Strengths and Weaknesses of Dataparallel C

Seetharamakrishnan S
Department of Computer Science
Oregon State University
Corvallis, OR 97331

seethas@cs.orst.edu

A research paper
submitted in partial fulfillment of
the requirements for the degree of
Master of Science

Major Professor : Dr. Michael Quinn
Minor Professor : Dr. Lawrence Crowl
Other Committee member : Dr. Walter Rudd

May 21, 1992

Acknowledgements

An exercise of this type is not possible without the information and help provided by many people. First among them were Robert Bjornson and Nicholas Carriero of Yale University, who patiently replied to many of my questions and also provided me with the neccessary source code, data and timing details for the DNA sequence matching problem. Calvin Lin of University of Washington also mailed me his source code and input information for the Jacobi problem. Karen Devine of Sandia National Laboratories and John L. Gustafson of Ames Lab answered my queries regarding the load balancing problem. Andrew Kwan of University of California, Irvine, provided me with the exact details of iteration counts and paper references for the Gauss-Seidel problem. Bradley K. Seevers of OSU, the author of the Sequent Dataparallel C compiler, fixed without delay whatever compiler bugs I brought to his notice and also helped me with general queries whenever I barged into his room. Tony Lapadula of the University of New Hampshire, the author of nCUBE Dataparallel C compiler, also fixed the compiler bugs whenever I brought them to his notice. Ray Anderson of OSU helped me with the compilation scripts needed for iPSC/2. Dr. Vikram Saletore of OSU, mailed me the "prime finder" program. Thanks to all the above people for contributing to this project.

Thanks to Dr. Bella Bose who gave me a paid LaTeX job; that experience came in very handy for writing this report. Thanks to Dr. Lawrence Crowl who suggested the addition of log scale graphs to this report which led to great many insights and also for going through my reports from the early stages. Thanks to Dr. Walter Rudd for providing me with financial support during the period of this research and also for introducing me to some interesting problems. Most of all, thanks to my major advisor, Dr. Mike Quinn, who was ready to discuss whenever I knocked at his door, and who meticulously reviewed my reports and kept me going by his encouraging words.

I thank my friends Satish, Gopal and Anil, who worked along with me during late nights, and I also thank Shesha for providing me with the initial LaTeX templates.

Abstract

Dataparallel C is a SIMD style data-parallel programming language for MIMD computers. Dataparallel C has been implemented on both shared memory (Sequent) and distributed memory (Intel and nCUBE) computers. Here we analyze the strengths and weaknesses of Dataparallel C by comparing the performance of compiled Dataparallel C programs with the performance of programs developed using other parallel programming environments.

Contents

1 Introduction 1

2 Overview of the approach 1

3 Problem cases 4

3.1 Matching Biological Sequences 5

3.2 Laplace Equation Solver using Jacobi approach 11

3.3 Laplace Equation Solver using Gauss Seidel approach 15

3.4 Quickmerge 19

3.5 Dynamic Load Balancing 24

3.6 Linear Equation Solver using Jacobi approach 29

3.7 Gaussian Elimination 32

3.8 Matrix Multiplication 34

3.9 Prime Finder 37

4 Strengths and Weaknesses of Dataparallel C 40

5 Conclusions 44

References 45

Appendix A – Performance timing charts 47

1 Introduction

A *data-parallel model* of parallel computation is a SIMD model with an unbounded number of processing elements called virtual processors and a global name space. Dataparallel C is a parallel programming language based on the data-parallel model of computation. It is a superset of the C programming language. Dataparallel C has been successfully implemented on different MIMD computers and reasonable performance has been achieved for several problems.

Programs can be written in assembly language to improve performance. But there are other things like programmer productivity, code portability and maintainability which leads us to consider high-level languages. As parallel computing enters the mainstream, extracting every possible parallel cycle will become less important, and other issues will become more important [10].

Reasonable performance is a relative term, because there might be other languages that perform better or worse for similar problems under similar environments. So we wanted to ascertain how well Dataparallel C compares with other existing parallel paradigms, and in the process identify the strengths and weaknesses of the Dataparallel C language.

In order to do this, we collected papers from conference proceedings and journals published in the last two years, which dealt with parallel paradigms for MIMD hardware. From this collection, we chose only papers which had performance results. In some cases we procured the source code from the authors and compiled and executed their programs on machines which supported both their compiler and a Dataparallel C compiler. This enabled us to directly compare the performance. In other cases we compared our times with the published execution times and speedups.

2 Overview of the approach

There are many factors one has to consider while comparing the performance of different languages or paradigms, especially if there are no ready-made benchmark problems.

Almost any problem could be a candidate for parallel computation, and so one cannot cover all types of problems. But we can parallelize some problems which are of practical importance and try to measure their speedup against the best sequential program for those problems. Also, we can compare the speedup achieved with the theoretical speedup possible for that problem, but calculating the theoretical speedup may not be possible for all problems.

Another way to evaluate the performance of a parallel language would be to take a problem and hand code it using a sequential language with message passing or synchronization primitives and then compare its performance with the performance of the high level parallel language to be evaluated, solving the same problem and

using the same algorithm. This way we can directly compare the execution time and program development time of the two approaches. This will indicate how efficient the parallel language is in comparison to the hand coded version in terms of performance and development time. But the difficulty in this approach is the complexity of developing an efficient hand coded version, though for toy problems this can be done.

So, in the absence of an absolute yardstick to measure the performance of our language, we decided to evaluate its performance by comparing it with other existing parallel paradigms, which also include sequential languages with message passing or synchronization primitives. In this way, we would expose our language to a wide variety of problems, since the problems we choose on our own might consciously or unconsciously favor our language model. We solved the problems described in each paper using Dataparallel C, and then we compared the performance results of our language with the performance results published for the other language.

We do several things to ensure a fair comparison.

First of all, the environment under which the performance is compared has to be same. This avoids many problems, which will otherwise lead to inaccurate inferences. For example, the speedup achieved for a problem on a shared memory machine will not necessarily be the same if the same problem is implemented on a distributed memory machine with the same number of processors. The clock rates of the processors in the shared memory machine and distributed memory machine might be different. Even if the clock rates are the same, the time spent in synchronizations may not exactly correspond with the time spent in communications. Still other factors like cache performance complicate the matter. So the best bet would be to use the same model of the machine on which the benchmarking was done for the other paradigm. This way we could directly compare the execution times, and it is more intuitive and ensures correct inferences. For this reason we tried to procure the source code from the authors of the paper and then we chose machine sites where both our compiler and theirs existed, in order to do the performance comparison. In situations where we could not do this, we depended on the published performance results, but we took care to solve the same problem on the same machine model for which the results were published.

Secondly, the general algorithm used for solving the problem in Dataparallel C has to be similar to the one described in the paper, though slight variations are unavoidable because of the differences in the paradigm, and also because of the availability or lack of certain features. This essentially captures the strengths or weaknesses of the language and its implementation.

Thirdly, the input data has to be identical or at least very similar, because in some cases variations in input data directly affects the time taken to solve the problem. One such example would be an iterative problem which has to meet a convergence criteria. Here the number of iterations performed depends on the input data itself.

Finally, optimization issues like using pointers in place of array access play a major role in deciding the performance, especially if these operations fall inside an

iterative loop. So a bad performance is not always due to communication overhead or inefficient parallelization. Hence, optimizations which apply to sequential programs should be considered for parallel programs also especially if the parallel code is purely local to a particular processor.

Since we did not choose the problems, but rather we selected the papers which had performance results, the problems explored do not fall under a set pattern, but instead have a wide variety. This really helped in testing the versatility of Dataparallel C.

3 Problem cases

This section consists of 9 subsections each dealing with an individual problem case. The general format of presentation for each problem case is as follows:

- Description of the problem
- Primary references
- Parallelism exploited
- Description of the algorithm
- Comparison of performance results
- Conclusions

The Appendix at the end of this report contains detailed timing information.

3.1 Matching Biological Sequences

Description of the problem

The problem to be solved is an assessment of the degree of similarity between two DNA sequences. The general method can be described as follows: Let s and t be two sequences. Construct a similarity matrix H, where $H[i,j]$ is the maximum similarity of all subsequences of s ending at index i as compared against all subsequences of t ending at j. $H[i,j]$ is compared by taking the maximum of three terms:

- $H[i-1,j-1]$ plus a weight that depends on $s[i]$ and $t[j]$. Intuitively this corresponds to lengthening the match.
- Consider all $H[i',j]$ for $i' < i$. These values modified by a suitable penalty function give similarity values consequent to the deletion of various lengths in t. Choose the maximum.
- Similarily consider all $H[i,j']$ for $j' < j$.

When H is complete, its maximum entry is the similarity of the two sequences. To put in simpler terms, the value of $H[i,j]$ depends on three values: the values of $H[i-1,j]$, of $H[i,j-1]$ and of $H[i-1,j-1]$. To start off with, we are given the value of $H[0,j]$ for all values of j, and of $H[i,0]$ for all values of i. These two sets of values are simply the two strings that we want to compare. In other words, we can understand the computation as follows: draw a two-dimensional matrix. Write one of the two comparands across the top and write the other down the left side. Now fill in the matrix; to fill in each element, you need to consider the element above, the element to the left, and the element above and to the left.

Primary references : Carriero and Gelernter [3], Bjornson [1] and Gotoh [9].

The above references did not contain the full details. Hence we communicated directly with the authors Robert Bjornson and Nick Carriero, to clarify our doubts and further procure their C-Linda source code.

Email references: bjornson@cs.yale.edu, carriero@cs.yale.edu.

Parallelism exploited

Study of the data dependencies in the above problem leads to the observation that a counter-diagonal of H can be computed as soon as the previous counter-diagonal is complete. The same observation holds if we deal with sub-blocks rather than individual elements of H: once the upper-leftmost block is completed, the blocks directly to the right and directly beneath can be computed, and so on. This is a wavefront type dependency. Individually each sub-block is treated just like the orignal matrix as a whole.

A given sub-block is computed on the basis of known values for the three sub-blocks immediately above, to the left, and above and to the left. When we say known

values, though, we don't need to know these entire sub-blocks, merely the upper block's lower edge, the leftward block's right edge, and the bottom-right corner of the upper-left block. To eliminate the need for this one datum from the upper-left block, we can define blocks in such a way that they overlap their neighbors by one row or column. Hence we can compute a new sub-block on the basis of one row (from above) and one column (from the left).

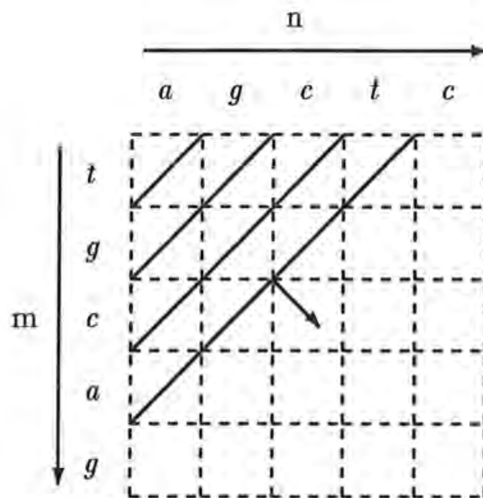

Figure 1: Wavefront Parallelism

Description of the algorithm

Each row of the H matrix is assigned to a Virtual Processor. The row elements are nothing but sub-blocks whose height and width have to be decided to attain the maximum parallelism.

If we are comparing two sequences of different lengths, potential efficiency improves. The shorter-sized sequence determines the maximum degree of parallelism. But whereas, for a square matrix, full parallelism is achieved during a single time step only (namely the time-step during which we compute the elements along the longest counter-diagonal), a rectangular matrix sustains full parallelism over many time steps. The difference between the lengths of the longer and the shorter sequence is the number of additional time steps during which maximum parallelism is sustained.

Suppose our two sequences are length m (the shorter one) and n, and suppose we are executing the program with m workers. The number of time steps for parallel execution, t_{par}, is given by

$$n + m - 1,$$

so for $m, n \gg 1$

$$t_{par} \approx m + n.$$

The sequential time, t_{seq}, is given by $m * n$, and thus speedup, S, is

$$\frac{t_{seq}}{t_{par}} = \frac{mn}{m+n}.$$

Suppose we define the *aspect ratio*, α, of a matrix to be the ratio of its width to its height, then

$$\alpha = \frac{n}{m}$$

and

$$S = \left(\frac{\alpha}{\alpha+1}\right) m$$

By adjusting the aspect ratio of the sub-block, the aspect ratio of the whole matrix can be set. First choose the height of the sub-block such that, all physical processors will have at least one row of the blocked matrix to themselves. If the number of blocked rows is R, then the sub-block width should be $n/\alpha R$, where a convenient α is chosen which leads to high efficiency and less communication or synchronization overheads.

Computation begins at the upper left block. After a virtual processor computes a sub-block, it proceeds to compute the next sub-block to the right. Thus the first virtual processor starts cruising along the top band of the matrix, computing sub-blocks. As soon as the upper-left sub-block is complete, the second virtual processor can start cruising along the second band, and so on. Only the bottom edge needs to be communicated to the next virutal processor, as the right edge of a sub-block will be used by the same virtual processor while working on its next sub-block.

Comparison of performance results

The following figures show the performance measured on a Sequent Symmetry and iPSC2. For the iPSC/2, the absolute timings for the C-Linda version were not available. Only the speedup data for the 7000 $\times$ 7000 self comparison was available. The sequential time for C-Linda version on one node of the iPSC/2 was 776 seconds. The speedup for Dataparallel C version was computed using the sequential time of C-Linda version. We had access only to a 32 node iPSC/2. Detailed timings for different aspect ratios are given in the Appendix.

