Evaluation of Characteristics of Three Programming Paradigms Supported by Existing CM5 Software

Dean Engelhardt

January 15, 1993

1 Introduction

This report details a series of experiments undertaken to gain an impression of the characteristics and performance of the currently available programming paradigms existing on the CM5. These paradigms are the Data Parallel paradigm as exemplified by the C* and CMFortran products, traditional synchronous message passing, as offered by the CMMD library, and the (asynchronous) Active Message paradigm of programming. The experiments we conducted centred around the problem of efficiently implementing a very simple grid based algorithm, of the kind often used in scientific computing to solve partial differential equations at discrete points. The particular algorithm we chose to implement was the common one used to solve Laplace's Equation over a two dimensional grid — this was chosen because it was a simple, yet fairly representative, algorithm which involved a fair amount of communication between elements of the grid: at each iteration of the algorithm, every element averages its four immediate neighbors (that is element $A(i,j)$ averages $A(i-1,j)$, $A(i+1,j)$, $A(i,j-1)$, $A(i,j+1)$); this average replaces the element for the next iteration.

The experiments reported within this document have been carried out using the following versions of software: CMOST version 7.1, CMF version 2.0, CMMD version 2.0, and CMAM version 1.99.

Our experiences with each of the three CM5 supported paradigms are given in separate sections below, followed by some conclusions that can be drawn from this admittedly fairly cursory evaluation of these facilities.

2 CMFortran

There are essentially two ways to implement the Laplace algorithm in CMFortran; either by using the Fortran90 `CSHIFT` function or using the non-Fortran90 `forall` construct. The difference between these two lies essentially in the fact that in the former the programmer is

specifying a conventional style `for` iteration over the grid but allowing the iterations to be evaluated in parallel, while in the second approach the programmer specifies the conceptual construction of a number of temporary grids (which hold shifted copies of the original grid) from which a final answer can be calculated simultaneously at each point.

As stated above, the `forall` method involves specifying an iteration over the grid which can be performed in parallel. The code for the specification of the Laplace iteration is given below; in this code the grid is represented by the two dimensional array `f` which is `maxx` by `maxy` in size. Two variables `i` and `j` conceptually act as iterators over this grid, and a new value for the array elements iterated over is calculated in terms of the current values.

```
      forall (i=2:maxx-1, j=2:maxy-1) f(i,j) = 0.25*(f(i-1,j) +
     &        f(i+1,j) + f(i,j-1) + f(i,j+1))
```

The Fortran90 style specification of the same calculation involves the usage of the `CSHIFT` intrinsic which performs a circular shift on an array along a particular dimension. The CSHIFT function takes three arguments; a source array, a dimension to shift along and a number of elements to shift by. The code for the Fortran90 style description of the Laplace algorithm is shown below. We generate four temporary copies of `f`, each shifted one element either up or down in one of the grid's two dimensions. A new value for the array `f` (really new values for every element of `f`) is specified in terms of these temporary arrays.

```
      where (cmask)
        f = 0.25 * (CSHIFT(F,1,1)+CSHIFT(F,1,-1)+ CSHIFT(F,2,1) +
     &   CSHIFT(F,2,-1))
      endwhere
```

In this example, `cmask` is an array of logical elements of the same extents as the `f` array. The elements of this array are false along the four borders (ie the perimeter) but true everywhere in the interior. The program uses the Fortran90 `where` construct to limit the computation of new values for elements to the non-border elements only; that is, the averaging function is only evaluated at points where the mask array has a true value in its corresponding element (this achieves the same result as the limiting of the iterators in the `forall` specification above went to the range 2 to max-1).

Results

The first experiment we conducted was to perform a single iteration of the Forall program for varying sizes of grids. The motivation behind this was that others had reported some anomalies in CMFortran performance between two similar, but slightly different, sized grids over which the same computation was run. We ran the Laplace iteration over square grids which varied in size from 8×8 to 1024×1024 with samples taken at intervals of 2. Two separate runs were made, one allowing the CMF compiler to generate code to take advantage

of the machine's vector units, once without. The results of these two runs are plotted as figures 1 and 2.

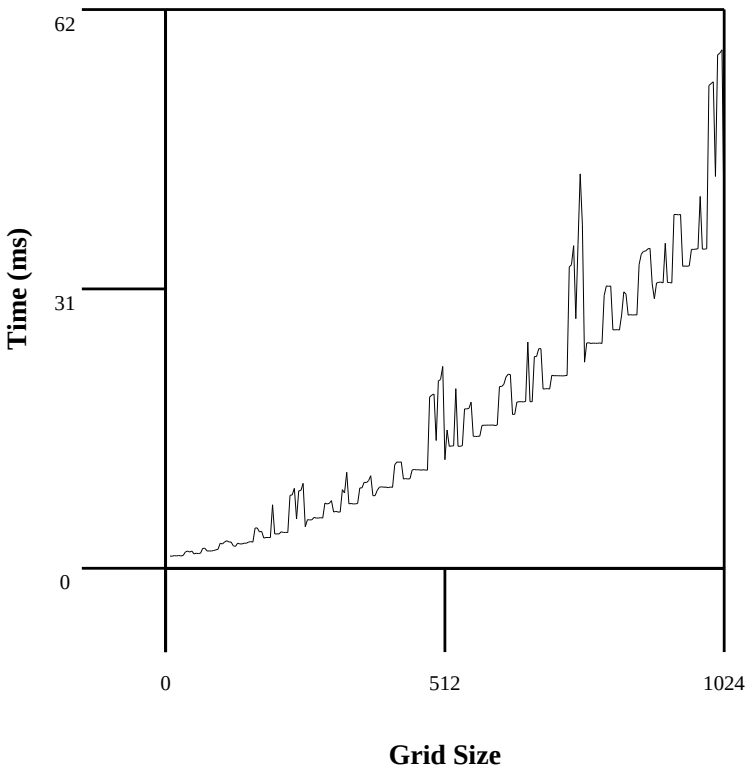

Figure 1: Time required for one iteration of (vectorised) Forall program

The horizontal axis of these graphs denote the side length of the square grid over which the algorithm was run, while the vertical axis represents the time required to complete a single iteration of the algorithm. Notice that, while the general trend is what we would expect (the algorithm is $O(n^2)$ if n is the length of the side of the grid, so we would expect a generally parabolic graph), the graphs are considerably less smooth than one would expect, particularly the graph of the non vectorised program. To highlight the jagged local behaviour in these two graphs, figures 3 and 4 shows enlargements of a region of the previous graphs. In these enlargements, samples are shown for each grid size between 300 and 400.

