Using UNE's Beowulf Clusters

Brett Carson & Ian A. Mason

Contents

1 Introduction

This paper has several aims. The first is to provide a clear and precise guide to programming the School's two Beowulf clusters using both PVM and MPI, qua `C` libraries. The second is to present a reasonably realistic case study of how one goes about designing and building a Beowulf application, again in both PVM and MPI, from *toy* beginnings to finished endings. The third is to compare the performance of our old Beowulf to that of our new one. While the fourth is to offer some guidelines for choosing between PVM, and MPI.

2 Beowulfs: The General and The Specific

3 Distributed Computing via PVM or MPI

4 The Road Ahead

This is a brief note explaining my recent experience with pvm and the Beowulf. It is partially based on my lectures in 381 so excuse the tone.

We will start with a *toy* version of an algorithm and tweak it until it hurts, or at least gets some smoke coming of those 16 cpu's.

Our tweaking will also probably make the thing break, so we will have to learn the gentle art of debugging PVM code. Something which is not as easy as it should be. All in all it should be a good learning experience for us all.

5 The Case Study: *Multiplying Matrices on a Torus*

This example is from [2] and its implementation in PVM is described in [1] pages 76 through 83. So you may want to read that part of the book. We will go through it in more detail than the book. A similar but not identical algorithm also makes an appearance with pictures on page 36 and 37 ibid.

Let us begin by reviewing some facts about matrix multiplication, and why it might be useful to do it on a Beowulf cluster using the torus block multiplication algorithm. Recall that a matrix is a rectangular collection of things:, consisting of m rows and n columns:

$$\{\text{thing}_{\text{row,column}} \mid 1 \leq \text{row} \leq m \quad 1 \leq \text{column} \leq n\}$$

The things will usually be numbers, but not always. Sometimes in this unit they will be other matrices, but in this project they will be `int`s. They are usually drawn thusly:

$$\begin{bmatrix} \text{thing}_{1,1} & \text{thing}_{1,2} & \dots & \text{thing}_{1,n} \\ \text{thing}_{2,1} & \text{thing}_{2,2} & \dots & \text{thing}_{2,n} \\ \vdots & \vdots & \dots & \vdots \\ \text{thing}_{m,1} & \text{thing}_{m,2} & \dots & \text{thing}_{m,n} \end{bmatrix}$$

OK so let: A be an $m \times p$ matrix, and B be an $p \times n$ matrix:

$$A = \{a_{r,c} \mid 1 \leq r \leq m \quad 1 \leq c \leq p\}$$
$$B = \{b_{r,c} \mid 1 \leq r \leq p \quad 1 \leq c \leq n\}$$

Then we can define the product $C = A \times B$ to be the following $m \times n$ matrix:

$$C = \{c_{r,c} \mid 1 \leq r \leq m \quad 1 \leq c \leq n\}$$

where

$$c_{r,c} = \sum_{1 \leq i \leq p} a_{r,i} \times b_{i,c}$$

Note how the number of columns of A must match the number of rows of B. Thus in pictures:

$$\begin{bmatrix} \vdots & \vdots & \dots & \vdots \\ a_{r,1} & a_{r,2} & \dots & a_{r,p} \\ \vdots & \vdots & \dots & \vdots \end{bmatrix} \times \begin{bmatrix} \dots & b_{1,c} & \dots & \dots \\ \dots & b_{2,c} & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \\ \dots & b_{r,c} & \dots & \dots \\ \dots & \vdots & \dots & \dots \\ \dots & b_{p,c} & \dots & \dots \end{bmatrix} = \begin{bmatrix} \dots & \vdots & \dots & \dots \\ \dots & \vdots & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots \\ \dots & c_{r,c} & \dots & \dots \\ \dots & \vdots & \dots & \dots \\ \dots & \vdots & \dots & \dots \end{bmatrix}$$

$$\text{where} \quad c_{r,c} = \sum_{1 \le i \le p} a_{r,i} \times b_{i,c}$$

The first really nice property of matrices that the algorithm uses is that one can divide the computation up into blocks. Suppose A and B are square matrices $2m$ by $2m$. Then to form $C = AB$, we can actually think of A and B as 2 by 2 matrices, whose entries are m by m matrices. Thus

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} & A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} & A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} \end{bmatrix}$$

We could also think of `A` and `B` as m by m matrices whose elements are 2 by 2 matrices. This is called *block* matrix multiplication, and we are going to do such a computation, on a torus.