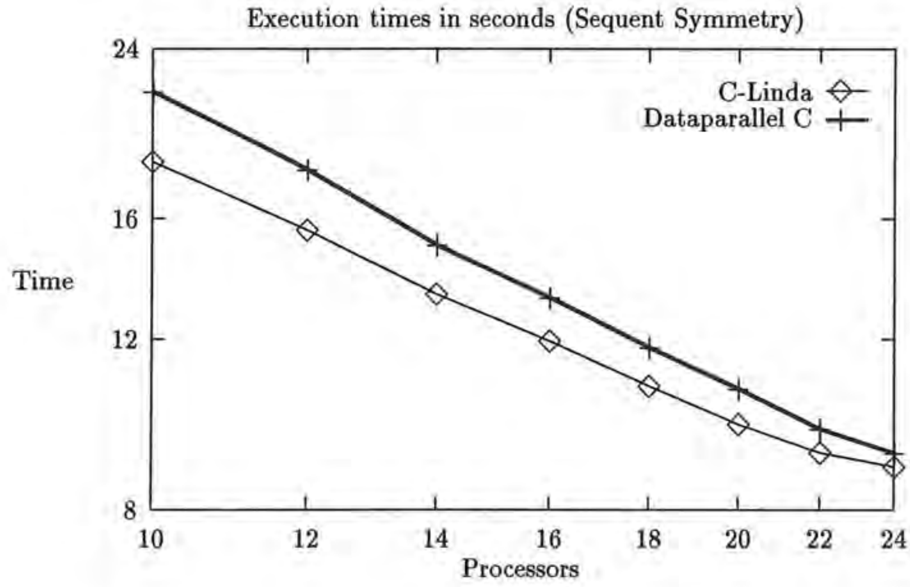

Figure 2: Side $\times$ Top = 3389 $\times$ 3389, Aspect Ratio = 10

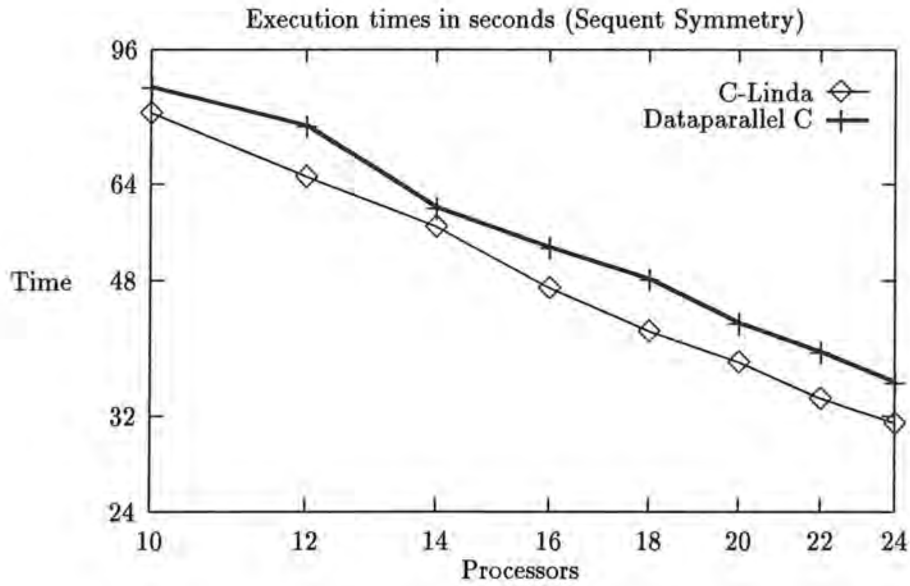

Figure 3: Side $\times$ Top = 6778 $\times$ 6778, Aspect Ratio = 10

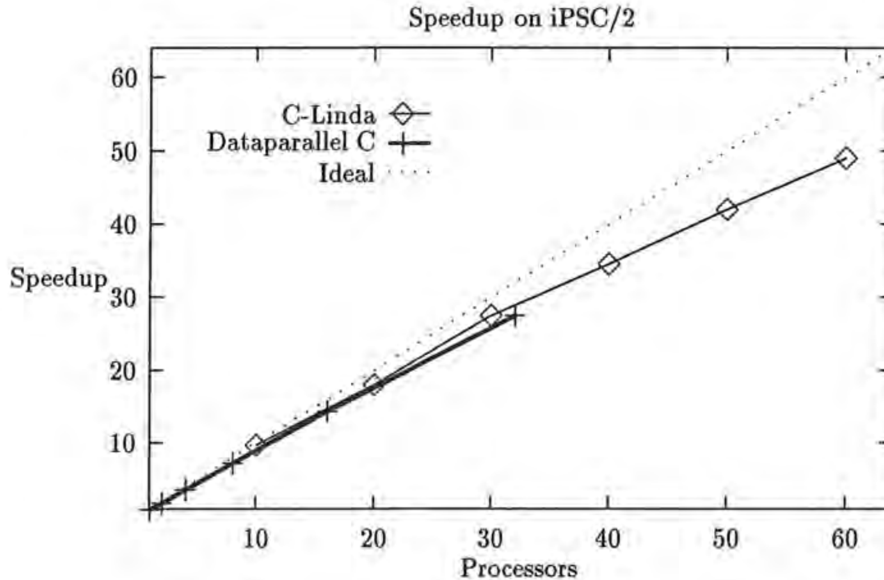

Figure 4: Side × Top = 7000 × 7000, Aspect Ratio = 10

Conclusions

- From the above performance graphs, we see that Dataparallel C performance is very close to C-Linda performance, but on the average C-Linda performs better than Dataparallel C. One of the main reasons for the performance loss is due to the parallelism lost due to synchronizations. A virtual processor can start processing its sub-block n as soon as it gets the bottom edge of the sub-block n owned by the predecessor of this virtual processor. But in our case, due to the barrier synchronization, a virtual processor sometimes has to wait for other virtual processors to arrive at the barrier, even if it has the necessary data to proceed. This results in a loss of efficiency.
- Keeping the problem size and aspect ratio constant, if the number of processors are increased, the size of the sub-block decreases. So the processing time for each sub-block decreases. Though the number of synchronizations increases, the processors reach the barrier faster, reducing the waiting time at the barrier. So the startup costs are reduced. This might be the reason why Dataparallel C timings are closer to C-Linda timings as the number of processors is increased.
- There was no direct mechanism for staggered start up of computation. So we had to do a small trick in the "for loop" to accomplish that.
- Initially, the Dataparallel C program performed poorly when compared to C-Linda version. The reasons, we found were the following:
  - C-Linda version was using pointers instead of array accesses.

- The C compiler used by the C-Linda people was different than the one we used.
- The similarity() subroutine was an optimized version of the compare subroutine which they had published. We realized this when we procured their source code. When we incorporated those optimizations, our timings improved.
- Also the change of data structure for storing the sub-block was changed from a matrix to two one dimensional arrays, which led to improvement in performance. The matrix representation was too large to fit in the cache and hence resulted in poor performance.

- The performance of Dataparallel C on iPSC/2 is almost same as C-Linda performance. Moreover the speedup for Dataparallel C version was measured using the sequential timings of C-Linda version. There is a decrease in efficiency as the number of processors increase. This is due to the sinking task granularity and also due to the decrease in amount of work per communication event.
- The Dataparallel C program needs to be compiled whenever the aspect ratio is changed, whereas C-Linda version needs them only at run time.
- Developing the Dataparallel C code for this problem was very easy. The syntax and semantics are exactly similar to the sequential model except for a few statements.
- Debugging the program was also very easy and 'printf' statements were more than sufficient for the task.
- The same code runs on iPSC/2 and nCUBE, which are distributed memory machines, and the Sequent Symmetry, which is a shared memory machine.

3.2 Laplace Equation Solver using Jacobi approach

Description of the problem

The problem is to solve Laplace's equation on a rectangle, where the rectangle is represented as a two-dimensional array of integers, V. The problem is governed by the equation

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = 0, \;\; V(x,y) = \text{constant on the boundary.}$$

We use the Jacobi iterative technique to solve this equation. There are three steps involved in this technique.

1. initialize the matrix V (i.e. the grid points).
2. repeat
   for each point in V
   $$V_{i+1}[x,y] = \frac{V_i[x+1,y] + V_i[x-1,y] + V_i[x,y+1] + V_i[x,y-1]}{4}$$
   until $|V_{i+1}[x,y] - V_i[x,y]| < \delta$ forall $V[x,y] \in V$
3. print results

In this particular problem all data points are initialized to a constant integer value and boundary conditions are kept constant.

Primary references:- Lin and Snyder [16].

We procured the source code from Calvin Lin, who is one of the authors of the paper, and we did the benchmarking of their program along with ours on a Symmetry machine. (Calvin Lin's email: linc@minke.cs.washington.edu)

Parallelism Exploited

For an iteration i, the computation of local average for each data point depends on the values of iteration $i-1$ of its four neighbors (north, east, west and south). Suppose we solve this problem on a sequential computer, we would normally represent the data points as a two-dimensional matrix where each element of the matrix would maintain the current and previous iteration value. Each iteration would be an n^2 loop which calculates the averages for each data point. Within an iteration, the averages for the data points can be computed in any order, the only condition being that all data points have to be considered. Unlike the Gauss-Seidel method, where values of the north and west neighbors have to be of the current iteration, here all the neighbor values used are of previous iteration. Hence a parallel algorithm can utilize this fact to simultaneously compute the averages of all data points during each iteration.

Depending on how we partition the problem, either cellwise, rowwise or blockwise, the neighbor values exchanged between virtual processors before each iteration will

be either single data point values or an edge of data point values. In the case of distributed memory machines, these neighbor values have to be communicated, whereas in the case of shared memory machines, they can be directly accessed, provided there is barrier synchronization between iterations. Essentially there is a communication phase and a compute phase. Both these phases can be totally parallel. But in order to test the convergence criteria, there is a reduce phase which tests to see if the maximum of the error difference is less than the tolerance value. If so the algorithm terminates, else the next iteration continues.

Description of the algorithm

The partitioning of the problem can be done either cellwise, rowwise or blockwise. Each has its own advantages and disadvantages. The major factors to be considered in order to choose the partitioning are size of the problem, number of physical processors available, number of edges or cells that will be exchanged between iterations and the length of the edges.

In case of cellwise decomposition each virtual processor is responsible for one data point. During the communication phase each virtual processor collects the values from its four neighbors (N, E, W, S). The virtual processors reponsible for the boundary points participate only during the communication phase. During the compute phase each virtual processor computes the new average of its data point and the error difference. Then during the reduce phase the maximum of the difference is found by the cooperation of all the virtual processors. Also since the virtual processor synchronize after or before each iteration, all virtual processors will be performing the same iteration step at any given time.

In case of rowwise decomposition each virtual processor is responsible for a row of data points. During the communication phase each virtual processor collects an edge of values from its two neighbors (N and S), because the points (E and W) lie in the same processor. The virtual processors responsible for the boundary edges remain active only during the communication phase. During the compute phase each virtual processor computes the average for all the data points in the row it owns and also simultaneously computes the local maximum of the difference. During the reduce phase the local maxima are gathered and the global maximum of the difference is found.

In case of blockwise decomposition each virtual processor is responsible for a square submatrix of data points. During the communication phase each virtual processor collects edges of values from its four neighbors (N, E, W, S), and this communication pattern is similar to the pattern due to cellwise decomposition the only difference being that instead of single value, 'edges of values' are exchanged between virtual processors. During the compute phase each virtual processor computes the average for all the data points in the submatrix which it owns and also simultaneously computes the local maximum of the difference. During the reduce phase the local maxima are gathered and the global maximum of the difference is found.

We notice that cellwise decomposition leads to fine grain parallelism, rowwise

decomposition leads to medium grain parallelism and blockwise decomposition leads to coarse grain parallelism.

Comparison of performance results

The following figure shows the performance on Sequent Symmetry for different square grid sizes. More timing details are given in the Appendix. The timings were taken on 4 and 16 processors. In the case of Dataparallel C, the decomposition is rowwise and in the case of Sequent C the decomposition is blockwise. The performance is plotted for rowwise decomposition because it gives almost the same performance as blockwise decomposition but the code size is pretty short compared to the blockwise approach.

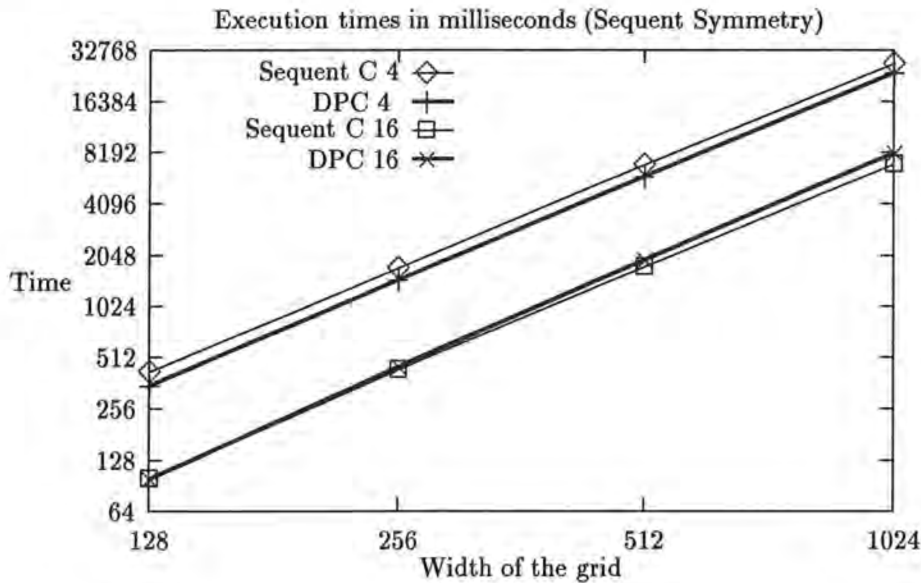

Figure 5: Laplace equation solver using Jacobi method (Rowwise)

Conclusions

- One of the first things we noticed was the vast difference in the length of the source code between Dataparallel C version and their version. Dataparallel C code is shorter by 4 to 8 times when compared with the Sequent C version. The cellwise decomposition leads to the shortest length code which is 72 lines and the blockwise decomposition leads to the longest which is 139 lines of code. The Sequent C version is 566 lines of code. Certainly the Dataparallel C version is compact and readable, and it has no machine dependent code.

- The development time was very short. The maximum time taken was for the blockwise decomposition due to the code pertaining to the communication of the edges. It took less than two hours to finish the coding and debugging of the program. The rowwise and cellwise versions took even less time. We do not

know how much time it took for the developers of the Sequent C version, but we are sure that even to type around 500 lines would take a lot of time. Though their intention is not to promote the Sequent C version, at least for comparison purposes we can see that Dataparallel C is convenient from the point of coding and debugging.