We have made some investigation into why these apparently regularly spaced perturbations occur, and have ruled out a number of possibilities. We have run versions of the program which time large numbers of iterations rather than the single iteration shown in the graphs above, in an effort to try to eliminate artifacts or inaccuracies in the timers. This produced very little change in the actual trend of perturbations. Similarly it was suggested that some of these anomalies could be attributed to time sharing costs; experimentation with running on very lightly loaded machines made no apparent difference. Also the rather smooth graphs obtained in later experiments seems to rule this out as a possibility. Discussions with Jacek Myczkowski on his brief visit to the department lent great weight to the theory that the jagged nature of the plots shown above is due to the complex algorithm the runtime system (CMRT) uses to determine whether a request for the value of an array can be satisfied on-processor or requires communication. This algorithm apparently handles things differently in the cases where the array is padded (ie the number of array elements isn't a factor of 32).

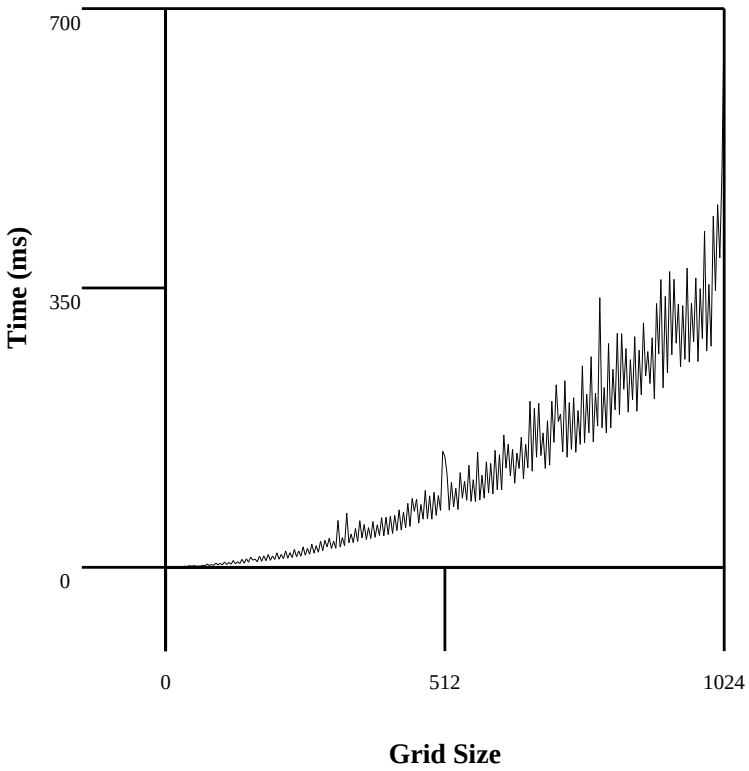

Figure 2: Time required for one iteration of (non vectorised) Forall program

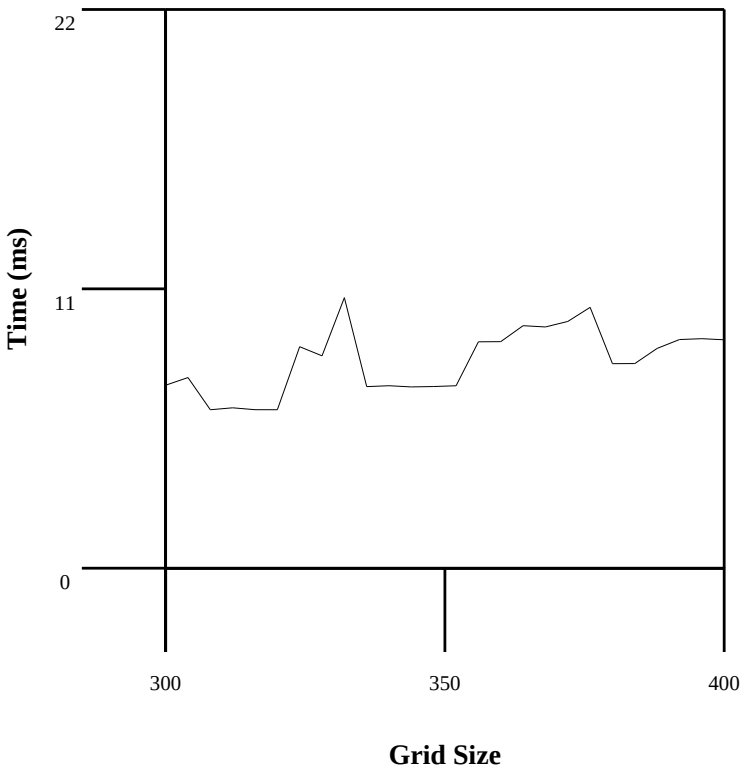

Figure 3: Detail of a region from figure 1

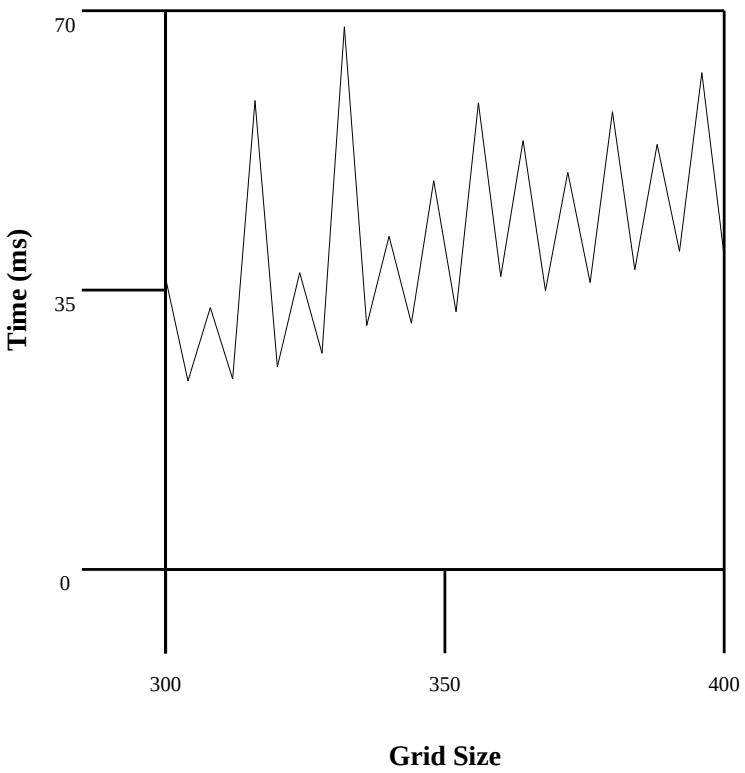

Figure 4: Detail of a region from figure 2

It is impossible to give a more satisfactory explanation for the phenomenon without knowing more about the runtime system and how it operates.

We also investigated the Fortran90 style version of this program (ie a program using `CSHIFT` to see how its performance differed from that of the `forall` program already discussed. The results of the experiments we ran (again one graph of the program allowing the compiler to vectorise the code and one without) are shown in figures 5 and 6

It is intriguing to note that while the non-vectorised `CSHIFT` program runs in about half the time of the non-vectorised `forall` program, the converse is true of the vectorised versions of the code. That is, the vectorised `forall` performed approximately twice as well as the vectorised `CSHIFT`. Again, no explanation can be given for this without recourse to documentation of the runtime system.

