Experiences with CDPEAC: Vectorising a Laplace Iteration on the CM5

Dean Engelhardt and Andrew Wendelborn*
University of Adelaide, South Australia

May 31, 1993

Abstract

We motivate and describe an algorithm we have designed to perform an efficient Laplace iteration on the CM5 using a synchronous message passing paradigm and programming the vector processing hardware directly. Some notes on the implementation of the algorithm and some performance statistics are included. These show our implementation to be more efficient than the equivalent program written with higher level paradigms like CM Fortran.

1 Introduction

Thinking Machines' CM5 supercomputer [5] is undeniably a potentially very powerful machine. It is also, however, a very complex one architecturally. RISC processing nodes, custom network interface boards, and custom vector processing units/memory controllers, combine to define a complicated environment which requires significant specialist knowledge to program at its lowest level. Fortunately a number of higher level software abstractions embodied by languages and libraries supplied by Thinking Machines negate the necessity of this knowledge by offering programmers far simpler concepts to reason about. A number of different levels of software abstraction are provided from relatively low level paradigms such as synchronous message passing, to quite high level paradigms such as CM Fortran [4] (a superset of Fortran 90). The question quite naturally arises as to what performance cost one pays in using such higher level platforms, and particularly how efficient different paradigms are in comparison to each other.

In a previous report [1] we have evaluated the performance of a series of implementations of a single algorithm (a simple Laplace iteration over a 2 dimensional grid) under a number of the different programming paradigms currently supported on the CM5. Our motivation for performing these experiments lay chiefly in evaluating a variety of paradigms with an eye towards using one as object code for a compiler we are constructing for a simple functional language. The paradigms we chose to look at embodied the data parallel model (CM

*Authors' Address: Department of Computer Science, University of Adelaide, Adelaide 5005, Australia. E-mail: {dean,andrew}@cs.adelaide.edu.au

Fortran), conventional synchronous message passing (CMMD [6]) and the active message paradigm (CMAM [8]). In the series of experiments we reported, we were able to establish that without the use of the vector processing hardware, examples hand coded using low level paradigms such as CMMD outperformed programs using high level paradigms like CM Fortran[1] by about 30%. It was, at that time, impossible for a similar comparison to be performed with the vector unit hardware utilized. The reason for this lay in the fact that the task of building a vectorised version of a message passing program requires the programmer to explicitly address the operation of the vector hardware at the assembly language level. This requires a knowledge of the assembly language of the vector chips and the peculiarities of the memory layout of the units. At the time of our earlier report we had none of the expertise required to build a vectorised message passing program.

This document is an account of our experiences with constructing just that; a hand-vectorised version of the (CMMD) message passing implementation of the simple Laplace iteration we used for previous comparisons. In Section 2 we briefly describe the non-vectorised algorithm we began with, concentrating in particular on the network communications involved in moving data from one node to another. Such internode communication is largely unchanged in the vectorised version. Section 3 then describes some basic details of the CM5's node architecture and vector units. This knowledge is necessary to an understanding of the vectorised algorithm we constructed (as described in Section 4) and notes on its implementation we present as Section 5. Finally, Section 6 presents a number of conclusions we can draw from our work.

2 Previous Work

Since the differences between the non-vectorised and vectorised Laplace programs lie exclusively in details of data motion on the individual nodes and vector arithmetic on the vector units, the code to move data from one node to another is essentially identical between the two. Hence it is worthwhile to briefly describe the non-vector algorithm here.

Firstly the processors of the machine are conceptually laid out in a two dimensional grid, that is each processor is granted an x and y co-ordinate in a 2D grid of processors. We then take the grid of values over which the iteration is to be performed and "lay it over" the processor grid. This involves dividing the original two dimensional grid into a number of two dimensional subgrids equal to the number of processors in the processor array. Each processor is assigned one of these subgrids. The particular scheme of grid partitioning we use divides the grid both vertically and horizontally and assigns the resulting subgrids on the grounds of position within the original grid. That is, the top left subgrid will be assigned to the top left processor in the processor array.

A simple example of this partitioning algorithm will clarify this somewhat. Suppose we have a grid $A_{i,j}$ which is 64 by 64 elements. We intend to divide that over a machine with 32 processors. Firstly we arrange the processors into a grid; of the several possibilities we choose a processor grid which is 8 processors in horizontal extent by 4 in vertical extent. Processor 0

[1] Performance results we cite for CM Fortran are from a beta release of version 2.0 of the compiler. Hence they do not necessarily reflect the performance of the commercial distribution of the CM Fortran system.

becomes the top left element of this grid, processor 1 it's right neighbour, processor 8 it's neighbour below. We divide the 64 by 64 grid into 32 subgrids each of which is 8 elements in horizontal extent and 16 in vertical extent. Processor 0 is assigned the subgrid which contains the elements $A_{1,1}$, $A_{1,2}$, ..., $A_{1,8}$, $A_{2,1}$, ..., $A_{16,8}$. Processor 1 receives the grid with grid elements $A_{1,9}$ and $A_{16,16}$ as upper left and lower right corners respectively.

The Laplace iteration requires every element in the grid to receive the average of its four neighbours at each iteration. For elements not on an edge of a subgrid this is simple, the element's four neighbouring elements are part of the same subgrid, hence on the same processor. For border elements, however, interprocessor communication is required for a new value to be computed for the element.

Experimentation with different methods for achieving this data motion under the various message passing paradigms we investigated, indicated that the most time-efficient method of performing such data motion was through an algorithm which swaps entire subgrid edges between processors via synchronous communication. The algorithm works as follows: each processor in the processor grid is coloured either black or white depending on its position in the processor grid. Colourings are assigned in a manner which paints the processor grid in the chess board pattern shown in figure 1.

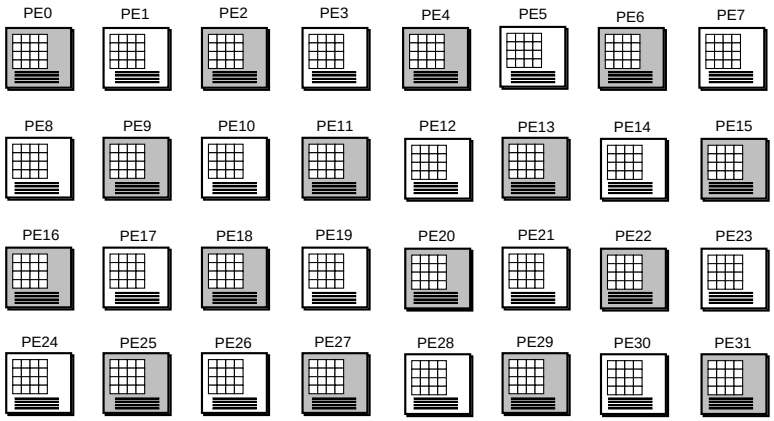

Figure 1: Pattern of processor colouring in the CMMD program

The edge swap algorithm supposes that each node allocates four buffers in which to receive copies of the four edges bordering on its subgrid. The communications involved in filling these buffers proceeds as a four step protocol. In step one, all black processors except those on the right hand edge of the processor grid engage in a rendezvous with their right neighbours during which the rightmost column of their subgrid is sent to the neighbour and the neighbour's leftmost column is received into a buffer. Once all these edge swappings have been completed the algorithm proceeds to stage two. In this stage all black processors except those along the left edge of the processor grid send their leftmost column to their left neighbour and receive their neighbour's rightmost column. Following the completion of these communications stages three and four are performed; these are simply vertical counterparts to the horizontal communications patterns described for the first two stages.

