Solving Partial Differential Equations on a Network of Workstations

Chi-Chung Hui, Gary Ka-Keung Chan, Michelle Man-Sheung Yuen,
Mounir Hamdi, and Ishfaq Ahmad

Department of Computer Science, Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

The use of a network of workstations as a single unit for speeding up computationally intensive applications is becoming a cost-effective alternative to traditional parallel computers. In this paper, we present the implementation of an application-driven parallel platform for solving partial differential equations (PDEs) on this computing environment. The platform provides a general and efficient parallel solution for time-dependent PDEs and an easy-to-use interface that allows the inclusion of a wide range of parallel programming tools. We have used two different parallelization methods in this platform. The first method is a two-phase algorithm which uses the conventional technique of alternating computation and communication phases. The second method uses a novel pre-computation technique which allows overlapping of computation and communication. Both methods yield significant speedup. However, the pre-computation technique is shown to be more efficient and scalable.

1 Introduction

Parallel computing environments based on networks of workstations have recently become effective and economical platforms for high-performance computing. This is due to a number of reasons. First, by providing controlled access to a much larger and richer computational resource, network environments can increase application performance by significant amounts. Second, the incremental enhancement of a network based concurrent computing environment is usually straightforward because of the availability of high-bandwidth networks. Third, many existing and projected applications are composed of heterogeneous sub-algorithms. On typical networks with a variety of architectures and capabilities, such applications can benefit by executing appropriate subalgorithms on the best suited processing elements.

In this paper, we use this computing environment for the implementation of a parallel platform for the solution of time-dependent PDEs which are generally regarded by scientists as one of the most computationally intensive tasks. Generally, PDEs can be classified as elliptic, hyperbolic or parabolic [2], [11]. For example, the Poisson equation is elliptic, while the Wave equation is hyperbolic, and the Heat equation is parabolic [9], [16]. The different classes of PDEs are typically solved using different numerical methods. In this paper, we will concentrate on parabolic PDEs which are also known as time-dependent PDEs. A parallel solution of such equations is not a straight-forward task. This is because the traditional methods of parallelization cannot be used due to the inherent recursive nature of time-stepping where the solution at the current step depends on the solution of the previous step [17]. Sequential numerical methods for solving time-dependent PDEs have been explored extensively [8], [11]. Attempts have also been made towards parallel solutions on distributed-memory MIMD machines [1], [3], [4], [10], [14], [16]. However, these parallel solutions are specific to particular kinds of problems and are not general in nature.

The prime objective of our platform is not to be specific to one problem, rather it should be able to solve a wide variety of time-dependent PDEs for various applications. The platform is interactive, portable, and scalable. The parallel solution in this platform uses one of the numerical approaches known as *the finite-difference approximation* [12]. The platform has been developed with a number of objectives in mind which are outlined below:

- To provide a general solution that solves a system of PDEs using the finite difference method. In the current implementation, the system is capable of solving systems of PDEs with a complexity of up to 3 spatial dimensions and 1 temporal dimension which is capable of solving most practical problems. Our implementation has been developed using the Parallel Virtual Machine (PVM) parallel programming environment [6] and is running on a cluster of SUN workstations.
- To provide an easy-to-use interface that includes a number of parallel programming tools including data partitioning and load balancing.
- To provide an efficient and effective parallel computation and communication design such that the speedup gained is as high as possible without sacrificing the generality of the platform.
- To obtain a thorough understanding of the overheads of

0-8186-6395-2/94 $4.00 © 1994 IEEE

the system (both software and hardware).

The rest of the paper is organized as follows. Section 2 presents a general overview of our parallel platform. Section 3 details the parallel methods used for the solution of PDEs. In Section 4, we present our experimental results. Finally, Section 5 provides some concluding remarks.

2 System Overview

The platform uses the finite difference method which provides approximate solutions of PDEs, such that the derivatives at a point are approximated by difference quotients over a small interval. In solving the initial-boundary value problems, this method determines approximations at a finite number of points (*grid points*) in the domain. Each data point in the grid is given an initial value at the beginning of the execution. As time goes by, the value at each grid point is updated according to the PDEs provided. Based on the time step used in time differentiation and the order used in spatial differentiation, different dependencies among the grid points are resulted. Typically, the data at each grid point is updated at each time step by the surrounding points in the past time steps, as depicted in Figure 1. This regular relationship among the data points is where data parallelization can be captured.

