MPI times are essentially the same as SOS times (and not significantly different from computation times) for four processes. MPI times rise gradually as the number of processes is increased, becoming about 10-15% greater than SOS for 8 or more processes as long as we have at least one physical processor per process. Once we have more processes than processors, MPI broadcast is faster than SOS broadcast in every case.

We feel that the observation that SOS times are equal to the computation times when we have only one process per processor is reliable because this relationship hold for 8, 10 and 16 processor cases, and also for a 6 processor run not graphed. In every case the SOS times were approximately the same as the pure computation times until the point where we needed to assign more than one process per processor, and at that point the curve turned up, away from the pure computation times.

10.4.2 More processes than processors

SOS is designed to hide or tolerate synchronization delays; in this particular problem it succeeds in doing this, hiding the communication time almost completely. MPICH implementation is designed for efficient collective communication, but makes no special attempt to hide latency or synchronization cost. Since a communication involving more processes is more complicated, and involves more steps (typically log(n)), MPI incurs increasing delays and therefore takes more time than SOS as the number of
processes increases. SOS times do not increase as rapidly because SOS both hides the communication costs and executes send-receive pairs opportunistically out of order.

If multiple processes in a single parallel execution are running on a single node, and these processes are not deadlocked, then there is no synchronization delay, since some process is always advancing the computation even if others must wait. In such a case, all the work that SOS does to prevent synchronization losses is wasted and simply increases the total computation time.

Since a broadcast is at least logically synchronizing, extra delays and work on one or a few nodes are likely to delay the whole computation; we believe this explains why SOS is significantly slower than MPI in cases where multiple processes must run on a single processor.

10.4.3 Scientific application

The comparison of execution times confirms the results of the broadcast comparisons in the more realistic setting of a real application. This includes reductions as well as broadcasts, and their location is dictated by the algorithmic requirements.

An earlier version of the same program was used in our previously reported results. At that time, we found a performance improvement on the order of 5\% when overlapping and short-cutting was applied. The current version improves on this by a factor of about 50. We believe the extremely large improvement is due to the following:

1. The current SOS library uses a more efficient and frequent interrupt scheme to handle run time communications, whereas the old version used polling at selected points in the program. Latency is greatly reduced and overlap can be better exploited due to this.

2. The internal logic of the SOS run time system (that allows out of order scheduling of communications when they become possible) has been greatly speeded up in the current library, and is finer grained. The original code in Fortran has been replaced with C.
3. The implementation of the overlap protocols has been improved and allows a greater amount of overlap to be exploited.

4. Better algorithms are used for broadcast and reduction in the current system. The original tests were performed using direct sends between every process involved; the current system uses collective algorithms.
CHAPTER 11
CONCLUSIONS AND CONTINUING WORK

We have that an MIPS is an entity in a parallel SPMD execution that can be efficiently and unambiguously identified by a combination of static analysis and a run time system. The static analysis identifies where a particular MIPS will execute on the CFG, and the run time system identifies the particular processes that are members of the MIPS process set in a particular execution of the program.

The identification of the MIPS is sufficient to avoid deadlock when communication occurs in sections of code that are selected for execution at only some of the total number of processes in a parallel run. Together with fusion communications, MIPS ensure determinism and provide a definite semantics of memory access for SPMD programs. For one-sided communications (shared memory), MIPS analysis provides a way of checking for correctness and freedom from deadlock of synchronizations required to ensure determinism.

The resulting execution model in which each process may execute different sections of code during execution naturally leads to synchronization delays when such processes merge to execute together on some later section of code. We have shown an overlapping technique that focuses on absorbing these synchronization waits, and which may also apply in the conventional sense of overlapping communication and computation.

The novel assignment of a separate finite state machine to control communication for each variable allows fine grained control of progress at each process, allowing decisions to continue or wait to depend in detail on the data dependences for each variable involved in communication. The run time system allows an opportunistic scheduling of communications according to which processes are ready to participate in communications. Although this means that the order of communication is non-deterministic
and will be different on repeated execution of the same program, determinism of the program execution is demonstrated.

Static analysis of the program code to determine where to insert calls to the run time system to define MIPS and to enforce the restrictions of the overlapping code can be done with straightforward extensions to standard compilation techniques, which are described. Therefore the required analysis can be done without excessive cost in a suitable compiler.

The run time system requires additional communication during MIPS splits and in passing the status messages used by the overlap protocol. These communications follow the same pattern of collective communication used by broadcast and reduction operations, therefore will scale in the same way. Experiments show the overhead required by this additional communication is small compared to the total communication and computation cost of our test programs.

Finally, we have shown that short-cutting is a technique that can be beneficial for some codes and has the possibility of performing less work in parallel execution than in a comparable serial execution of the same algorithm.

11.1 Continuing work

11.1.1 Compiler support

In the experiments carried out so far, program analysis has been done by hand. Calls to the SOS system have been manually inserted in the program following the algorithms we have described. We propose to do this analysis and call insertion automatically in the Planguage compilers. Communication in Planguage is by fusion statements, which map to SOS communication calls which are also fusion statements, so the resulting compilation system will provide support for communication restricted to occur inside implicit program sets. This support is presently lacking in all the SPMD programming systems of which we are aware.