A slightly more extensive discussion of this communications strategy is offered in [1].

3 DPEAC and CDPEAC

Programming the vector accelerator hardware of the CM5 directly requires a knowledge of not only the specialised assembly language of the vector units themselves but also a fairly detailed understanding of how the address space of the units themselves map into the virtual address space of the node to which they are attached. This section briefly discusses these details to provide the reader with sufficient background for the sections which follow. For a full treatment of these issues the reader is referred to [5] and [7]

The node architecture of VU equipped CM5 is as follows: the node itself (that is, the SPARC unit) has no memory of its own, but instead has four memory mapped vector units attached to it via a 64 bit bus. These four units are divided between two boards: VUs 0 and 1 are on the first board, and VUs 2 and 3 on the second. The vector units each have a memory bank associated with them (at present each bank is 8Mb) for which they act as memory controllers through which the SPARC may access the memory. Additionally each of the vector units has 64 double precision (64-bit) registers which can also be addressed as 128 single precision registers. The VUs can be instructed to read values from the memory they control into these registers, write values from these registers into the controlled memory, or execute arithmetic operations over their registers. These operations can be either vector operations (i.e., between vectors of registers) or scalar ones. The node can issue instructions to its VUs in essentially three ways: it can issue a command that all four VUs are to perform in parallel, it can issue a command that only a specific VU is to perform, or it can issue a command that two VUs on the same board are to perform in parallel.

The assembly language of the vector units is known as DPEAC [7]. It is a relatively complex instruction set which supports several addressing modes for its operands. Each line in a DPEAC program consists of one memory operation and one arithmetic operation. A typical vector memory operation might be to load a vector of 8 values from memory into VU registers R8 to R15. A typical vector arithmetic operation might be to take the two vectors defined by the registers R8 through R15 and R24 through R31 and add them elementwise placing the resultant vector in the vector defined by the registers R16 through R23. Operands which define vectors of registers can be specified either by a starting register and a stride into the register file (e.g., one of the above could be defined as starting at R8 with a stride of 1) or by the following special notation: V0 refers to a vector starting at R0 with a stride equal to one, V1 refers to a vector starting at R8 with a stride equal to one, and so on. Operands which define vectors to be read from or written to memory are specified through a base address in the VUs memory and a stride. In both cases of vector specification the length of the vector (i.e., how many elements are added together or read or written) is determined by the value stored in a special control register on the board called the Vector Length Register. Another control register allows for the specification of a default memory stride. It is allowable to omit either the memory operation or the arithmetic operation from a DPEAC instruction, however if both are present the memory operation is defined to be executed prior to the arithmetic one. Hence an instruction can load a vector to a series of registers and use those registers in an arithmetic operation on the same DPEAC line.

Whether a particular instruction is to be executed on one, two or four vector units is inferred from the memory address operand if the instruction is a memory load or store, or from the memory address operand of the memory instruction on the same line in the case of an

arithmetic instruction. The virtual address space of the SPARC processor is divided into 42 regions, some of which are used as stack and heap space for the SPARC, but most of which are used for the so-called parallel stack and parallel heap; memory spaces which the vector units can read and write in parallel. One virtual region is devoted to the parallel stack on VU0, one for the parallel stack on VU1, and so on. Using addresses in the former region as address operands to a memory operation will cause only VU0 to execute the operation. An additional region is defined to be the parallel stack for VUs 0 and 1; operations using addresses in this region will cause both VUs 0 and 1 to execute the instruction. Another exists for the parallel stack of VUs 2 and 3. Finally a virtual memory region for all VUs is defined; operations on addresses in this region will be executed by all four VUs in parallel. The offset of the address in a virtual region defines the physical address the VU or VUs will access. If a DPEAC instruction does not include a memory operation the participation of all VUs occurs by default, although the inclusion of a dummy memory address to alter these semantics is possible.

It is perhaps worthwhile clarifying these concepts with a series of simple examples of DPEAC code. Let us consider the case that we have a vector of 8 single precision numbers in the parallel stack region of each of the four VUs of a node. Each vector begins at hex offset `0x1000` into the VU's individual parallel stack region. If the DPEAC instruction

```
floadv [0x80001000], V2
```

is executed by the SPARC, it will cause a vector load to be executed only on VU0; since the address `0x80001000` is in the virtual region for the parallel stack of VU0 only. The vector unit will take a series of values from its own memory bank starting at offset `0x1000` into its parallel stack region and load them into a series of its registers R16, R17 and so on. The number of values that will be read will be equal to the value currently in VU0's Vector Length control register. If instead the instruction had have been

```
floadv [0x94001000], V2
```

both VUs 0 and 1 would have executed the load from addresses `0x1000` onwards in their parallel stack regions into their registers. Finally, had the instruction been

```
floadv [0x90001000], V2
```

all four vector units would have loaded values from their parallel stack regions into their registers. If a line of DPEAC code has both arithmetic and memory operations, the address operand of the memory instruction also determines which VUs will execute the arithmetic instruction. So if a line of DPEAC reads:

```
faddv V1, V2, V3; floadv [0x94001000], V1
```

the read from memory to register will be executed on both VUs 0 and 1, then those same two vector units would execute the addition instruction. If no memory instruction appears on a line of DPEAC code, one may supply a dummy address to specify which vector units we wish to have execute the instruction. For example the command

```
faddv V1, V2, V3; maddr = [0x80001000]
```

causes a vector addition to only be executed on VU0.

The memory operations discussed above and illustrated in the examples allow vector units to read and write from their own memory bank. If it is necessary for a vector unit to access another vector unit's memory bank or registers, things are not so simple. There are essentially two means by which this may be achieved. Firstly, there exists a command modifier that causes a vector register write (such as that which occurs late in the execution of a vector arithmetic operation) to behave differently; instead of the vector being written

to the registers on the same VU, they are written to the registers of the other VU on the same board. That is the two VUs on the board `exchange` values. To carry on the example from above, consider the line of DPEAC code:

```
faddv V1, V2, V3; floadv [0x90001000], V1; exchange
```

The operation of this line proceeds as follows, firstly all four vector units fetch vectors of single precision numbers from their own memory banks into their V1 registers (i.e., their R8, R9, ... registers), they then perform a vector addition pairwise adding values from the registers just filled to values from the V2 registers. The values obtained from these additions is written to the V3 registers, however not the V3 registers on the same VU, but rather those on the other VU on the same board. Hence VU0 stores it's solutions to VU1's registers and vice versa, and similarly VUs 2 and 3 exchange solutions.

The `exchange` modifier allows a very limited transfer of data between VUs, but at least performs that data motion in parallel. The only other method of moving data from memory or registers of one VU to the memory or registers of another is through a series of inherently sequential "VU Accessor Instructions". These allow values to be read from a register or memory location on one VU into a register on the SPARC, or written from a SPARC register to a register or memory on one VU. Hence if we wish to copy a single precision value from VU1's memory to a register on VU2 (which are not on the same board), the code would look something like the following:

```
dpld [0x80001000], %l2
dpwrt DP_2,%l2,R8
```

These VU accessor instructions is significantly slower than the previously discussed memory and register operations, since they make no use of the pipelined nature of the vector units. They also have the disadvantage of potentially causing pipeline hazards or requiring significant delays be introduced to avoid such hazards.