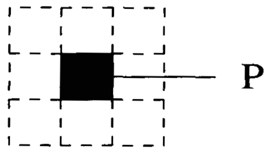


Figure 1. The central shaded point is the location of the point *P* that needs to be calculated.

One common area involving the use of parabolic PDEs is the theory of heat conduction [11]. The so called heat equation which is the fundamental component of the theory of heat conduction is one of the troublesome equations to deal with in scientific computing. In our experimental results, we use this equation to analyze our platform. In its generic form, the heat equation is defined as

$$\frac{\partial u}{\partial t} = \Delta u = \sum_{j=1}^{n} \frac{\partial^2 u}{\partial x_j^2} \quad .$$

One simple example of the heat equation is the temperature distribution $u(x,y,t)$ of a two-dimensional grid as a function of the coordinates x, y and the time t.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad .$$

Using an explicit forward difference approximation to the equation, we obtain the following solution

$$u_{i,j}^{k+1} = r_x u_{i-1,j}^k + r_x u_{i+1,j}^k + (1 - 2r_x - 2r_y)\, u_{i,j}^k + r_y u_{i,j-1}^k + r_y u_{i,j+1}^k$$

where $r_x = \Delta t / (\Delta x)^2$, $r_y = \Delta t / (\Delta y)^2$ and $u_{i,j}^k$ denotes the value of the temperature at grid point (i, j) and time $t_k = t_0 + k \times \Delta t$.

Our parallel processing environment consists of SUN IPX workstations connected by an Ethernet network. A single file system is being shared by all the workstations in the network. The PVM system has been used in the implementation of the platform. The parallel programming model used is SPMD (Single Program Multiple Data). The parallelization process consist of a number of steps. This includes data partitioning, processor allocation, load balancing, I/O scheduling, overlapped computation and communication routines, and message-passing procedures.

3 The Parallel Platform Design

One of the main objectives of the platform is to make it flexible enough to accommodate a wide variety of applications. The specifications of the PDEs are given through the user interface which are then fed into a parser. The parser parses the specification and builds the corresponding executable programs. The user specifications are divided into 3 categories, which are the *parameter section*, the *definition section*, and the *auxiliary section*. The interface is flexible enough to allow the user to define C-style parameters, variables and functions inside the appropriate sections. The parameter section includes the structure of a data point, initial values, dimension of the grid, data dependency, checkpointing options, computational mode, and boundary conditions. In the definition section, the user needs to define the function which is used by the platform to perform iterative updates of the grid points. The auxiliary section contains any user supplied sub-programs that are needed by the three functions defined in the definition section. Moreover, the platform carries out a number of tasks including processor allocation, data partitioning, load balancing, computation, communication and disk I/O. These are elaborated below.

3.1 Data I/O

As mentioned earlier, all workstations share a single file system. The platform takes advantage of this to drastically reduce the amount of communication. Since the programming model is SPMD model, there is a host process which sets the parallel processing environment. Based on the data partitioning scheme, the node processes are informed of the region of grid points assigned to them. Each node process can read the corresponding portion of the initial data file concurrently because the file system is shared. Similarly, the results of computations are also written back into the data files concurrently. Then, the results are combined to get the required solution.

3.2 Processor Allocation and Data Partitioning

The number of processors used by the platform is determined by two factors. The first is the *user-stated maximum* which must be less than or equal to the number of processors the PVM system is started with. The second factor is that of data dependency. Since the grid points are distributed across different processors, communication is needed to pass the grid points at the processors' boundaries so that computations can be performed for the boundary grid points. However, in order to reduce the communication cost, it is better if the computation in a processor does not depend on data from more than one processor from a side along a dimension. The above factors limit the maximum number of processors to be used. The minimum of these two maximum is used to determine the number of processors.

The data is distributed in a load balanced fashion into approximately equal-sized blocks. A simple cost function is used to decide the processors allocation scheme. The function considers the size of the cross-sections when the grid is split into regions. The size of the cross-sections is minimized as it is directly proportional to the amount of interprocessor communication. The method of computing the cost is straight-forward. Given the number of processors N and the dimensionality d, we find all possible factorizations of N into d integers, i.e. $n_1 \times n_2 \times ... \times n_d = N$. For each factorization, we can easily compute the size of the cross-section. The cross-section will be a point, a line and a plane for 1-D, 2-D and 3-D problems, respectively. As an example, consider the two allocation schemes for a 3-D grid shown in Figure 2. The resolution of the grid is 4 × 4 × 4 and the number of processors used is 4. Two possible partitioning schemes are shown in this figure. The communication cost is reflected by the cross-sectional areas in the above schemes. For first scheme, the total shaded area is 4 × 4 + 4 × 4 = 32, while for the second scheme , the total shaded area is 4 × 4 + 4 × 4 + 4 × 4 = 48. Hence, the cost function would allocate the processors in scheme 1's fashion. The cost function keeps track of the cost of different allocations (factorizations of N) and selects the one that gives the minimum.