3 CMAM — Active Messages

Our implementation of the Laplace iteration using C with communication routines from the CMAM library is considerably more complex than the Fortran programs described above. A source listing is included as an appendix to this document. The program is fairly simple in concept; at the startup of the program, the front end processor calls an initialisation routine which causes the processing nodes to set processor variables which make them view themselves as being conceptually laid out in a grid structure. The main data structures of the program are two identically sized arrays; one of which is the grid from which values are

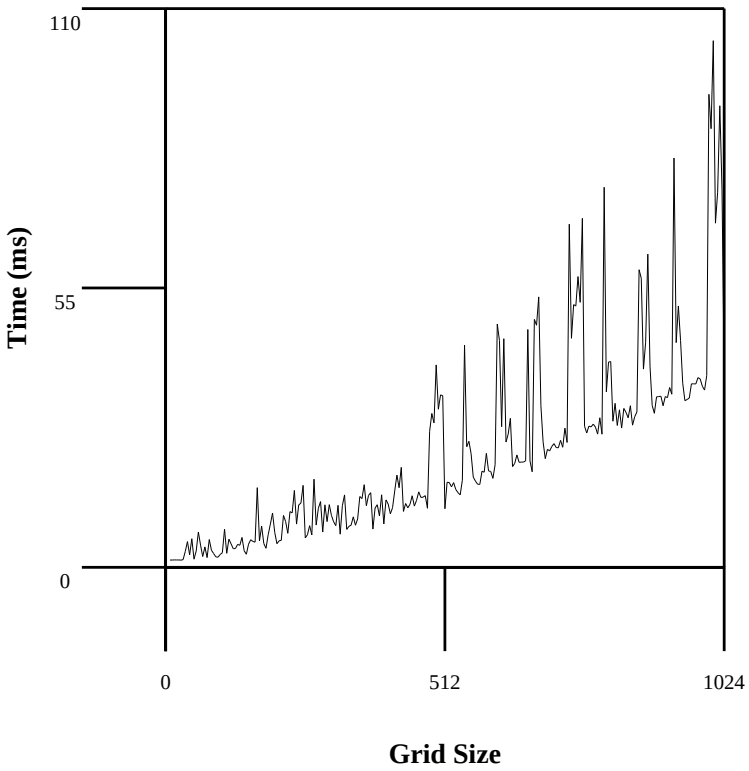

Figure 5: Performance graph for the (vectorised) Fortran90 style program

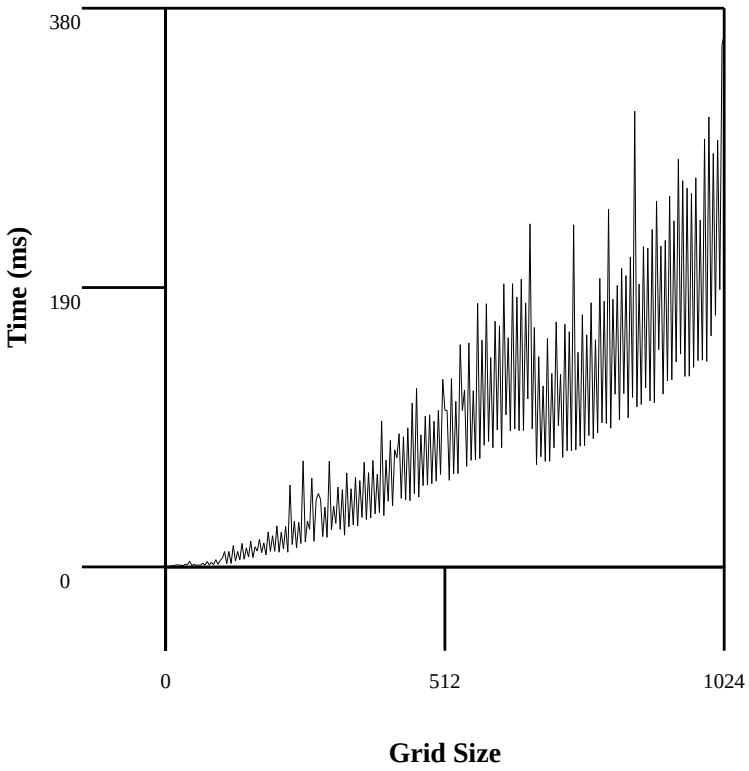

Figure 6: Performance graph for the (non-vectorised) Fortran90 style program

read during the current iteration while the other is where result values are written during the current iteration. Once an iteration has completed, the roles of these two arrays is swapped. Once the processors have been conceptually laid out as a grid, the two arrays are identically partitioned into a number of blocks (each block being a submatrix of the original grid) and these blocks distributed to the processors in such a way that adjacent processors in the grid receive adjacent sub-regions of the two grids. Each processor calculates the leftmost and rightmost column of the original arrays it has in its block as well as the topmost and bottommost rows. Once every processor has performed these operations, they pass through a synchronisation barrier and the front end processor is signalled that they are ready to begin computation.

The actual Laplace iteration over the grids is initiated by the front end calling a procedure and passing the number of iterations which should be performed as well as a pointer to a function to be called (by the front end) when the iteration is complete. The front end processor then suspends, and the processing nodes begin iterating over their local subregions of the grid. At each step in the iteration the processor gathers the values to the left, right, above, below the element being updated in that iteration and stores these values in temporary variables. When a processor needs a value which is not stored in the same subgrid it sends an active message to the processor who owns the subgrid which contains the desired value. The active message sent contains the name of a routine to handle the message when it arrives, the processor number of the processor who sent the message, a code to determine whether the value is needed as a left neighbor, right neighbor, top neighbor or bottom neighbor, and finally the address of the required value in the receiver's subgrid. This message is delivered by CMAM and causes the appropriate handler to begin executing. This handler looks up the requested location in its local subgrid, extracts the value and sends an active message back to the requester. This active message contains the name of a different interrupt handler, the value and the direction code passed to it. The requesting process receives this message and fills the appropriate temporary variable (based on the direction code). If this was the only off-processor value needed to calculate a new value for the element in question then the calculation will proceed and the resultant value be stored in the local subgrid of the second array. If this isn't the last of-processor value required then other active messages would already have been despatched to the appropriate processors requesting the desired values, so the processor waits for the replies to these messages to arrive and checks again to see whether it now has all the data it needs to compute the new value.