- From the above performance graph we see that rowwise decomposition offers the best of both worlds; i.e, good timing and ease of programming. Cellwise decomposition is good for smaller problem size since the overhead of virtual processors is not much. But for large problem size, the cellwise version is not suited. The blockwise decomposition may perform better for even larger problem sizes. It also depends on whether we are running the program on a shared memory or distributed memory machine. The number of edges communicated before each iteration, the length of each edge communicated do matter in case of distributed memory machines. In case of shared memory machines the communication time may be equated with the time taken to copying of the buffers since the synchronizations depend only on the number of iterations and not on the number of edges communicated.

- One more additional advantage of Dataparallel C lies in the communication macros like **north()**, **south()**, etc. These macros are independent of machine architecture from the point of view of the programmer and also they help in optimizing the communication since they indicate the communication pattern. The macros available can be used for large variety of problems, because they form a fairly comprehensive set of communication patterns exhibited by many parallel algorithms.

- One more thing which is not reflected in the timing chart is the time taken for the initialization phase. For some reason the Sequent C version takes very long time for the initialization compared to the Dataparallel C version.

- The cellwise version took around 18 seconds for 512×512 on 4 processors and around 33 seconds on 16 processors. This was with the default interleaved layout of virtual processors. When we changed the layout to contiguous, the timings were drastically reduced. It took around 7 seconds on 4 processors and around 4 seconds on 16 processors. The contiguous distribution reduces the number of times processors write to the same cache block.

- For large sized problems, the swap space in the partition matters. This was revealed when we tried to run 1024×1024 problem using cellwise decomposition. So we copied the executable to /tmp and then executed the program. We have not yet found the reason for the extraordinarily slow performance of cellwise version for 1024×1024 problem size.

3.3 Laplace Equation Solver using Gauss Seidel approach

Description of the problem

The problem is to find the steady-state temperature distribution in a square plate, one side of which is maintained at 100°, with the other three sides maintained at 0° as shown in the figure below. This is achieved by solving Laplace's equation in a rectangle. If each side of the square is divided into n increments $\triangle x(=\triangle y)$, the problem involves the solution of a system of $(n-1)^2$ simultaneous equations with the temperatures at the interior grid points as the unknowns.

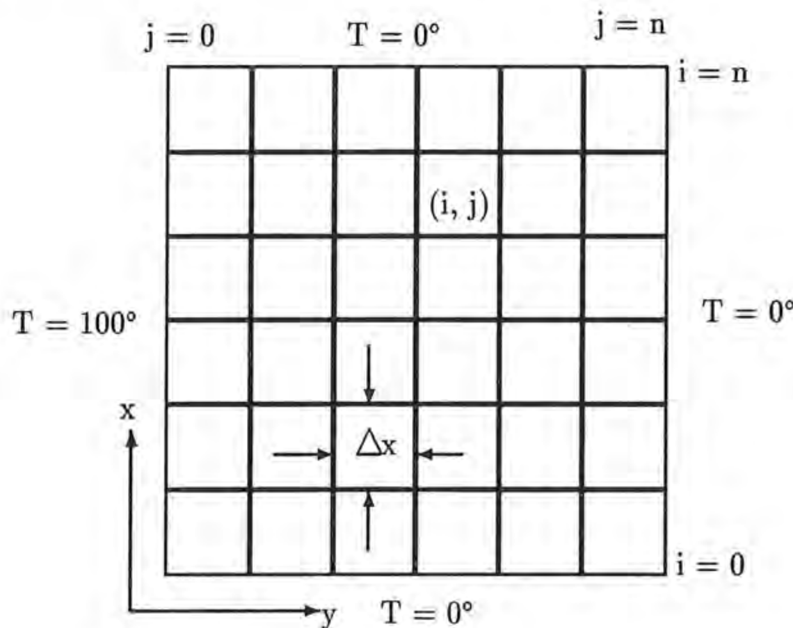

Figure 6: Heat conduction in a square plate

The method of solution is to iterate through all the grid points, calculating a better approximation to the temperature at each point (i,j). The following equation represents the computation at a grid point (i,j) for kth iteration.

$$T_{i,j}^{k} = \frac{T_{i-1,j}^{k} + T_{i,j-1}^{k} + T_{i,j+1}^{k-1} + T_{i+1,j}^{k-1}}{4}$$
$$i = 1,2,\ldots,n-1, j = 1,2,\ldots,n-1$$

As soon as a new value of T is calculated at a point, its previous value is discarded. This is the Gauss-Seidel method of iteration. To start, a temperature of 0° is assumed everywhere within the plate. The process of iteration through all grid points is repeated until further iterations would produce very little change in the computed temperatures. The process stops when the maximum deviation of the temperatures from their previously computed values, over the entire grid falls below a small quantity ϵ_{max}.

Primary references : Kwan and Bic [14, 15], Carnahan, Luther, and Wilkes [2]. Email reference: kwan@ics.uci.edu.

Parallelism exploited

For an iteration k, the new temperature at a grid point can be computed as soon as the computed values of its north and west neighbors for that iteration are available. This is assuming that new values are generated from left to right, top to bottom of the rectangular grid. The usual method for solving this type of problem is to use a checker board block pattern.

In this method, each virtual processor is responsible for a single grid point. There will be an intial delay before all virtual processors become active, due to the staggered start. But after that all virtual processors are kept busy till the convergence criteria is met. This method is too fine grain and may not be suitable for distributed memory machines. Also for each computation, four short messages need to be exchanged, and hence the time taken to initiate the message may be more than the time taken to exchange the message itself. On the iPSC/2, overhead for generating messages is the dominant factor in message latency, and not message length or message transmission distance [5]. So even if we partition the grid into smaller subblocks, and each responsible for a subblock of grid points instead of single grid point, still the message length may not be big enough to compensate for the overhead in generating the message.

The algorithm used here partitions the grid into rows and each virtual processor is responsible for an even number of rows. A row can be computed as soon as the row above it has generated its new values. This scheme also has the initial startup delay, but the message passing is more efficient and leads to coarse grain parallelism, which is essential for good performance on distributed memory computers.

Description of the algorithm

Each virtual processor is responsible for an even number of rows. The set of rows each virtual processor owns is split into two halves, namely upper and lower, each containing an equal number of adjacent rows. To start with, virtual processor zero computes the new values for the rows in its upper half. To do this it does not need any neighbor values from the north, since the topmost row corresponds to the top edge of the plate, and the neighbor values from west, east and south lie on the same processor. Once it computes the upper half, the lower half can be computed. This is possible because the last row of the upper half is the northern neighbor of the first row of the lower half. However to compute the values of its last row, it needs values from the virtual processor to its south. After virtual processor zero finishes with its lower half, virtual processor number one can start working due to the availability of new values from its northern neighbor. At the same time, virtual processor zero reverts to its upper half to calculate the new values for its next iteration.

After the initial startup delay, all processors become busy and they alternately compute their upper and lower halves exchanging values of the bottom and top edges with their neighboring processors in between the computation phases. During the computation of the new values, the difference between the old value and the currently

generated new value is found and a local maximum of that difference is maintained. At the end of each iteration, the global maximum of the difference is found to check if the convergence is achieved. A single iteration is said to be complete when all virtual processors finish computing their upper and lower halves once.

Comparison of performance results

The following graphs show the performances of CAB paradigm and Dataparallel C for both 128 × 128 and 256 × 256 size Laplace Equation Solver on the iPSC/2.

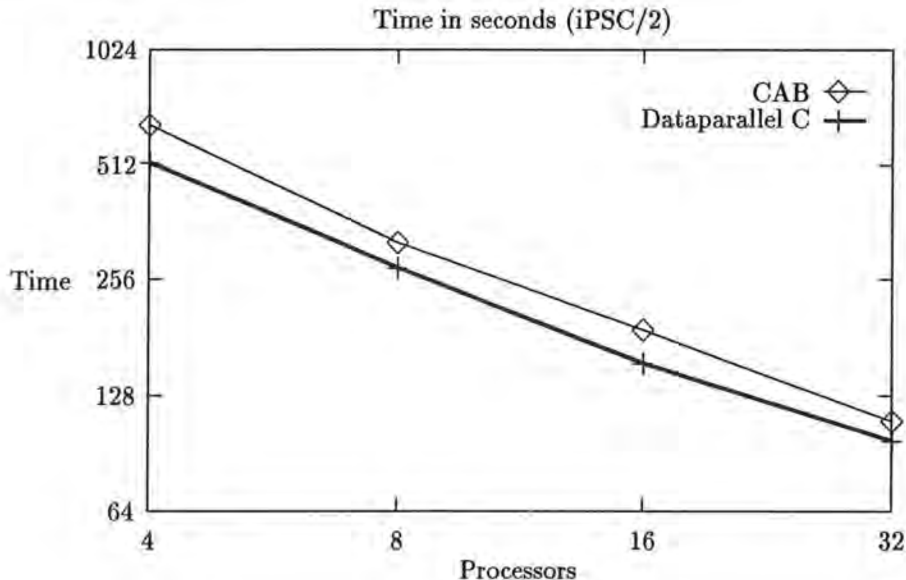

Figure 7: Laplace Solver 128 × 128

The detailed timings are given in the Appendix. The timings for processors less than 4 were not taken due to the long time it will take to complete (on the order of 6 hours for 256 × 256 size equation).

Conclusions

- Compute-Aggregate-Broadcast paradigm is meant for problems of this type which have a compute and aggregate-broadcast phase. Dataparallel C is a generic language for solving parallel problems, with a special stress on numerical iterative problems. The performance of Dataparallel C is better than CAB paradigm. We used the same algorithm used by the authors of CAB paradigm.
- There are only three communication statements in Dataparallel C: two for exchanging neighbor values and one for reduction and the rest of the code is like regular 'C' code.

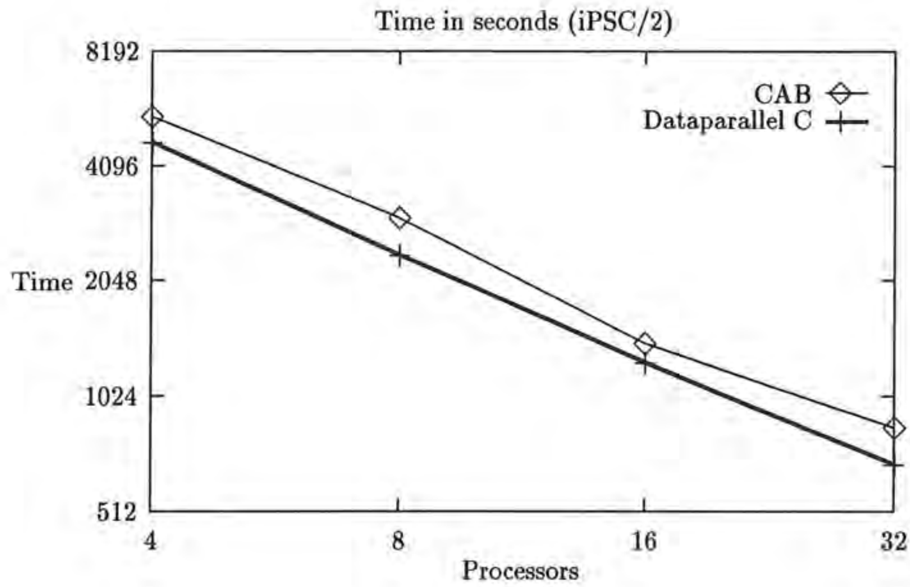

Figure 8: Laplace Solver 256 $\times$ 256

- Dataparallel C is ideal for such problems. Though there is a startup delay, it is amortized by the huge number of iterations needed for convergence (in the order of 12,000 for 256 $\times$ 256 size problem).

3.4 Quickmerge

Description of the problem

Quickmerge, a parallel sorting algorithm, is a combination of quicksort and a multi-way merge designed for tightly coupled multiprocessors. This algorithm was originally developed by Quinn [18] and implemented on a single-PEM Denelcor HEP using HEP FORTRAN. The performance of this algorithm was further investigated by Evans [7] on a Balance 8000 multiprocessor for various distributed pseudo-random data sets consisting of uniformly, exponentially, normally, ordered and reverse ordered data. Now we have implemented this algorithm using Dataparallel C and have run benchmarks on a Balance 8000 multiprocessor. We describe the algorithm and compare the performance of Dataparallel C with the performance results published in [7].

Primary references : Evans [7], Quinn [18] and Knuth[13].

Parallelism Exploited

The quickmerge algorithm contains 3 distinct phases. The processes are totally independent of each other during each phase and need to synchronize only at the end of each phase. There is no need for any synchronizations during the execution of a phase, though the processes do access non-local variables. These non-local accesses are read only, since the variables defined in the preceeding phase are used in the current phase. All variables which are read/write during a phase are local to a process. Since this is a shared memory implementation, the cost of the non-local memory accesses is negligible, which may not be the case if this algorithm is implemented on a distributed memory machine.

Description of the algorithm

Quickmerge has three phases. In the first phase each of p processes sorts a contiguous set of no more than $\lceil n/p \rceil$ keys using the fast sequential quicksort algorithm. After this phase all processes synchronize. The n keys can now be seen to form p independent sorted lists of size approximately $\lceil n/p \rceil$.

In the first sorted list of $\lceil n/p \rceil$ keys, $p-1$ evenly-spaced keys are used as dividers to partition each of the remaining sorted lists into p sublists. The second phase accomplishes the partitioning as follows. Each process i, where $1 \leq i \leq p-1$, finds, for lists 2 through p, the index of the largest key no larger than the key located at index $\lfloor in/p^2 \rfloor$ in list 1. After this phase all processes synchronize. At this point each of the sorted lists has been divided into p sorted sublists with the property that every key in every list's ith sorted sublist is greater than any key in any list's $(i-1)$st sorted sublist, for $2 \leq i \leq p$.

In the third phase, each process i, where $1 \leq i \leq p$, performs a p-way merge of the ith sorted sublists. Note that unlike phase one, in which each process sorts a contiguous block of keys, in phase three each process merges p lists stored in p different areas. Because of the demarcations established in phase two, these merges are completely independent of each other. After this phase all processes synchronize,

and the list is sorted.

