Performance Modelling for High Order Finite Difference Methods on the Connection Machine CM-2

Yu-Chung Chang

Tony F. Chan

March 1993

(Revised December 1993)

CAM Report 93-04

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024-1555

PERFORMANCE MODELING FOR HIGH-ORDER FINITE DIFFERENCE METHODS ON THE CONNECTION MACHINE CM-2

Yu-Chung Chang

DEPARTMENT OF APPLIED MATHEMATICS
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA 91125

Tony F. Chan

DEPARTMENT OF MATHEMATICS
UNIVERSITY OF CALIFORNIA,
LOS ANGELES
LOS ANGELES, CALIFORNIA 90024

Summary

This paper is concerned with modeling the performance of high-order finite-difference schemes for hyperbolic problems on the Connection Machine CM-2. Specifically, we would like to determine whether the higher communication cost of higher-order methods makes them less favorable in a parallel setting than in a sequential setting. Since most difference methods are implemented using the *cshift* operator, we first derive a timing model for it in CM-Fortran under the new slicewise compiler model. This model is then used to predict the performance of the difference methods with different orders applied to the 2D Bürgers' equations. In addition, we study the effect of varying different machine performance parameters, such as the communication time and floating-point operation time, as well as problem parameters such as mesh size. Our analysis and numerical results indicate that among high-order finite difference methods, the fourth-order one is the most efficient method in that it achieves a moderate error tolerance (a few percent) with least running time.

The International Journal of Supercomputer Applications, Volume 9, No. 1, Spring 1995, pp. 40–57
© 1995 Massachusetts Institute of Technology.

Introduction

There are many physical problems whose solution requires large-scale computations. The computation of turbulent flows is one example. Typically, these computations demand an extremely high numerical resolution to resolve the physically interesting small-scale structures. Naturally one would like to perform such calculations on a computer that can provide both extremely high speed arithmetic operations and large memory storage. A massively parallel supercomputer exhibits both of these features. By dividing the computational work into many individual processors, a parallel computer can, in effect, achieve a very high speed-up over conventional sequential machines. The result is a very powerful addition to our arsenal of computational environments, one that makes many difficult computational problems more tractable.

In practice, the actual performance of an application running on a massively parallel computer can depend in crucial ways on the communication overhead and on the data layout (see Leiss and Lee, 1991), and in some cases, the communication time may even exceed the time it takes to perform the arithmetic operations. For a given computational problem, the algorithm should be designed in such a way as to minimize the communication time. Very often this time is problem dependent. From a general user's point of view, it would be very desirable to have a timing model that can predict the overall performance by taking into account both the volume of arithmetic operations and the interprocessor communication time. The design of such a timing model is the subject of this paper.

Finite-difference methods are perhaps the most commonly used numerical methods in computational fluid dynamics. A higher-order method achieves a given error tolerance with fewer grid points than a lower-order method, but requires more computations per grid point. The interesting question is to decide what is the optimal order in the sense of achieving a given error tolerance with maximal computational efficiency. In Chang (1992) and Chang and Kreiss (1992), the performance of different finite-difference methods was compared with the pseudo-spectral method for computations with strong shear layers. It was observed

that on a sequential computer (SUN SPARC) the fourth- and sixth-order finite-difference methods are more efficient in achieving a given error tolerance, say 1%, than the pseudo-spectral method. Empirically, a similar comparison result was observed on the Connection Machine CM-2. In this paper we will undertake a more systematic and more quantitative study of the performance of finite-difference methods with different orders for these types of calculations on CM-2. For this purpose we develop a timing model for a basic communication kernel using the nearest-neighbor NEWS communication network, *cshift*, which is used in most finite-difference codes. We develop this using the current slicewise model of CM-2. In deriving our timing model we propose a method of estimating both the purely internal communication time and the purely external communication time. One new feature of our model is that it accounts explicitly for the startup time in the internal communication which depends on the geometry of the data. Our numerical experiments indicate that the relative error of the *cshift* timing model is less than 15% in the three-dimensional geometry on a 16K CM-2 and less than 20% in two- and three-dimensional geometries on an 8K CM-2.

"Finite-difference methods are perhaps the most commonly used numerical methods in computational fluid dynamics. A higher-order method achieves a given error tolerance with fewer grid points than a lower-order method, but requires more computations per grid point."

