1. Introduction

PC is an extension of C. The original motivation was to simplify the writing of message-passing code for multiprocessors with distributed memory. The objectives were two-fold: first to make it easier to write programs for parallel machines; second to insure that exchange of data among different processors is done correctly. The language provides an efficient approach to a unified address space for distributed memory processors as well as a programming model for general parallel computing.

Single Code Programming

PC makes the important assumption that each processor knows the names of the variables in all processors. To this end we require, in the current implementation, that all processors run the same PC code. For acronym addicts, our programming model is Single Code Multiple Datastream or SCMD. Thus the programming style is similar to SIMD languages (e.g., *-Lisp on the Connection Machine) used for SIMD programs. However, SCMD programs need not proceed in lock-step, operation by operation, as would occur on SIMD machines.

deterministic versus chaotic

In implementing a language such as PC, one must decide whether to assume that everyone knows what everyone else is doing, or not. For example, if you know in advance that a neighbor may want to borrow a cup of sugar, you can put one by the back door when you have a spare moment and then go on about your business. Otherwise, you would have to wait at home indefinitely for the knock at the back door. Moreover, you can make sure that there is sugar available; otherwise the operation of dipping the cup in the sugar bin may have undesired results. We refer to a parallel programming environment in which the implementor assumes that we know in advance what others will require of us a deterministic environment. An environment in which we must constantly be prepared for arbitrary demands on our resources we call a chaotic environment. For simplicity, we have limited our implementation of a compiler for PC to the deterministic case. There is nothing (that we know of) in PC that prevents implementation of a compiler in the chaotic case, but it is clear that the demands on the operating system and node architecture are much greater in order to do so efficiently.
How to Use PC

For the iPSC/2

In the current implementation, the compiler for PC produces C code. This compiler is called pcc on karam. For now it is in the directory /home/babak/bin. For example, if codename.pc is a PC code, then

```
/home/babak/bin/pcc codename.pc
```

produces a C code called codename.c. This file contains message-passing C code that can be compiled for the iPSC/2 using the C compiler:

```
cc codename.c -node
```

with the usual options available. The resulting executable file a.out can be executed via

```
load a.out
```

on the iPSC/2.

For the iPSC simulator

In the current implementation, the compiler for PC produces C code. This compiler is called pcc on karam. For now it is in the directory /home/babak/bin. For example, if codename.pc is a PC code, then

```
/home/babak/bin/pcc codename.pc
```

produces a C code called codename.c. This file contains message-passing C code that can be compiled for the iPSC/2 simulator using the C compiler:

```
cc codename.c -1bsim
```

with the usual options available. The resulting executable file a.out can be executed via

```
load a.out
```

in the simulator.

In the Cosmic Environment

In the current implementation, the compiler for PC produces C code. Compilation for the sun network is done via pcc -c. The compiler is in the directory /usr/local/bin. For example, if codename.pc is a PC code, then

```
/usr/local/bin/pcc -c codename.pc
```

produces a C code called codename.c. This file contains message-passing C code that can be compiled for the Cosmic Environment using the C compiler:
/usr/local/cube/ccgh -o codename codename.c -litcube -ltrk -lcube

with the usual additional options available. The compiler ccgh is located in the directory /usr/local/cube so this directory must be included in your path if you do not want to use the full path every time. This compiler will produce an executable named codename.gh.

To run on the Suns in the Cosmic Environment, you must first get some resources. For example,

    getcube ghost 2n

will acquire two Suns on which to execute your code(s). The executable file a.out resulting from the above compile can be executed via

    spawn codename -1

on the Suns.

You should not have either of the following in your PC code:

    #include <stdio.h>
    #include <malloc.h>

Inclusion of either of them can cause unpredictable results.

When you are done running your code you should release the ghost cube by saying freecube.
2. PC Constants

For its own and your convenience, PC defines the following constants.

- **THISNODE** is the processor number of the processor running the code,
- **THISPROC** is the process number of the process running the code,
- **NUMNODES** is the total number of processors,
- **HOST** is the processor number of the host processor,
- **HOSTPROC** is the process number of the host process.

We assume there is only one process on the host which can communicate with the nodes (processors). We make the convention that the processors are identified by consecutive integers, starting at 0 and continuing to **NUMNODES**−1.