Programming directly in DPEAC means essentially programming completely at the level of assembly code. One writes a program with mixed SPARC assembly instructions (non vector operations to be executed by the node) and DPEAC instructions, pass it through a special assembler `dpas` which produces the appropriate object file. For the purposes of this project we would like, however, to program the non-vector portion of our program in C rather than assembly language. Fortunately there exists a macro package which cleverly uses a feature of GCC [2] allowing inclusion of inline assembly code in C code. The macro package, known as CDPEAC [3] essentially allows the easy inclusion of DPEAC instructions in C programs compiled under GCC.

4 Vectorised Laplace Algorithm

The algorithm we presented earlier for the non-vectorised Laplace iteration assumed that once edges had been exchanged between nodes of the machine, the computation could take place through simple memory fetches from the SPARC's memory, arithmetic operations performed on SPARC registers and memory writes back to the SPARC's memory. This algorithm does not, however, take advantage of either the possible parallelism from multiple vector units working at once, or the speedup available through executing the arithmetic instructions as vector operations on the VUs. In an effort to rectify these missed opportuni-

ties for performance gain, we have developed an algorithm which partitions data across the parallel stack spaces of the VUs and uses VU arithmetic instructions on vectors of values. Figure 2 describes this algorithm at a fairly abstract level, while the remainder of this section goes on to provide a considerably more detailed description. An extended diagrammatic example of the operation of the computation portion of the algorithm is available as a separate document.

Algorithm Step 1: Setup

As before, we partition the grid over which we are working into a number of subgrids over which evaluation is to occur in parallel. However in this algorithm we consider individual vector units as elemental processing units, rather than considering nodes to be the elemental unit of processing. The partitioning algorithm is essentially the same as that presented earlier, however we insist that the four vector units on a node are granted portions of the grid in such a manner that vertical grid coordinates of corresponding elements in different VUs are identical, and the horizontal grid coordinates run consecutively (increasing) from VU0 to VU3. Figure 3 shows one potential partitioning scheme that obeys these criteria.

Once we have decided on a valid partitioning scheme we allocate memory on each VU of each node and copy the appropriate subregions of the original grid to this memory. We choose to allocate slightly more memory than required: if the size of a VU's subgrid has been determined to be x by y we allocate enough physical memory for a rectangular region $x + 2$ by $y + 2$. The subgrid data is placed in this memory in such a way that it occupies the interior of this rectangular space leaving an unused border of one element all around the data. This is shown in figure 4. Space not used for data is indicated in grey in this diagram. This space will be used later in the algorithm to hold copies of edges from other processors.

Algorithm Step 2: Internode communication

As in the non-vectorised algorithm we presented earlier, it is obviously necessary for copies of edges to be exchanged between nodes. In the vectorised algorithm this proceeds almost identically to the earlier scheme; nodes are coloured in a chess-board pattern and edges are swapped through a series of four phases of synchronous communication. The only difference lies in where the data sent comes from and where the data received is stored. The edge communicated to the node's left neighbour is the leftmost (non-border) column of the VU0 subgrid on that node, similarly the edge communicated to the right is from VU3's rightmost (non-border) column. The top edge is obtained by conceptually concatenating the four topmost (non-border) rows from each of the four VU subgrids. Similarly for the bottom edge. Data received from other nodes is stored as follows: the edge received from the left neighbour is stored in the border column on the left of VU0's memory space, the edge from the right neighbour is stored as the rightmost column of VU3's memory region. The top and bottom conceptual edges received are split into four and assigned to the top and bottom border regions of all the VU's memory spaces respectively. Hence after all the communication from this step of the algorithm has taken place, the VU subgrid shown earlier would look as shown in figure 5.

```
Algorithm inputs: x and y, the horizontal and vertical subgrid extents
Algorithm variables: i, the subgrid column being calculated
                     count, the number of column elements yet to be computed
                     v, the vector length used in current computation
                     start, the address of the vector of column elements being computed

  Partition grid over the vector units of the machine [step 1]

  Colour each node either black or white depending on position in conceptual processor array
  Each node communicates edges with its neighbours in the processor array. This is performed
  as 4 phases of synchronous communication [step 2]

  for i = 1 to x
      count = y
      v = (if count >= 8 then 8, else if count >= 4 then 4,
           else if count >= 2 then 2 else 1) [step 3b]
      while count >= 0 do
           compute start
           set vector length control register to v [step 3c]

           load vector of left neighbours of column i into registers V1 [step 3d]

           load vector of right neighbours of column i into registers V2 [step 3e]

           load vector of above neighbours of column i into registers V3
           load vector of below neighbours of column i into registers V4 [step 3f]

           Perform vector addition: V4 = V4 + V3
           Perform vector addition: V4 = V4 + V2
           Perform vector addition: V4 = V4 + V1
           Perform elementwise vector division: V4 = V4 / 4.0 [step 3g]

           write registers V4 to memory as column i of a copy of subgrid [step 3h]

           count = count - v
           v = (if count >= 8 then 8, else if count >= 4 then 4,
                else if count >= 2 then 2 else 1) [step 3i]
      end while
  end for
```