It may also be possible to design a less powerful system which could be built on top of a scripting language to perform a less complete analysis of program code than
a compiler would do. This is based on the observation that the analysis described in Chapter 5 is performed on the CFG, so any system that is capable of extracting a CFG from a program text can be used. Overlapping protocol calls can be inserted ahead of all references to variables being communicated, which can be identified in a lexical scan. Although such a system would not be able to perform as complete an analysis as a full compiler, it may be useful in that it may be applied more quickly to a larger variety of languages.

Our work on implicit process sets and overlapping has concentrated on identifying splits, merges and variables that appear in communication in existing program code, and then inserting suitable calls to the SOS library. Some optimization may also be useful. In the case of overlapping, optimizations similar to those used in systems that separate sends and receives (such as [SPLITC], or Ironman [CHAMBERLAIN 97]) should be applicable. In the case of implicit process sets, it may be possible to optimize to, for example, reduce the number of splits (which involve communication). Other optimizations should be investigated.

11.1.2 Hierarchical sets of processes

In some programs it may be possible to identify sets of MIPS in addition to identifying the individual process sets that make up each MIPS. This follows from the fact that MIPS are nested within MIPS. We note in section 7.1.2 that a split produces a set of MIPS that are immediate descendants of that split. This may be considered a set of MIPS. A further split in any of those MIPS creates another set nested within the original set. All these sets must merge when the original MIPS is reconstituted at the first complete merge for the original split. The short-cutting API at present recognizes these sets through a nesting level, which allows a short-cut message to go to all the processes in the current MIPS (nesting level 0), or by using a higher nesting level, to all processes that are descendants of a particular split of which the short-cutting process is also a descendant. It may be possible to do the same sort of thing in a standard fusion communication, by extending the set participating in
the communication to all descendants of a particular split; that is, all members of a particular MIPS.

11.1.3 Error handling

At present, communication errors are not handled in SOS, but are allowed to crash the computation. Evaluation of the membership function (def: 3.2.1) permits more graceful error handling. If the membership function evaluates to FALSE at any node, this fact may be propagated and allow aborting an individual communication without necessarily aborting the entire computation. Even if the computation is halted, membership function information may allow easier debugging.

11.1.4 SOS development

The run time system requires additional work. At present, the SOS library is implemented only on top of MPI, and supports only point to point, broadcast and all-reduce, the collective operations acting inside the current MIPS. This needs to be extended to include at least reductions at a single node, and possibly operations like shifts and multiple parallel point to point sends. Additionally, data structures need to be added to support merges after N-way splits that do not reconstitute the original MIPS.

SOS has only been tested up to 32 processes. Memory requirements to support up to several thousand processes can be estimated to be of the order of two megabytes per thousand processes, based on the data structures described in 7.1.2. Although this is not excessive by the standards of today's computer memory sizes it could be improved by, for example, storing protocol flags (one per process) in an array of bits instead of integers. Scalability needs to be tested up to large numbers of processes. SOS uses the same collective communication algorithms used in the Planguages, so we anticipate that scalability properties should be similar; nevertheless this needs to be verified.
11.1.5 Communication algorithms

The collective communication algorithms used in SOS are based on algorithms which are efficient for processes in which each collective communication is carried out by itself in order, not mixed with other communications, and without the possibility that parts of a collective transfer will be dynamically rescheduled. It is possible that some particular patterns of collective communication will be better, because for example they allow the system to better exploit dynamic scheduling, or reduce synchronizing effects. This has not been studied in the present work, but it should be a subject for continuing research.

The SOS communication queuing mechanism was designed to allow different priorities to be assigned to particular communications. This is not used in the present system, but its effect needs to be studied, possibly in the context of real-time communications.

11.1.6 Algorithms and computation

We have experienced difficulty in finding suitable test problems for SOS. We believe this is because the lack of support for implicit process sets in current systems, and the inefficiency with which such systems deal with computational imbalance, has led scientists to avoid codes which use implicit process sets, even if such codes could be scientifically advantageous.

Irregular computation appears to be a fruitful field for the application of MIPS techniques. Short-cutting may be applicable to various types of search algorithms (see for example problems such as described in [AIX 98]). Further investigation is needed, probably including application to multiple problems, to determine what to do in the case of multiple short-cuts. We propose to seek collaborations with computational scientists to develop applications that could be scientifically fruitful and that would be supported by SOS.
11.1.7 Implications for Computer Architecture

The overlapping protocol described in chapter 8 requires the assignment of multiple finite state machines, one for each variable that appears in a communication statement. Further exploration of this concept may suggest the need for additional FSMs controlling aspects of communication in parallel execution. Architectures designed to implement parallel computation may usefully include hardware support for FSMs.
CHAPTER 12
APPENDICES

12.1 SOS API

The following are functions of the programmer visible API for the SOS (MIPS, Overlapping and Short-cutting) systems. Functions are designed to be callable from Fortran. When calling these functions from C, the function name should be written in lower case only with a trailing under bar, and all parameters must be passed by address.

The current system is built on top of MPI (compiled and tested using MPICH 1.2). MPI must be initialized before calling any SOS functions, unless using the Planguages. SOS is compatible with Pfortran and PC when these are built with MPI communications; when used in this way the Pfortran and PC initialization is sufficient and no extra calls to MPI are required.