In many cases, our default names for these constants may be undesirable, due to their length or conflict with other variable names. The name for **THISNODE** can be redefined at compile time by using the “−T” compiler option. For example, to compile the code **nodecode.pc** which uses **ME** to mean **THISNODE**, simply compile with the command

```
  pcc −TME nodecode.pc
```

A special, architecture–dependent variable is defined for hypercube–topology machines, namely,

- **CUBEDIM** is the dimension of the cube, i.e., \( \log_2(\text{NUMNODES}) \).
3. The Operator @

An essential requirement in many parallel programs is to access variables in other processors. This can be done in PC using a new binary operator, @, which indicates that a variable is “at” another processor. PC defines this new infix operator for accessing nonlocal variables by adding the following to the grammar for C:

\[
<\text{primary expression}> \rightarrow \\
<\text{primary expression}> \, @ \, <\text{expression or identifier}> \mid \\
<\text{primary expression}> \, @ \\
\quad [ \, <\text{expression or identifier}> \, : \, <\text{expression or identifier}> \, ] \mid \\
<\text{primary expression}> \, [ \, <\text{expression or identifier}> \, : <\text{expression or identifier}> \, ]
\]

where \(<\text{expression or identifier}>\) is defined by the rule

\[
<\text{expression or identifier}> \rightarrow \langle \text{constant} \rangle \mid \langle \text{identifier} \rangle \mid [ \, <\text{expression} \, ]
\]

For now these constructs are only allowed in assignment statements, with at most one @ per assignment statement.

Using a nonlocal variable: the simplest case

The construct

\[
<\text{primary expression}> \, @ \, <\text{expression or identifier}>
\]

returns the value of \(<\text{primary expression}>\) at the processor whose number is given by \(<\text{expression or identifier}>\). (So the \(<\text{expression or identifier}>\) must be integer valued.) For example,

\[
x = y@0;
\]

means: assign the value of variable \(y\) of processor 0 to \(x\) (in every processor). This is simply a broadcast from processor 0 to all other processors. Similarly

\[
x = y@[n+1];
\]

means: assign the value of variable \(y\) of processor \(n+1\) to \(x\) (in every processor). If the value of an expression is not in the range \(0, \ldots, \text{NPROCS}-1\), then assignment does not occur.
PC Reference Manual

Ranges of processors and arrays

We can now give a simple example of a code to compute the sum of values contained in each processor, say in the variable \( y \). The idea is simply for everyone to broadcast their own value, and then each processor sums the values independently.

```c
for(n=0; n < NUMNODES; n++)
    x[n] = y @ n;
sum=0.0;
for(n=0; n < NUMNODES; n++)
    sum = sum + x[n];
```

This technique may not be very efficient, but it serves to illustrate an important point concerning our assumption about a deterministic environment. In the first loop in the code above, the processor from which we receive data is a variable. However, at each step in the iteration the value of this variable is the same in each processor. Thus \( n \) denotes a value universally agreed upon at each iteration. If the value of \( n \) were different in different processors, a chaotic environment would be needed, which is not currently supported.

It is currently not allowed to mix communication expressions with ordinary expressions, e.g., an expression such as \( \text{sum} = \text{sum} + y @ n; \) is not allowed at present.

Ranges of processors and arrays

There are many instances in which sets of processors or arrays of values are useful objects. We have implemented one class of such sets, namely *ranges*. The construct

\[
<\text{primary expression}> @[<\text{expression or identifier}>_1 : <\text{expression or identifier}>_2 ]
\]

refers to the value of \(<\text{primary expression}>\) in the range of processors (nodes) from the \(<\text{expression or identifier}>_1\) to \(<\text{expression or identifier}>_2\). Naturally, such a construct cannot return a value in the usual sense. Therefore its use is very restricted. It can appear on the left hand side of an assignment statement. For example,

\[
x@[1:5] = y@0;
\]

means: “Assign the value of variable \( y \) in processor 0 to variable \( x \) in processors 1 through 5.”

The construct

\[
<\text{primary expression}> [ <\text{expression or identifier}> : <\text{expression or identifier}> ]
\]
refers to ranges of values in arrays. For example, consider the code

\[ x[0:5]@[1:5] = y[1:6] @ 0; \]

Here we treat \( x[0:5] \) and \( y[1:6] \) as vectors and assign the elements consecutively. We do the same consecutive assignment for arrays with higher dimension. For example

\[ x[ni:nf][mi:mf]@[1:5] = y[ji:jf][ki:kf] @ 0; \]

has the same ultimate effect as

\[
\text{for}(n=ni; n <= nf; n++)
\text{for}(m=mi; m <= mf; m++)
\quad x[n][m]@[1:5] = y[ji+ni-n][ki+mi-m] @ 0;
\]

but its implementation is much more efficient since block moves are utilized.