Lets start by describing the toy algorthm, then you'll see why its done on a torus.

We are going to have $m \times m$ tasks, and each task will manage the appropriate entry in the matrices. I.e. each task corresponds to an entry in the matrix. Now the entries in the matrix will be themselves square matrices of size k. Thus in reality we will be multiplying a $m \times k$ square matrix.

The algorithm will be using passes blocks of (of `B`) up, and receives blocks (of `B`) from below. In addition to this we will be (broadcasting) blocks (of `A`) horizontally. Thus we have rings up, and rings around. Thats a torus.

Here is how to visualize it. We start with a square matrix

```
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
+ + + + + + + + + +
```

Each entry in the matrix will actually be a block. Furthermore for each block there will be a task. Thus tasks will manage three blocks, an `A` block, a `B` block, and a `C` block, the corresponding answer block. So we can think of the `+`'s as either blocks or tasks. Now the bottom rows nearest neighbours lie within the row in the rom above, and the top row. Thus we can think of the thing as rolled over to form a tube. The same reasoning goes for the columns, thus the tube its rolled over and joined. A donut results. Donuts are toruses, or torii if you want to be snooty.

Now this is how the algorithm works. Each task starts out with an `A` block, a `B` block, and a (iniitalized to zero) `C` block.

In step one, the tasks on the diagonal (i.e. row i equalling column j) broadcast their `A` block along their row.

They each then multiply this `A` block with their `B` block and add it to their `C` block.

Now we rotate the `B` blocks. We pass our `B` block to the task above us (if we are in the top row we pass our `B` to the entry in the bottom row) but if you think of the thing as a torus we are just passing it up.

The process is now repeated with the next diagonal along. I.e tasks in the diagonal (i.e. row i , column $i + 1$) broadcast their `A`'s, do the multiplication, add it to `C`, then pass the `B`'s up. After m iterations everything returns to it origonal place and we have our result `C = AB`.

6 The Algorithm: Fox's Matrix Multiplication

The following is straight out of the book. There are some obvious toy aspects to the code. Like creating `A`, `B`, and `C` on the fly. The toy program generates a random `A` and uses the identity as `B`, thus we can catch errors on the fly by comparing the answer with the `A` matrix.

We'll obviously want to get them from files. In fact we'll have to think about this aspect of the algorithm in detail. But we will retain this choice of `A` and `B` as a convenient way to monitor the correctness of our program.