MIPS functions

MIPS support sets of processes implicitly formed by program logic. Processes in an MIPS may be identified by process number PN or by MIPS index SI. SI and PN are the same for the initial MIPS.

\[ \text{int } IOSstreamStart(\text{int } P) \]

Start an MIPS including all processes that have matching values of P. P must be a positive integer or 0 value that may be different at different nodes (for example TRUE=1 or FALSE=0 depending on data).
Increments the MIPS number of the calling process by 1. MIPS may be defined inside other MIPS.

\[ \text{int } \text{SOSstreamEnd}(\text{int } D) \]

Ends the current MIPS. If the current MIPS is the initial MIPS, does nothing.

Decrement the MIPS number by D. Normally there will be one streamEnd statement paired to each streamStart statement (like open and close parenthesis). However, if more than one MIPS has been opened in sequence so that there are several nested MIPS, and it is desirable to end more than one MIPS at the same point in the logic, D may be greater than one.

\[ \text{int } \text{SOSstreamLevel}(\text{int } L) \]

Returns the number/nesting level of ME in the parameter L. 0 means the initial MIPS.

Each process keeps a fixed array of MIPS levels, indexed by PN. When a process enters an MIPS, it increments its MIPS level by one, and increments the MIPS levels of other processes as it finds out that they have joined the same MIPS. Upon exiting an MIPS, a process decrements its MIPS level, and that of all other processes that have the same MIPS level. So in the MIPS array of a process, all processes that are in the same MIPS are marked at the same level, processes that were part of the MIPS that split to form the current MIPS are at a level equal to the current MIPS -1, and so forth.

\[ \text{int } \text{SOSstreamMember}(\text{int } PN, \text{int } R) \]

Returns R=1 if process number PN is in the same MIPS as THISNODE, R=0 if it is not. If the MIPS membership is not fully defined, is a synchronizing call.
\begin{verbatim}

int SOSme(int SI)

Returns the index SI within the current MIPS of the current process. If the composition of the MIPS is not completely known, this is a synchronizing call. If the current MIPS is the initial MIPS, returns THISNODE.

int SOSid(int PN, int SI)

Returns the index SI of process PN within the current MIPS. If the composition of the current MIPS is not completely known, this is a synchronizing call. If PN is outside the current MIPS, returns SOSme() /* Rationale: this allows writing deterministic node specific code inside an MIPS, by translating calls outside the MIPS to local access. Calls using raw PN would be nondeterministic, get value of requested item at current clock time */. Returns PN if current MIPS is the initial MIPS.

int SOSpn(int SI, int PN)

Returns the process number PN of MIPS member whose index in the current MIPS is SI. If SI is greater than the MIPS count, returns THISNODE. Synchronizing if MIPS not yet defined.

int SOSstreamCount(int N)

Returns the number of processes N in the current MIPS. If the current MIPS is the initial MIPS, returns NUMNODE. Synchronizing if MIPS not yet defined.

Overlap functions

Unless otherwise indicated, all process identifiers in the following refer to process number PN rather than MIPS index SI. All variable indices are those returned by
\end{verbatim}
SOSaddVar. Size is number of words for variable or array (on Intel/Linux systems, word = 32 bits)

For final version, the variable Count will be replaced by Offset,Count,Stagger. The assumption is that AddVar will be used to define an array, where Count gives the array size; then in order to pass part of the array it will be possible to specify O:S:C (as in Fortran 90 syntax). It may be beneficial to add API functions that omit Count, to allow more convenient passing of the entire array or of single variables; this may also be handled by a compiler or preprocessor that takes a higher level (such as P language syntax) into SOS calls.

\[ int \text{SOSpoint2point}(int \text{FromVar}, int \text{ToVar}, int \text{Size}, int \text{Offset}, int \text{SOSType}, int \text{FromP}, int \text{ToP}) \]

Declare a point to point communication of Size words of FromVar+Offset at process FromP to ToVar+Offset at process ToP. Returns TRUE if successful, FALSE on error.

\[ int \text{SOSbroadcast}(int \text{FromVar}, int \text{ToVar}, int \text{Size}, int \text{Offset}, int \text{SOSType}, int \text{FromP}) \]

Declare a broadcast of Size words of FromVar+Offset at process FromP to ToVar at all processes that are members of the same MIPS as FromP.

\[ int \text{SOSget}(int \text{FromVar}, int \text{ToVar}, int \text{FromP}, int \text{ToP}, int \text{Size}, int \text{Offset}) \]

Declare a read of Size words of FromVar+Offset at process ToP stored into ToVar+Offset at process ToP. Returns TRUE if successful, FALSE on error. One sided execution at process ToP, gets asynchronously from process FromP. Nondeterministic unless explicit synchronization is used.
\textit{int \texttt{SOSput}(int FromVar, int ToVar, int FromP, int ToP, int Size, int Offset)}

Declare a write of Size words of FromVar+Offset at process FromP to ToVar+Offset at process ToP. Returns TRUE if successful, FALSE on error. One sided execution at process FromP, puts asynchronously to process ToP. Nondeterministic unless explicit synchronization is used.

\textit{Reduction operations:}

These operations apply a binary operator or function to scalars or vectors distributed across all processes in a set, returning a single scalar or vector at all processes or a single process. SOS includes predefined operators for addition, multiplication, maximum and minimum (see SOSreduce.h), and users may write their own functions in either Fortran or C and call them in reduction operations. All functions used in this way must be initialized in SOS with SOSaddFun, and variables used in a reduction must be initialized with SOSaddVar.