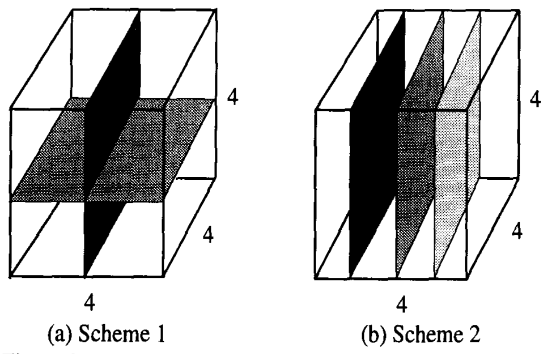


Figure 2. Two possible schemes for distributing the data across 4 processors and the corresponding communication costs.

3.3 Computation

A simple methodology for computing the point values is a two-phase algorithm. This algorithm uses alternating phases of computation and communication.

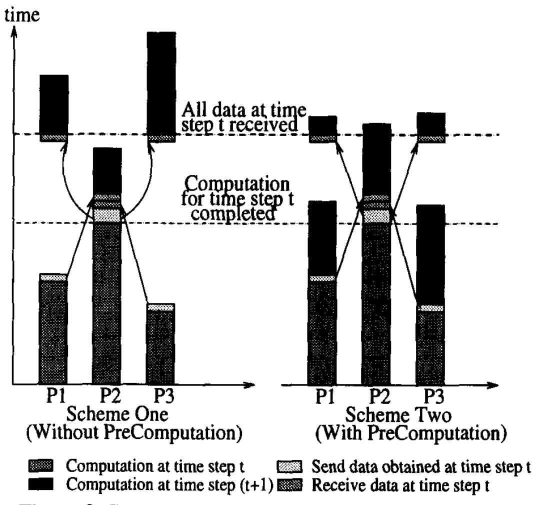
3.3.1 The Two-Phase Algorithm

In order to calculate the points at time t, the following steps apply:

Communication Phase: The nodes wait until all boundary data in time (t - *LEVEL*) to time (t - 1) arrive, where *LEVEL* correspond to the number of time steps used up to that point.

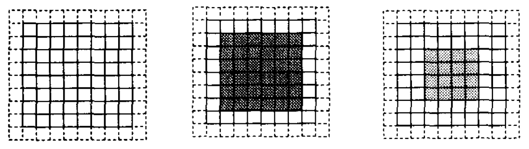
Computation Phase: The nodes compute the values at time t.

Since the time used in each computation phase and communication phase may be different and there is no overlap between them, a node must wait until it receives all of the required data from other nodes. Consequently, this algorithm is inefficient since part of the elapsed time is used in just waiting. This is illustrated in the left diagram of Figure 3. As can be noticed from this figure, processor 2 is slower than other processors. As a result, processors 1 and 3 must wait until all the boundary data from processor 2 have arrived. The communication phase consists of two parts: 1) sending and receiving messages, and 2) waiting for the messages to arrive.


time
All data at time step t received
Computation for time step t completed
P1 P2 P3
Scheme One (Without PreComputation)
P1 P2 P3
Scheme Two (With PreComputation)
Computation at time step t
Computation at time step (t+1)
Send data obtained at time step t
Receive data at time step t
Figure 3. Comparison of the elapsed times for the two-phase and pre-computations methods.

3.3.2 The Pre-Computation Algorithm

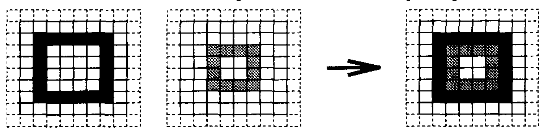
In order to alleviate the problems of the two-phase algorithm, we use the technique of *pre-computation.* For each node, if the boundary data from time (t - *LEVEL*) to time (t - 1) have not yet arrived, the node computes part of the points in time t which do not make use of the boundary data. This can be noticed from Figure 3 by contrasting the right diagram with the left diagram. In this case, processors 1 and 3, instead of waiting for the data from processor 2, utilize the idle time to pre-compute portions of data at time step (t+1). The idea of pre-computation can be further explained by considering Figure 4 which shows the data allocated to a node represented by solid lines. This is a 2-dimensional region with size 8 × 8. We call it the *local region.* The extent of points that a node needs to receive in this case is represented by the dotted boundary. Figure 4(a) shows that in order to calculate the point values at time t, and the node has not yet received all the boundary points it needs, it can still calculate the point values in the central 6 × 6 region (it is assumed that the calculation of each point depends on itself and its surrounding points only, see Figure 1). This region is shown as the shaded region in Figure 4(b). This technique is called *pre-computation* since a portion of points in time t are calculated before all the boundary points have arrived.