The code is not particularly complex. It uses dynamic groups for identification purposes. Each task joins the group `mmult`, and receives its membership number (from 0 upto the size of group). From this number it figures out its row and column number, the identity of the other tasks in its row, and the tasks immediately above it and below it (columnwise) on the torus.

```
// origonal version
#include <stdlib.h>
#include <pvm3.h>
#include <stdio.h>
/* Maximum number of children this program will spawn */
#define MAXNTIDS    100
#define MAXROW      10
/* Message tags */
#define ATAG        2
#define BTAG        3
#define DIMTAG      5
void InitBlock(float *a, float *b, float *c, int blk, int row, int col){
    int len, ind;
    int i,j;
    srand(pvm_mytid());
    len = blk*blk;
    for (ind = 0; ind < len; ind++)
        { a[ind] = (float)(rand()%1000)/100.0; c[ind] = 0.0; }
    for (i = 0; i < blk; i++) {
        for (j = 0; j < blk; j++) {
            if (row == col)
                b[j*blk+i] = (i==j)? 1.0 : 0.0;
            else
                b[j*blk+i] = 0.0;
            }
        }
}

void BlockMult(float* c, float* a, float* b, int blk){
    int i,j,k;
    for (i = 0; i < blk; i++)
        for (j = 0; j < blk; j ++)
            for (k = 0; k < blk; k++)
                c[i*blk+j] += (a[i*blk+k] * b[k*blk+j]);
}
int main(int argc, char* argv[]){
    /* number of tasks to spawn, use 3 as the default */
    int ntask = 2;
    /* return code from pvm calls */
    int info;
    /* my task and group id */
    int mytid, mygid;
```

```
/* children task id array */
int child[MAXNTIDS-1];
int i, m, blksize;
/* array of the tids in my row */
int myrow[MAXROW];
float *a, *b, *c, *atmp;
int row, col, up, down;
/* find out my task id number */
mytid = pvm_mytid();
pvm_setopt(PvmRoute, PvmRouteDirect);
/* check for error */
if (mytid < 0) {
    /* print out the error */
    pvm_perror(argv[0]);
    /* exit the program */
    return -1;
    }
/* join the mmult group */
mygid = pvm_joingroup("mmult");
if (mygid < 0) {
    pvm_perror(argv[0]); pvm_exit(); return -1;
    }
/* if my group id is 0 then I must spawn the other tasks */
if (mygid == 0) {
    /* find out how many tasks to spawn */
    if (argc == 3) {
        m = atoi(argv[1]);
        blksize = atoi(argv[2]);
        }
    if (argc < 3) {
        fprintf(stderr, "usage: mmult m blk\n");
        pvm_lvgroup("mmult"); pvm_exit(); return -1;
        }
    /* make sure ntask is legal */
    ntask = m*m;
    if ((ntask < 1) || (ntask >= MAXNTIDS)) {
        fprintf(stderr, "ntask = %d not valid.\n", ntask);
        pvm_lvgroup("mmult"); pvm_exit(); return -1;
        }
    /* no need to spawn if there is only one task */
    if (ntask == 1) goto barrier;
    /* spawn the child tasks */
    info = pvm_spawn("mmult", (char**)0, PvmTaskDefault, (char*)0,
        ntask-1, child);
    /* make sure spawn succeeded */
    if (info != ntask-1) {
        pvm_lvgroup("mmult"); pvm_exit(); return -1;
        }
    /* send the matrix dimension */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&m, 1, 1);
    pvm_pkint(&blksize, 1, 1);
    pvm_mcast(child, ntask-1, DIMTAG);
```

```
        }
    else {
        /* recv the matrix dimension */
        pvm_recv(pvm_gettid("mmult", 0), DIMTAG);
        pvm_upkint(&m, 1, 1);
        pvm_upkint(&blksize, 1, 1);
        ntask = m*m;
        }
    /* make sure all tasks have joined the group */
barrier:
    info = pvm_barrier("mmult",ntask);
    if (info < 0) pvm_perror(argv[0]);
    /* find the tids in my row */
    for (i = 0; i < m; i++)
        myrow[i] = pvm_gettid("mmult", (mygid/m)*m + i);
    /* allocate the memory for the local blocks */
    a = (float*)malloc(sizeof(float)*blksize*blksize);
    b = (float*)malloc(sizeof(float)*blksize*blksize);
    c = (float*)malloc(sizeof(float)*blksize*blksize);
    atmp = (float*)malloc(sizeof(float)*blksize*blksize);
    /* check for valid pointers */
    if (!(a && b && c && atmp)) {
        fprintf(stderr, "%s: out of memory!\n", argv[0]);
        free(a); free(b); free(c); free(atmp);
        pvm_lvgroup("mmult"); pvm_exit(); return -1;
        }
    /* find my block's row and column */
    row = mygid/m; col = mygid % m;
    /* calculate the neighbor's above and below */
    up = pvm_gettid("mmult", ((row)?(row-1):(m-1))*m+col);
    down = pvm_gettid("mmult", ((row == (m-1))?col:(row+1)*m+col));
    /* initialize the blocks */
    InitBlock(a, b, c, blksize, row, col);
    /* do the matrix multiply */
    for (i = 0; i < m; i++) {
        /* mcast the block of matrix A */
        if (col == (row + i)%m) {
            pvm_initsend(PvmDataDefault);
            pvm_pkfloat(a, blksize*blksize, 1);
            pvm_mcast(myrow, m, (i+1)*ATAG);
            BlockMult(c,a,b,blksize);
            }
        else {
            pvm_recv(pvm_gettid("mmult", row*m + (row +i)%m), (i+1)*ATAG);
            pvm_upkfloat(atmp, blksize*blksize, 1);
            BlockMult(c,atmp,b,blksize);
            }
        /* rotate the columns of B */
        pvm_initsend(PvmDataDefault);
        pvm_pkfloat(b, blksize*blksize, 1);
        pvm_send(up, (i+1)*BTAG);
        pvm_recv(down, (i+1)*BTAG);
        pvm_upkfloat(b, blksize*blksize, 1);
```

```
    }
    /* check it */
    for (i = 0 ; i < blksize*blksize; i++)
        if (a[i] != c[i])
            printf("Error a[%d] (%g) != c[%d] (%g) \n", i, a[i],i,c[i]);
    printf("Done.\n");
    free(a); free(b); free(c); free(atmp);
    pvm_lvgroup("mmult");
    pvm_exit();
    return 0;
}
```

7 The Goal

Ok as I mentioned last time, we are going to try and tweak `mmult` until we get it running fast and smooth on the cluster. Thus these notes will be pretty informal, and pretty thin. All the programs are in the

```
~comp381/pvm3/mmult/
```

directory.