Figure 2: A Top Level Description of the Vectorised Laplace Algorithm

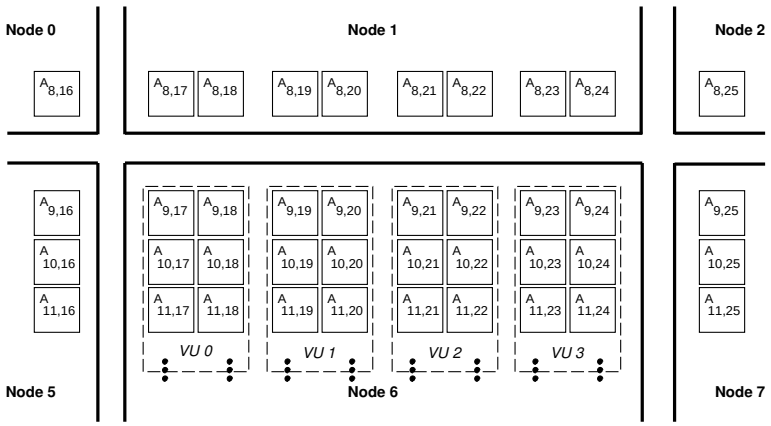

Figure 3: A Valid Partitioning of the Grid A in the Vectorised Algorithm

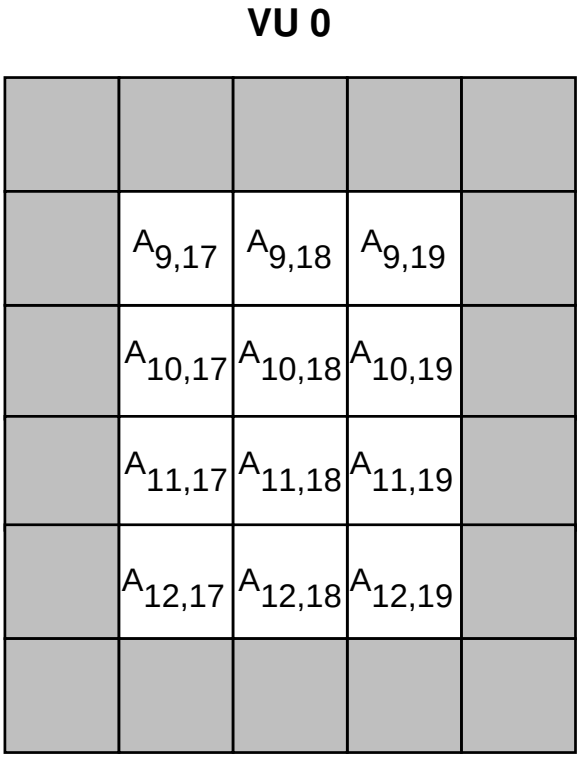

Figure 4: Memory layout on a Vector Unit

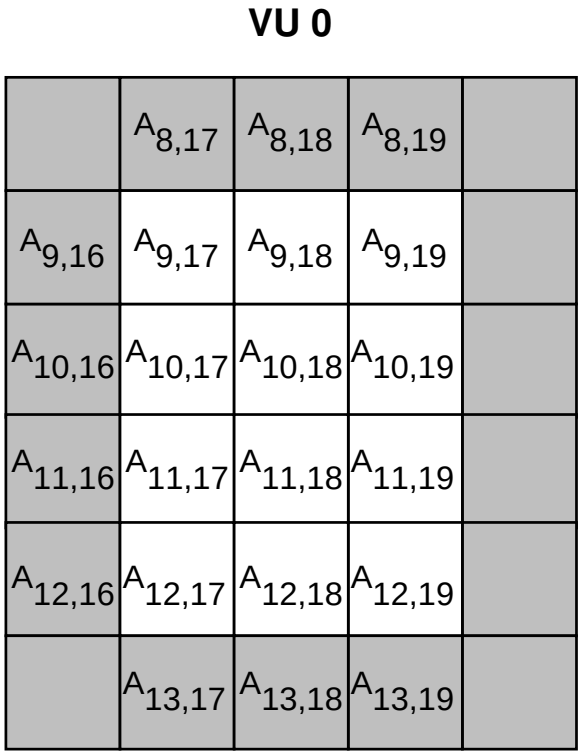

Figure 5: A Vector Unit's Subgrid After Receiving Copies of Off-Node Edges

Algorithm Step 3: Vector Calculation

The procedure we follow to perform the actual vector computation is relatively complex and is laid out in ten substeps below. We assume that each of the four vector units has been granted an equally sized subgrid which is x elements in horizontal extent and y in vertical extent. The general strategy we adopt to perform the entire iteration is as follows: we first consider computing new values for all elements in column 1 of each of the VU's subgrids, we then proceed to column 2 of each VU's subgrid and compute new values for those elements, and so on until all x subgrid columns on each VU have been processed. All nodes of the machine perform this iteration over subgrid columns in parallel with no need for synchronisation or communication since all required values are available on the node.

The computation of new values for a subgrid column may actually require a series of vector arithmetic computations, since the vectors of VU registers we deal with are constrained to a maximum size (in our program we use vectors of up to 8 elements long). Hence we iterate down the column currently under consideration processing vectors of maximal length where possible until we have processed all y elements of the column.

The substeps which follow use a number of variables; `i` to denote the column currently being processed (on each VU), `count` to denote how many elements in that column have yet to be processed, `v` to denote the vector length to be used for the current set of vector operations, and `start` to denote the memory address of the subgrid element which is in the i'th column, `count` elements from the bottom – that is, the topmost element of the vector of positions currently being solved for. For the sake of conciseness we define `leftcol(start)` to be the memory address in the same vertical position as this element but in the first column of the

subgrid. Similarly `rightcol(start)` is the element in the x'th column of the subgrid and the same row as the element in question. The notations `l(start)`, `r(start)`, `a(start)` and `b(start)` denote the left, right, above and below neighbours of the element with address `start`. Note that these last four notations may, in fact, denote elements not in the subgrid but rather in the border region surrounding the subgrid. These notations are summarised in figure 6.

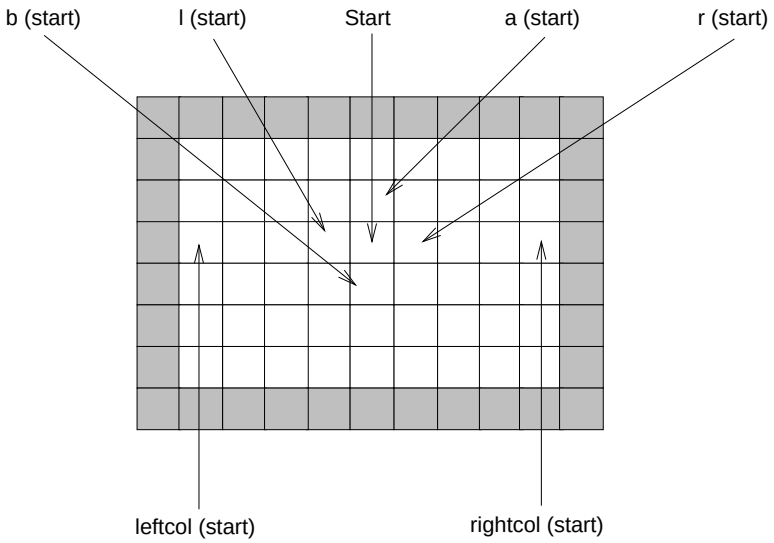

Figure 6: Notations used to denote Memory addresses

Algorithm Step 3a

We first consider the first subgrid column on each of the four VUs, hence by way of initialisation we set `i = 1`

Algorithm Step 3b

This is the initialisation portion of the algorithm which is encountered at the beginning of the processing of each column. We compute starting values for two variables `count` and `v`. `count` = y, and `v` is assigned to be the largest power of 2 less than or equal to y and less than or equal to 8.

Algorithm Step 3c

We set the Vector Length control register on all four VUs to `v`. We then compute the address of the element in the `i`'th column `count` positions from the bottom. We store that address as `start`. Note that `start` is an address in the parallel stack region for all VU's, so any DPEAC operation which uses this address will cause instructions to be executed on all four vector units in parallel. We also, at this stage, set the memory stride control register on each VU such that all vector memory operations will be strided so as to access columns of the subgrid.