Although there do exist timing models for the CM-2, these were designed for the older compilation model, namely the fieldwise model (e.g., Levit, 1988; Pozo, 1991). Recently, Thinking Machines Corporation developed a new slicewise model based on the slicewise storage strategy as a way of speeding up the floating-point operations. Because of differences in the data mapping strategies, the timing model developed for the fieldwise model cannot be applied directly to the slicewise model. A new timing model for the slicewise model is, therefore, needed. To the best of our knowledge, no such timing models for the slicewise model now exist.

We use our timing model to perform a number of studies. First, we use it to predict the performance of high-order finite-difference methods for an interesting model problem that arises in the study of high Mach number flow, namely, the 2D Bürgers' equations. The results validate the observation in Chang (1992) and Chang and Kreiss (1992) that, for the case of solutions that develop small-scale structures, the fourth-order method is the most efficient in terms of reasonable error tolerance, say 1%,[1] with least computational effort. Next, we use the timing model to explore the effect of varying different machine and problem parameters in our model, e.g., communication startup time, internal and external communication time, floating-point operation time, and the mesh sizes of the problem. Our study seems to suggest that the fourth-order method remains the best one assuming reasonable variations of these parameters.

In the next section we give a brief introduction to the architecture of CM-2. Then we develop the *cshift* timing model. In the following section we develop a simple timing model for arithmetic operations on CM-2 after which we apply the timing model to the finite-difference code presented in Chang (1992) and Chang and Kreiss (1992) and perform a parameter study. The final section of the paper summarizes our main conclusions.

Brief Introduction to CM-2 and the Slicewise Model

The Connection Machine Model CM-2 is a massively parallel computing system, of the single-instruction, multiple-data (SIMD) type, with a distributed memory hypercube structure. The CM system consists of a parallel processing unit containing thousands of data processors each with its own memory and all acting under the direction of a serial processor, called the front end. There is also an I/O system that supports mass storage and graphic display devices and other types of peripherals. CM-2 supports this with three kinds of interprocessor communication: Nearest-neighbor (NEWS) communication, General-purpose (router) communication, and Global communication (scan) (see CM Fortran Reference Manual, (Version 1.0) 1991, and CM Fortran Optimization Notes: Slicewise Model, (Version 1.0) 1991).

There are two execution models that a programmer can choose from when compiling a CM Fortran program, the Paris (fieldwise) model or the slicewise model. These differ in the way the compiler maps CM arrays onto the underlying hardware.

Our timing model is established based on the new slicewise model. This model performs much faster than the standard fieldwise model. In the fieldwise model, storage of a 32-bit word is allocated in 32 consecutive bits of a physical bit-serial processor's memory. In each clock cycle a bit-serial processor can send out only one bit of a single-precision (32-bit) floating-point value. However, the Weitek 3132 floating-point unit (FPU) can operate on a 32-bit word in each clock cycle, which means that it takes 32 cycles for a bit-serial processor to send out the whole value of a 32-bit floating-point number to FPU. To balance the bandwidth between these two processors (32 to 1), each pair of CM chips (total 32 bit-processors) is designed to attach to one FPU. Thus these 32 bit-processors send out 32 single-precision words in 32 clock cycles, which, on average, means one single-precision word per clock cycle.

However, in this manner of storage the 32 bits sent out by the associated 32 bit-serial processors in each clock cycle do not represent any single precision number. Each bit just comes from a different bit-serial processor. Therefore, an interface is required to transpose these 32 bits into a proper format for FPU. This interface between the common processor memory and FPU is called the "Sprint Chip."

In practice, it is not very efficient to store the data fieldwise and then have to do transposition back and forth between the common memory area and the Weitek FPU. Recently, CM architects have designed a new slicewise model to view processors arranged in a slicewise configuration. A 32-bit word is stored in a 32-bit slice across the memory of those associated 32 physical bit-processors in the processing node (a processing node consists of 32 bit-serial processors sharing a single FPU). In other words, each bit of a 32-bit number is stored in a different processor associated with the par-

1. We are interested in the large time behavior in turbulent flow calculations. The solutions here have many different scales, and, in general, it is not possible to resolve all the details. Therefore, we will be interested only in the qualitative behavior of the solutions. In such situations an error tolerance of a few percent is good enough from a practical point of view. Throughout this paper, our comparisons of the performance of calculations assume an error tolerance of 1–2%.

ticular processing node, one bit per processor. Therefore, in the slicewise model FPUs actually access data from their associated processors in each clock cycle. That is, in each clock cycle a 32-bit slice across processors is read into FPU. Thus the slicewise model makes it unnecessary to implement transpositions between the main memory and the Weitek FPU. This distinctive feature of the floating-point architecture represents a significant improvement on the floating-point computational power of the CM.