**Functions of THISNODE**

The construct

\[ x = y@(\text{THISNODE+1}) \% \text{NUMNODES}; \]

means: assign the value of variable \( y \) of processor \( \text{(THISNODE+1)} \% \text{NUMNODES} \) to \( x \) in THISNODE. Within the parentheses we can have any expression which mentions THISNODE explicitly (including functions which return integers, arithmetic operators, etc.).

An example of the use of functions can easily be constructed to give a slightly more efficient summation algorithm, as follows. Suppose we think of our nodes as making up a simple ring. Then we can sum by accumulating and passing partial sums around this ring:

\[
\text{sum=}y;
\text{for}(n=0; n < \text{NUMNODES} - 1; n++)
\text{\{ sum = sum@(\text{THISNODE+1}) \% \text{NUMNODES};
\text{\quad sum = sum + y; \}}
\]

This has the effect of computing the \( n \)-th partial sum

\[ \ldots (y@\text{THISNODE} + y@[\text{THISNODE+1}]) + \ldots + y@[\text{THISNODE+n+1}] \]

at the \( n \)-th iteration, where node indices in the latter representation must be identified modulo NUMNODES.
4. Set Operations

Having introduced a formalism for distributed variables via the \(@\) operator, it is natural to consider appropriate operators on this new data type. In this section we introduce one class of such operations based naturally on corresponding operations on each component of the distributed data structure. Such operations may, moreover, be defined on a restricted set of distributed data. As mentioned above, an expression like \(x@[1:5]\) refers to many variables. We cannot expect it to return a value, but we often want to perform an operation on the set of values of these variables. To allow such operations, PC introduces the following rule:

\[
\text{<assignment expression> } \longrightarrow \\
\text{<binary operator>}{\text{<expression or identifier> } } | \\
\text{<binary operator>}{\text{<expression or identifier> } } \otimes \\
\quad [\text{<expression or identifier> } : \text{<expression or identifier> } ]
\]

The construct

\[
\text{<binary operator>}{\text{<expression or identifier> } }_1 \otimes \\
\quad [\text{<expression or identifier> }_2 : \text{<expression or identifier> }_3 ]
\]

returns the result of applying \(<\text{binary operator}>\) to the set of values of \(<\text{expression or identifier}>_1\) at the processors in the range from \(<\text{expression or identifier}>_2\) to \(<\text{expression or identifier}>_3\). Since we are allowed to say

\[
\text{<binary operator>}{\text{<expression or identifier> } }
\]

we can omit the range of processors; when the range is missing, we take it to mean all processors. The \(<\text{binary operator}>\) is +, *, or any function of two parameters both of which have the same type as the \(<\text{expression or identifier}>_1\). The function must return a value of the same type as \(<\text{expression or identifier}>_1\) and assignment must be defined for this type. Of course, since the order of application of the function to the set of values is not specified, the function must be associative and commutative. (More precisely, we are assuming that the set of values of type \(<\text{expression or identifier}>_1\) forms an Abelian semi-group with respect to \(<\text{binary operator}>\).)
For example, the summation problem for which we have given two codes previously can now be written succinctly as

\[ \text{sum} = +\{y\}; \]

This means: “Assign to \text{sum} in every processor the sum of the values of \text{y} in all processors.”

Correspondingly,

\[ x = \max\{y@[0:7]\}; \]

means: “Assign to \text{x} in every processor the maximum of the values of \text{y} in processors 0 through 7,” provided \text{x} and \text{y} are integers and we have defined a function called \text{max} as follows:

```c
int max(i,j)
int i,j;
{
    return((i>j)?i:j);
}
```

Quite complex commutative operators can be utilized. Consider the following approach to computing a matrix times a vector (\text{matvec}) that uses the “globalized” version of \text{addvec}, a function that adds vectors. We begin by introducing a data structure for vectors:

```c
typedef struct {double v[D];} VECTOR;
```

For this new data type, we define \text{addition} by introducing a function that adds two vectors and returns their sum:

```c
VECTOR addvec(a,b)
VECTOR a,b;
{
    int i;
    VECTOR r;
    for (i=0;i<D;i++) r.v[i] = a.v[i] + b.v[i];
    return(r);
}
```