Figures 7 and 8 show this remote value fetching protocol diagrammatically for a small region of a processor grid. Figure 7 shows processor 10 gathering the four neighboring values it requires to calculate a new value for it's local top left element. Note that two of these required values are located on the same processor so require no communication to fetch. The other two values, however, need to be obtained from other processors (the left neighbour value from processor 9 and the above neighbour value from processor 2). Hence it sends out two active messages requesting values; each of these contains (in order on the diagram) fields containing the interrupt handler to handle the message, the number of the sending processor, a code indicating what relation the required value is to the element being computed (0 represents left neighbour, 2 represents above neighbour), and the receiving processor's local address for the value to be fetched. Note that these two messages may or may not be sent off simultaneously. Figure 8 shows the receiving processors being interrupted and looking up the requested values (shown as filled and empty circles) and sending active messages back

to the requesting processor; the fields in these messages are handler name, relationship code and value. Again these messages are sent as the others are received, so the two receivers will most likely not be sending at the same time. When all outstanding requests have been replied to, processor 10 computes a new value for the array element being updated and moves on to the next element in its local block of the array.

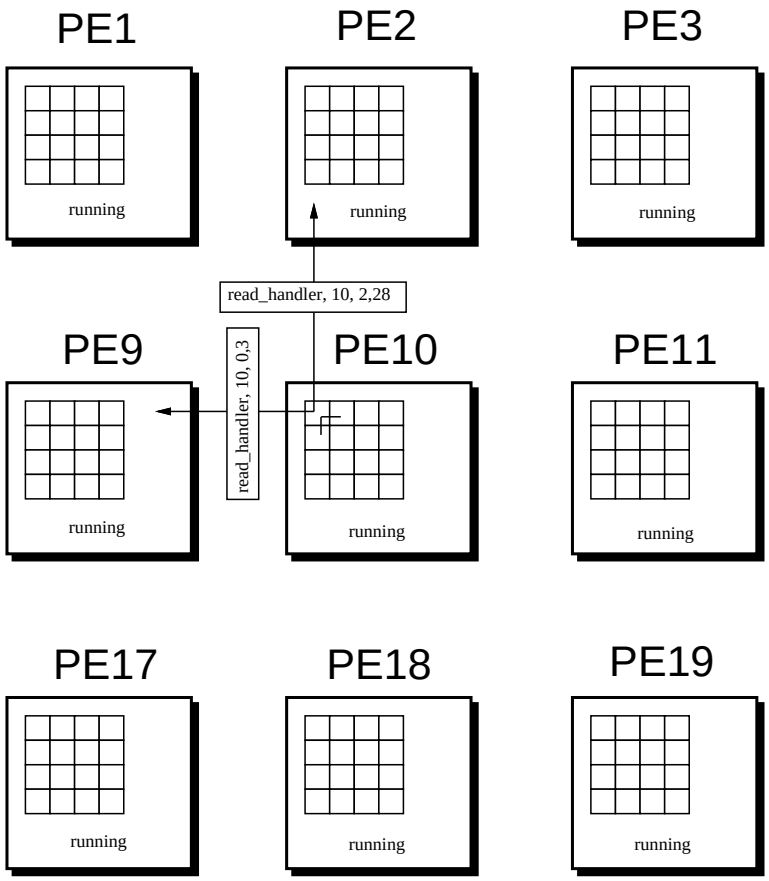

Figure 7: Processor 10 sending out requests for foreign values

It is worthwhile noting that in this program the processors are acting very much independently and the patterns of communication between processors is essentially nondeterministic.

Results

We ran the CMAM program under identical circumstances to the Fortran programs already described, over the same range of grid sizes (8×8 to 1024×1024) and recorded the time required to complete a single iteration of the algorithm. The results we obtained are shown plotted as figure 9.

The most notable thing about this graph is its smoothness in comparison to the Fortran programs' profiles. The performance of this program is much more predictable although there are still some slight perturbations from the smooth parabola one would expect. The performance of the CMAM program is slightly worse than the Fortran 90 style non-vectorised