	Virtual processor 1	Virtual processor 2	Virtual processor 3
Initial	99 2 22 33 44 57 24 8	9 66 24 43 42 1 10 20	31 99 22 24 1 73 16 84

	Virtual processor 1	Virtual processor 2	Virtual processor 3
After Phase I	2 8 22 24 33 44 57 99	1 9 10 20 24 42 43 66	1 16 22 24 31 73 84 99

	Virtual processor 1	2 8	22 24 33	44 57 99
After Phase II	Virtual processor 2	1	9 10 20 24	42 43 66
	Virtual processor 3	1	16 22 24 31	73 84 99

	Virtual processor 1	Virtual processor 2	Virtual processor 3
After Phase III	1 1 2 8	9 10 16 20 22 22 24 24 31 33	42 43 44 57 66 73 84 99 99

Figure 9: Example of quickmerge

The above figure illustrates a three-processor sort of a list of twenty-four elements using quickmerge. After the initial quicksort phase, the list consists of three sorted sublists of length 8. During the second step binary search is used to determine where the elements one-third and two-thirds of the way through the first sublist would fit in the other sublist. In the final phase each processor performs a three-way merge on its own set of sorted sublists. The merge step can be done using a heap, but currently we are using a simpler method. In this method virtual processor i repeatedly scans the ith sorted sublists of all the virtual processors and removes the element with the lowest magnitude and appends it to a result array which will be the final sorted list.

The worst-case time complexity of this algorithm as calculated by Quinn [18] is $O((n/p)^2 + p\log(n/p) + np + p)$. The worst-case time complexity for the merge phase is $O(np)$ for the method used here as opposed to $(n \log p + p)$ when a heap is used.

Comparison of performance results

To compare the performance of the quickmerge version written using Dataparallel C with that of the performance results published in [7], we generated distributed pseudo-random data sets consisting of uniformly, normally (using the Polar method [13]), and ordered data. The ordered data was simply the sorted output of the uniformly distributed data. For each value of N where $N = 10^3, 10^4$ and 10^5, one hundred runs were performed. We benchmarked the algorithm on 1 to 9 physical processors for each N on a Sequent Balance 8000 multiprocessor. The following figures plot the mean

execution times of the quickmerge algorithm for uniformly, ordered and normally distributed pseudo-random data consisting of 10^5 elements. We do not know the language in which the version published in [7] was written. For reference purposes we indicate the non-Dataparallel C version as Evans's version. Detailed timings for different values of N are given in the Appendix.

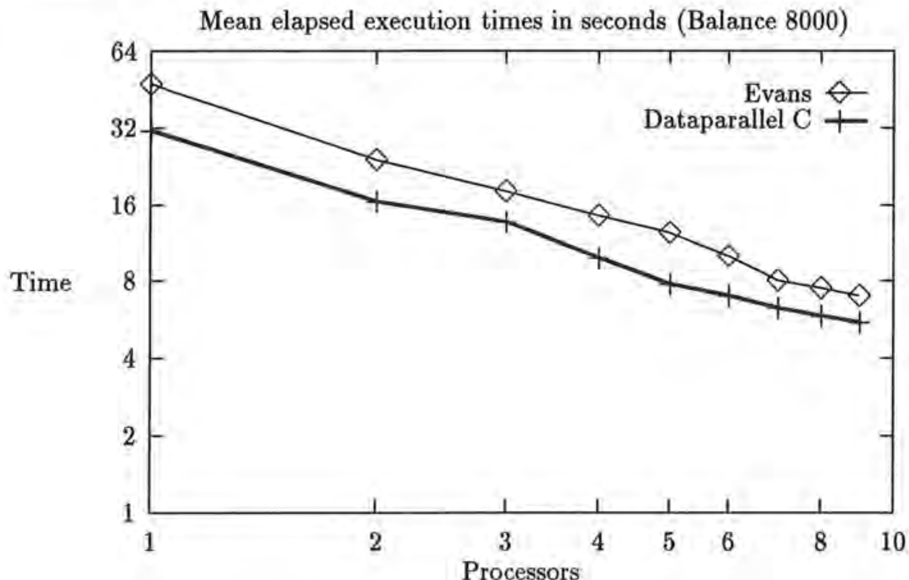

Figure 10: Uniform Distribution 10^5 elements

Conclusions

- We were able to code this algorithm using Dataparallel C in a neat and easy way. Accessing a non-local variable through the virtual processor index is a nice feature that simplified coding this algorithm.
- The member functions were useful in the sense that they clearly demarcate the three phases and hence the synchronization points.
- From the performance graphs we see that except for the ordered distribution the Dataparallel C timings are better than the Evans's version of the same algorithm. This might be due to differences in the operating system or compiler, the pseudo-random data set generated, or the load variation of the system during the benchmarking.
- We tried to use `malloc()` to dynamically allocate memory inside the member functions. But it was unpredictable, and the program stopped due to segmentation fault many times. So we used `shmalloc()`, which worked fine without any segmentation fault.

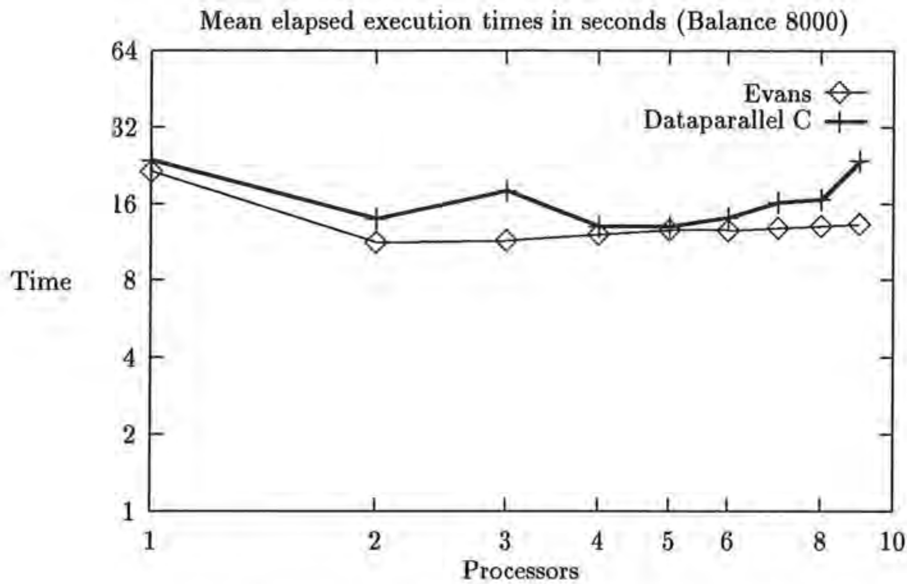

Figure 11: Ordered Distribution 10^5 elements

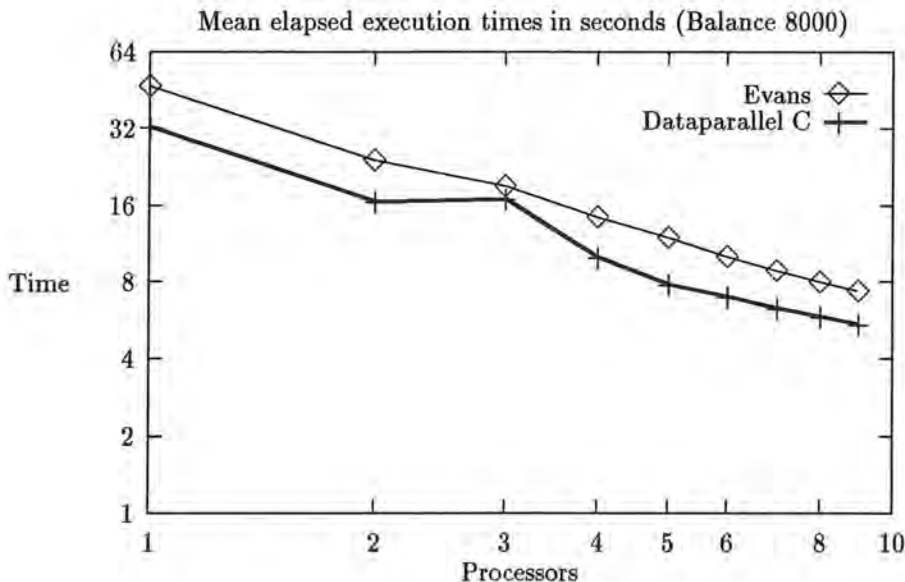

Figure 12: Normal Distribution 10^5 elements

- The Dataparallel C program has to be compiled each time the number of physical processors change, to get a better performance, instead of having a fixed number of virtual processors and running the once compiled program on a different number of physical processors. This is due to the overhead of managing more number of virtual processors than needed. Suppose we run the example case shown in Figure 9 on one physical processor, then that physical processor has to emulate 3 virtual processors. Instead if we make the number of virtual processors equal to one, then the virtual processor overhead is reduced.

- We should be able to perform I/O in the context of the parallelism of the problem. There should be capability to read input data directly into poly variables and write output data from the poly variables in a predetermined order. The predetermined order might depend on the virtual topology of the problem. When support for new topologies is added, it should automatically support I/O also for that topology. For implementing the I/O for Quickmerge, the data from the input is read in a sequential mode, but the data itself goes into the memory of appropriate virtual processors. This was done using the virtual processor index. Similarily, the sorted data is output in sequential mode by taking data from different virtual processor memories and outputting in a predetermined order. For a distributed machine architecture, the above method may not work efficiently. So the language should have the feature for doing I/O in the context of the parallelism of the problem.

3.5 Dynamic Load Balancing

Description of the problem

This is an implementation of a low-cost hypercube load-balance algorithm on an nCUBE/10 using Dataparallel C. The algorithm for this dynamic load balancing was originally developed by Gustafson et al. and implemented on an nCUBE/10. This algorithm was tested by them using a simple particle simulation application. We also tested the Dataparallel C version of the load balance algorithm using a particle simulator. The original authors claim that this technique appears applicable to the efficient parallelization of particle-in-cell methods, finite difference and finite element methods with adaptive meshes, molecular dynamics calculations, and other timestepping applications where dynamic load balancing is required.

This is a divide-and-conquer approach to dynamic load balancing. The balancing is done with global information, and it is distributed over the ensemble rather than performed by a single processor. The model chosen to test this load balance algorithm is a two-dimensional particle simulation where the positions and velocities of the particles, initialized with a uniform random distribution, are a source of dynamic load imbalance throughout the simulation. The particles provide a measure of the work to be done.

In a perfectly balanced state, every node controls an equal number of particles, so that nodes do not sit idle at each time step, waiting for other nodes to finish moving their particles. At every timestep, balance is achieved by changing particle-to-processor assignment. Hence each processor is responsible for a region whose boundaries change, if required, to encompass a fixed number of particles for an ideal load balance. The key feature of this divide-and-conquer approach is the binary domain decomposition.

The load-balance step requires $O((\log_2 P)^2)$ additional operations and $O(1)$ additional storage.

We used X-windows to display the particle simulation graphically.

Primary references : Dragon and Gustafson [5], Devine and Gustafson [4].

We also communicated with the authors for clarifying some of our questions.
The email addresses of the authors are

Jonh L. Gustafson: gus@tantalus.scl.ameslab.gov
Karen D. Devine : kddevin@cs.sandia.gov

Parallelism exploited

In the problem described above, we see that the particles move over a rectangular domain. We can divide the domain into regions containing equal amounts of work, and map these regions onto the nodes of the hypercube. This leads to parallelism since all the nodes will be simultaneously working on their respective regions instead of one single processor working on the whole rectangular domain. However, since

the particles are constantly moving there is bound to be load imbalance after certain number of iterations. If the load balance algorithm uses the master slave method to maintain the balance then then we might lose the parallelism gained from distributing the work. This load balance algorithm depends only on the nodes and uses only neighbor to neighbor communication to achieve the balance.

For each load balance operation there are $D = \log_2 P$ levels of balancing. At the highest level, the balancer operates on the entire hypercube of dimension D. On the next level, subcubes of dimension $D - 1$ are balanced, and so on, down to the lowest level, where subcube of dimension 1 are balanced. At each level, balancing between any subcube pair is independent of other subcube pairs. Also the nodes in any subcube communicate only with their peers in the other subcube and are independent of other node pairs. This pattern of communication leads to high degree of parallelism which results in a low-cost load balancing algorithm.

Description of the algorithm

Consider a general two-dimensional problem to be solved on a hypercube of dimension D. The binary domain decomposition algorithm computes and compares the amount of work in each of the subcubes of dimension $D - 1$. If an imbalance exists, work is moved so that, after the operation, the two subcubes have equal amounts of work. Each of the $D - 1$ dimensional subcubes is then divided into two $D - 2$ dimensional subcubes; the amounts of work in the new subcubes are calculated; and work is exchanged between the subcubes. This dividing and redistributing process is repeated until the algorithm reaches one-dimensional subcubes. If a load imbalance still exists at some level of the hypercube, the algorithm repeats the redistribution process at the same level ; otherwise, the load balancing is done. All nodes in the hypercube have the same amount of work to do, within the granularity of the task size. Thus, the balance is "ideal", not partial.

In particle simulations, the work to do is the calculation of the particles' positions and velocities. Let N be the total number of particles and P be the total number of processors. After the load balancing procedure, every node has N/P particles. The first level of the decomposition divides the N particles into two sets of $N/2$ particles along a line drawn vertically between the two particles nearest the middle of the particle distribution. The second level of the decomposition divides each set of $N/2$ particles into two sets along horizontal lines drawn between the two middle elements in each set. This process continues, with alternating vertical and horizontal divisions, until the entire distribution of particles is divided into sets of size N/P.

No node has knowledge of particles in the other nodes. The only shared information is the "extreme values" of particle coordinates in a subcube. For example, if a vertical division is to be established, the subcube to the right of the division identifies its leftmost two particles as extreme values 'smallest' and 'next-to-smallest'; the subcube to the left of the division identifies its rightmost two particles as extreme values 'largest' and 'next-to-largest'. Each node determines its local extreme values. Then, as the total number of particles in the subcube is calculated, the local extreme values are compared to the extreme values of the other nodes in the subcube, and are

updated by the most extreme values of the subcube.

After the calculations, each node in the subcube knows the total amount of work in the subcube and the extreme values of the subcube. The two subcubes then exchange their work totals and extreme values, and compare them with their neighbor's values. If no redistribution of work is needed, a new boundary is drawn between the subcubes' regions at the average of the extreme values for the subcubes, and the balancing procedure is done on the next level of subcubes. If the subcubes have the same amount of work, but their extreme values show their regions overlap, the extreme particles are exchanged, and the process is repeated at the same level. If a redistribution of work is needed, the subcube with less work takes its neighbor's extreme particle; the subcube with more work removes its extreme particle; the new boundary is established at the average of the extreme value and the next-to-extreme value of the subcube that had more work. If an imbalance exists after the redistribution, a flag is set indicating that the balancer should return to that level. The process then moves to the next finer level.