(a) The local region and extent of data required in computation.

(b) The shaded region indicates the points which can be computed with pre-computation.

(c) The lightly shaded region indicates the points at time (t-LEVEL) that are no longer required.



Extra points at time step t

Extra points at time step (t+1)

Final content of the extra region

(d) The concept of extra regions: points resulted in different level of pre-computations can be stored.

Figure 4. The data management schemes during the pre-computation method.

When the pre-computation is completed at time t and the boundary data have not yet arrived, the node can pre-compute the data at time (t + 1) using the available data from time (t - *LEVEL* + 1) to time t. This process is repeated until no data can be pre-computed anymore. In the implementation of the pre-computation algorithms, it is necessary to allocate additional memory to hold the pre-computed results. However, the size of additional memory after a number of steps can grow very large. Consequently the memory may eventually run out. To deal with this problem, we free a region of the memory that holds the data which is no longer required. This is shown in Figure 4(c) where the lightly shaded region shows the data that is not required at time step t and is freed. Note that the freed region (with size 4 × 4) is slightly smaller than the pre-computed region (with size 6 × 6). However, the extents of these extra points shrink as the level of pre-computation increases, and more importantly, they do not overlap. Thus, it is possible to allocate a region with size equal to the local region and place all the pre-computed points into that region. This region is called the *extra region.* See Figure 4(d) for an illustration.

In the following, it is shown that only 1 level of pre-computation is sufficient such that a maximal overall reduction in elapsed time can be achieved. For the two-phase algorithm, the total time used in time step n, $T_{2P}(n)$, is given by

$$\begin{aligned} T_{2P}(n) &= T_{comp}(n) + T_{send}(n) \\ &\quad + T_{wait}(n) + T_{recv}(n) \end{aligned}$$

where T_{comp} is the computation time, T_{send} is the time used in sending messages, T_{wait} is the time used by the platform in waiting for the incoming messages, and T_{recv} is the time used in processing the incoming messages. Clearly, there is a period of idle time during the $T_{wait}(n)$ period, and it is possible to perform pre-computation so as to overlap $T_{wait}(n)$ and $T_{comp}(n+1)$. It is impossible to overlap $T_{wait}(n)$ and $T_{recv}(n)$ since the platform cannot process the incoming message before they arrive. Similarly, it is impossible to overlap $T_{wait}(n)$ and $T_{send}(n+1)$ since the data required to be sent in time (n+1) is not ready. This is shown in Figure 5 where two cases are considered:

Case 1. $T_{wait}(n) \le T_{precompute}(n+1)$: After pre-computation at time step (n+1) is completed, the message at time step n arrives. Thus, no waiting time is needed for incoming messages, and the elapsed time is:

$$\begin{aligned} T_{best}(n) &= T_{postcompute}(n) + T_{send}(n) \\ &\quad + T_{precompute}(n+1) + T_{recv}(n) \\ &= T_{comp}(n) + T_{send}(n) + T_{recv}(n) \end{aligned}$$

where $T_{postcompute}(n)$ is the computation time of the points in time step n that cannot be computed during the pre-computation phase. Note that the term $T_{compute}(n)$ does not appear in the equation because it is decomposed into two terms, $T_{precompute}(n)$ and $T_{postcompute}(n)$. The above time is the shortest elapsed time achievable since all the three components involve computations and thus cannot be overlapped.

Case 2. $T_{wait}(n) > T_{precompute}(n+1)$: In this case, the computation time is less than the waiting time. The shortest time achievable in each time step is:

$$T_{best} = T_{send}(n) + T_{wait}(n)$$
$$+ T_{recv}(n) + T_{postcompute}(n)$$

Here, $T_{precompute}(n)$ is included into $T_{wait}(n-1)$ while $T_{postcompute}(n)$ remains distinct. Although it is possible to perform more levels of pre-computation during the $T_{wait}(n)$ period, this waiting period cannot be reduced further since the incoming messages arrive only after this period (i.e., $T_{recv}(n)$ cannot be started before $T_{wait}(n)$ ends).