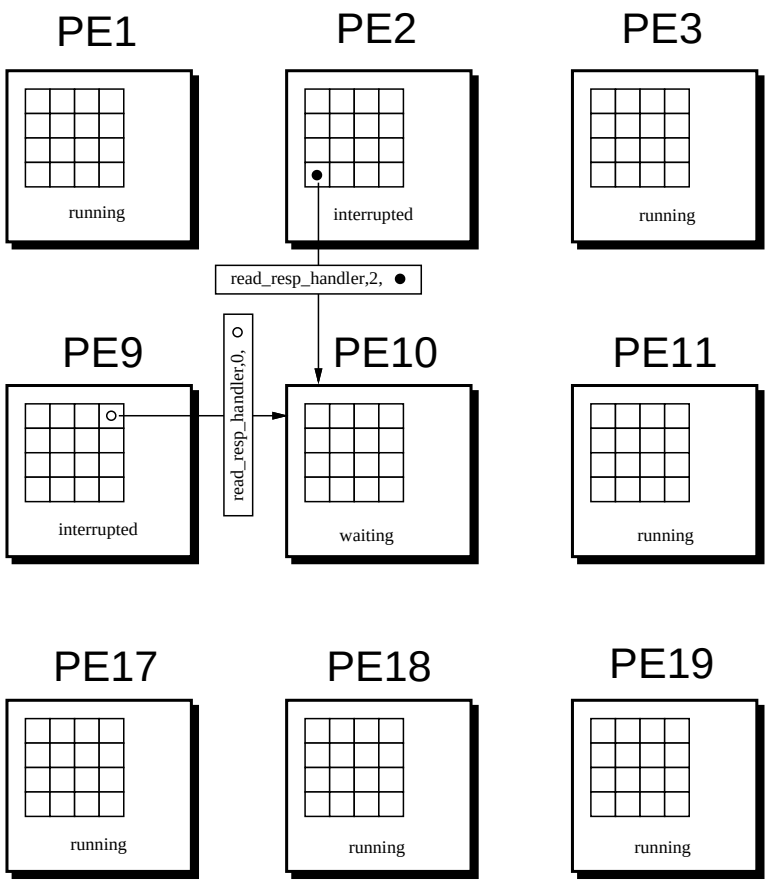

Figure 8: Processors 2 and 9 sending replies to processor 10's requests

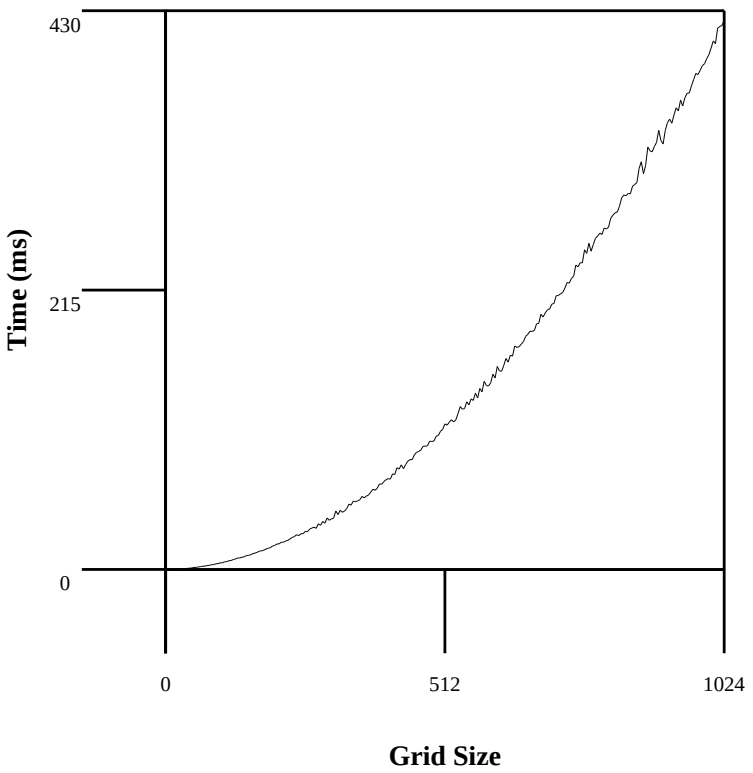

Figure 9: Time required for one iteration of CMAM program

program, and roughly on par with the `forall` non-vectorised program. The vectorised versions of these programs are several times faster. It should be remembered that this program uses the nodes purely as SPARC's each running their own C program; hence they can make no use of the vector unit functionality.

4 CMMD

The program we constructed to perform the Laplace iteration using the CMMD message passing library were originally written using version 1 of the library which only provided proper support for synchronous message passing. We have recompiled these programs under the new version of CMMD to produce the results included below. We have not, however, rewritten the program to utilise the new support for asynchronous messages that the new product provides. For completeness sake this should be done in the future and the results of the asynchronous program included in this section.

The CMMD program is in some ways similar to the CMAM program described above. Like the CMAM program, source code is appended to this document rather than in this section. We will here discuss in more abstract terms the operation of the algorithm. Like the CMAM program the CMMD program begins by initialising the nodes and partitioning two arrays over the conceptual grid of processors. The initialisation also sets local variables denoting the extents of the local subgrid. Additionally during this setup phase, each processor is given a colour based on where it is in the grid of processors. This colouring will determine what

the processor does at different stages of the computation. The colours are allocated as a chess board (see figure 10).

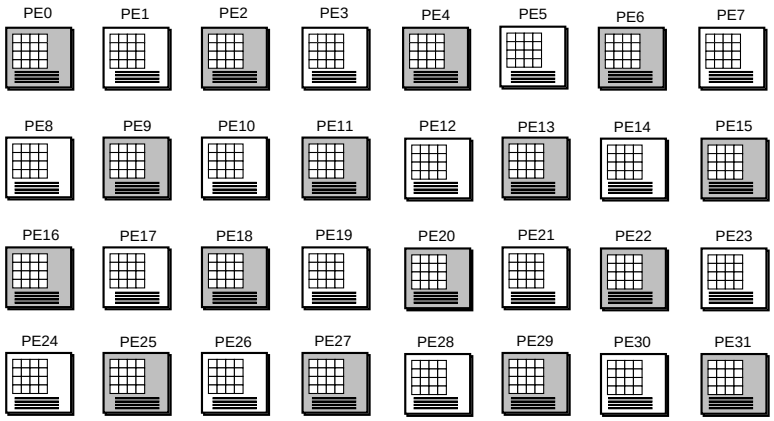

Figure 10: Pattern of processor colouring in the CMMD program

This program differs from the CMAM program in that rather than requesting an off-processor value when it is required, it transfers whole edges which will be needed for later computation before any iteration over the grid begins. Thus it essentially separates the communications section of the program and the computation section. The main motivation for trying this originally lay in the fact that to swap whole edges prior to computation was that it greatly reduced the number of synchronous transfers which needed to be scheduled (the nature of these transfers mean they must be scheduled statically, and a minor error in scheduling will almost certainly cause the program to deadlock). This relatively regular communications pattern should be more efficient than the rather random pattern occurring in the CMAM program.

The edge transferring protocol described above is implemented as follows: each processor maintains four extra buffers in addition to the memory it has allocated for the subgrids of the two arrays. These four buffers will eventually hold copies of edges adjacent to the leftmost and rightmost columns on processor and the topmost and bottom most rows on processor. The transfer of edges into neighboring processors' buffers takes place in four stages of synchronous swapping (the program uses the routine `CMMD_swap_v` to swap edges between processors). During the first stage, depicted in figure 11 every white processor which has a right neighbour engages in a rendezvous with its right neighbour during which its rightmost edge is transmitted and its right neighbours leftmost edge is received and placed in the appropriate buffer. Similarly during this phase, every black processor with a left neighbour transmits its left edge and receives its left neighbours rightmost edge.

Stage two of the protocol is the reverse of stage one; white processors transmit their leftmost edges to their left neighbours while receiving their left neighbours rightmost edge. This is shown in figure 12.

The third and fourth phases of the protocol are vertical analogies of phases one and two respectively. That is top and bottom edges of adjacent processor local memories are exchanged with direction determined by processor colouring.

Once these four phases of edge transfer are complete (with each individual phase necessarily being finished by all processors before the next is permitted to start), every processor now has