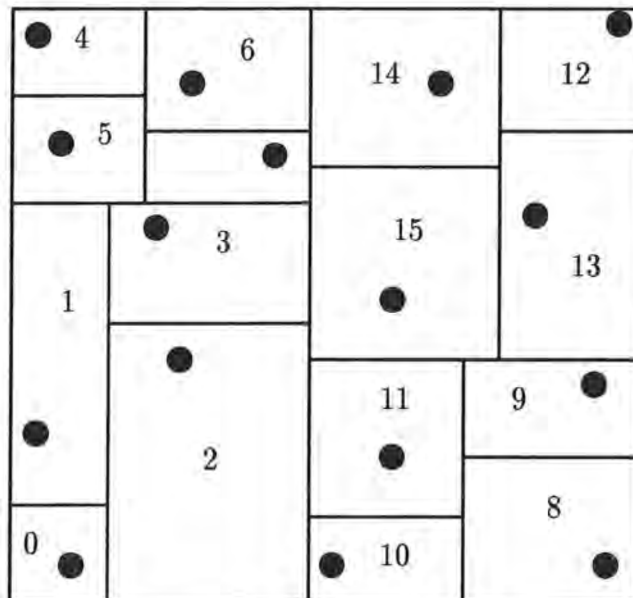

Figure 13: Binary domain decomposition for particle simulation

More detailed description of the algorithm is given in the original reference.

Comparison of performance results

The following figure shows the time taken to perform a single load balance step for different hypercube dimensions. In addition, the time spent on communication by the Dataparallel C version is shown. The timings are given in the Appendix.

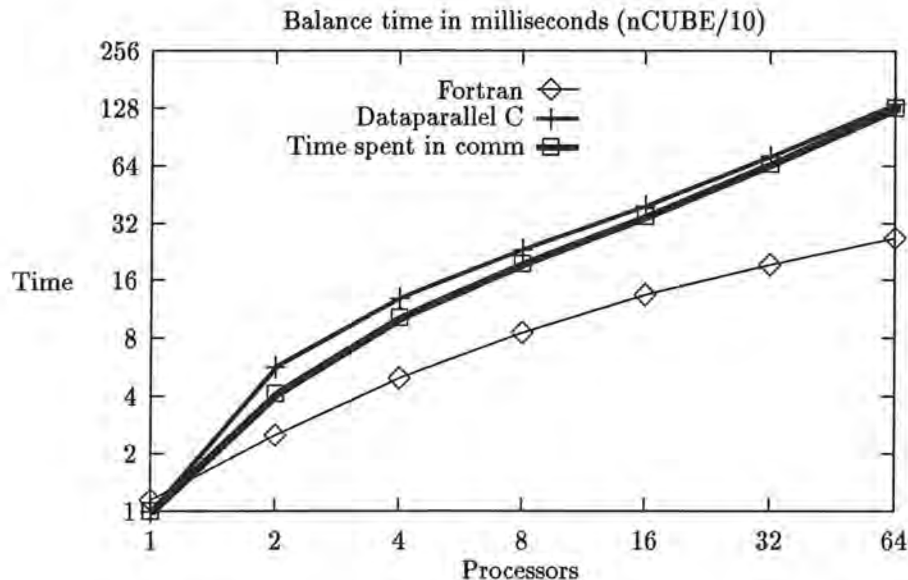

Figure 14: Load balancing times on nCUBE/10

Conclusions

- From the timings, we observe that almost 80 to 90 percentage of the time is spent in communication. The communication is always between neighbors in the hypercube. Most of the time two subcubes communicate with each other. This cube communication is achieved using point-to-point communication which is more efficient than random read or random write. But still it did not give the expected performance, due to the the large time spent copying of buffers during communication.

- When we used member functions inside a while loop, the performance was very poor. This happened because the compiler does not do interprocedural data flow analysis, instead every call to a member function results in a "global or" to synchronize the processors. When the member function was expanded and included inside the loop the "global or" vanished and the performance improved.

- The point-to-point communication had to be achieved in a strange way by way of calling a sequential function inside a domain select, because the only virtual topologies supported by Dataparallel C are ring and mesh, not hypercube.

- We had to use the nCUBE system call `nfwrite()` in order to output the graphics display data. Also we had to use the `ntime()` system call to measure the cumulated time spent in communication.

- Since the compiler maps the virtual processors to the hypercube nodes using binary reflected gray code, when defined as a one dimensional array, the mapping naturally fitted this problem.

- This problem can be used as a real test case for a parallel debugger. The debugging was a little tedious due to the dynamic nature of the problem.
- The graphics display output had to be redirected to a file and then displayed using an X window program. We could not directly pipe the output of the parallel program to the display program because it resulted in a **Segmentation fault** for unknown reasons.

3.6 Linear Equation Solver using Jacobi approach

Description of the problem

The problem is to solve a system of simultaneous linear equations expressed in the following form:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & = & b_2 \\ \vdots & & & & & & & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \ldots & + & a_{nn}x_n & = & b_n \end{array}$$

which is expressed in matrix notation as $\mathbf{Ax} = \mathbf{b}$. The matrix $\mathbf{A}$ is called the coefficient matrix of the system of equations. A system of equations is completely defined by its coefficient matrix $\mathbf{A}$ and right-hand-side vector $\mathbf{b}$.

The system can be solved using 'direct methods' like Gauss-Jordan elimination where the solution is obtained after a predeterminable number of steps. But here we are using the 'Jacobi iterative approach' where successive estimates of the solution is made to arrive at the desired solution eventually.

The Jacobi iterative solution method determines an approximate solution for each element of the vector $\mathbf{x}$, with the $(k+1)$th approximation expressed as

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)}}{a_{ii}}$$

The above equations will converge to a solution provided that

$$\sum_{j=1, i\neq j}^{n} |a_{ij}| < |a_{ii}|$$

Primary references : Kwan and Bic [14, 15], Steinberg [20].
Email reference: kwan@ics.uci.edu.

Parallelism exploited

From the above equations we see that, for an iteration k, the computation of new approximations for the entries of vector $\mathbf{x}$ can be done in parallel. To compute an entry i of the vector $\mathbf{x}$ we need the coefficients from the row i of matrix $\mathbf{A}$, an entry from row i of vector $\mathbf{b}$ and all entries of vector $\mathbf{x}$ of iteration $k-1$. Each process owns a set of rows and computes the approximations for the entries of vector $\mathbf{x}$ corresponding to the rows it owns. This way work is shared among processes and computation for different entries can proceed in parallel.

Description of the algorithm

Each virtual processor owns a single row of coefficient matrix $\mathbf{A}$ and a single entry of vector b corresponding to that row. Two $\mathbf{x}$ vectors, for previous and current

iterations, are defined as a global array. At the end of each iteration, each virtual processor updates its **x** value on the corresponding location of the global array. Then all the virtual processors calculate the maximum deviation, since they all have access to the old and new generation values of the **x** vector. After convergence criteria is met, the program terminates after outputting the solution vector.

Comparison of performance results

The performance for the linear equation solver using Jacobi method for a 128×128 system on an iPSC/2 for both CAB and Dataparallel C versions are shown below. The performance on an nCUBE is also shown in the next page. The reason for this is given in the next subsection. The timings are for 10 iterations in all the cases. CAB stands for Compute-Aggregate-Broadcast paradigm. The timings for the CAB version were noted down from a graph.

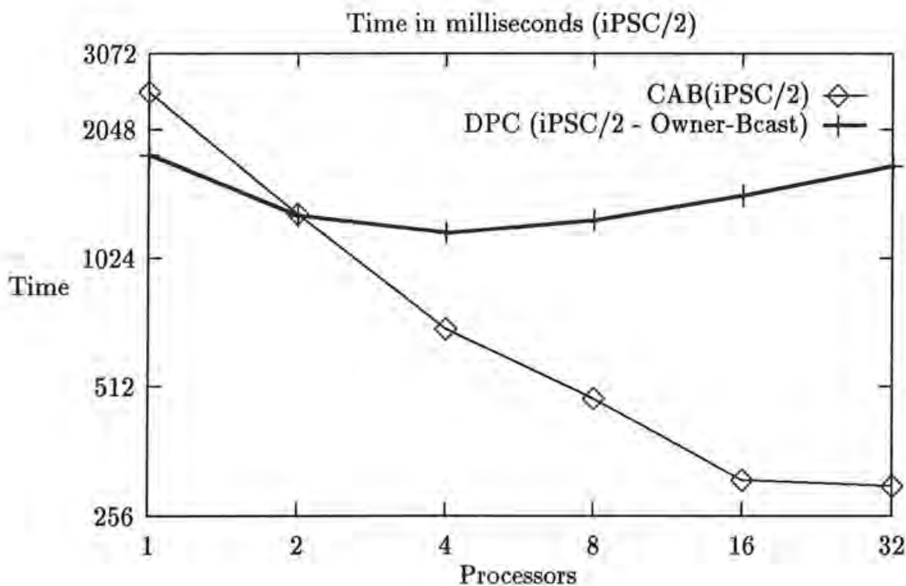

Figure 15: System Solver 128×128 using Jacobi (iPSC/2)

Conclusions

- The **x** vector, which is stored globally, has to be updated after each iteration. The less expensive way for this situation is to use multi-reduce where each virtual processor updates the **x** value corresponding to its virtual processor index, on the global array. Another way is to use owner-broadcast where the global **x** vector is broadcast after each element of the vector is updated. At the time of this benchmarking multi-reduce feature was malfunctioning on iPSC/2. Hence we used owner-broadcast to achieve the same functionality. On the nCUBE, multi-reduce feature was working properly, and so we have shown the performance of Dataparallel C on the nCUBE to give an indication of how

the performance on iPSC/2 will change when owner-broadcast is changed to multi-reduce.

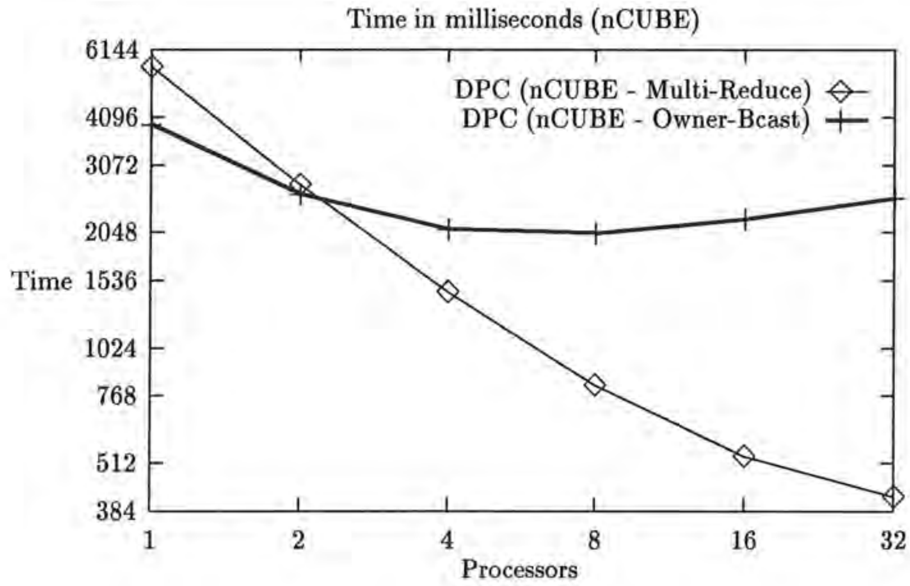

Figure 16: System Solver 128 $\times$ 128 using Jacobi (nCUBE)

3.7 Gaussian Elimination

Description of the problem

The problem is to solve a system of simultaneous linear equations expressed in the following form:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & = & b_2 \\ \vdots & & & & & & & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \ldots & + & a_{nn}x_n & = & b_n \end{array}$$

which is expressed in matrix notation as $\mathbf{Ax} = \mathbf{b}$. The matrix $\mathbf{A}$ is called the coefficient matrix of the system of equations. A system of equations is completely defined by its coefficient matrix $\mathbf{A}$ and right-hand-side vector $\mathbf{b}$.

Elsewhere we have solved this problem using the Jacobi iterative approach. Here we will be using a direct method called Gaussian Elimination method to solve the above system of equations. Essentially, this method reduces the augumented matrix $[\mathbf{A}, \mathbf{b}]$ to an upper triangular matrix possessing identical solution as the original system of equations. Then by using back substitution the unknowns are found.

Primary references : Rivera [19], Hatcher and Quinn [10], Steinberg [20].

Parallelism exploited

The performance of this problem depends on the way the matrix is partitioned. Each virtual processor can own a row, a column or a single element. In any case the maximum parallelism available is during the reduction of the augumented matrix to upper triangular form. One of the main considerations in solving this problem will be to reduce the communication without affecting the parallelism. If we are solving the problem on a shared memory machine, the communications will reduce to non-local accesses and we can get away with very few synchronization points (in the order of number of rows or columns).

Description of the algorithm

Given an $N \times N$ matrix $\mathbf{A}$, where rows are numbered $1, 2 \ldots, N$. The algorithm has N iterations. During iteration i, where $1 \leq i \leq N$, Gaussian elimination forces to 0 the column i elements in rows $i+1, i+2, \ldots, N$ by multiplying the elements in each of these rows j by some multiple of the elements in row i. The *pivot* row is the one with the largest value in the column i, the pivot column. The element i in the pivot row is the pivot element.

The back substitution solves $\mathbf{Ux} = \mathbf{B}$, where $\mathbf{U}$ is an upper triangular matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors.

Each virtual processor is associated with a row of matrix $\mathbf{A}$. The pivot row is identified by means of a tournament and then broadcast so that every virtual processor can reduce its rows in parallel. Then back substituition is achieved by means of repeated broadcasts.

Comparison of performance results

We compared the performance of Dataparallel C with that of another SIMD style language called Array C LANguage (ACLAN) for this problem. The problem size was 120×120 and it was solved on an nCUBE hypercube.