\textit{int \texttt{SOSallReduce}(int FromVar, int ToVar, int Size, int Offset, int \textit{SOSType}, int FunP, int Sel)}

Declare a reduction operation using the subroutine pointed to by FunP. Let X(i) be an element of FromVar, and Y(i) be an element in ToVar. Let \( \oplus \) be an operator implemented in FunP. Then, Y(i) everywhere = X(i)@SI_0@X(i)@SI_1 \( \oplus \) ...@X(i)@SI_{\text{numstream}-1}; where numstream is the number of processes in the same MIPS as the current process. Sel is an additional parameter to the function; for example it could indicate the size of a vector to which the function is to apply.

FromVar and ToVar may be the same. If they are not the same, only ToVar will be modified.
int SOSpointReduce(int FromVar, int ToVar, int Size, int Offset, int SOSType, int FunP, int Sel, int Root)

Declare a reduction operation using the subroutine pointed to by FunP, with result at Root. Let X(i) be an element of FromVar, and Y(i) be an element in ToVar. Let @ be an operator implemented in FunP. Then, Y(i) everywhere = X(i) @ SI_0 @ X(i) @ SI_1 @ ... @ X(i) @ SI_{numstream-1}, where numstream is the number of processes in the same MIPS as the current process. Sel is an additional parameter to the function.

FromVar and ToVar may be the same. If they are not the same, only ToVar will be modified. ToVar at Root will contain the result of the reduction; ToVar at every other process is used for work space. Since the system may execute the reduction in a different order each time, the contents of ToVar everywhere except at Root are undefined.

Operations will be performed on the section of ToVar of Size words at Offset words from the base of ToVar

int SOSdoRH(int SOSvarIndex)

Var is used or read in the program, i.e. appears on the RH side of an equation. Variable indexed by SOSvarIndex should appear in a communication operation, otherwise does nothing. SOSdoRH should be placed in the program immediately before any statement in which the variable appears on the right-hand side of an equation, or is read from.

int SOSdoLH(int SOSvarIndex)

Var is written to or updated in the program, i.e. appears on the LH side of an equation. Variable indexed by SOSvarIndex should appear in a communication operation, otherwise does nothing. SOSdoLH should be placed in the program immediately before any statement in which the variable appears on the left-hand side of an equation, or is written to.
Short-cut functions

\[ \text{int \textit{SOSshortCut}}(\text{int \textit{SOSvarIndex}}, \text{int \textit{DStream}}) \]

Assert a short-cut at the current process, indicating that processes that are short-cutted will receive Var before starting to execute the short-cut section. short-cut is sent to all processes with streamLevel equal or greater to the current process SOSstreamLevel -DStream; this is the short-cut level. Normally DStream is equal to 1, because in a typical short-cut case several MIPS will be executing different code to get trial answers, and once a process in one MIPS short-cuts, we want to interrupt all the trials in the other MIPS also.

\[ \text{int \textit{SOSshortCutStart}}(\text{int PN}) \]

Join the short-cut set of process PN. Place at the start of the short-cut code section. This acts like a merge of all the processes at the short-cut level, immediately followed by a split according to how many different processes have asserted a short-cut.

\[ \text{int \textit{SOSshortCutEnd}}() \]

End a short-cut code section. Is a merge for all processes that were short-cutted, MIPS level is the short-cut level.

\[ \text{int \textit{SOSshortCutTest}}() \]

Returns the number PN of the short-cutting process if a short-cut has been received from another process or asserted by this one, else returns -1. Use to test whether short-cut has occurred and therefore should jump to short-cut code.

\[ \text{int \textit{SOSshortCutClear}}() \]

Clear short-cut, allowing new short-cuts to be received. Intended for use in cases where short-cut section is not used. Otherwise SOSshortCutEnd cleans the short-cut.
Interrupt system

Current run time system uses the timer interrupt to set the polling routine to check for communications activity. This occurs by default every 2000 microseconds.

\[ \text{SOSalarm()} \]

Start the timer. Must be done once at the start of the program to initialize the timer interrupt system, may be done after functions and variables are initialized. This is distinct from SOSinit() which must be done before any other SOS call.

\[ \text{setSOSalarm(int INTERVAL)} \]

Set timer interrupt interval, in microseconds. If not used, default is 2000.

System initialization and control

The SOS run time system is based on timer interrupts. The library is linked into each run time image, and the SOSpoll() routine is set to execute periodically. Default periodicity is every 2000 microseconds, but this may be modified during program run.

If not using an interrupt or thread based run time system, it is necessary to poll the SOS code periodically to check for messages and data. Current version uses interrupts, a poll call may still be used to force a check of pending communications.

\[ \text{SOSinit()} \]

Initializes the SOS system: sets up MIPS arrays, buffers, etc. System dependent. May also be software dependent; currently assumes that Pfortran and MPI have been initialized before calling SOSinit.


int setSOSalarm(int Interval)

Set interval in microseconds between interrupts that call the SOS run time system. Default is 2000 microseconds. Call may be repeated to adjust interrupt rate in different sections of the program.

int SOSalarm()

Start the SOS interrupt system. This should be done only after defining variables and functions to be called in reductions. If this is not done, explicit SOSpoll() calls must be inserted in the code.