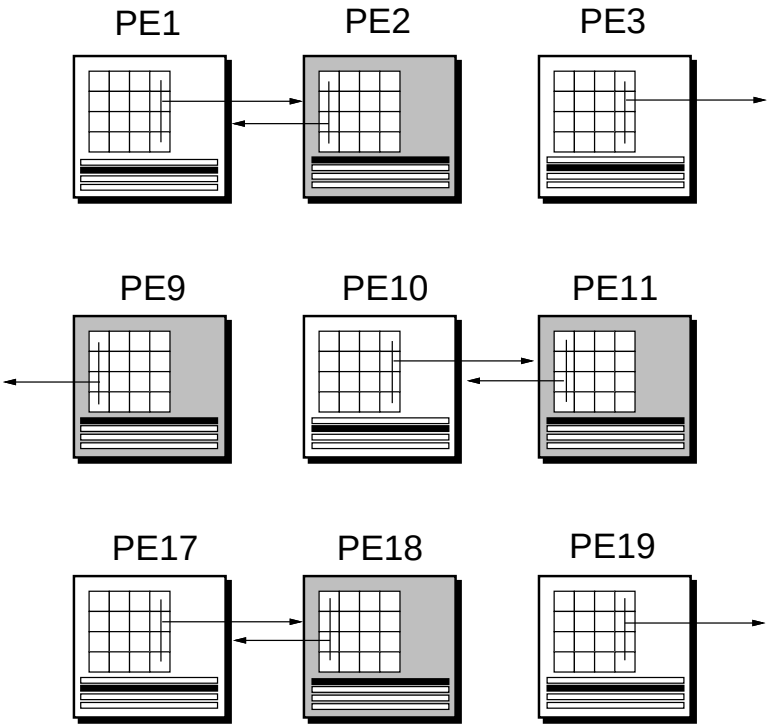

Figure 11: Stage One of the edge swapping protocol

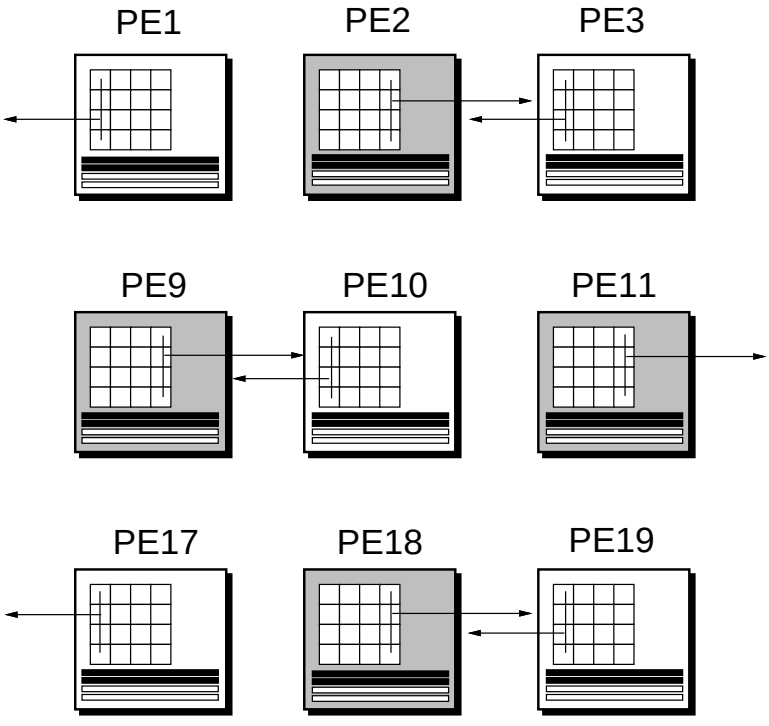

Figure 12: Stage Two of the edge swapping protocol

all the values it needs to iterate through its local subgrid on processor, hence the computation can proceed without further communication. If more than one iteration is being run over the grid, the edge communication protocol described above is performed at the beginning of each iteration to get copies of the latest foreign edge values on processor.

Results

Figure 13 below shows the execution times for grids of between 8×8 and 1024×1024 under CMMD 2.0. These times are roughly half of the corresponding CMAM times and compare favourably with the non-vectorised CMFortran programs. Once again this program does not make use of the vector units on the processing elements, so predictably performs significantly worse than the vectorised Fortran programs.

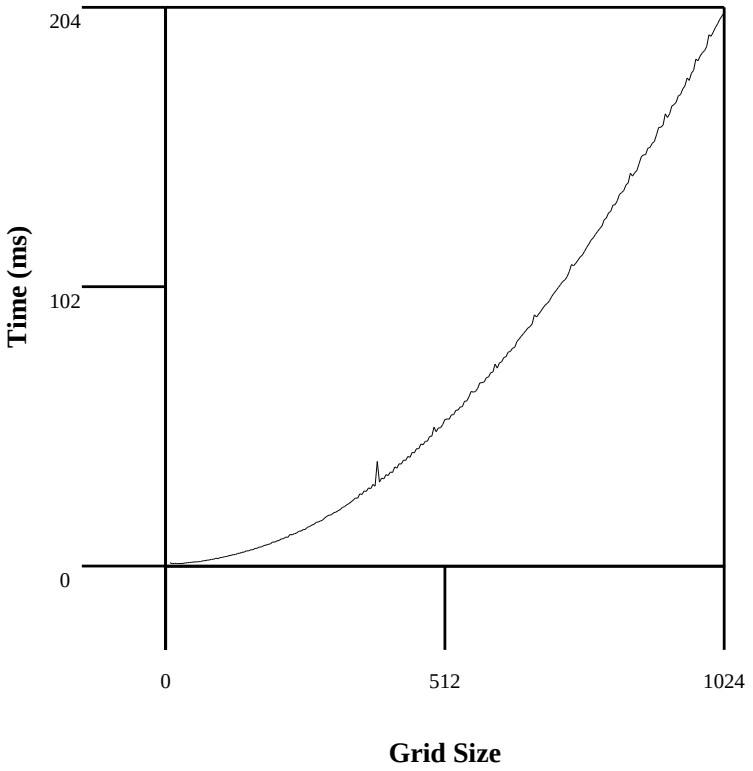

Figure 13: Time required for one iteration of the CMMD 2.0 program

5 Conclusion/Future Work

It would be dangerous to draw any general conclusions from the experiments we have performed to date, since they deal with but a single problem to be solved. We can, however, clearly point out some impressions of relative performance and predictable scaling of performance as they relate to the class of grid based cooperative computations.

Firstly, from the figures we have collected there seems to be a considerable performance gain

to be had by using the CM5's vector processors. The two CMFortran programs we wrote, the `forall` program and the `CSHIFT` program, ran considerably faster when compiled to use the vector units; the former ran about 6 times faster while the latter ran 3 times faster. The CMMD and CMAM programs, which did not use the units, performed slightly better than the non-vectorised CMFortran programs but nowhere near as well as the vectorised versions of the same programs. To better compare the different paradigms it would be desirable to produce hand-vectorised CMMD and CMAM programs (with specific instructions for the vector units embedded in the C source) and evaluate the performance characteristics of such programs in comparison to the vectorised CMFortran programs.

A notable observation that stands out when viewing the performance profiles for the different programs we have produced is that the CMFortran programs have a considerably more jagged profile than the C/Message Passing programs. These wide variances in performance between similarly sized grids can not be put down to artifacts of time sharing or system overhead since if this was the case then all programs we have written would exhibit such characteristics. It is theorised (a theory supported by Jacek Myczkowski when he visited our department) that the cause of this unpredictable scaling of performance lies in the algorithm the CMRT runtime system uses to determine how data must be moved around the system to compute a result. It seems that in cases where the array is padded (that is, the user array does not evenly divide over processing elements requiring a slightly larger array to be allocated and partially padded with unimportant data) the decision algorithm takes considerably longer to execute, hence the peaks in the graph. It is impossible to hypothesize further than this without a considerably better understanding of the inner workings of the CMRT runtime system. If such documentation can be attained from Thinking Machines we would like to look into this matter further.

An unexpected result to arise from this series of experiments was the fact that the CMMD program ran faster than the CMAM program, despite the claims made by the CMAM developers that their message passing system is considerably faster than that provided by Thinking Machines. Admittedly the algorithm we used for the CMMD programs conducted bulk data transfers rather than requesting individual values; this bulk transfer can most likely take advantages of hardware optimisations that the other, more chaotic pattern of data motion cannot. To investigate further whether our CMAM programs were being unfairly penalised for this type of communication pattern we developed a modified CMAM program which exchanged edges prior to the start of a grid iteration in a similar manner to the CMMD program we have described. This modified CMAM program used the bulk data primitives `CMAM_open_segment` and `CMAM_xfer`. This program performed approximately 25% faster than the single value transfer CMAM program, but was still considerably slower than the CMMD edge transfer program.