8 The Goal Posts

We can expect to get an order of magnitude speed up, and an order of magnitude increase in the size of the matrices we can multiply. Maybe more if we are optimists. The best would be a speed up of 16, and a size increase of two orders of magnitude. We could probably do better on this last figure, after all we could subdivide the blocks to multiply them, and so on and so forth. Thats for another time.

```
// single process matrix multiplication
// written  by ian a. mason
// may 21 @ u.n.e
//
#include <stdio.h>
#include <time.h>
#include "sblock.h"

#define VERBOSE  0

int main(int argc, char* argv[]){
    float *a, *b, *c;
    int i, dim, row, col;
    int start_time = time(NULL);
    if((argc != 2) || ((dim = atoi(argv[1])) < 0)){
      fprintf(stderr, "usage: spmm dim\n");
      exit(1); }
    a = (float*)malloc(sizeof(float)*dim*dim);
    b = (float*)malloc(sizeof(float)*dim*dim);
    c = (float*)malloc(sizeof(float)*dim*dim);

    if (!(a && b && c )) {
      fprintf(stderr,
      "%s: out of memory!\n",
      argv[0]);
```

```
      free(a);
      free(b);
      free(c); };

    InitBlock(a, b, c, dim);
    BlockMult(c,a,b,dim);

    for (i = 0 ; i < dim*dim; i++)
      if (a[i] != c[i])
printf("Error a[%d] (%g) != c[%d] (%g) \n",
       i, a[i],i,c[i]);

    if(VERBOSE){
      printf("Block C");

      for(row = 0; row < dim; row++){
for(col = 0; col < dim; col++)
printf("%5g ", c[(row*dim) + col]);
      printf("\n");  }

      printf("Block A");

      for(row = 0; row < dim; row++){
for(col = 0; col < dim; col++)
printf("%5g ", a[(row*dim) + col]);
printf("\n");  }
    }
    free(a);
    free(b);
    free(c);
    printf("Done smoothly after %d seconds\n",
            time(NULL) - start_time);
    return 0;
}
```

So thats the yard stick. Pretty rough one, but it will do. There are some preliminary timings and comments in the files:

```
  -rw-rw-r--   1 comp381  admin       3041 Jun  1 12:46 times
  -rw-rw-r--   1 comp381  admin       2355 Jun  2 14:53 times.5
  -rw-rw-r--   1 comp381  admin        916 Jun  4 11:05 times.6
```

9 What are the PVM versions?

The program evolved in a series of several steps.

9.1 Version `mm.0.0.c`

This is the origonal version downloaded from Oakridge National Laborotory.

9.2 Version `mm.0.1.c`

I added some primitive timing information and a verbose mode.

9.3 Version mm.0.2.c

I tailored it to single architecture mode and enlarged MAXROW to something more reasonable. This had no real effect on the times.

9.4 Version mm.0.3.c

I added real argument arrays and got rid of the silly broadcasting of the things that should be in the argument arrays. I added three file arguments for the input and output, and got rid of the absurd goto added VERSION prior to making it automatic put everything in a library

9.5 Version mm.0.4.c

I incorporated real live manipulation, and cleaned it up a bit in preparation for debugging – sigh. The longest run I've done was (all 16 nodes online):

```
b1 % mkMatrices ~/A ~/B ~/C 4 1000
Done writing files
b1 % mm.0.4 ~/A ~/B ~/C 4 1000
mm.0.4 4 1000 task 262154 done successfully after 3682 seconds.
b1 % cmp ~/A ~/C
```

and this will be slowed down by NFS bottleneck to bourbaki.

We are also beginning to hit the limitation created by my inability to make matrices much bigger than 4000 by 4000. Since such a matrix is 64 megabytes. Thus we move on to mm.0.5.c.