SOSclear()

Clear SOS system queue and status arrays, does not reinitialize variables, functions or communications channel.

int SOSflush()

Poll SOS system. Does not return until all pending communications have terminated.

int SOSsync(int Dstream)

Barrier synchronization: blocks all processes in a set until all have reached and executed this statement. If Dstream is 0, process set is the current implicit process set. If Dstream is -1, this is a global barrier that involves all processes. If Dstream is >0, then affects all processes with streamLevel equal or greater to the current process SOSstreamLevel -Dstream (if this value is less than or equal to 0, then acts like a global barrier).
int Sosblock()

Block SOS interrupt system from interrupting running program. Normally not needed if SOSdoLH and SOSdoRH are correctly inserted, since system will block by itself before reaching a conflict.

int SOSunblock()

Unblock interrupt system after SOSblock.

int SOSpoll()

Poll SOS system. If the system is in a MUST receive or send state, does not return until the state has cleared (i.e., the data transfer has completed). Users should never see this behavior because SOSdoRH and SOSdoLH check the system, and do not return until any MUST data transfers are clear.

Variable and function initialization

In order to be able to use user-written subroutines in reduction operations, SOS needs function pointers to them and type info. SOS also needs information on variables to be used in communications.

int Sosaddvar(void *Buffer, long *Size, int *SOStype, int *Sosivar);

Buffer is a variable or array name. Size is the size in words of the variable or array (eg. on Intel based systems, a word is 32 bits = 4 bytes).

On the present MPI based version, SOStype is replaced by the corresponding MPI type.

This function returns Sosivar (index of variable), which is an integer that should be used to refer to the variable Buffer in SOS calls.
int SOSaddFun((void *Function()), int *SOSfunP);

In C, Function is a function pointer, as indicated. In Fortran, Function is the name of a function, with no parameters or parenthesis. The name must be declared as external in the program that executes SOSaddFun, even if the function is defined in the same file.

This function returns SOSfunP (pointer to function), which is an integer that should be used to refer to the function in SOS reduction calls.

**User-written function definitions template**

Operations to be performed in reductions should be binary operations; associative and commutative, since the system will execute them in arbitrary order, of the form A op B => C (for example, sum and multiplication.

The current version of the SOS library requires that user written functions to be used in reductions be expressed as subroutine calls, such that all parameters are passed by address (this is the Fortran standard) and the return value (or vector) is a parameter.

User written functions take two input data parameters and one output data parameter, plus one extra parameter which would normally be used to indicate a vector size if the input and output parameters are vectors. Both input parameters and the output parameter must be of the same type; if they are of different sizes then the user must take care that the function operates only on a subset of the data that will fit in all three buffers, and that its operation is limited to memory that is actually in the specified buffers.

The first input parameter takes the local value of the variable; the second input parameter will be filled with values passed from other processes, and the result will be placed in the output parameter. The extra parameter can provide control information; for example a vector size, or the position of a particular element of a vector.
Parameters:

Formal parameters are: \( \text{Op(In1, In2, Out, Extra)} \); Such that the operation described is: \( \text{In1 Op In2 } \rightarrow \text{Out} \).

All parameters must be passed by address, and Extra must be an integer. \( \text{In1, In2 and Out must be of the same type;} \) and must be defined to the SOS system with \( \text{SOSaddVar} \). Extra must be an integer.

System Implementation Notes

Communication completion:

A communication declared by any of the functions that move data between processes is queued on the SOS run time system and actually happens when the processes involved are able to perform it.

Actual data transfer may occur in different order from the declarations. To preserve the semantics of the program, SOS provides instructions \( \text{SOSdoRH(indexV)} \) and \( \text{SOSdoLH(indexV)} \) to tell the run time system when the program needs to read (SOSdoRH) or update (SOSdoLH) the variable indexed by indexV. If the affected communication has not happened when the SOSdoRH or SOSdoLH call is asserted, the call will block and wait for that communication to complete. (The run time system also checks for all other pending communication while blocked).

Pending sends for a variable will be forced to complete by SOSdoLH, so the previous value will be sent before an update. Pending receives will be forced to complete by either SOSdoRH (so the system can use the latest remote value) or SOSdoLH (so a pending receive will not overwrite a new value). Use of both together forces pending communication with respect to a particular variable to complete, but communication of other variables may remain in the queue.

To force completion of all pending communication, use \( \text{SOSflush()} \).
Mixing SOS calls with MPICH or Planguage functions:

This advice applies only to programmers wishing to mix SOS calls with direct MPI calls or Planguage fusion statements (statements with @ or \{\}). Tests of this nature have been performed with MPICH version 1.2.

When using MPICH as the communication layer, either by itself or underneath Planguages, it is necessary to block the SOS interrupt system before calling any MPI function, or before using the Planguage extensions to C or Fortran: @ or \{\}. For example, in a Fortran program, you should write:

\begin{verbatim}
call SOSblock()
(Pfortran or direct MPI call)
call SOSunblock()
\end{verbatim}

In a C program, the equivalent calls are sosblock\_() and sosunblock\_(). This is because MPICH (at least up to version 1.2) is not reentrant; if SOS interrupts an MPICH call and itself calls MPICH, results are unpredictable and generally fatal.

Planguages may include an SOS version of the libraries linked at compilation; in this case SOS and Planguage calls may be freely mixed, and sosblock and unblock calls are automatically generated as needed.

If Planguages and SOS are used with a version of MPI that is not MPICH, these precautions may or may not be needed.

Type and boundary checking:

SOS performs no checking of types, array boundaries or size of memory areas. You’re on your own.

Limitations on communication of arrays:

The present system controls communication protocols through finite state automata assigned to each variable entered into the system. It is possible to transfer or operate on a contiguous section of an array, but since the system is controlling communication on the basis of the entire array, it cannot at present overlap communication on two
different sections of the same array. Care should be taken to force completion of communication of any part of an entire array before attempting to communicate some different part of the same array.