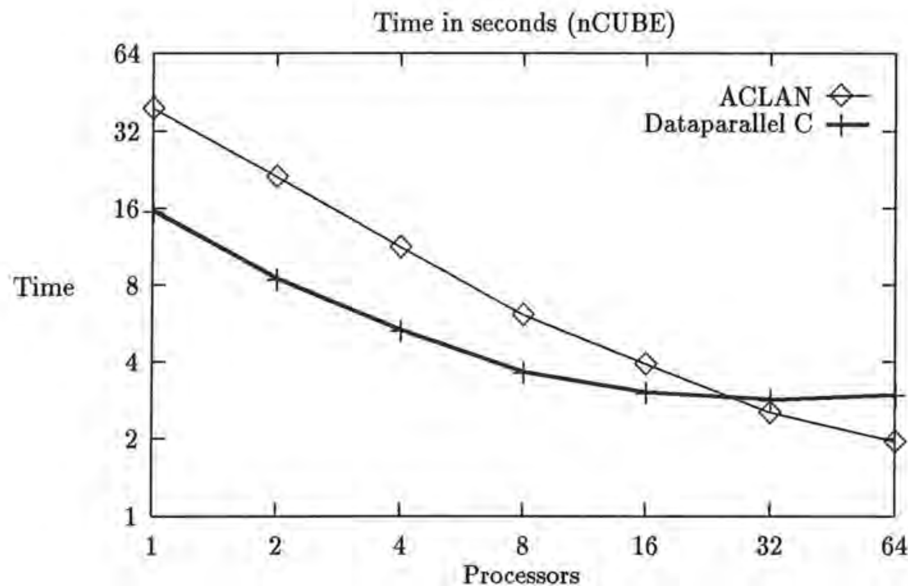

Figure 17: Gaussian elimination with partial pivoting (120×120)

Conclusions

- From the performance graphs we see that Dataparallel C performs better than ACLAN except for 5 and 6 dimensions. This might be explained by the load imbalance because of some physical processors owning lesser number of rows than others due to the problem size chosen. (120×120). And this imbalance is magnified as the dimensions increase. But we do not know the exact reason yet.
- The Dataparallel C code is more intelligible than ACLAN code because ACLAN extensively uses MASKS to enable or disable a processor from participating in a computation or communication step.

3.8 Matrix Multiplication

Description of the problem

The problem is to multiply two matrices **A** and **B** to yield matrix **C**.

Primary references : Kalé [12], Hatcher and Quinn [10].

Parallelism exploited

Matrix multiplication is embarassingly parallel, because each element of the result matrix **C** can be computed independently of the others. Due to this high degree of parallelism, there are many ways to parallelize matrix multiplication. We are using an algorithm which is familiarly known as systolic matrix multiplication.

Description of the algorithm

The systolic matrix multiplication algorithm described here is better suited for distributed memory multicomputers, though it performs well on shared memory multiprocessors as well. On a shared memory machine, the matrices **A** and **B** can be stored globally and each processor need not have an individual copy of them. But on distributed memory computers, the global variables are stored in each physical processor. Given a small node memory, storing large matrices would be ruled out. For this reason we assume that every element of **A, B,** and **C** is stored only once in the parallel computer. Also the costs should not outweigh the reduction in memory costs. The systolic algorithm addresses both these above issues and hence gives a good performance.

Though we have talked about physcial processors, we will be dealing with virtual processors while writing the algorithm. How big a matrix we can multiply and how efficient the algorithm can perform depends on the number of physical processors we choose to run the program.

Let us consider the multiplication of two two-dimensional matrices **A** and **B** whose product will be another two-dimensional matrix **C**.

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \times \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00} \times b_{00} + a_{01} \times b_{10} & a_{00} \times b_{01} + a_{01} \times b_{11} \\ a_{10} \times b_{00} + a_{11} \times b_{10} & a_{10} \times b_{01} + a_{11} \times b_{11} \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix}$$

Each virtual processor owns an element (usually a block of elements) of matrix **A, B** and **C**.

To start with, each virtual processor computes the product of a and b they own and add the result to their own c. In the next step, virtual processors send their a's to their west and b's to their north. Virtual processors on the west extreme send their a's to the virtual processors on the east extreme of same row and similarily virtual processors on the extreme north send their b's to the virtual processors on the extreme south of same column. Then, each virtual processor computes the product $a \times b$ and adds the result to their own c. The following figure illustrates the process for a two-by-two matrix multiplication. When the values in the east and south extremes reach the west and north extremes respectively, the product is computed like in the

previous steps and the process stops, and the result is available in the **C** matrix which is distributed among the virtual processors.

The initial distribution of the elements a and b has to be skewed to get the right answers.

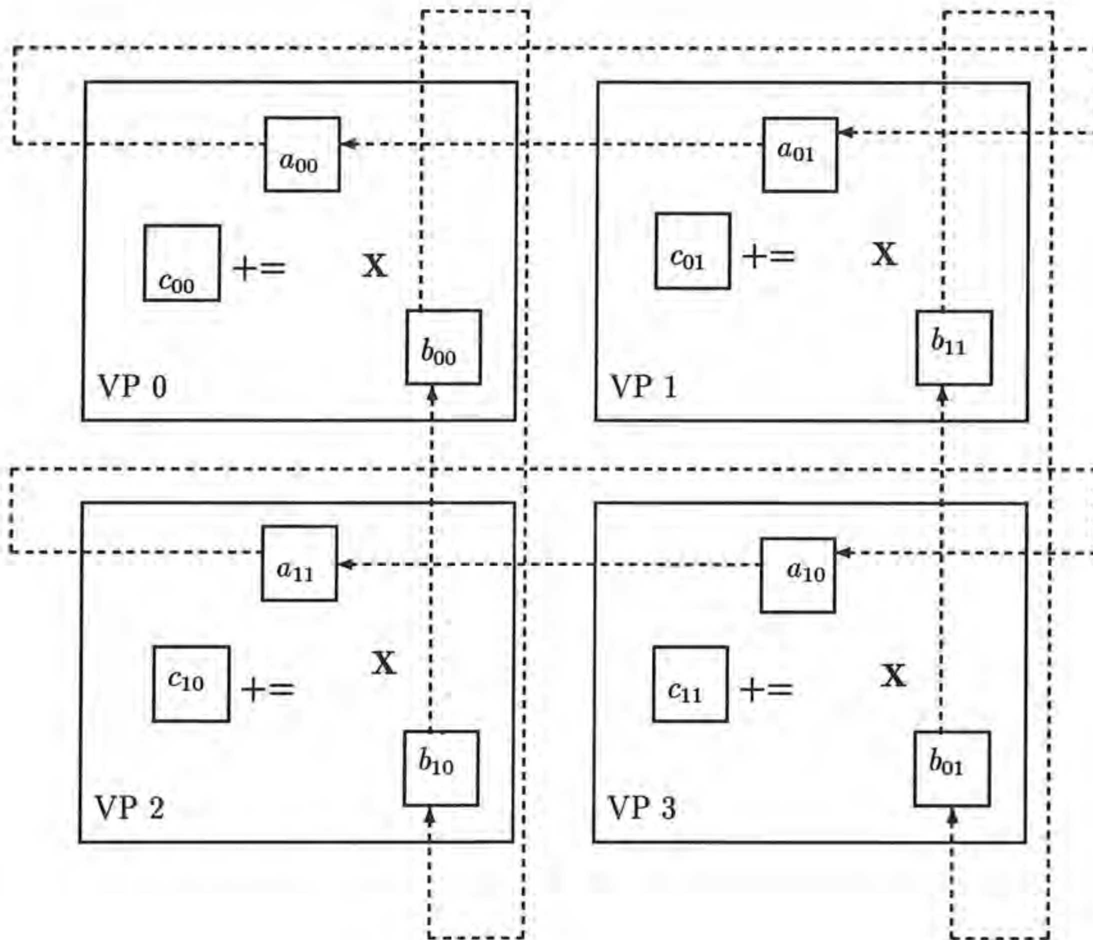

Figure 18: Systolic matrix multiplication

Comparison of performance results

The performance comparison was done on a Sequent Symmetry and on an iPSC/2 for the multiplication of 256 $\times$ 256 matrices. We have compared the Dataparallel C performance with that of the Chare Kernel performance. The detailed timings are given in the Appendix. The graphs appear in the next page.

Conclusions

- We see from the graphs that the algorithm performs equally well in both the shared memory and distributed memory cases.

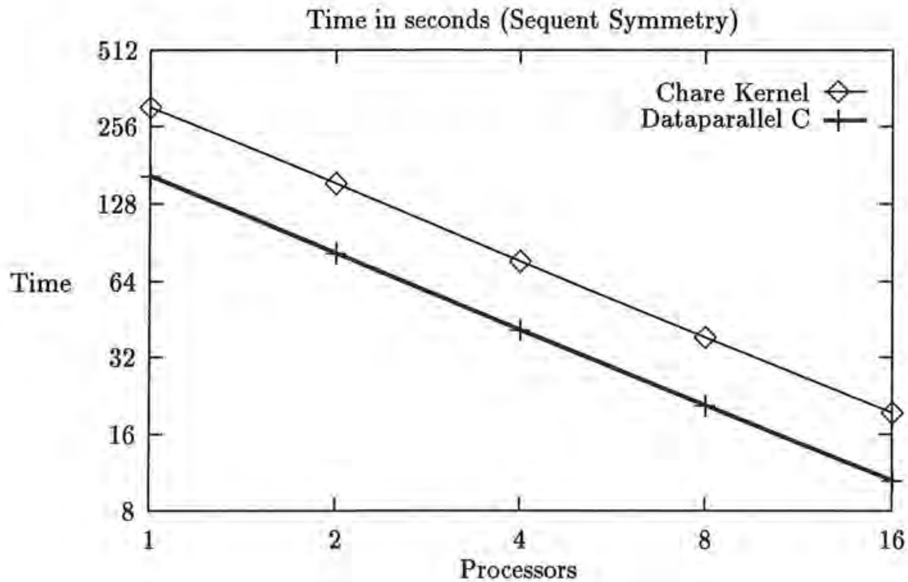

Figure 19: Performance of 256 $\times$ 256 matrix multiplication

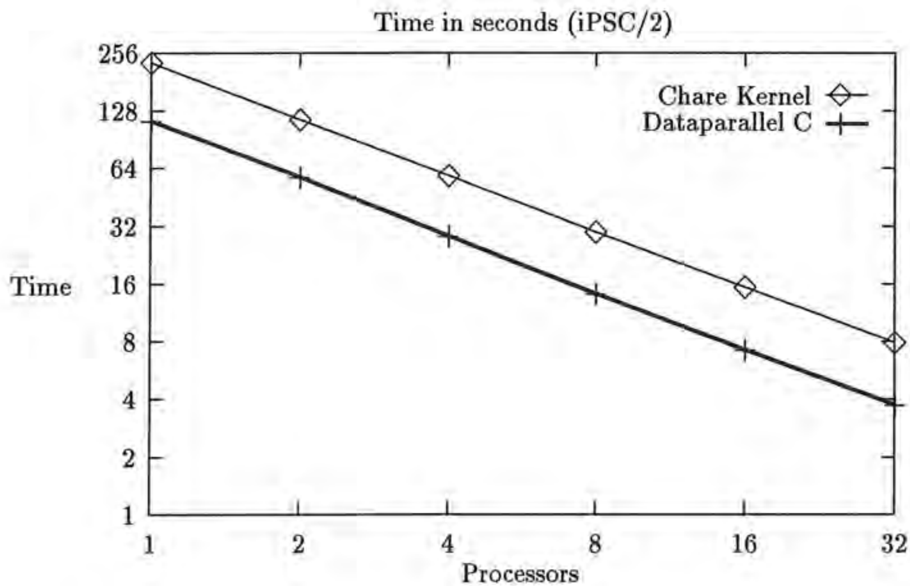

Figure 20: Performance of 256 $\times$ 256 matrix multiplication

- When we did a linear plot of Time vs Processors, it seemed that Dataparallel C started off with a better timing than Chare Kernel but the performance difference seemed to narrow down as the number of processors were increased. But when we did a linear log plot, we found out that Dataparallel C performance was better than Chare Kernel by a constant factor for all number of processors.

3.9 Prime Finder

Description of the problem

The problem is to find all the primes under a given number N. A prime number has the property that it is not divisible by any number other than itself and 1. To say if a given number i is prime or not we have to divide i by all possible factors it can have and if we find that there is no factor for i other than itself and 1 then we conclude i is prime. The largest factor a number can have other than itself is its square root. So it is enough to eliminate the factors up to the square root of a given number to verify its primality.

If we directly implement the above method to test the primality, we will be wasting a lot of computation time in repeated divisions. This method might be useful to test if a given number is prime or not. But in order to find all the primes under a given N, we need a better algorithm which will avoid duplication of effort and also expensive computations. Here we are parallelizing the classical prime-finding algorithm, the Sieve of Eratosthenes, which uses less expensive computations, avoids duplication of effort and scales very well.

Primary references : Kalé [12], Bjornson [1], Hatcher and Quinn [10].

Parallelism exploited

Each process is associated with a chunk of natural numbers. When a new prime value is found, each process independently strikes out the multiples of the newly found prime number. This way all processes can work in parallel and synchronize only to get the new prime value. Any increase in the problem size will increase the grain size also.

Description of the algorithm

Each virtual processor owns an array of boolean values whose indices correspond to the natural numbers they represent. Even numbers are implicitly left out from being stored, since they all are divisible by 2. To start with, all virtual processors eliminate the multiples of 3, which is the first prime number after 2. Usually the virtual processor zero will own all numbers $\leq \sqrt{N}$. So the successive prime values will be broadcast always by virtual processor zero. All the elements in the array are initially true. Making an element false is equivalent to striking out the number which it represents. After all the possible factors $\leq \sqrt{N}$ have been considered, the natural numbers whose corresponding array elements are true are the prime numbers under N.

Comparison of performance results

The performance of Dataparallel C for finding primes less than 3 million and 7 million on an iPSC/2 are plotted below. The timing for the C-Linda version is available only for finding primes under 7 million and that too only on 32 nodes of iPSC/2. The performance of Chare Kernel, C-Linda and Dataparallel C version using a naive algorithm for finding primes is also shown.