A Source Code for CMFortran Programs

The source code found in this appendix and those that follow is available in electronic form via anonymous ftp to `ftp.cs.adelaide.edu.au`. The files appear in a single compressed tar file `pub/cm5laplace.tar.Z`.

This program implements the Laplace iteration in CMFortran using the Non-Fortran 90 `forall` statement.

```
      subroutine doit (maxx,maxy,acc1,acc2)
      integer maxx,maxy
      double precision acc1,acc2
      intent (inout)::acc1,acc2

      INCLUDE '/usr/include/cm/timer-fort.h'

      real f (maxx,maxy)
      logical cmask (maxx,maxy)
      integer i,j

cmf$  layout f(:news,:news), cmask(:news,:news)

      cmask = .false.
      cmask (2:maxx-1,2:maxy-1) = .true.

      f = 2.0
      f(:,maxy) = 1.0
      where (cmask) f = 0.0

      call cm_timer_clear(0)
      call cm_timer_start(0)

      do (50) times

      forall (i=2:maxx-1, j=2:maxy-1) f(i,j) = 0.25*(f(i-1,j) +
     &        f(i+1,j) + f(i,j-1) + f(i,j+1))

      enddo

      call cm_timer_stop(0)
      acc1 = acc1 + CM_timer_read_elapsed(0)/50.0
      acc2 = acc2 + CM_timer_read_cm_busy(0)/50.0

      return
      end


      program main

      integer x,y
      double precision acc1,acc2

      x = 8
      y = 8
```

```
      do while (x <= 1024)

      acc1 = 0.0
      acc2 = 0.0

      call doit (x,y,acc1,acc2)

      print *, x,acc1*1000,acc2*1000

      x = x + 4
      y = y + 4

      enddo

      end
```

The Fortran 90 version of the same program replaces the `forall` part of the program with:

```
      where (cmask)
        f = 0.25 * (CSHIFT(F,1,1)+CSHIFT(F,1,-1)+ CSHIFT(F,2,1) +
     &   CSHIFT(F,2,-1))
      endwhere
```

B Source Code for CMAM Program

The code for the CMAM program consists of two C files; one for the program running on the Control Processor and another for the program running on the individual Processing Elements.

The Control Processor code is as follows:

```
#include <stdio.h>
#include <cmam/cmam.h>

static volatile int w=0;
static volatile double time = 0;

static void done ()
{
  w = 1;
}

static void set_time(double t) {time =t; w = 1;}

void iter (int n)
{
  int bx,by;

  if (n%8 == 0) bx = n/8;
  else bx = (n/8) + 1;

  if (n%4 == 0) by = n/4;
  else by = (n/4) + 1;

  init(8,bx,by,n-1,n-1,done);
  CMAM_wait(&w,1);

  w = 0;
  iterate (1,set_time);

  CMAM_wait(&w,1);

}

void main(int argc, char **argv)
{
  int i;

  CMAM_enable();
  CMAM_enable_barrier();

  for (i=atoi(argv[1]);i<=atoi(argv[2]);i+=4)
    {
      iter (i);
      printf ("%d  %.5f  %.5f\n",i,time*1000,0.0);
    }
```

```
  CMAM_disable();
  exit(0);

}
```

The Processing Element code is:

```
#include <cmam/cmam.h>
#include <cm/timers.h>
#include <cm/cmna.h>

#define BUFFER_SIZE 40000

static float A [BUFFER_SIZE];
static float B [BUFFER_SIZE];
static int T1[BUFFER_SIZE];
static int T2[BUFFER_SIZE];
static int pro_arr_x,block_x,block_y,max_x,max_y,me;
static int left_border,right_border,top_border,bottom_border;
static int sw = 0;

#define ijn(i,j) ((int) ((j/block_y)*pro_arr_x + (i/block_x)))
#define ijo(i,j) ((int) ((j%block_y)*block_x + (i%block_x)))
#define noi(n,o) ((int) ((n%pro_arr_x)*block_x + (o%block_x)))
#define noj(n,o) ((int) ((n/pro_arr_x)*block_y + (o/block_x)))

static double time (int t)
{CM_TIME ct = {0,t}; return CM_ni_time_in_sec(ct);}


void CMPE_init (int p, int x, int y, int mx, int my, void (*done)())
{
  static int i;

  pro_arr_x = p;
  block_x = x;
  block_y = y;
  max_x = mx;
  max_y = my;
  me = CMAM_self_address;

  srandom (me);

  for (i=0;i<(block_x*block_y);i++)
    {
      A[i] = (float) (random()%1000) ;
      T1[i] = noi(me,i);
      T2[i] = noj(me,i);
    }

  left_border = noi(me,0);
  right_border = noi(me,(block_x-1));
  top_border = noj (me,0);
```

```
  bottom_border = noj (me, (block_x*block_y-1));

  CMAM_barrier (0);

  if (me == 0) CMAM_host_4 (done);

}


static float left,right,up,down;

void read_resp_handler(int ty, float data)
{
  switch (ty){
  case 0: {left = data;
           break;}
  case 1: {right = data;
           break;}
  case 2: {up = data;
           break;}
  case 3: {down = data;
           break;}
    }
}


void read_handler(int ret_node, int ty, int index)
{
  if (sw)
   CMAM_reply_4(ret_node,(void *)read_resp_handler,ty,A[index]);
  else
   CMAM_reply_4(ret_node,(void *)read_resp_handler,ty,B[index]);
}


inline void wait()
{
 while (left < 0.0 || right < 0.0 || up < 0.0 || down < 0.0)
  CMAM_poll();
}


CMPE_iterate (int it, void (*set_time)())
{
  static int i,j,index1,index2,tmp;
  static float *base,*other;
  static double t2;

  CMNA_timer_clear(0);
  CMNA_timer_start(0);

  for (j = 0;j<it;j++)
    {
     if (sw) {base = A; other = B;}
     else {base = B; other = A;}

     for (i=0;i<(block_x*block_y);i++)
```

```
      {
         left = right = up = down = -1.0;
         index1 = T1[i];
         index2 = T2[i];

         if (index1 > left_border)
           {left = base[i-1];}
         else
           { CMAM_4(me-1,read_handler,me,0,
                    (index2+1-top_border)*block_x-1);}


         if (index1 < right_border)
           {right = base[i+1];}
         else
           { CMAM_4(me+1,read_handler,me,1,
                    (index2-top_border)*block_x);}


         if (index2 > top_border)
           {up = base[i-block_x];}
         else
           { CMAM_4(me-pro_arr_x,read_handler,me,2,
                    (block_y-1)*block_x+index1-left_border);}


         if (index2 < bottom_border)
           {down = base[i+block_x];}
         else
           { CMAM_4(me+pro_arr_x,read_handler,me,3,
                   (index1-left_border));}


         wait();

         other[index1,index2] = (float) (left+right+up+down) / 4;

       }
      }

      CMAM_barrier (0);

      if (sw) sw = 0;
      else sw = 1;

   }

    CMNA_timer_stop(0);
    t2 = CMNA_timer_busy(0);
    if (me == 0) CMAM_host_4(set_time,t2);

}
```

C Source Code for CMMD Program

Once again we include two C files here, one for the Control Processor and one for the Processing Elements. The CP program is in this case very similar to that given above for CMAM with some calls to communications routines altered slightly.