A workaround is to define sections of the array as separate variables. The programmer should still make sure that communications of one section of an array complete before communications of any other part of an array that is defined so that there is an overlap in memory. It is, however, safe to treat two disjoint sections of an array that are defined as such to SOS as separate variables.

**Barriers:**

SOS communications (except for SOSput and SOSget) do not need explicit synchronization, they will block processes as needed to preserve communication semantics. It may be necessary nevertheless to synchronize processes for some other reason.

MPIBarrier (MPI) and pf_sync (Planguages) are both global barriers (depending on Planguage build options, they may be the same call). Neither is safe to use in the presence of implicit process sets (supported by MIPS), which may not include all the processes in a parallel execution. Additionally, MPIBarrier blocks execution of the SOS background communication process. If you must use either of these calls for a global synchronization, make sure to do an SOSflush first to force completion of all pending communications. It is preferable to instead use SOSsync.

SOS interface to application:

Application must:

- Declare variables to be used in communication
- Declare functions to be used in reductions
- Call communication statements to declare p-p send, broadcast, reduction
- Indicate to SOS when a communicated variable is read (RH)
- Indicate to SOS when a communicated variable is updated (LH)
• Indicate to SOS logic statements that split MIPS

• Indicate to SOS where MIPS merge occurs

Comment: broadcast and reduction both act on IPS (implicit process set), as defined by MIPS primitives. Only SOSput, SOSget, SOSshortCut and SOSsync allow communications outside the IPS of a process.

A communication statement in the program source code indicates that data is ready; LH and RH indicators on variable following communication statement indicate when communication must be complete (following SOS automaton logic).

SOS internals

The following information is provided to help the programmer understand how the SOS system processes communication requests.

Status array:

Each process keeps a status array indexed by variable and node, used to store clear-to-send and short-cut information. This array may be updated without regard to history since it records only current status. Currently, status messages use buffered MPI sends on a separate SOS status communicator. The system periodically polls to get pending status messages. (A non-MPI implementation would have status in shared memory or post updated status information without receiver intervention by active/fast messages).

Data

All communications are built up from point-to-point synchronous data transfers. Except for reductions, the “buffer” at both sender and receiver is defined as the application variable being sent/received (in the case of reductions, receives from multiple
nodes are in separately declared temporary buffers. The binary operator in the reduction places incoming data in a work buffer and applies the binary operation to that buffer and the local application variable, placing the result into the application variable).

Data sends are coordinated by a clear-to-send/send protocol: the receiver indicates readiness to receive with a “CS” handshake message to the sender; the sender replies by sending actual data. The receiver confirms that the sender got the handshake when the actual data is received.

\textit{FSM}

The system assigns a finite state machine to each variable to be communicated; this is done with an SOSaddVar call. In this version of the system there is no way to remove a variable once defined into the system. The FSM consists of a single integer for each variable at each node; and an 11x13 transition table. State changes at each FSM are performed by table lookup.

The system assumes that different FSMs are assigned to each variable, and bases its guarantee of non-interference between communications with respect to different variables on this assumption. There is presently no bound or alias checking, so it is possible to define the same variable into the system more than once and to define overlapping areas of the same array as different variables. The results of doing this are undefined and probably incorrect.

It may, however, be desirable to define disjoint areas of the same array as different variables, if those areas may be communicated or updated independently of each other.

\textit{Queue}

An SOS communication is a set of point-to-point communications. The collective communication pattern is currently defined by a hypercube algorithm on the indices of an array that contains the current implicit process set (IPS). The required set of
p-p sends to define the complete communication is calculated at each node; each node selects out the sends and receives it must carry out and builds an ordered list of sends and receives. Each item on the list includes:

- variable pointer to application data info (location, size, type)
- size of this communication (may be subset of an array)
- offset of this communication (point into array)
- send-flag true if this item is a send
- to-process
- from-process
- function pointer to function for reductions, present only in calculation step - else NONE
- pointer to work buffer for reduction (only if function exists) - else NIL
- send-phase flag indicating all communications remaining are sends
- group flag, shared by a group of sends or receives that can go in any order
- next-pointer-queue
- prev-pointer-queue
- next-in-this-communication-pointer
- prev-in-this-communication-pointer

The first node on the list is pushed on the SOS communication system queue (actually implemented as a combined stack and queue), and each communication on the queue can be a linked list of p-p sends and receives all of which refer to the same variable and all of which are controlled by the same finite state machine.
Queue processing

The SOS run time system runs on a periodic timer interrupt. On each interrupt, the system checks for status changes and examines the queue (status is always checked, the queue is only scanned if there is a status change or application program call to a communication routine, an RH, LH or short-cut call). At present, status change is determined by checking the MPI system for buffered message reception on the SOS status communicator.

SOS checks the queue starting from top (front). SOS looks at the first item on a list of actions for a particular communication. If that communication can be carried out, it is done and removed from the queue - its memory is freed, and any pointers it uses are reset.

If the first item has the group communication flag set, and the next item in the same communication also has the group flag set, SOS scan the linked list on the same communication until either the group flag is not set or the end of the list.

SOS then scans the next item on the queue and repeats the above logic.