Following the above argument, it is easy to see that 1 level of pre-computation is sufficient to obtain the shortest elapsed time since performing more than 1 level of pre-computation can only shift some computations earlier as is illustrated by the right diagram of Figure 5.

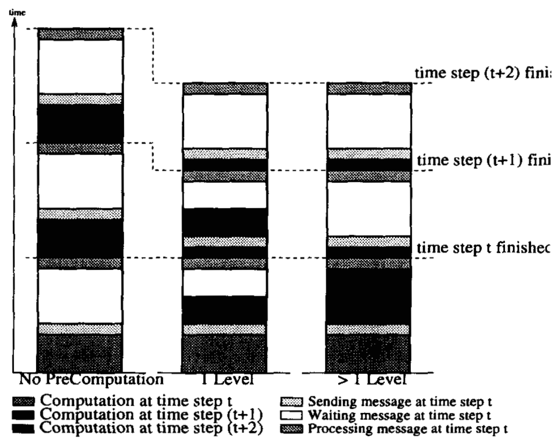


Figure 5. The effect of number of levels used in pre-computation on the elapsed time.

3.4 Load Balancing

Our platform can provide a load balancing scheme which can be used to perform even or uneven distribution. Even distributions, such as those described in Section 3.2, are required if all processors are identical and are equally loaded. When the workloads on processors are different, load balancing is performed to migrate some of the workload from heavily loaded processors to lightly loaded ones. The load balancing scheme is designed in such a manner that it can also be used in a heterogenous environment where the speeds of processors are not identical. In order to minimize the complexity of the buffer management scheme and interprocessor communication scheme, the local regions kept by the nodes are always rectangular in shape. This way, our load balancing scheme considers a grid to be a collection of planes and columns in a 3-dimensional Cartesian space. Partitioning along the x-axis partitions the grid into planes. Partitioning along the y-axis further partitions these planes into blocks. Figure 6 illustrates one simple example where the grid consists of two planes and each plane is divided among two processors. With load

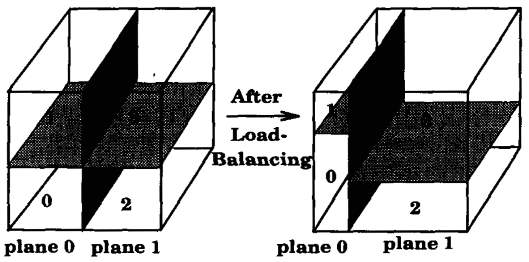


Figure 6. The load balancing scheme partitions the data across two dimensions.

balancing, the division of data can be unequal. If the grid is very large and there are large number of processors available, load balancing allows further partitioning of data along the z-axis. This is shown in Figure 7 where the grid is divided along all three dimensions. Here the planes can be viewed as collections of columns.

3.5 Interprocessor Communication

To accommodate the load balancing schemes described above, the platform is designed to have communication in interplane, intercolumn and interblock directions for problems up to 3 dimensions. Interplane communication deals the communication between each processor and the processors of the plane in front and the plane behind. Whereas, intercolumn communication is for communication between processors in the same plane but neighboring columns of processors. For interblock communication, it deals with communication of each processor and its neighboring processors in the same column. For example, as illustrated in Figure 7, for the interplane communication, processor 11 at location of plane 1, column 0 and block 1 needs to communicate with neighboring processors in plane 0 and plane 2. The communication process is composed of the sending and receiving steps.

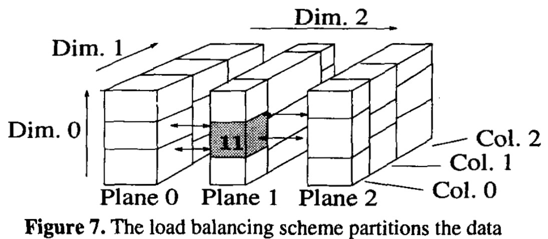
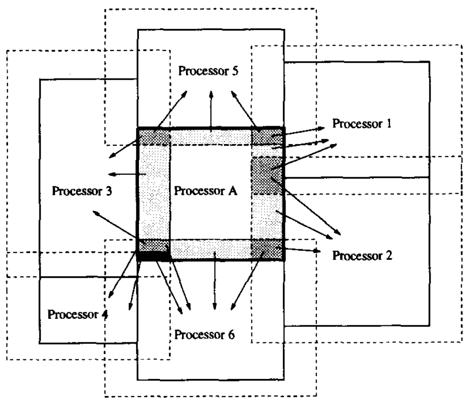


Figure 7. The load balancing scheme partitions the data across three dimensions.