Algorithm Step 3d

The task of this step is to have each VU execute instructions such that the vector of left

neighbours of the elements currently being computed end up in the VU registers V1. In the case that we are looking at a column other than the first one this is simple; the required values are part of the same subgrid, so the required data is on the same VU. Hence all that needs be done is to execute the DPEAC instruction:

```
floadv l(start), V1
```

In the special case that we are dealing with the first column of the subgrid (i.e., `i = 1`), the columns of left neighbours are from another subgrid. This means that on each node VU3 needs to read a column from VU2's subgrid, VU2 needs to read a column from VU1's subgrid, VU1 needs to read a column from VU0's subgrid and VU0 needs to read a column from its left node neighbour's VU3 subgrid. We have, however, already arranged that the appropriate column from the node neighbour's VU3 subgrid be copies into the border region of VU0's memory space. So in actuality VU0 does not require access to another VU's subgrid during this step. We can perform the required memory copy from foreign subgrid into the V1 registers of VU's 1 and 3 in parallel by means of the following DPEAC line:

```
fmovev V15, V1; floadv [rightcol(start)], V15; exchange
```

This code causes each VU to load the appropriate elements from its rightmost subgrid column into a temporary set of registers V15. The VU's then copy these registers to the V1 registers, but values are swapped between VUs on a grid; that is, VU1 receives VU0's right column values, VU3 receives VU2's right column values. However VU0 also receives VU1's values into its V1 registers and VU2 receives VU3's right column, neither of which was desired. These values will need to be overwritten by the correct values. In the case of VU0 these values come from its own border region, hence if we adopt the notation `VU0{a}` to represent the memory address corresponding to `a` in the parallel stack space of VU0 only, the DPEAC instruction:

```
floadv [VU0{l(start)}], V1
```

will overwrite the incorrect values in VU0's V1 registers with the correct ones. The work required to load a elements from VU1's rightmost subgrid column to overwrite the incorrect ones written during the exchange, is somewhat more. We must transfer this data through the node itself elementwise via a series of instructions like:

```
dpld [VU1{rightmost(start)}], %l2
dpwrt DP_2, %l2, R8
```

The number of such pairs of accessor instructions that must be executed is `v`.

Algorithm Step 3e

This step of the algorithm arranges for the vector of right neighbour elements of column `i` to be loaded into registers V2 on each of the vector units. This is essentially analogous to step 3e above, and similarly is divided into two cases. Firstly if `i` is not the rightmost column of the subgrids, then the right neighbour elements are all part of the same subgrid and may be loaded with the DPEAC command:

```
floadv [r(start)], V2
```

If we are, however, currently considering the rightmost subgrid column (i.e., `i` $= x$) we need to communicate data between VUs on the node. As before, we do not require interprocessor communication at this stage since we have previously arranged for data from the node's right neighbour to be copied onto VU3. The DPEAC instructions to perform this data motion are analogues of those presented for the data motion required to fetch vectors from the memory

of left neighbouring subgrids:

```
fmovev V15, V2; floadv [leftcol(start)], V15; exchange
floadv [VU3{r(start)}], V2
```

followed by a series of v accessor pairs of the form:

```
dpld [VU1{rightmost(start)}], %l2
dpwrt DP_2, %l2, R8
```

Algorithm Step 3f

In this step we arrange for the vectors of above and below neighbours of the subgrid elements we are currently calculating, to be fetched into the registers V3 and V4 respectively. Because of the properties we demanded from the partitioning scheme, neither of these memory accesses require data to be communicated between VUs. Furthermore, all off-node values required for this load have been copied into border regions on each VU as part of step 2. Hence, the DPEAC code required to load these neighbour vectors to registers is relatively straightforward:

```
floadv [a(start)], V3
floadv [b(start)], V4
```

Algorithm Step 3g

We have now loaded all the values necessary to compute the updated value for each of the elements currently under consideration. Each element must be updated with the average of its four grid neighbours; these neighbour values have been loaded into vector registers V1, V2, V3 and V4. Hence the computation that must take place at this stage is as follows:

```
(V1 + V2 + V3 + V4) / 4.0
```

where the plus operator denotes an elementwise addition and the division operator an elementwise real division. One possible DPEAC implementation of this would be:

```
faddv V4, V3, V4    !  V4 = V4 + V3
faddv V4, V2, V4    !  V4 = V4 + V2
faddv V4, V1, V4    !  V4 = V4 + V1
fdivv V4, 4.0       !  V4 = V4 / 4.0
```

Algorithm Step 3h

In the previous algorithm step we calculated new values for v subgrid elements on each of the VUs of the machine. These values are held in the V4 registers on the vector units. The task of this step is to write these values back to memory. The semantics of the Laplace iteration are such that we cannot write these directly back to the subgrids, since that would affect the computation of other grid elements who use these elements as neighbours. Instead we store the values to a region of parallel stack space on each VU which will hold the grid of results. The operation of this step of the algorithm then becomes essentially the calculation of start′, the address in the result memory to which the vector of solutions is to be written, followed by the DPEAC store command

```
fstorev [start′], V4
```

Algorithm Step 3i

We have now completely processed v elements of column i. Hence we decrement the variable

`count` which holds the number of column elements to be processed, and use this new value of `count` to calculate a new vector length `v` for the next segment of the column. The new `v` is computed here in identical fashion to the computation showed earlier. Once new values for `count` and `v` have been determined, we look at the new `count`; if it is greater than zero, we branch back to algorithm step 3c.

Algorithm Step 3j

We reach this stage of the algorithm when we have completely processed an entire column of each VU's subgrid. We increment `i`, that is, we turn our attention to the next column in sequence, and if this new `i` is less than or equal to x we determine that we have yet to completely process the entire grid. In this case we branch back to algorithm step 3b to process another column of the grid. In the case where the whole grid has been processed, we terminate having completely built up a solution in the result space of each VU.

Note that the algorithm as presented performs only a single iteration of the Laplace algorithm over a grid. The necessary modifications to alter this algorithm to one that performs n iterations of Laplace are elementary. Essentially all that is required is for steps 2 and 3 of the presented algorithm be performed n times. The memory to which results are written and the subgrid memory from which values are read change roles at the end of each of these passes through the algorithm. That is, the result grid of pass j becomes the source subgrid over which calculation is made in pass $j + 1$.

5 Implementation Issues

The algorithm we have presented in the previous section has been implemented as a CDPEAC/CMMD program on a 32 node CM5 with vector units. This section describes briefly a couple of the implementation issues that arose during the construction of the program, and presents some performance statistics for both the non-vectorised programs (both CM Fortran and CMMD) and their vectorised counterparts.

5.1 Edge Transfer Options

One of the issues that arose during the programming of the communications step of the vectorised algorithm (step 2) concerned the efficient edge swapping of top and bottom subgrid edges. The algorithm requires the top row of each subgrid on a node to be copied into the bottom border spaces of VUs on the node's above neighbour in the processor grid. Similarly the bottom-most rows of the four VU subgrids on each node must be copied to the top border spaces of the four VU's of the below neighbouring node. There are two obvious ways to implement this in terms of the CMMD message passing library: one is to arrange for the node to pass four synchronous messages to the appropriate neighbour, each message containing the appropriate edge from one of the four VU subgrids. The neighbour knows which order these four messages will be sent, so it can set the buffer address into which the message is to be received to be the address of the corresponding border region. Once all four

messages have been received by the neighbour into the four corresponding border regions, the communications phase is complete. The operation of this particular implementation strategy is illustrated in the series of diagrams in figure 7. Diagram (a) shows a node communicating it's VU0 subgrid's top edge to its neighbour above, diagrams (b), (c) and (d) show the remaining three communications.