The Control Process source is as follows:

```
#include <stdio.h>
#include <sys/types.h>
#include <cm/cmmd.h>

static double time = 0;

void iter (n)
int n;
{
  static int bx,by,dummy;

  if (n%8 == 0) bx = n/8;
  else bx = (n/8) + 1;

  if (n%4 == 0) by = n/4;
  else by = (n/4) + 1;

  init(8,bx,by,n-1,n-1);
  CMMD_receive (0,0,&dummy,sizeof(int));

  iterate (1);

  CMMD_receive(0,0,&time,sizeof(double));

}


void main(argc,argv)
int argc;
char **argv;
{
  int i;

  CMMD_enable();

  for (i=atoi(argv[1]);i<=atoi(argv[2]);i+=4)
    {
      iter (i);
      printf ("%d  %.5f  %.5f\n",i,time*1000,0.0);
    }

  CMMD_disable();

}
```

The individual Processing Elements run the following program:

```
#include <cm/cmmd.h>
#include <cm/timers.h>

#define BUFFER_SIZE 40000

static float A [BUFFER_SIZE];
static float B [BUFFER_SIZE];
static int T1[BUFFER_SIZE];
static int T2[BUFFER_SIZE];
static int pro_arr_x,pro_arr_y,block_x,block_y,max_x,max_y,me;
static int left_border,right_border,top_border,bottom_border;
static int proc_x, proc_y, parity;
static int sw = 0;

#define ijn(i,j) ((int) ((j/block_y)*pro_arr_x + (i/block_x)))
#define ijo(i,j) ((int) ((j%block_y)*block_x + (i%block_x)))
#define noi(n,o) ((int) ((n%pro_arr_x)*block_x + (o%block_x)))
#define noj(n,o) ((int) ((n/pro_arr_x)*block_y + (o/block_x)))

static double time (t)
int t;
{CM_TIME ct;
 ct[0] = 0;
 ct[1] = t;
 return CM_ni_time_in_sec(ct);}


void CMPE_init (p,x,y,mx,my)
int p,x,y,mx,my;
{
  static int i, dummy;

  pro_arr_x = p;
  pro_arr_y = CMMD_partition_size()/p;
  block_x = x;
  block_y = y;
  max_x = mx;
  max_y = my;
  me = CMMD_self_address();

  proc_x = me%pro_arr_x;
  proc_y = me/pro_arr_x;

  parity = (proc_x+proc_y)%2;

  srandom (me);

  for (i=0;i<(block_x*block_y);i++)
    {
      A[i] = (float) (random()%1000) ;
      T1[i] = noi(me,i);
      T2[i] = noj(me,i);
    }
```

```
  left_border = noi(me,0);
  right_border = noi(me,(block_x-1));
  top_border = noj (me,0);
  bottom_border = noj (me, (block_x*block_y-1));

  CMMD_sync_with_nodes ();

  if (me == 0) CMMD_send (CMMD_host_node(),0,&dummy,sizeof(int));

}


static int left, right, up, down;


CMPE_iterate (it)
int it;
{
  static int i,j,k,index1,index2,tmp;
  static double t2;
  static float *base,*other;
  static float T_L[1100],T_R[1100],T_U[1100],T_D[1100];

  CMNA_timer_clear(0);
  CMNA_timer_start(0);

  for (j = 0;j<it;j++)
    {
     if (sw) {base = A; other = B;}
     else {base = B; other = A;}

     if (parity == 0 && proc_x<(pro_arr_x-1))
       CMMD_swap_v(me+1,T_R,sizeof(float),1,block_y,
                   base+(block_x-1)*sizeof(float),sizeof(float),block_x,
                   block_y);


     if (parity == 1 && proc_x>0)
       CMMD_swap_v(me-1,T_L,sizeof(float),1,block_y,
                   base,sizeof(float),block_x,block_y);


     CMMD_sync_with_nodes ();

     if (parity == 0 && proc_x>0)
       CMMD_swap_v(me-1,T_L,sizeof(float),1,block_y,
                   base,sizeof(float),block_x,block_y);

     if (parity == 1 && proc_x<(pro_arr_x-1))
       CMMD_swap_v(me+1,T_R,sizeof(float),1,block_y,
                   base+(block_x-1)*sizeof(float),sizeof(float),block_x,
                   block_y);

     CMMD_sync_with_nodes ();

     if (parity == 0 && proc_y<(pro_arr_y-1))
       CMMD_swap_v(me+pro_arr_x,T_D,sizeof(float),1,block_x,
```

```
              base+(block_y-1)*block_x*sizeof(float),sizeof(float),1,
              block_x);

if (parity == 1 && proc_y>0)
  CMMD_swap_v(me-pro_arr_x,T_U,sizeof(float),1,block_x,
              base,sizeof(float),1,block_x);

CMMD_sync_with_nodes ();

if (parity == 0 && proc_y>0)
  CMMD_swap_v(me-pro_arr_x,T_U,sizeof(float),1,block_x,
              base,sizeof(float),1,block_x);


if (parity == 1 && proc_y<(pro_arr_y-1))
  CMMD_swap_v(me+pro_arr_x,T_D,sizeof(float),1,block_x,
              base+(block_y-1)*block_x*sizeof(float),sizeof(float),1,
              block_x);

CMMD_sync_with_nodes ();


for (i=0;i<(block_x*block_y);i++)
  {
     index1 = T1[i];
     index2 = T2[i];

     if (index1 != 0 && index1 < max_x && index2 != 0 && index2 < max_y)
       {
          if (index1 > left_border)
            {left = base[i-1];}
          else
            {left = T_L[index2];}


          if (index1 < right_border)
            {right = base[i+1];}
          else
            {right = T_R[index2];}


          if (index2 > top_border)
            {up = base[i-block_x];}
          else
            {up = T_U[index1];}


          if (index2 < bottom_border)
            {down = base[i+block_x];}
          else
            {down = T_D[index1];}


        other[index1,index2] = (float) (left+right+up+down) / 4;

       }
  }
```

```
        CMMD_sync_with_nodes ();

        if (sw) sw = 0;
        else sw = 1;

    }

     CMNA_timer_stop(0);
     t2 = CMNA_timer_busy(0);
     if (me == 0) CMMD_send (CMMD_host_node(),0,&t2,sizeof(double));

}
```