3.5.1 Send Procedure

In the sending step, the location of the processor is determined in terms of its plane, column and block number. The sending routine determines the grid point values needed by the neighboring processors in each plane, column and block, packs the data into messages and sends them to each processor. Note that the same data may be required to be sent to more than one processor due to overlapping regions. This can be seen from Figure 8, where processor A needs to send different chunks of data to each of the neighboring processors. The sending process is done in the 3 communication steps mentioned above to ensure that the grid point values on each face of the block are sent systematically.



□ Data to be sent from Processor A to 1 neighbouring processor
▨ Data to be sent from Processor A to 2 neighbouring processors
■ Data to be sent from Processor A to 3 neighbouring processors

Figure 8. The flow of data from a processor to its surrounding processors.

3.5.2 Receive Procedure

In the receiving step, a counter is initialized as the total number of grid point values needed to perform calculation of all grid points. The counter then keeps track of the number of grid points that have not yet arrived. When the counter becomes zero, data are sufficient for completing computation for the next time step. During the communication phase, computation and communication are executed alternatively depending on when the data arrives. First, the sending procedure for all the processors is called. Then, the computation process starts for the points where sufficient data exist for completing the calculation. During the computation, non-blocking receive is called from time to time to receive data from neighboring processors. Computation continues until no more points at the next time step can be calculated. At this time, blocking receive is used to wait for arrival of all the grid points. The process continues until all the data needed has arrived. Then the computation for the grid points at the boundary of the processor can be started for the next time step.

4 Experimental Results

In this section, we present the experimental results for solving 2-D and 3-D PDEs on our platform using the heat equation as an example. We measured all the factors that contributed to the total elapsed time for the two-phase method and the pre-computation method. Table I and Table II show the timing results for the 2-D heat equation using the two-phase method and the pre-computation method respectively. We used a grid size of 3800 x 100 points and 500 time steps using different number of SUN IPX workstations. The setup time is the time spend by the host in setting up the nodes, sending and receiving information to and from the nodes. Thus, it increases as the number of nodes increases. However, it is small compared to the total elapsed time and is similar for both methods. The I/O time is the time that the node program spends in reading the initial data and writing the final results. This time is the same for both methods, and it decreases as the number of nodes increases due to the overlapping of disk I/O processing among the processors.

Table I. Timing results (in seconds) of the 2-D heat quation using the two-phase method for 3800 x 100 grid and 00 time steps.

No. of Nodes	Setup Time	I/O Time	Comp. Time	Comm. Time	Elapsed Time	Speedup
1	0.78	16.26	555.26	0.18	572.48	1.00
5	1.21	10.90	112.91	11.80	136.81	4.18
10	3.54	8.19	57.57	12.55	81.86	6.99
15	3.96	7.00	38.47	14.13	64.06	8.94
20	4.78	7.08	29.83	14.84	56.52	10.13

Table II. Timing results (in seconds) for the 2-D heat equation using the pre-computation method for a 3800 x 100 grid and 500 time steps.

No. of Nodes	Setup Time	I/O Time	Comp. Time	Comm. Time	Elapsed Time	Speedup
1	0.55	14.86	555.26	0.28	570.95	1.00
5	1.62	9.49	173.84	6.21	191.16	4.24
10	3.36	7.78	60.77	6.63	78.54	7.27
15	4.14	6.74	42.06	8.04	60.99	9.36
20	4.79	6.84	32.39	8.12	52.14	10.95

The major difference between the two-phase method and the pre-computation method is in the computation time and the communication time. The computation time is the time spent in actual computation including the time spent in setting up buffers and actual calculations. Typically, this time is smaller for the two-phase method because the pre-computation method incurs higher overhead because of additional needed buffer management due to the overlapping of computation and communication. Figure 9 illustrates the computation time of the two-phase method and the pre-computation method. The communication time is

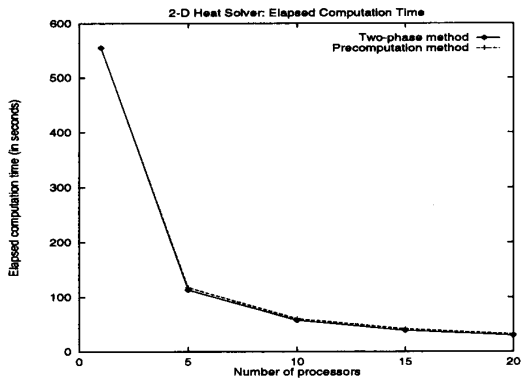


Figure 9. The computation times in seconds for the two-phase method and the pre-computation method.

the time that a node spends in communication including processing the messages and waiting for the messages. The communication time of the pre-computation method is smaller than that of the two-phase method since it reduces the waiting time to a minimum by overlapping computation and communication. This is illustrated in Figure 10.