Finally, we put all this together to define matrix multiplication using the “globalized” version of \text{addvec}. Note that \text{D} is the dimension of the matrix, and \text{N<<D} is the number of columns stored in each node. The index \text{global}(i) (i= 1,\ldots,N) indicates how the columns are distributed to each processor, but the details of a particular choice of such a distribution is left unspecified. One such choice is defined in the following:
global(j) int j; { return(TISPQDE + j*NN); }

where $NN = D/N$. The matrix is assumed to be distributed in a similar way.

```c
void matvec(res, mat, vec)
    double res[N], mat[D][N], vec[N];
{
    int i,j,li,lj;
    VECTOR s;
    for(i=0;i<D;i++) {
        s.v[i] = 0.0;
        for(j=0;j<N;j++) {
            s.v[i] += mat[i][j]*vec[j];
        }
    }
    s = addvec(s);
    for(i=0;i<N;i++) res[i] = s.v[global(i)];
}
```

The function `addvec` is clearly associative and commutative; its global version adds all the components among all processors. A similar approach can be used to “accumulate” other distributed objects, such as pixel values in a digital image.
5. Sample Programs

We begin with a simple implementation of the bitonic sort algorithm. We assume only that NUMNODES is a power of two, although the algorithm is designed with hypercube topology in mind. For simplicity, the code assumes there is only one piece of data per processor; for a complete code, a local sort would need to be added. It begins by generating random integers in the range 0, ..., 31 to be sorted and prints the initial values.

```c
main()
{
    int i,j,d,dim,n,x,cudim;
    cudim = 4;
    n = 7*THISNODE % 31; (random number generator)
    printf("\n proc \%d has \%d", THISNODE , n);
    dim = 1;
    for(j=0;j< cudim ;j++)
    {
        d = dim;
        dim = dim*2;
        for(i=0;i< j+1 ;i++)
        {
            x = n@[ THISNODE ^ d ];
            if((bit(THISNODE,dim)+bit(THISNODE,d))%2) {n = min(x,n);} else {n = max(x,n);} d = d/2;
        }
    }
    printf("\n processor \%d has \%d", THISNODE, n);
}

bit(m,d)
int m,d;
{
    if( m & d ) { return( 1 );} else { return( 0 );}
}
```

The function max was defined previously and min must be defined analogously.

If one wants to print the sorted numbers in order, a simple “baton passing” technique will do this; replace the last printf statement by
int baton = 0;
for(i=0; i < NUMNODES; i++) {
    baton[i] = baton[i-1]; (this is a null statement for i=0)
    if (baton == THISNODE)
        printf("\n processor %d has %d", THISNODE, n);
    baton = baton + 1;
}
}

The most computationally intensive part of the conjugate gradient method consists of the multiplication of a matrix times a vector (matvec). Using the techniques introduced in the chapter on "global" operations, the following solves a linear system approximately via this iterative technique:

int cg(x,a,b)
double x[N],a[D][N],b[N];
{
    int i,k;
    double bnorm, prevrho, rho, alpha, beta, p[N], r[N], w[N];
    bnorm = dot(b,b);
    for(i=0;i<N;i++) {x[i] = 0.0; r[i] = b[i];}
    prevrho = rho = dot(r,r);
    for(k=1;(k<MAXITER) && (rho>EPS*bnorm);k++) {
        beta = (k==1)?0.0:(rho/prevrho);
        for(i=0;i<N;i++) p[i] = r[i] + beta*p[i];
        matvec(w,a,p);
        alpha = rho/dot(p,w);
        for(i=0;i<N;i++) {
            x[i] += alpha*p[i];
            r[i] -= alpha*w[i];
        }
        prevrho = rho;
        rho = dot(r,r);
    }
    #ifdef DEBUG
    if(THISNODE==0) printf("%d: rho=%e\n",THISNODE,rho);
    #endif
}
return(k);
}
matvec (defined previously) and dot. The “dot” product of two vectors used in cg is defined in the usual way as follows.

```c
double dot(v,w)
double v[N],w[N];
{
    int i;
    double d=0.0;
    for (i=0;i<N;i++)
        d += v[i]*w[i];
    d = +{d}; // (global summation over all processors)
    return(d);
}
```
6. Typical Programming Errors in PC

One source of difficulty with any parallel programming system concerns communication statements that occur inside blocks that are “guarded” by conditional expressions. For example, the code

```c
if( THISNODE == 7 ) { x = y@11 ;}
```

is not the same as

```c
x@7 = y@11 ;
```

in deterministic PC. The latter correctly assigns the value of the variable y in processor 11 to the variable x in processor 7, whereas the former will simply cause processor 7 to dead–lock waiting for processor 11 to do something that it will never know about. Variants of the former statement can be correct, e.g.,

```c
if( THISNODE == 7 || THISNODE == 11 ) { x = y@11 ;}
```

will set x in both processors 7 and 11 to the value of y in 11. However, the latter programming style in PC is discouraged. Rather, the expression `x@7 = y@11 ;` is preferred.