9.6 Version mm.0.5.c

Here we never actually create the matrices A, B, and C. Rather we make their blocks. For this we use the mkBlocks program.

```
bourbaki.une.edu.au.mmult> mkBlocks /tmp/A /tmp/B /tmp/C 4 100
Done writing files
bourbaki.une.edu.au.mmult> ls -l /tmp/A*
-rw-rw-r--   1 comp381  admin           0 May 27 13:27 /tmp/A
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.0.0
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.0.1
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.0.2
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.0.3
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.1.0
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.1.1
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.1.2
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.1.3
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.2.0
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.2.1
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.2.2
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.2.3
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.3.0
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.3.1
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.3.2
-rw-rw-r--   1 comp381  admin       40000 May 27 13:27 /tmp/A.3.3
bourbaki.une.edu.au.mmult>
```

Thus each task on the torus in the i row and j column only maniplulates the appropriate A.i.j, B.i.j, and C.i.j. The program is run like thus:

```
bourbaki.une.edu.au.mmult> mkBlocks /tmp/A /tmp/B /tmp/C 4 300
Done writing files
bourbaki.une.edu.au.mmult> mm.0.5 /tmp/A /tmp/B /tmp/C 4 300
mm.0.5 4 300 task 262213 done successfully after 163 seconds.
bourbaki.une.edu.au.mmult>
```

9.7 Version `mm.0.6.c`

This was my first attempt to map the problem to the Beowulf. Since *most* communication is within each row (the broadcasting) of the `A blocks`. Each row `i` is located on machine `bi` in the area

```
/disks/bi/homes/courses/comp381
```

in files `A.*.*`, `B.*.*` and `C.*.*` `parse_args` checks that the paths are ok relative to `DATAPATH` with no `.i.j` extension numbers. these file should be created by `mkBlockRows`. To make these blocks we use a script `datasetup`

```
bourbaki.une.edu.au.mmult> datasetup A B C 16 100
```

will write the appropriate files to the cluster. You can look at the results via:

```
bourbaki.une.edu.au.mmult> brun -v ls "/scratch/homes/courses/comp381/"
b1      A
b1      A.0.0
b1      A.0.1
b1      A.0.10
b1      A.0.11
b1      A.0.12
b1      A.0.13
b1      A.0.14
b1      A.0.15
b1      A.0.2
b1      A.0.3
b1      A.0.4
b1      A.0.5
b1      A.0.6
b1      A.0.7
b1      A.0.8
b1      A.0.9
b1      B
b1      B.0.0
b1      B.0.1
.
.
.
b16     B.15.5
b16     B.15.6
b16     B.15.7
b16     B.15.8
b16     B.15.9
b16     C
b16     C.15.0
b16     C.15.1
b16     C.15.10
b16     C.15.11
```

```
b16     C.15.12
b16     C.15.13
b16     C.15.14
b16     C.15.15
b16     C.15.2
b16     C.15.3
b16     C.15.4
b16     C.15.5
b16     C.15.6
b16     C.15.7
b16     C.15.8
b16     C.15.9
bourbaki.une.edu.au.mmult>
```

We then start pvm on any node. Its best to start it from the directory:

```
~comp381/pvm3/bin/LINUX
```

directory. Like so

```
bourbaki.une.edu.au.mmult> rlogin b1
Last login: Thu Jun 17 13:47:26 from b0.une.edu.au
b1 % cd pvm3/bin/LINUX/
b1 % pvm setup
pvm> conf
16 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                      b1    40000    LINUX    1000
                      b2    80000    LINUX    1000
                      b3    c0000    LINUX    1000
                      b4   100000    LINUX    1000
                      b5   140000    LINUX    1000
                      b6   180000    LINUX    1000
                      b7   1c0000    LINUX    1000
                      b8   200000    LINUX    1000
                      b9   240000    LINUX    1000
                     b10   280000    LINUX    1000
                     b11   2c0000    LINUX    1000
                     b12   300000    LINUX    1000
                     b13   340000    LINUX    1000
                     b14   380000    LINUX    1000
                     b15   3c0000    LINUX    1000
                     b16   400000    LINUX    1000
pvm> quit

pvmd still running.
```