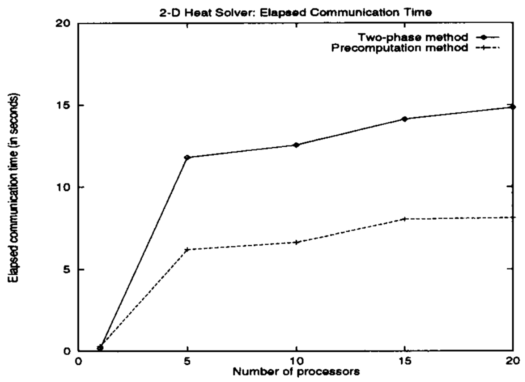


Figure 10. The communication times in seconds for the two-phase method and the pre-computation method.

Thus, as long as the time saved in communication time by using the pre-computation method is bigger than the additional time for buffer management when compared to the two-phase method, using the pre-computation method would result in a smaller total elapsed time and a better speedup. Figure 11 shows the speedup curves of the two-phase method and the pre-computation method, where the precomputation method compares favorably on this measure especially when using a large number of nodes. However, both methods record significant speedups. Hence,

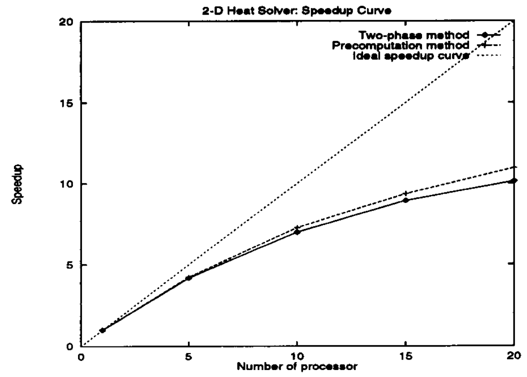


Figure 11. The speedup curves of the two-phase method and the pre-computation method.

whenever the communication time is crucial in determining the overall elapsed time of the application, using the pre-computation method in our platform would be preferred over the two-phase method. This would usually be the case using a network of workstations where the network (e.g. Ethernet) has a low communication bandwidth.

Table III and Table IV show the timing results for solving the 3-D heat equation using a grid of 1000 × 20 × 20 points and 500 time steps for both the two-phase method and the pre-computation method. The setup times and

Table III. Timing results (in seconds) for the 3-D heat equation using the two-phase method for a 1000 × 20 × 20 grid and 500 time steps.

No. of Nodes	Setup Time	I/O Time	Comp. Time	Comm. Time	Elapsed Time	Speedup
1	0.34	15.48	935.58	0.20	951.61	1.00
4	0.77	10.86	240.76	16.91	269.32	3.53
8	3.04	10.20	123.00	20.10	156.35	6.09
12	3.50	9.21	79.23	20.13	112.06	8.49
16	3.46	7.72	63.24	21.20	95.62	9.95
20	2.85	6.89	49.87	24.84	84.46	11.27

the I/O times are the same for both methods as expected. However, the difference in computation times and communication times between the two methods is even greater than that of the 2-D heat equation. However, the same conclusions for solving 2-D PDEs hold when solving 3-D PDEs using both methods. That is, if the gain in communication time by the pre-computation method is bigger than the gain in computation time by the two-phase method, then the pre-computation method would be the preferred method in our platform.

Table IV. Timing results (in seconds) for the 3-D heat equation using the pre-computation method for a 1000 × 20 × 20 grid and 500 time steps.