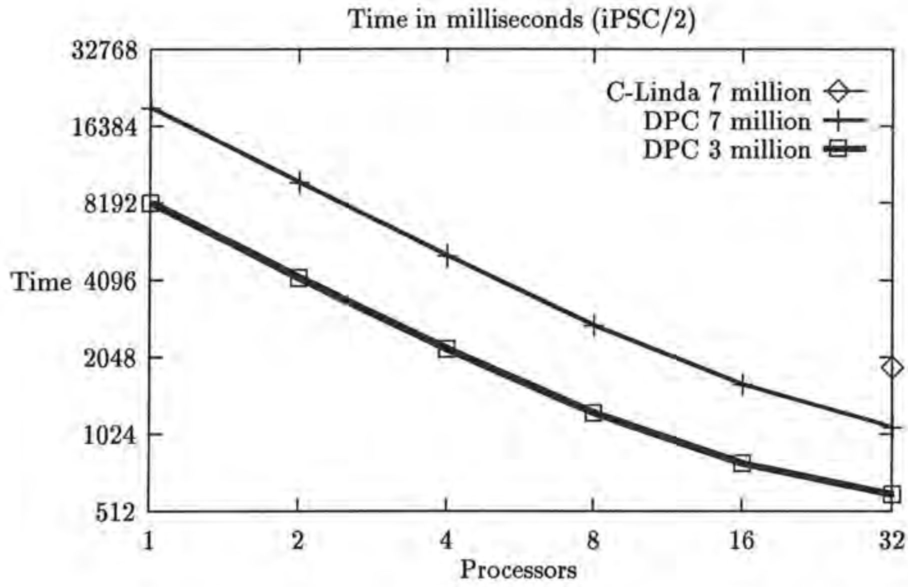

Figure 21: Sieve of Eratosthenes

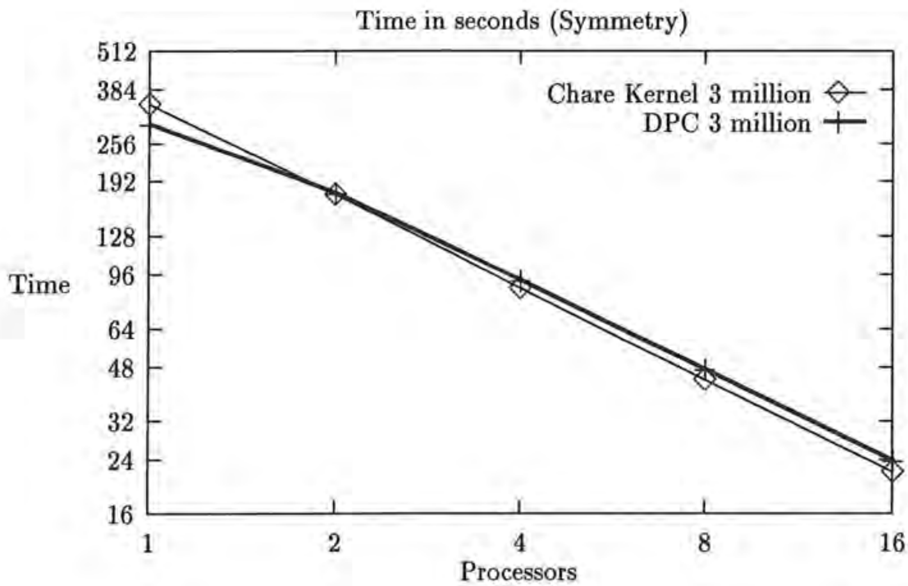

Figure 22: Performance of Prime Finder (naive algorithm)

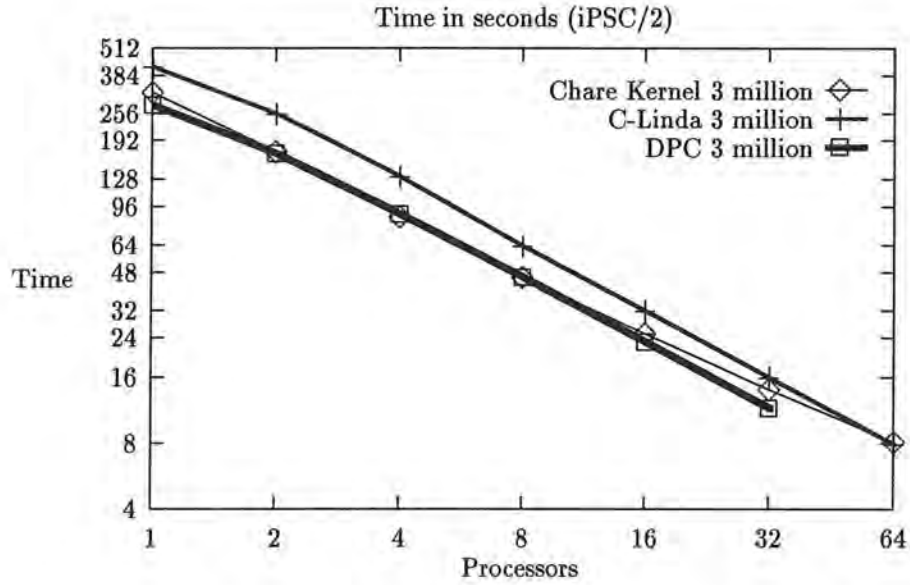

Figure 23: Performance of Prime Finder (naive algorithm)

Conclusions

- We see from the above performance graphs that Sieve algorithm is much faster when compared to the naive algorithm to find primes. However, Dataparallel C performs equally well for the naive algorithm when compared to Chare Kernel and C-Linda.

4 Strengths and Weaknesses of Dataparallel C

From our experience with high-level sequential languages we have seen that flexibility, ease of programming, portability, modularity and maintainability are some of the major features of a well evolved language. These features are more important from the point of programmer productivity and maintenance costs, though they do not prevent generation of efficient code. So the primary consideration should be to design a language which is easy to program and maintain. The compiler should be mainly responsible for generation of efficient code.

The above features are even more important for a high-level parallel programming language because it has to support expressions to specify parallel operations and interactions among those operations. This new dimension leads to a high degree of complexity in developing a parallel program. Though in the case of parallel programming, it is very hard to expect the compiler to do everything, it is still possible for the compiler to generate highly efficient code, especially if the language is explicitly parallel.

Dataparallel C was designed with some of the above considerations. Dataparallel C has the following characteristics [11].

- Imperative style.
- Explicit parallelism
- Local view of computation
- Synchronous execution of a single instruction stream.
- Global name space

As a result, Hatcher and Quinn claim that it has the following desirable attributes [11].

- Versatility
- Practicality
- Programmability
- Portability
- Reasonable performance

From the experience of solving 9 different problems using Dataparallel C, we try to verify how well Dataparallel C has satisifed the above claims and what are its strengths and weaknesses when compared to other paradigms which we have encountered while

solving these 9 problems and what further improvements need to be done to the language and its compiler to make it more useful.

All the problems we had chosen naturally fitted into data parallel category though some had control parallel solutions also. This corroborates with the view of Fox [8] that "domain decomposition" or "data parallelism" is the key to the success of most parallel applications. Also Fox [8] says that $\approx 90\%$ of the scientific and engineering applications are *synchronous* or *loosely synchronous*. The problems we had chosen belong to the category *loosely synchronous* because the grain size between synchronizations was macroscopic. Now we shall consider each claim in detail.

Versatility

The problems were chosen from papers published with benchmark results on shared memory (Sequent Symmetry & Sequent Balance) and distributed memory machines (nCUBE & iPSC/2). The following is the list of problems solved, the languages originally used to solve the problem and the machines on which performance was measured.

Problem	Language	Machines
3.1 Sequence Matching	C-Linda	Symmetry, iPSC/2
3.2 Laplace equation using Jacobi	Poker	Symmetry
3.3 Laplace equation using Gauss-Seidel	CAB	iPSC/2
3.4 Quickmerge	Low-level	Balance 8000
3.5 Load Balancing	Fortran	nCUBE
3.6 Linear equation using Jacobi	CAB	iPSC/2
3.7 Gaussian elimination	ACLAN	nCUBE
3.8 Matrix multiplication	Chare Kernel	Symmetry, iPSC/2
3.9 Prime finder	C-Linda Chare Kernel	iPSC/2

We were able to solve all of the above problems using Dataparallel C and measure the performance on the same machines for which performance was published. Except for Problem 3.3 we could program all other problems with least difficulty. Problem 3.3 in itself was a little complex problem due to its dynamic nature and the code size was around 850 lines. The problems encompass a wide range of communication patterns and different decompositions. So we can say that Dataparallel C is quite versatile.

Practicality

Sometimes we found that it is easy to directly write the parallel code instead of converting the sequential code to parallel code. This way we could avoid at least one level of nesting in loops. However some problems we wrote a sequential version and conversion to parallel version took little effort. After a little practice, one will find it easy to write the parallel code directly unless the sequential code is needed to verify the solution etc.

Programmability

Synchronous execution removes the main headache of race conditions and with a minimum understanding of machine architecture (at least to the extent of knowing if we are programming for shared or non-shared memory) and the Dataparallel C model one can implement all the above algorithms without worrying about messages and locks. However for problem 3.3 we had to be aware of hypercube architecture because the algorithm exploits the node arrangement. This is also because Dataparallel C currently supports ring and mesh topologies only. It is debatable if a topology like hypercube need to be supported by a high-level language at all or a three-dimensional toroidal mesh mapped onto a hypercube would be sufficient for all practical purposes.

Dataparallel C programs are pretty short when compared with other high-level and low-level parallel languages.

Debugging is one issue which needs attention. When a Dataparallel C program is compiled and executed on a parallel machine and if the program comes out saying Segement violation or if the program hangs midway, it is hard to locate where exactly the problem would be. At present "printfs" are the only mechanism to locate a bug. A primitive mechanism to locate system errors and hangs would be of great help because other things can be managed with "printfs" at least till a complete debugger is developed.

Since interprocedural analysis is not yet implemented in the compiler, using member functions inside loops generates synchronizations even in cases where it is not necessary.

Portability

We see from the above table that Dataparallel C works on nCUBE, iPSC/2, Sequent Symmetry and Sequent Balance. The same program runs on all the above machines without need to change. However, a Dataparallel C program written for a distributed memory machine will be more portable without sacrificing performance rather than writing first for shared memory and using it for distributed memory machine. This difference is due to the cost free non-local reads in the shared memory machines a feature which one would like to exploit.

Reasonable performance

From the performance results we see that Dataparallel C is close to the performance of other languages and in some cases better. There is still more scope for optimizing the code. Except for the Problem 3.3 where communication time is predominant, Dataparallel C performance is definitely reasonable and if coding effort is taken into consideration, the performance is more than reasonable.

We see that Dataparallel C satisfies the above claims to a large extent and it is evolving in the right direction as we are able to program with ease many parallel problems. Now let us see what is its strength and weakness when compared to other parallel paradigms we have encountered.

C-Linda [3] vs Dataparallel C

The C-Linda program for the Problem 3.1 (Sequence matching) requires separate code

to be written for master and worker tasks. The data has to be dumped to the tuple space by the master process specifying the correct actuals and the worker processes have to read from the tuple space using the correct formals and appropriate tuple key. After that, the edges are communicated through the tuple space. The comparison itself is done by the program called "similarity.c". In contrast, the Dataparallel C program uses only the logic of "similarity.c" and in addition has a **successor()** macro to communicate the edges. The C-Linda code is not synchronous in execution. The C-Linda model is not very intuitive for "data parallel" problems. So short code size, synchronous execution and intuitive to solve "data parallel" problems are some of the strengths of Dataparallel C which was helpful in programming this problem.

The Dataparallel C supports only static decomposition whereas C-Linda can handle decompositions at run time. Hence we had to compile Dataparallel C programs each time a problem size or aspect ratio was changed. This was one of the weaknesses we encountered in this problem.

Also we had a small loss of performance on Sequent due to the synchronization costs. Whereas on iPSC/2 the message passing itself acted as synchronization and so there was no extra cost to perform synchronizations.

C-Linda offers a debugging tool called Tuplescope, but we do not know how useful it is for practical debugging.

CAB [17] vs Dataparallel C

Compute-Aggregate-Broadcast is a paradigm specially meant for iterative problems which have a compute phase, aggregate phase and a broadcast phase [17]. The structuring technique [14] used for CAB paradigm has the concept of master and slave processes. The programmer has to keep in mind what the master process has to do and what the slave processes have to do. In contrast, Dataparallel C needs an initial specification of the decompostion and the same set of operations are performed on different data elements in a synchronous manner. The problem used for the comparison was solving Laplace rectangle using Gauss-Seidel iteration. We used the same algorithm as they had used. The decomposition was by rows. Again the communication macros **successor()** and **predecessor()** were sufficient to exchange the edges. Dataparallel C does not have a means to specify staggered start. Whether it should be taken care of by the language or the programmer is an issue to be decided.

ACLAN [19] vs Dataparallel C

Array C LANguage is a SIMD style language for MIMD computers like Dataparallel C. However ACLAN has a notion of MASKS, which are used to enable or disable processes from participating in the computation or communication. These masks make the program hard to read. The virtual processor index used by Dataparallel C is a better way of accomplishing the same task.

Poker [16] vs Dataparallel C

The Laplace rectangle using Jacobi was solved by Snyder [16] using the C language and Poker style message passing using ports. Their code length is almost 5 times

longer than Dataparallel C code, since it was a complete implementation using Sequent C language. From this we can see how difficult it is to program in a low-level language even with the help of some message passing constructs.

5 Conclusions

In general, the local view of computaion provided by Dataparallel C allows you to treat parallel programming just like sequential programming. Global name space allows you to forget about details of message passing. Synchronous execution facilitates easy debugging and coding.

The problems considered here are small sized except for Problem 3.3, which we can say is medium sized. These problems did not require more than one domain instance, did not have multiple modules interacting with each other and there was no need for complex communication. But from this exercise we find that Dataparallel C performs equally well and sometimes better when compared with other paradigms for small sized problems. The future work would be to program big applications or benchmarks which are fairly complex and at least around 5000-10,000 lines. This way we can really test all the features and also know exactly what else needs to be done to make Dataparallel C a full-fledged parallel language. Of course, to develop bigger applications we need some preliminary debugging tools also.