For some reason the group server does not start automatically, so we do this manually:

```
b1 % pvmgs &
[1] 6205
```

and now we can fire up the multiplier:

```
b1 % mm.0.6 A B C 16 100
spawned 15 tasks for row 0 on b1 with tids:
        262148
```

```
        262149
        262150
        262151
        262152
        262153
        262154
        262155
        262156
        262157
        262158
        262159
        262160
        262161
        262162
spawned 16 tasks for row 1 on b2 with tids:
        524289
        524290
        524291
.
.
.
spawned 16 tasks for row 15 on b16 with tids:
        4194305
        4194306
        4194307
        4194308
        4194309
        4194310
        4194311
        4194312
        4194313
        4194314
        4194315
        4194316
        4194317
        4194318
        4194319
        4194320
```

and if it suceeds we see:

```
mm.0.6 16 100 task 262147 managing block [0,0] on b1 done successfully after 158 seconds.
b1 %
```

The other tasks output ends up in the log file:

```
b1 % tail /tmp/pvml.794
[t80040000] [t80010] mm.0.6 16 100 task 524304 managing block [1,15] on b2 done successfully after 150 seconds.
[t80040000] [t8000f] mm.0.6 16 100 task 524303 managing block [1,14] on b2 done successfully after 150 seconds.
[t80040000] [t80007] mm.0.6 16 100 task 524295 managing block [1,6] on b2 done successfully after 153 seconds.
[t80040000] [t8000a] mm.0.6 16 100 task 524298 managing block [1,9] on b2 done successfully after 152 seconds.
[t80040000] [t40005] mm.0.6 16 100 task 262149 managing block [0,2] on b1 done successfully after 157 seconds.
[t80040000] [t8000b] mm.0.6 16 100 task 524299 managing block [1,10] on b2 done successfully after 152 seconds.
[t80040000] [t80001] mm.0.6 16 100 task 524289 managing block [1,0] on b2 done successfully after 153 seconds.
[t80040000] [t4000e] mm.0.6 16 100 task 262158 managing block [0,11] on b1 done successfully after 154 seconds.
```

```
[t80040000] [t40004] mm.0.6 16 100 task 262148 managing block [0,1] on b1 done successfully after 158 seconds.
[t80040000] [t40011] mm.0.6 16 100 task 262161 managing block [0,14] on b1 done successfully after 154 seconds.
b1 %
```

794 is the user id of comp381. The thing seems to be quite delicate. Sometimes it just hangs for no reason that i've managed to figure out. When it does this it takes sometime to restart. If we exceed the swap space limitations the whole cluster may have to be rebooted. If we are lucky and the system is still alive. We need to do a reset of pvm.

```
b1 % pvm
pvmd already running.
pvm> reset
pvm> halt
[1]  + Terminated                    pvmgs
b1 %
```

and then a cleanup of the system:

```
b1 % brun -v killall pvmd3
b1 % brun -v rm -f "/tmp/pvm*.794"
```