Although it may appear that the only difference between the fieldwise model and the slicewise model lies in the performance of floating-point operations, it is not clear whether fieldwise timing models can be used for computations under the slicewise model. In the fieldwise model, a special Paris function call can be employed to specify the number of virtual processors simulated by a bit-serial processor in each dimension. The specific description of each bit-serial processor may then be referred to in order to estimate the communication time. Appropriate parameters can be obtained, for example, by a least-square fit. On the other hand, in a slicewise model each floating-point number is stored slicewise across the memory of the associated 32 bit-processors at a single floating-point node. There are no data completely stored in any single bit-serial processor, and it is irrelevant to speak of the subgrid sizes of each bit-serial processor. Clearly, it is more appropriate to view a *floating-point node* as the basic unit in the slicewise model of CM-2. Thus a 64K CM-2 should be viewed as one with 2,048 floating-point nodes and an 8K CM-2 as one with 256 floating-point nodes. The VP ratio[2] should be defined in terms of the number of virtual processors in this basic element. Therefore, the fieldwise models provided by Levit (1988) and Pozo (1991) cannot be applied directly in the slicewise model. New

"Although it may appear that the only difference between the fieldwise model and the slicewise model lies in the performance of floating-point operations, it is not clear whether fieldwise timing models can be used for computations under the slicewise model."

2. The ratio of the number of virtual processors required by the application to the number of physical PEs (processing elements) is called the VP ratio. PE is the basic element. For the fieldwise model, a PE is a bit-serial processor. For the slicewise model, it is a processing node consisting of 32 bit-serial processors with a floating-point chip. The VP ratio indicates how many times each PE must perform a certain task in order to simulate the appropriate number of virtual processors.

"To account for the dependence of the data on the geometry for the internal communication time, we propose a more sophisticated model to correct for the discrepancy in the previous model. The idea is to separate the study of the internal communication from the external communication."

timing models for both the communication operations and the arithmetic operations are required in order to study the performance of the slicewise model. These will be derived in the next two sections.

Performance Model for Grid Communication

Now we would like to study the communication part of the timing model. For a typical finite-difference code running on CM-2, the main communication can be done via the *cshift* (circulation shift) function. This is one of the nearest-neighbor communication functions that is naturally used for periodic boundary-value problems; it can be also used for nonperiodic boundary-value problems. There also exists an *eoshift* (end-off shift) function that can be used for boundary-value problems. Since *eoshift* is not as fast as *cshift* and since many boundary-value solvers can be programmed so as to make use of *cshift*, we will concentrate on the communication timing model for *cshift*.

To derive the timing mode for communication, we first need to know the *data layout* and the size of the chunk of data in each processing element. In addition, we need to know how much of the data movement in a *cshift* operation is internal (within the same node) or external (between two different nodes).

NEWS GRID COMMUNICATIONS AND DATA LAYOUT

Unlike the fieldwise model, in the new slicewise model there are fewer layout directives for specifying a data structure. In the fieldwise model, the way in which the dimensions of a data structure are to be mapped to the CM processors may be specified by calling the Paris subroutine CM_creat_detailed_geometry(). However, this cannot be done in the slicewise model. The CM Fortran compiler uses a canonical layout, called the NEWS layout, to allocate arrays in order to achieve nearly optimized performance. We need only use CMF_DESCRIBE_ARRAY() to find out how the data have been distributed to those floating-point nodes (for examples, see Chang, 1992).

The NEWS grid communication can be decomposed into two major parts:

- On-Node communication: Communication between two virtual nodes on the same physical node. Since there is a common memory area for a node, it only costs memory moving (copying) within the same node. We denote the time required for transferring a 32-bit word within the same node by t_M.
- Off-Node communication: Communication between two different physical processing nodes. This requires moving (or memory copying) data to a temporary buffer and transferring data through the off-node hypercube network. We denote the time required for transferring a 32-bit word off-node as t_{OFF}.