When the system reaches the end of the queue, SOS checks if any of the finite state machines has indicated that program cannot continue (is in a blocked state) until communication has completed. If blocked, SOS does a status check again and repeats the check of the queue from the top. These actions are repeated until any communications that were placing the system in the blocked state have completed, then control returns to the application program.

12.2 Lexical analysis of communication statements

We here give some examples to illustrate the kinds of communication statements with which we will be concerned. We wish here to justify the idea that, in typical cases, the information we will need can in fact be extracted by reasonably standard analysis of the program text. Further, we are using communication statements as leaders, which will affect the block structure of the CFG. We therefore need to support the feasibility of their identification and analysis.
Control flow and logic statements are identified by standard lexical analysis of program text. It is worth considering the lexical analysis of communication statements, since this is not standard. We would like not only to determine where the communication statements are, but also to determine the processes involved in the communication statement (or at least the variables that will hold the process identifiers at run time). We assume that the program text can be compiled into an executable program, therefore it must be possible to either determine what the identifiers of communicating processes are, or where this information will be stored at run time. Depending on the particular system, however, this information may be more or less difficult to obtain by a lexical scan of the program.

As is the case with program statements in general, the analysis of communications is strongly dependent on both the syntax and the semantics in which communication is expressed in different systems. We can distinguish between communication based on library functions, such as MPI [MPI 95], and based on language features, such as PBlanguage [PLANG 99] and Co-Array Fortran [NUMRIC 97]. Within the language based communication set, we can further classify communication as explicit, if some particular construct always indicates a communication between processes, or implicit, in which communication may or may not underlie some standard expression in the language. Each type of communication may require different considerations for analysis.

Consider MPI library based communication. MPI library calls are indicated in the text by a naming convention that uses the prefix MPI followed by an underscore in the names of all calls. It is straightforward to obtain a list of all the calls, mark those calls that perform communication or synchronization and scan for them in the program text.

MPI classifies communication as point-to-point or collective. Within point-to-point, we have, for example:

\[
\text{MPI\_SENDRCV}(\text{sendbuf}, \text{sendtype}, \text{dest}, \text{sendtag}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \\
\text{source}, \text{recutag}, \text{comm}, \text{status})
\]
This is a transfer of `sendbuf` at `source` to `recvbuf` at `dest`; it is straightforward to parse the function parameters and obtain these values. Similar information is available when sends and receives are separate calls, using `MPI_SEND` and `MPI_RECV`. In particular, the send statement includes the identifier of the intended receiver. It is, however, possible for the receive statement to identify the sender as `MPI_ANY_SOURCE`. Without matching a particular send statement to a particular receive statement, we can not know the source at compile time. Matching sends with receives requires substantial additional analysis and may not always be possible.

Within collective communications, MPI recognizes two types of process set, the `group` and the `communicator`. Collective communication functions (in MPI version 1) are implemented only on `communicators`, so we will limit the current discussion to this type of set. We have statements such as:

```c
MPI_BCAST(buffer, count, type, root, comm)
```

This is a transfer of the contents of `buffer` at process `root` to `buffer` at all processes in communicator `comm`. Collective communication in MPI does not allow the use of identifiers like `MPI_ANY_SOURCE` which leave process identities unspecified, so at least in theory we should have access to all the process identifiers.

A problem with the analysis of collective calls is identifying the processes in `comm`. MPI initializes a standard communicator called `MPI_COMM_WORLD` to include all processes in a particular program run. To specify a set of processes different from this for collective communication, MPI provides calls such as `MPI_COMM_CREATE` which allows construction of a new set as a subset of some current set, `MPI_COMM_DUP` which allows creation of a new set identical to an old set, and `MPI_COMM_SPLIT` which allows creation of several disjoint subsets from an existing communicator. Functions `MPI_COMM_SIZE` and `MPI_COMM_RANK` provide information on the number of processes in a communicator and on the identifier of the process calling `MPI_COMM_RANK` in a particular communicator. Access to identifiers of all processes in a communicator can be obtained by creating a temporary set from the particular communicator with `MPI_COMM_GROUP` and using
the handle of the set to extract the identifiers of processes in it. (MPI has more varied functions to deal with groups than it does with communicators, so it may appear that we should deal with them directly; however there is no function to directly create a communicator from a group specification. We can only create communicators by starting from other communicators. For details and the rationale behind this, the reader is referred to [MPI 95]).

Except in the simplest cases, for example where \texttt{MPI\_COMM\_WORLD} is used, information about the group of processes involved in collective communication may require the programmer to somehow provide this information to the compiler. In general, although library systems must have internal information about things like process sets for collective communication, the availability of this information depends on the design of the particular system. We can expect, then, that the use of library functions may not allow for complete analysis of processes participating in communication without added information from the programmer.

Language based systems are more amenable to analysis at compilation, since data structures that hold run time information must be created by the compiler and are therefore known. Textual analysis must be able to capture which statements require communication, because the compiler has to generate the appropriate communication code. The analysis and code generation may be more or less elaborate depending on the nature of the language.

Co-array Fortran implements a form of shared memory in which interprocess communication is textually indicated by an array reference in square brackets. Any statements that do not include square brackets refer only to local memory. There is no direct way of expressing a point-to-point send and receive, access to memory not owned by the local process has the semantics of either a put or a get. A send and receive such as exhibited by our \texttt{MPI\_SENDRECV} example may be expressed as either a get at process \texttt{dest}:

\begin{verbatim}
recvbuf = sendbuf[source],
\end{verbatim}

or as a put at process \texttt{source}:

\begin{verbatim}
recvbuf[dest] = sendbuf.
\end{verbatim}
In this case, the get must be executed by the destination process, or the put must be executed by the source process. In either case, a synchronization statement involving both processes must precede the transfer statements, to insure the same semantics as the MPI code; otherwise we may have a race condition because the point that either the source or the destination process has reached in execution is undetermined (synchronization statements in Co-array Fortran look like function calls rather than language syntax extension).