9.8 Version `mm.0.7.c`

10 So Whats the state of the art?

```
// version 0.6 by ian a. mason
// june 1  @ u.n.e
//
// row i located on machine bi in the area
// /disks/bi/homes/courses/comp381
// in files A.*.* B.*.* and C.*.*
// parse_args check that the paths are
// ok relative to DATAPATH with no .i.j
// extension numbers. these file should
// be created by mkBlockRows

#include <pvm3.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <time.h>
#include <limits.h>
#include "mmlib.h"

#define CLUSTER     1
#define VERSION     "mm.0.6"
// data is stored on each node with relative path
// /scratch/homes/courses/comp381
// with absolute names:
#define DATAPATH    "/disks/%s/a8/homes/courses/comp381/%s"
#define DATA        "/disks/%s/a8/homes/courses/comp381/%s.%d.%d"
#define ENCODING    PvmDataDefault
#define MAXNTIDS    1025
#define MAXROW      100
#define ATAG        2
```

```
#define BTAG        3
#define DEBUG       1
#define PERMS       0666


int parse_args(int, char*[], int*, int*, int *, int []);

int main(int argc, char* argv[]){
    int ntask, info, mytid, mygid;
    int i, j, m, blksize, matrix_size;
    int row, col, up, down;
    int child[MAXNTIDS-1];
    int myrow[MAXROW];
    char hostname[100];
    // paths
    // A = P[0] B = P[1] C = P[2]
    //
    char   P[3][PATH_MAX];
    // file descriptors
    // A = F[0] B = F[1] C = F[2]
    //
    int    F[3];
    // data
    // A = M[0] B = M[1] C = M[2]  TMP = M[3]
    //
    float *M[4];

    //note real time
    int start_time = time(NULL);

    //check everything
    if(parse_args(argc, argv, &m, &blksize, &ntask, F) < 0) exit(0);


    //enroll in pvm daemon & note id
    if((mytid = pvm_mytid()) < 0){
      pvm_perror(argv[0]);
      return -1; }

    //tinkerville
    //too many files?
    //    pvm_setopt(PvmRoute, PvmRouteDirect);


    //join the group
    if((mygid = pvm_joingroup("mmult")) < 0){
      pvm_perror(argv[0]);
      pvm_exit();
      return -1; }

    //spawn the torus
    if ((mygid == 0) && (ntask > 1)) {
      //tailored to the cluster?
```

```
      if(CLUSTER){
char nodes[MAXROW][64];

for(i = 0; i < m; i++)
  sprintf(nodes[i], "b%d", (i%16) + 1);

info = 0;
for(i = 0; i < m; i++){
  int ns;

  ns = pvm_spawn(VERSION,
    &argv[1],
    PvmTaskHost,
    nodes[i],
    ((i == 0) ? m-1 : m),
    &child[((i > 0) ? m*i - 1 : 0)]);

  info += ns;

  if(DEBUG){
    fprintf(stderr,
    "spawned %d tasks for row %d on %s with tids:\n",
    ns,
    i,
    nodes[i]);
    for(j = 0; j < ((i == 0) ? m-1 : m); j++)
      fprintf(stderr, "\t%d\n", child[((i > 0) ? m*i - 1 : 0) + j]);
  }

}
      }
      else {
info = pvm_spawn(VERSION,
 &argv[1],
 PvmTaskDefault,
 (char*)0,
 ntask-1,
 child);       }

      //check torus
      if(info != ntask-1){
perror("spawn failure");
fprintf(stderr,
"only spawned %d of the %d requested tasks\n",
info,
ntask-1);
pvm_lvgroup("mmult");
pvm_exit();
return -1; }

      //what about the orphans?
      //not my bug, their bug
    }
```

```
  // wait at barrier
  if((info = pvm_barrier("mmult",ntask)) < 0)
    pvm_perror(argv[0]);


  //figure out my row
  for (i = 0; i < m; i++)
    myrow[i] = pvm_gettid("mmult",
                          (mygid/m)*m + i);

  //allocate space for the blocks
  for(i = 0; i < 4; i++)
    M[i] = (float*)calloc(sizeof(float),
                          blksize*blksize);

  //bail if calloc fails
  if (!(M[0] && M[1] && M[2] && M[3])) {
    fprintf(stderr,
    "%s: out of memory!\n",
    argv[0]);
    for(i = 0; i < 4; i++)free(M[i]);
    pvm_lvgroup("mmult");
    pvm_exit();
    return -1; }

  //determine my coordinated on the torus
  row = mygid/m;
  col = mygid % m;

  //determine my machine
  gethostname(hostname, 64);

  //make the file names
  for(i = 0; i < 3; i++){
    sprintf(P[i],
    DATA,
    hostname,
    argv[i+1],
    row,
    col);

    //open them
    F[i] = open(P[i],
( i < 2 ?
  O_RDONLY :
  O_WRONLY) );
  }

  //size of virtual matrix
  matrix_size = m * blksize;

  //load in the data
```

```
    for(j = 0; j < 2; j++)
      get_block_row(F[j],
    blksize,
    0,
    0,
    0,
    blksize*blksize,
    M[j]);

    //find out who is above me on torus
    up = pvm_gettid("mmult",
    ((row)?
     (row-1):
     (m-1))*m+col);

    //find out who is below me
    down = pvm_gettid("mmult",
      ((row == (m-1))?
       col:
       (row+1)*m+col));


    //broadcast A's along, and rotate B's up
    //incrementing answer as we go.
    for (i = 0; i < m; i++) {
      if (col == (row + i)%m) {
pvm_initsend(ENCODING);
pvm_pkfloat(M[0], blksize*blksize, 1);
pvm_mcast(myrow, m, (i+1)*ATAG);
block_mult(M[2],M[0],M[1],blksize);  }
      else {
pvm_recv(pvm_gettid("mmult",
    row*m + (row +i)%m),
          (i+1)*ATAG);
pvm_upkfloat(M[3], blksize*blksize, 1);
block_mult(M[2],M[3],M[1],blksize); }

      pvm_initsend(ENCODING);
      pvm_pkfloat(M[1], blksize*blksize, 1);
      pvm_send(up, (i+1)*BTAG);
      pvm_recv(down, (i+1)*BTAG);
      pvm_upkfloat(M[1], blksize*blksize, 1);  }

    //look right (A == C) ?
    for (i = 0 ; i < blksize*blksize; i++)
      if (M[0][i] != M[2][i])
printf("Error a[%d] (%g) != c[%d] (%g) \n",
       i,
       M[0][i],
       i,
       M[2][i]);
```

```
    //close A and B
    for(i = 0; i < 2; i++)
      close(F[i]);


    //write out my answer
    set_block_row(F[2],
  blksize,
  0,
  0,
  0,
  blksize*blksize,
  M[2]);


    //log the details
    printf("%s %d %d task %d managing block [%d,%d] on %s done successfully after %d seconds.\n",
   VERSION,
   m,
   blksize,
   mytid,
   row,
   col,
   hostname,
   time(NULL) - start_time);



    //shut up shop
    for(i = 0; i < 4; i++)
      free(M[i]);

    close(F[2]);

    pvm_lvgroup("mmult");
    pvm_exit();
    return 0;

}


int parse_args(int argc, char *argv[], int *m, int *blksize, int *ntask, int F[]){
  int i, j;
  char   P[3][PATH_MAX];
  char hostname[100];
  gethostname(hostname, 64);

  if ((argc != 6)                       ||
      ((*m   = atoi(argv[4])) <= 0)    ||
      ((*blksize   = atoi(argv[5])) <= 0)){

    fprintf(stderr,
    "Usage: %s matrixA matrixB matrixC m blk\n", argv[0]);
```

```
    return(-1); };

  if(*m > MAXROW){
    fprintf(stderr, "m = %d not valid.\n", *m);
    return(-1); };

  *ntask = (*m)*(*m);

  if ((*ntask < 1) || (*ntask >= MAXNTIDS)) {
    fprintf(stderr, "ntask in parse_args = %d not valid.\n", *ntask);
    pvm_exit();
    return(-1); };


  for(i = 0; i < 3; i++)
    sprintf(P[i],
    DATAPATH,
    hostname,
    argv[i+1]);

  for(j = 0; j < 3; j++)
    if ((F[j] = open(P[j], ((j < 2) ? O_RDONLY : O_WRONLY))) == -1) break;

  if(j < 3){
    fprintf(stderr, "Data not detected at fd %d : %s\n", j, P[j]);
    for(i = 0; i < j; i++)close(F[i]);
    return(-1); }


  for(i = 0; i < 3; i++)close(F[i]);
  return(0);

}
```