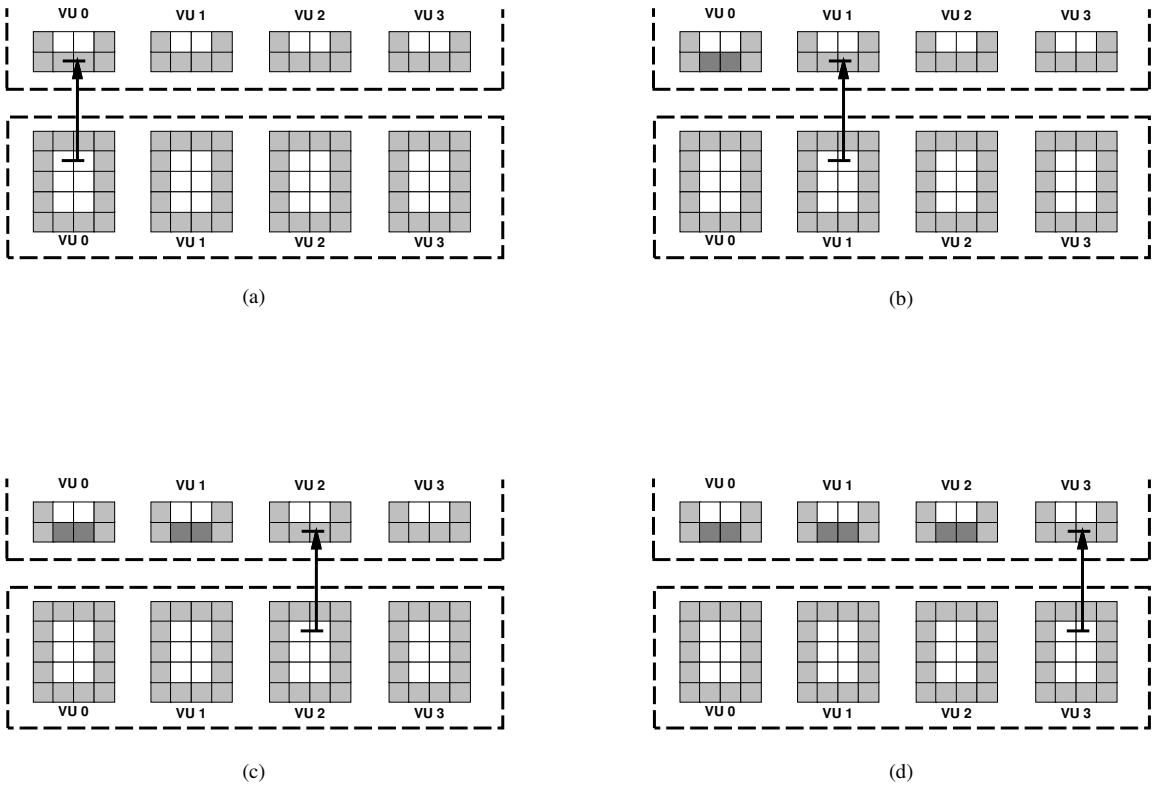

Figure 7: The Individual Edge Strategy of Above/Below Communications

The second method of performing the necessary communication is for the SPARC processor to copy the four subgrid edges from its VUs into a temporary buffer (in the SPARC memory space) and communicate that buffer to the neighbour as a single message. The neighbour receives this message into another SPARC buffer, which is subsequently conceptually divided and the four portions copied to the appropriate border regions. This alternative is depicted as figure 8. Diagram (a) shows a node gathering values from the top edges of its VU subgrids, diagram (b) shows the entire buffer being communicated as a single message, and diagram (c) shows the buffer the neighbour received the values into being scattered to the corresponding VU border regions.

The relative performance of these two communications schemes is entirely dependent on the time required to initiate a synchronous communication versus the time required to perform a series of memory copies. We implemented both strategies and recorded the performance statistics shown in figures 9 and 10. These diagrams show profiles of how the time required for all communication to take place under each of the two above/below edge transfer strategies varies with grid size. They quite clearly show that the time required to set up four relatively short synchronous messages under the individual edge strategy is consistently longer than the time required to set up one long message and perform two sets of memory copies as required by the single message strategy. Hence for the implementation of the full vectorised

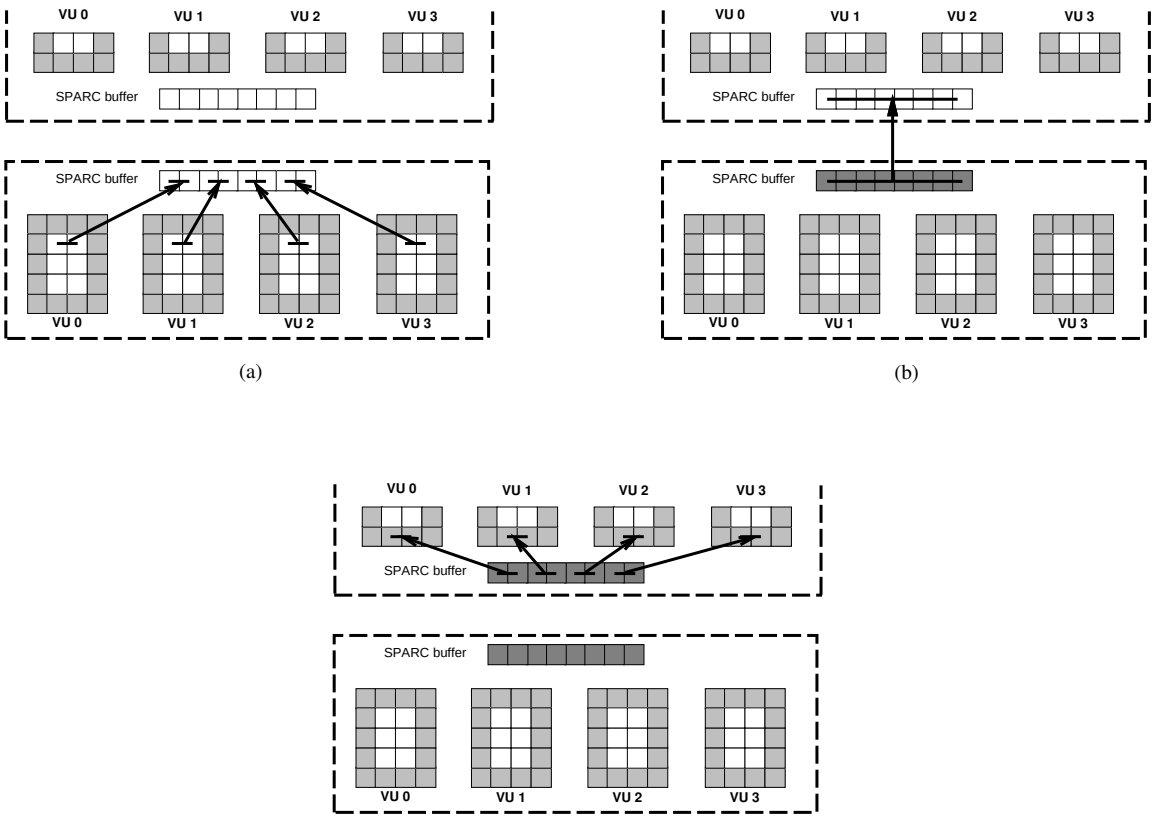

Figure 8: The Single Message Buffered Strategy of Above/Below Communications

Laplace algorithm (whose performance profile is presented below) we chose to use the single message protocol.