No. of Nodes	Setup Time	I/O Time	Comp. Time	Comm. Time	Elapsed Time	Speedup
1	1.44	17.90	935.19	0.29	954.82	1.00
4	1.05	10.98	262.42	6.77	281.22	3.40
8	1.93	8.76	136.78	7.61	155.07	6.16
12	4.35	8.54	92.94	8.44	114.27	8.36
16	3.08	7.46	74.22	8.64	93.41	10.22
20	2.74	6.22	60.95	9.61	79.52	12.01

5 Conclusions

A parallel platform for solving time-dependent PDEs is designed and implemented on a network of workstations. The platform is general in such a way that different applications can be easily developed through a very effective user interface. Significant speedups were achieved using two parallel methods, a two-phase method and a pre-computation method. The pre-computation method is an efficient technique that allows the overlapping of computation and communication. It has been shown that it can outperform the traditional two-phase method especially when the communication time is significant and/or the number of nodes used is large. On the other hand, the two-phase scheme is simple and easier to implement. We have presented the detailed timing of all the components that contribute to the total elapsed time. This is important as it helps the programmer identifying the major bottlenecks in the system, and gives him/her guidance on where an improvement is needed most.

References

[1] M. Berger, J. Oliger and G. Rodrigue, "Predictor-Corrector Methods for the Solution of Parabolic Problems on Parallel Processors," in *Elliptical Problem Solvers*, Ed. M. Schultz, Academic Press, New York, 1981, pp. 197-202.

[2] K. Black, "Polynomial Collocation Using a Domain Decomposition Solution to Parabolic PDEs via the Penalty Method and Explicit/Implicit Time Marching," *Journal of Scientific Computing*, vol. 7, no. 4, 1992, pp. 313-338.

[3] Z. Cvetanovic, E.G. Freeman and C. Nofsinger, "Efficient Decomposition and Performance of Parallel PDE, FFT, Monte Carlo Simulations, SImplex and Sparce Solvers," *Proceedings of Supercomputing '90*, Nov. 1990, pp. 465-474.

[4] A. Fijany, "Time Parallel Algorithms for Solution of Linear Parabolic PDEs," in Proc. of *Int'l Conf. on Parallel Processing*, 1993, vol. III, pp. 51-55.

[5] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker. *Solving Problems on Concurrent Processors, Vol. I: General Techniques and Regular Problems*. Prentice Hall, 1988.

[6] G. A. Geist and V. M. Sunderam, "Network-based Concurrent Computing On The PVM System," *Concurrency: Practice And Experience*, 1992, pp. 293-311.

[7] M. A. H. MacCallum, "An Ordinary Differential Equation Solver for REDUCE," *International Symposium ISSAC'88*, pp. 115-123.

[8] S. McFaddin and J.R. Rice, "RELAX: a Software Platform for Partial Differential Equation Interface Relaxation Methods," *Proceedings of the 2nd IMACS International Conference on Expert Systems for Numerical Computing* (1992), pp. 175-194.

[9] M. A. Pinsky, *Introduction to Partial Differential Equations with Applications*, McGraw-Hill Publishing co., 1984.

[10] A. Schuller, "Parallelizing Particle Simulations based on the Boltzmann Equation," *Parallel Computing* vol. 18, no.3, (March 92), pp. 269-279.

[11] G. D. Smith, *Numerical Solution of Partial Differential Equations: Finite Difference Methods, 3rd Edition*, Oxford Applied Mathematics And Computing Science Series.

[12] J. Noye: *Finite Difference Methods for Partial Differential Equations, Numerical Solutions of Partial Differential Equations*. North-Holland Pub. Co., pp. 3-137.

[13] H. A. Riphagen, "Numerical Weather Prediction, Numerical Solution of Partial Differential Equations: Theory, Tools and Case Studies," pp. 246-274.

[14] J. Saltz and V. Nail, "Towards Developing Robust Algorithms for Solving Partial Differential Equations on MIMD Machines," *Parallel Computing*, vol. 6, 1988.

[15] E. Verhulst, "A Prototype of a User Friendly Partial Differential Equation Solver on a Transputer Network," *Proceedings of the User 1 Working Conference*, 1998, pp. 232-239.

[16] G. R. Wightwick and L. M. Leslie, "Parallel Implementation of a Numerical Weather Prediction Model on a RISC System/6000 Cluster," *Fifth Australian Supercomputing Conference*, Oct 12, 1992, pp. 135-142.

[17] D. Womble, "A Time-Stepping Algorithm for Parallel Computers," SIAM Journal of Sci. & Stat. Computing, vol. 11(5), pp. 824-837, 1990.

[18] E. Zirman, M. Rao and Z. Segall, "Performance Efficient Mapping of Applications to Parallel and Distributed Architectures," 1990 *International Conference on Parallel Processing*, 1990, pp. 147-154.