11 The Results

```
version spmm (single process)

dimension  seconds minutes
1000       267       4.45
```

```
1600       1224     20.4
4000       24078   401.3
5000       39901   665.0

(a single unix process can't get much bigger than this
size > 3 * 5000^2 * 4 = 60 megabytes)

----------------------------------------------------------
version mm.0.5 on bourbaki (dual processor)
(m^2 tasks, using nfs to read write data to /tmp)

dimension  m  blk seconds  minutes
2000       4  500 2018     33.63


version mm.0.5 on the cluster  (16 processor)
(m^2 task, using nfs to read write data to bourbaki)

dimension  m  blk seconds  minutes
2000       4  500 368      6.1
2400       4  600 719     11.98
2800       4  700 1149    19.15
3200       4  800 1737    28.95
3600       4  900 2429    40.48
4000       4 1000 3333    55.55
8000       4 2000 19733  328.88

----------------------------------------------------------

version mm.0.6 on the cluster
(m^2 tasks, each row on a single machine writing to local disks)

dimension  m  blk seconds  minutes

1600       16 100 125      2.08
2400       16 150 254      4.23
3200       16 200 499      8.37
4000       16 250 876     14.60
4800       16 300 1540    25.67
5600       16 350 2594    43.23
6400       16 400 4728    78.8
7200       16 450 5629    93.82
8000       16 500 7982   133.03
8800       16 550 11683  194.72
9600       16 600 17815  296.92


(size of swap limits larger computations)



mm.0.6 32 10 task 262147 managing block [0,0] on b1 done successfully after 333 seconds.
mm.0.6 32 20 task 262147 managing block [0,0] on b1 done successfully after 310 seconds.
```

References

[1] Al Geist et al. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1996.

[2] G. C. Fox, S. W. Otto, and A. J. G. Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.