In put statements, the source process is implicitly the process executing the put; this creates a potential race condition if more than one process executes the statement. In get statements, communication is potentially to a set of processes which execute the get. It is left up to the programmer to ensure that all information transfer occurs between the correct processes at the correct points of execution.

Planguages feature explicit communication statements that must be executed by all involved processes. All communication statements use either the explicit process reference operator @, or the curly brackets {} for collective reduction operations. Separate synchronization statements are not required because @ and {} include synchronization between all participating processes. For example, the following statement has the same effect as the MPI_SENDRECV statement in the above discussion:

\[
\text{recvbuf}@\text{dest} = \text{sendbuf}@\text{source}
\]

As in the MPI code, source and destination processes and memory locations are known from parsing the statement.

The effect of a broadcast is obtained in Planguages by not specifying the location of the destination buffer; thus the MPI_BCAST instruction above could be written:

\[
\text{buffer} = \text{buffer}@\text{root}
\]

Although the root process (the source of the broadcast) is explicit in the statement, the destination processes are only known implicitly. In fact, the semantics of this statement indicate that all processes that execute the statement receive the value of buffer at root, and set their local value of root to the received value. To generate code for this statement, the compilation system must be able to know which processes will in fact execute this statement at run time (even a simple implementation in which the
statement is written as a set of point to point communications from root directly to each target process can be problematic, since the sender requires a list of destinations).

Both Co-array Fortran and Planguage syntax permit the creation of implicit communication sets that are a subset of the total number of processes executing the program; neither presently provides any mechanisms for the programmer to determine what these sets are. Co-array Fortran potentially generates highly inefficient communication for broadcasts, in which the amount of data transferred scales as the number of participating processes. Planguages generate efficient code for broadcast and reduction operations, in which the data communicated increases as the logarithm of the number of processes involved; however the cost of this is the requirement that the process set be known (this would also be the case for a Co-array Fortran implementation that attempted to implement more efficient collective operations instead of a succession of gets at each process). In practice, this restricts Planguage collective communications to code sections that are executed by all processes.
INDEX

After, 41
deterministic, 36
all-gather, 107
DOM, 86
back edge, 101
dommates, 86
barrier synchronization, 56
double-arrow, 73
basic blocks, 80
equivalent executions, 45
Before, 41
exit node, 102
block, 33
Finite State Machine, 139
BSP, 8, 125
first merge node, 84
CFG, 33
Flynn taxonomy, 4
circular dependence, 62
forward edge, 101
closed under communication, 53
fusion statement, 59
code analysis, 110
fusion statements, 119
collective communications, 138
get, 14
collective data transfer, 148
handshake, 17, 48, 54
communication statements, 50
header, 102
communication time, 125
implicit process groups, 26
Communications, 17
Independent, 42
complete merge node, 84
information transfer, 49
concurrent, 77, 78
interruptible communications, 159
Conflict, 42
IPG, 32
Irregular, 11
data transfer, 13
key, 18
data view, 36
Keys, 20
deadlock, 58
lazy buffering, 127
deadlock-freedom, 121
deriving the CFG, 80
key, 18
derterminism, 121
211
leader, 75
Lockstep, 65
logical synchronization, 144
loosely synchronous, 67, 127
membership, 52
merge node, 84
merge node tree, 96
merge nodes, 85
merge tree, 85, 92
MIMD, 4
MIPS, 24, 74
MIPS level, 99, 108
MIPS merge, 76
MIPS split, 75
MISD, 4
natural loop, 102
non-uniform, 27
observable, 51
one sided transfer, 54
overlap protocol, 134
Overlapping, 24, 124, 127
P model, 27
parallel data view trace, 44
parallel efficiency, 155
parallel state, 37
parallel state relations, 41
parallel transition, 39
Pfortran buffered messaging, 19
Planguages, 27
point synchron, 75
point-to-point, 51
point-to-point communications, 146
progress, 35
put, 14
race condition, 62
reducible, 101
Runtime, 114
send-receive, 119
serial execution, 35
serial state, 34
serial step, 35
serial transition, 34
set of all valid states, 44
shared memory, 20
short-cutting, 24, 155
SIMD, 4
SISD, 4
SOS, 3, 26, 130, 132, 138, 142, 145, 166
split node, 84
SPMD, 7
SPMD execution, 40
subroutine call, 80
successor, 75
synchron, 74
synchronization, 55, 57
synchronization time, 125
task parallelism, 7
time-like, 35
total execution, 39
trailer, 75
Trivial, 65
two-sided transfer, 53
uniform loop, 103
valid state, 41
vector combine, 107
while, 102
REFERENCES


214


[DEO 74] Narsingh Deo; Graph theory with Applications to Engineering and Computer Science, Prentice-Hall 1974


[FLASH] ASCI/Alliances Center for Astrophysical Thermonuclear Flashes, http://flash.uchicago.edu/


[PLANG] Planguages web site: http://planguages.cs.uchicago.edu/