References

[1] R. Bjornson. Experience with Linda on the iPSC/2. *The Proceedings of The Fourth Conference on Hypercubes, Concurrent Computers and Applications*, 1989, Vol 1, pages 493-500.

[2] B. Carnahan, H. A. Luther and J. O. Wilkes. Applied Numerical Methods, 1969, John Wiley & Sons, Inc.

[3] N. Carriero and D. Gelernter. How to Write Parallel Programs. A First Course. The MIT Press, 1990. Chapter 7.

[4] K. D. Devine and J. L. Gustafson. A Low-Cost Hypercube Load-Balance Algorithm. *MPCRL Research Reports*, DOE's Massively Parallel Computing Research Laboratory, Sandia National Laboratories, Albuquerque.

[5] K. M. Dragon and J. L. Gustafson. A Low-Cost Hypercube Load-Balance Algorithm. *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications*, 1989, Vol 1, pages 583-589.

[6] T. Dunighan. Performance of a Second Generation Hypercube. Oak Ridge National Laboratory, Technical Report ORNL/TM-10881, Nov 1988.

[7] D. J. Evans. A parallel sorting-merging algorithm for tightly coupled multiprocessors. *Parallel Computing*, 14, 1990, pages 111-121.

[8] G. C. Fox, " What have we learnt from using real parallel machines to solve real problems ?," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 897-955, ACM Press, 1988.

[9] O. Gotoh. An improved Algorithm for Matching Biological Sequences. *Journal of Molecular Biology*, 162, pages 705-708.

[10] P. J. Hatcher and M. J. Quinn. Data-Parallel Programming. The MIT Press, 1991.

[11] P. J. Hatcher, M. J. Quinn, A. J. Lapadula, R. J. Anderson, and R. R. Jones. Dataparallel C: A SIMD Programming Language for Multicomputers. *The Sixth Distributed Memory Computing Conference Proceedings*, 1991, pages 91-98.

[12] L. V. Kalé. The Chare Kernel Parallel Programming Language and System. *1990 International Conference on Parallel Processing*, II, pages 17-25.

[13] D. E. Knuth. The Art of Computer Programming, Vol 2, Seminumerical Algorithms. Addison-Wesley Publishing Company.

[14] A. W. Kwan and L. Bic. A Structuring Technique for Compute-Aggregate-Broadcast Algorithms on Distributed Memory Computers. *Proceedings of the Sixth International Parallel Processing Symposium*, March 23-26, 1992.

[15] A. W. Kwan and L. Bic. Using Parallel Programming Paradigms for Structuring Programs on Distributed Memory Computers. *The Sixth Distributed Memory Computing Conference Proceedings*, 1991, pages 210-214.

[16] C. Lin and L. Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. *International Conference on Parallel Processing*, 1990, pages II-163-170.

[17] P. A. Nelson and L. Snyder. Programming Paradigms for Nonshared Memory Parallel Computers. *The Characteristics of Parallel Algorithms* (L. H. Jamieson, D. B. Gannon and R. J. Douglass, editors). The MIT Press, 1987, pages 3-20.

[18] M. J. Quinn. Parallel sorting algorithms for tightly coupled multiprocessors. *Parallel Computing* 6, (1988), pages 349-357.

[19] F. F. Rivera, R. Doallo, J. D. Brugerra, E. L. Zapata and R. Peskin. Gaussian elimination with pivoting on hypercubes. *Parallel Computing*, 14, 1990, pages 51-60.

[20] D. I. Steinberg. Computational Matrix Algebra, McGraw-Hill, Inc.,New York, 1974.

Appendix A – Performance timing charts

Timings for problem 3.1

The original C-Linda version of the algorithm was implemented both on a shared memory machine (Sequent Symmetry) and a distributed machine (iPSC/2). Since Dataparallel C was available on both the machines, we could compare our performance results with that of the C-Linda performance. The iPSC/2 timings for C-Linda version was not available. Instead we have compared the speedup achieved by C-Linda and Dataparallel C for a 7000 × 7000 self-comparison problem elsewhere in the document.

Execution times on Symmetry (seconds)										
Side × Top	Aspect Ratio	Language	Number of Processors							
			10	12	14	16	18	20	22	24
3389 × 3389	10	C-Linda	18.31	15.58	13.36	11.94	10.73	9.81	9.16	8.86
		DPC	21.65	17.99	15.03	13.26	11.75	10.66	9.70	9.17
	20	C-Linda	17.88	14.94	12.91	11.42	10.49	9.61	8.83	8.65
		DPC	20.66	16.81	14.49	12.75	11.48	10.25	9.38	8.88
	30	C-Linda	17.53	14.96	12.73	12.28	10.29	9.49	8.78	9.72
		DPC	20.22	16.73	14.36	12.57	11.23	10.16	9.29	8.73
	40	C-Linda	17.36	14.75	12.71	11.28	10.38	9.57	8.92	10.56
		DPC	19.63	16.52	14.20	12.55	11.22	10.18	-	-
6778 × 6778	10	C-Linda	79.32	65.39	56.35	46.91	41.14	37.55	33.71	31.31
		DPC	85.69	76.27	59.67	53.06	48.25	42.23	38.70	35.36
	20	C-Linda	71.46	64.24	55.79	45.91	41.47	35.12	32.25	30.03
		DPC	80.17	67.37	57.33	55.67	44.51	40.77	37.05	34.47
	30	C-Linda	71.93	58.5	49.17	44.48	38.46	37.07	32.76	29.55
		DPC	79.50	67.18	56.37	52.64	44.56	41.47	36.22	33.36
	40	C-Linda	70.14	58.25	49.54	42.91	40.70	37.20	31.94	29.99
		DPC	83.63	74.14	62.52	50.11	45.14	39.59	35.92	33.58

The above timings include the time taken to read the input data file from the disk. Due to some unknown bug, the DPC program gave segmentation fault, in two cases. Hence the two blank columns.

Timings for problem 3.2

We did the benchmarking of their program and ours on a Sequent Symmetry. Their program was written using the C language + Sequent parallel library. Since they used **getusclk()** for taking the timing we also did the same thing to ensure a fair comparison.

Execution times on Symmetry (seconds)				
Language	Decomposition	Data Points	Number of Processors	
			4	16
DPC	Blockwise 139 lines of source code	8×8	0.006	0.009
		128×128	0.46	0.12
		256×256	1.92	0.48
		512×512	7.72	2.01
		1024×1024	31.07	8.00
	Rowwise 83 lines of source code	8×8	0.005	0.007
		128×128	0.35	0.10
		256×256	1.47	0.45
		512×512	5.88	1.92
		1024×1024	24.12	8.09
	Cellwise 72 lines of source code	8×8	0.005	0.006
		128×128	0.72	0.35
		256×256	2.86	1.63
		512×512	7.95	3.68
		1024×1024	1025	800
Sequent C	Blockwise 566 lines of source code	8×8	0.005	0.005
		128×128	0.42	0.10
		256×256	1.72	0.43
		512×512	6.90	1.74
		1024×1024	27.38	6.96

The recorded timing is the time taken elapsed from the start to termination of the iterative loop. We noticed that if the decomposition is interleaved which is the default, then the performance becomes bad due to cache trashing. But contiguous decomposition improves the performance. This is prominent in cellwise decomposition.

Timings for problem 3.3

The timings for the 128 $\times$ 128 and 256 $\times$ 256 Laplace Equation Solver is given here for both Dataparallel C and Compute-Aggregate-Broadcast paradigm. The timings were taken on a 32 node iPSC/2. The timings for processors less than 4 are not given because it would approximately take around 6 hours for a 256 $\times$ 256 problem to complete. And the timings given here are sufficient indication of the performance. The timings for the CAB paradigm were noted down from a graph and hence the actual timings may be $\pm 2\%$ of the timings given below.

Execution times on iPSC/2 (seconds)					
Size	Language	Number of Processors			
		4	8	16	32
128 × 128	CAB	650	320	190	110
	DPC	520.34	275.24	156.18	97.90
256 × 256	CAB	5500	3000	1400	850
	DPC	4712.71	2396.51	1251.33	682.29

Timings for problem 3.4

Mean elapsed times for the Quickmerge algorithm on Sequent Balance 8000 processor for number of elements $N = 1000$, 10 000 and 100 000 for three kinds of distributed pseudo random data sets namely uniform, ordered and normal. Data-parallel C version timings are indicated by DPC and the timings published by Evans [3] is indicated by Evans.

Execution times on Balance 8000 (seconds)											
Dist	Size	Lang	Number of Processors								
			1	2	3	4	5	6	7	8	9
UNIF	1000	Evans	-	-	-	-	-	-	-	-	-
		DPC	.21	.12	.08	.08	.07	.07	.08	.08	.13
	10000	Evans	-	-	-	-	-	-	-	-	-
		DPC	2.55	1.49	1.25	.998	.912	1.04	1.39	.71	.81
	100000	Evans	47.50	24.00	18.00	14.50	12.40	10.00	8.00	7.50	7.00
		DPC	31.03	16.44	13.67	9.82	7.76	6.99	6.24	5.83	5.50
ORD	1000	Evans	.11	.16	.26	.35	.43	.51	.59	.68	.75
		DPC	.21	.11	.10	.11	.12	.13	.16	.17	.22
	10000	Evans	1.73	1.03	1.17	1.35	1.44	1.56	1.67	1.77	1.85
		DPC	2.21	1.18	1.10	1.15	1.24	1.54	1.79	1.88	2.38
	100000	Evans	21.44	11.24	11.43	12.12	12.60	12.54	12.83	13.08	13.30
		DPC	23.79	13.97	18.02	13.03	13.07	14.15	16.23	16.75	23.83
NORM	1000	Evans	.27	.25	.30	.35	.42	.48	.55	.62	.69
		DPC	.20	.15	.12	.09	.08	.09	.09	.09	.18
	10000	Evans	3.64	1.99	1.74	1.46	1.31	1.23	1.21	1.21	1.24
		DPC	2.54	2.28	1.85	2.21	2.27	2.17	2.80	.71	.84
	100000	Evans	46.82	23.95	18.99	14.42	11.97	10.07	8.86	7.99	7.37
		DPC	32.35	16.54	16.92	9.99	7.79	6.98	6.29	5.85	5.46

The dashes indicate data not available.

Timings for problem 3.5

Execution times on nCUBE/ten (milliseconds)								
Language	Timing breakup	Number of Processors						
		1	2	4	8	16	32	64
DPC	Total balance time	0.44	5.70	12.91	23.28	39.70	71.50	135.04
	Time spent in comm	0.00	4.12	10.21	19.43	34.74	65.25	127.49
Fortran	Total balance time	1.14	2.51	4.97	8.53	13.40	19.32	26.72

The global domain was the rectangle with corner points (0,0) and (1024, 768). The number of particles per processor was held constant (8 particles per processor). The timings are for one time step excluding the graphics. Each component of the velocity was in the range (-1, 1).

Timings for problem 3.6

The timings for the linear equation solver using Jacobi method for a 128 $\times$ 128 system on an iPSC/2 for both CAB and Dataparallel C versions are given below. At the time of this benchmarking, multi-reduce feature was malfunctioning on iPSC/2. Hence we used owner-broadcast to achieve the same functionality. On the nCUBE, multi-reduce feature was working properly, and so we have shown the performance of Dataparallel C on the nCUBE to give an indication of how the performance will change when owner-broadcast is changed to multi-reduce. The timings are for 10 iterations in all the cases. CAB stands for Compute-Aggregate-Broadcast paradigm. The timings for the CAB version were noted down from a graph.

Execution times on iPSC/2 and nCUBE (milliseconds)							
Language	Machine	Number of Processors					
		1	2	4	8	16	32
CAB	iPSC/2	2500	1300	700	480	310	300
DPC	iPSC/2 Owner Broadcast	1789	1290	1175	1260	1442	1688
	nCUBE Owner Broadcast	3907	2574	2082	2033	2213	2512
	nCUBE Multi-Reduce	5534	2724	1429	818	533	417

Timings for problem 3.7

The timings for the Gaussian elimination method to solve a 120 $\times$ 120 size system of equations on an nCUBE are given below for both Dataparallel C and ACLAN (Array C LANguage). The timings for ACLAN were noted down from a graph.

Execution times on nCUBE (seconds)							
Language	Hypercube Dimension						
	0	1	2	3	4	5	6
ACLAN	39.81	21.87	12.02	7.41	4.67	3.46	2.81
DPC	16.37	9.09	5.92	4.32	3.85	3.82	4.10

Timings for problem 3.8

The timings for the 256 $\times$ 256 matrix multiplication on a Sequent Symmetry and iPSC/2 are given below for both Dataparallel C version and Chare Kernel version.

Execution times on Symmetry (seconds)						
Language	Sequential	Number of Processors				
		1	2	4	8	16
Chare Kernel	305	304.8	152.6	76.2	38.3	19.4
DPC	156.11	164.00	82.12	41.20	20.76	10.54

Execution times on iPSC/2 (seconds)							
Language	Sequential	Number of Processors					
		1	2	4	8	16	32
Chare Kernel	228	228	115	58.1	29.8	15.4	7.87
DPC	87.882	112.733	57.308	28.440	14.334	7.255	3.728

Timings for problem 3.9

The timings on an iPSC/2 for the sieve algorithm for both C-Linda and Data-parallel C versions are given below. The dashes indicate timing not available.

Execution times on iPSC/2 (seconds)							
Language	Size	Number of Processors					
		1	2	4	8	16	32
C-Linda	7 million	-	-	-	-	-	1.87
DPC	7 million	19.40	9.82	5.07	2.74	1.60	1.09
	3 million	8.14	4.16	2.19	1.23	0.78	0.59

The timings for the naive algorithm to find primes under 3 million, is given below for C-Linda, Chare Kernel and Dataparallel C versions for Sequent Symmetry and iPSC/2.

Execution times on Sequent Symmetry (seconds)					
Language	Number of Processors				
	1	2	4	8	16
Chare Kernel	344	174	87	43.7	21.9
DPC	295	175.90	91.86	47.12	23.77

Execution times on iPSC/2 (seconds)							
Language	Number of Processors						
	1	2	4	8	16	32	64
C-Linda	421	255.15	131.56	63.78	31.89	15.94	7.97
Chare Kernel	319.5	170.4	86	45.5	25	14	8.1
DPC	279.447	167.416	88.531	45.39	22.90	11.524	-