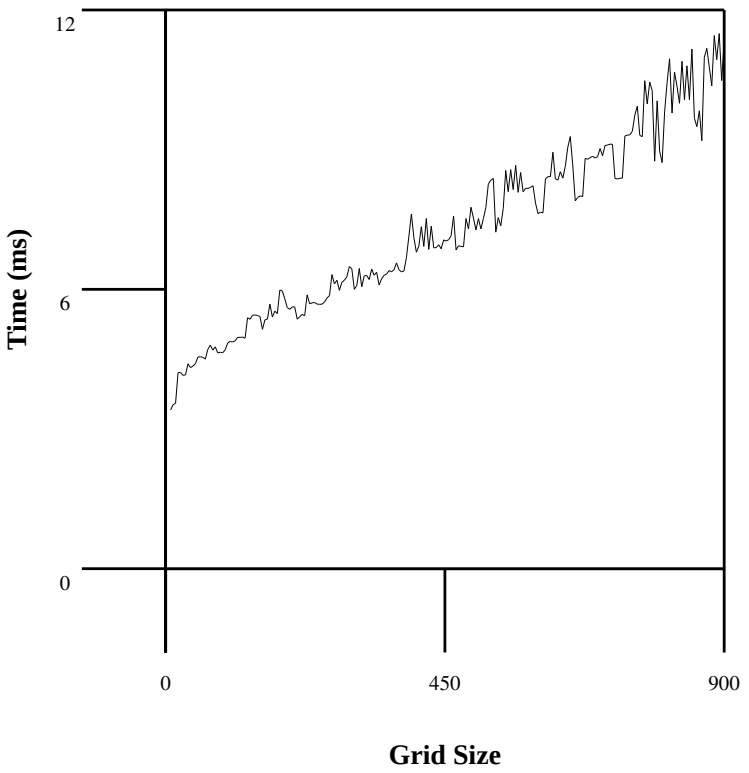

Figure 9: Performance Profile of Individual Edge Strategy

5.2 Single Precision Memory Issues

The DPEAC manual [7] warns that writing single precision values to VU memory is significantly less efficient than writing double precision ones. This difference arises because of the fact that the memory on the VUs is actually 64-bits wide which, despite the fact that we can address it as 32-bit wide memory. The task of writing a 32-bit value (essentially a half-word as far as the memory is concerned) is one of reading the old 64-bit value to a register, modifying either the upper or lower 32 bits, and writing this new 64-bit value back to memory. Writing a double precision (64 bit) value obviously requires none of this reading and modifying. The performance difference is quite large: a double precision store is rated as taking 2 machine cycles, while the single precision store takes 7 cycles.

In the context of our implementation of the vectorised Laplace (over a grid of single precision reals), this means that we are paying a relatively high price for the single precision write we perform for each grid location. The only way to reduce this overhead, however, is to modify the algorithm to make the vector calculation phase handle two adjacent subgrid columns simultaneously. That is, to read neighbour vectors into registers for portions of columns i and $i+1$ in the same iteration of the algorithm, and to calculate new values for these column portions as part of that same iteration. Results from these computations would be written

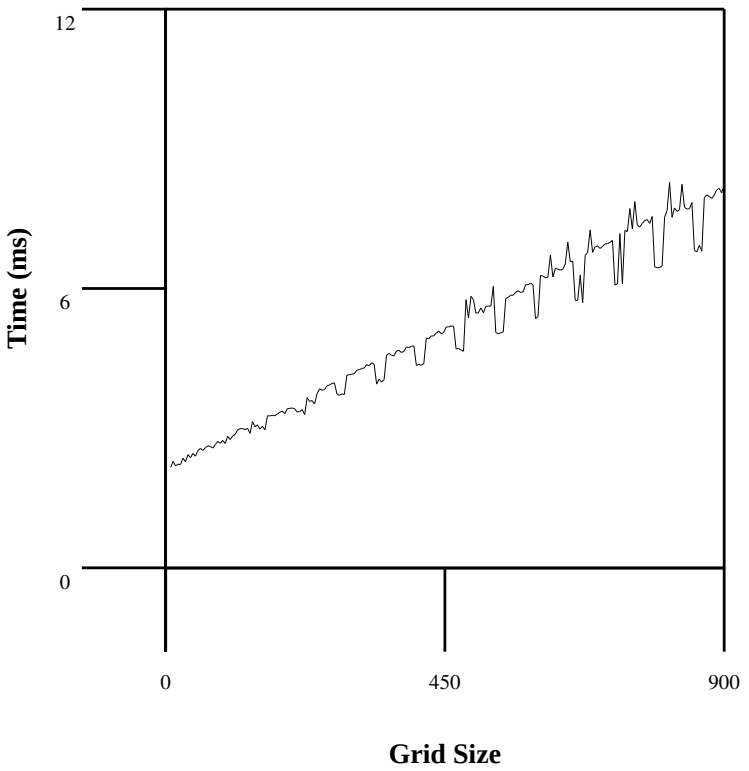

Figure 10: Performance Profile of Single Message Buffered Strategy

to registers in such a way that solutions can be considered pairwise as 64-bit values which once written to memory have the same effect as performing the corresponding two single precision writes normally required. To date we have yet to attempt this modification, hence cannot comment on it's effect on overall performance.

5.3 Algorithm Performance

Before we present a performance profile for the vectorised Laplace program described in this report, we first present, for the sake of comparison, some statistics gathered from earlier work. We include a profile for a CM Fortran program which implements the same iteration, compiled to take advantage of the vector accelerator hardware of the machine (figure 11). We also include a profile of a non-vectorised CMMD program which implements the algorithm described in Section 2. This is shown as figure 12. Both these profiles show the time required to perform one complete iteration of the Laplace algorithm as it varies with the size of the grid over which we are iterating. The grids for each of the program runs recorded were square, with the side length of the square being the unit of the horizontal axis of the profiles. The vertical axis is time in milliseconds.

The performance profile our CMMD/CDPEAC implementation of the vectorised message passing Laplace algorithm, as presented in this report, follows these previous results as figure 13.

From a comparison of these three profiles we can draw the following conclusions:

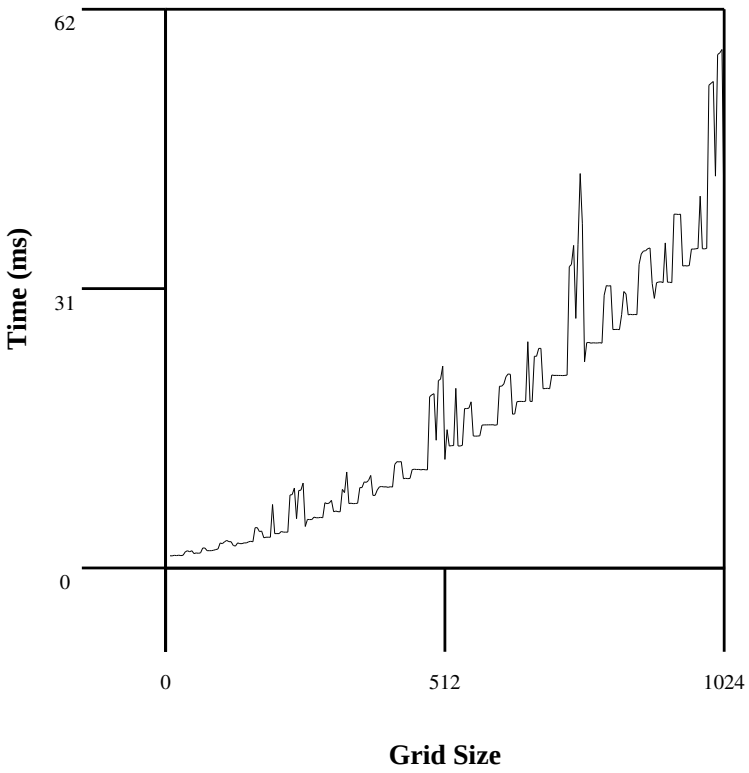

Figure 11: Performance Profile for the Vectorised CM Fortran Program

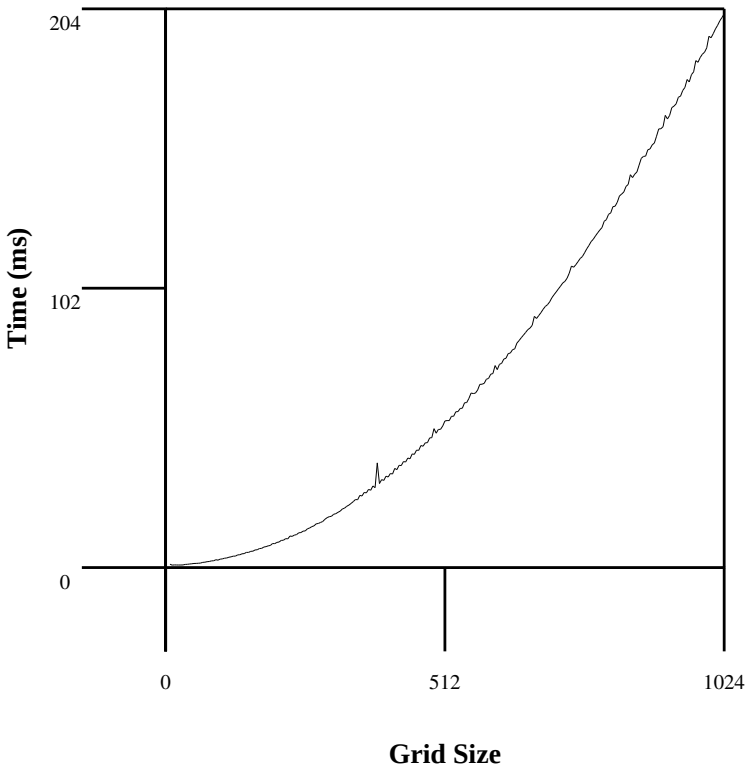

Figure 12: Performance Profile for the Non-Vectorised CMMD Program

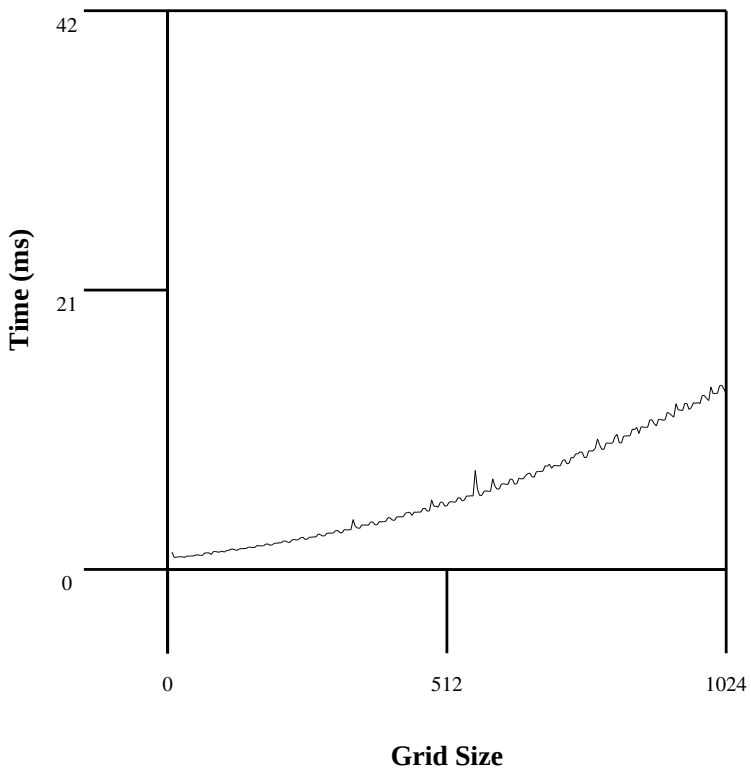

Figure 13: Performance Profile for the Vectorised CMMD Program

- There is most definitely a great benefit to be gained from using the vector processing ability of the CM5 in the implementation of algorithms of this kind. The non-vectorised CMMD program is outperformed by the vectorised CM Fortran program by a factor of approximately 5, and outperformed by the vectorised CMMD program (using CDPEAC) by a factor of 10.

- Both implementations using the message passing model to explicitly handle data motion exhibit considerably more consistent performance than the CM Fortran implementation which uses CMRT runtime library calls for its data motion. This is illustrated by the relatively smooth parabolic profiles recorded for the CMMD programs compared with the jagged profile of the CM Fortran program.

- The low level coding of the algorithm in terms of message passing, explicit memory partitioning and DPEAC programming gains one a performance factor of approximately 2 to 2.5 over constructing the equivalent program using higher level tools such as CM Fortran.

6 Conclusions

We have presented an algorithm for performing a Laplace iteration on the CM5 using message passing and explicit programming of the vector processing capacity of the machine's nodes. This algorithm has been implemented on a 32 node CM5 and its performance recorded and presented here.

Despite an inherent inefficiency in our implementation of the algorithm (due to the usage of a single precision vector store), the performance displayed by our vectorised Laplace is encouraging. It maintains the predictable profile we observed for the non-vectorised message passing algorithm, but improves computation performance to the extent that the presented algorithm is more time efficient than any other of the Laplace iterations we have constructed.

The techniques we utilise in the algorithm presented in this paper, and indeed the general strategy we adopt in computing the Laplace iteration, is easily generalisable to other problems of a similar nature. Essentially any "stencil" type computation (which include a large number of partial differential equation solvers) over a grid is amenable to efficient solution through similar algorithms to that described herein.

Finally from the point of view of our original motivation for this work, the evaluation of a target language for a compiler, this work has built valuable understanding of low level programming techniques and expertise in algorithm vectorising. These will hopefully prove useful to our ongoing project to efficiently implement a simple functional language on the CM5.

References

[1] Dean Engelhardt. Evaluation of characteristics of three programming paradigms supported by existing CM5 software. Unpublished summary of a series of performance analysis experiments, January 1993.

[2] Richard M. Stallman. *Using and porting GNU CC (version 2.2)*, May 1992.

[3] Thinking Machines Corporation, Cambridge, Massachusetts. *CDPEAC: Using GCC to program in DPEAC*, December 1992.

[4] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Reference Manual (version 2.0 Beta)*, December 1992.

[5] Thinking Machines Corporation, Cambridge, Massachusetts. *CM5 Technical Summary*, January 1992.

[6] Thinking Machines Corporation, Cambridge, Massachusetts. *CMMD Reference Manual (version 2.0)*, August 1992.

[7] Thinking Machines Corporation, Cambridge, Massachusetts. *DPEAC Reference Manual*, October 1992.

[8] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

N.B. A diagrammatic example of the operation of the Algorithm described herein is also available via FTP from the site you obtained this report.