Since we now view an ensemble of 32 bit-serial processors sharing a single FPU as the basic unit of the slicewise model of CM-2, the communication mechanism becomes much simpler than in the fieldwise model. Suppose we would like to compute the finite-difference operator, ∂x or ∂y, on a 2D 512×512 data grid on an 8K CM-2. In the slicewise model, an 8K CM-2 becomes 256 floating-point nodes. By calling CMF_DESCRIBE_ARRAY(), we find that the data layout is a two-dimensional 16×16 hypercube that preserves the nearest-neighbor property. Each physical processing node gets a chunk of data with subgrid size 32×32 and simulates 32×32 virtual processing nodes. A simple first-order, forward-difference approximation for u_x with periodic boundary, at grid (i,j), is given by

```
do j = 1,n
   do i = 1,n-1
      ux (i,j) = (u(i+1,j) - u(i,j))/h
   end do
   ux (n,j) = ( u(1,j) - u(n,j) )/h
end do
```

where h is the mesh size. In the CM-Fortran command, this is translated into the following operation using the *cshift* function:

```
(cshift (u,dim = 1,shift = 1) - u)/h
```

This means that each grid point must send data to its nearest-neighbor node[3] with distance one along the first dimension.

One way of estimating the communication time of this difference operation is as follows. Suppose we need to perform the NEWS communication with distance one along the first axis, assuming that every (virtual) node wishes to communicate one floating-point value (32-bit) to its left neighbor. Those virtual nodes located in the left boundary have to be sent out sequentially through the off-node (external) hypercube network to its left neighbor. Therefore, there is a total 32×1 of external communications. The other virtual nodes are sent to their destinations within the same physical node, and this only involves memory moving (or memory copying) within a node, which is very fast. Finally, there are 32×31 internal memory movings (memory copying). By a straightforward calculation, we would expect that the overall communication time is given by $32 \times t_{OFF} + 32 \times 31 \times t_M$, as in the fieldwise models used by Levit (1988) and Pozo (1991). However, the drawback of such a communication model is that it ignores the startup time in internal memory moving. As will become apparent, the startup time in the internal communication can grow proportionally to the product of the second and third subgrid dimension of the data set in the layout (:serial, :news, :news).

MODEL FOR INTERNAL COMMUNICATION

To account for the dependence of the data on the geometry for the internal communication time, we propose a more sophisticated model to correct for the discrepancy in the previous model. The idea is to separate the study of the internal communication from the external communication. For this purpose, we use the layout (:serial, :news, :news) or the layout (1000:news,

3. Because the data are mapped into the hypercube in the NEWS ordering so as to map all neighboring grids into the neighboring nodes of the hypercube.

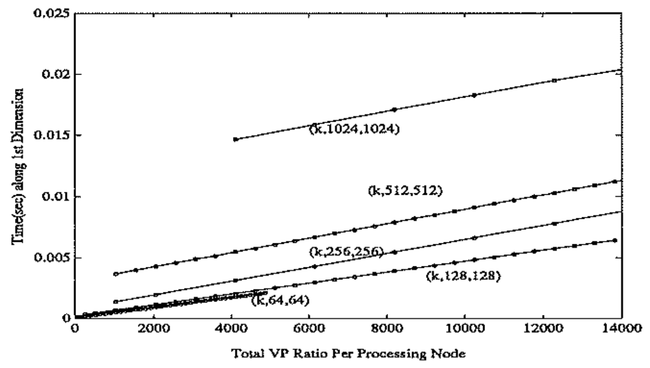

Fig. 1 NEWS Communication on CM-2 for 3D (:serial, :news, :news) data.

:news, :news)[4] with different data sizes as test cases. We perform the *cshift* operation of distance d along the first dimension. Because of the special layout this corresponds to purely internal communication. For simplicity, we consider the case $d = 1$, where d is the distance; other cases can be treated similarly. For measuring the time, we use the function call **cm_timer_read_cm_busy.**

One important observation in our study is that the startup time in the internal communication depends on the geometry in a subtle way. In Figure 1 each line corresponds to a set of data with similar geometry. For those data of size $k \times 128 \times 128$, with layout (:serial, :news, :news) and k varying from 4 to a few hundred, the performance of *cshift* of distance 1 along the first dimension is linear with respect to the total subgrid size of data. We observe the same behavior for data of size $k \times 256 \times 256$, $k \times 512 \times 512$, and $k \times 1024 \times 1024$, respectively. The corresponding startup times are also different in each case.

A careful study shows that *the startup time of internal communication of each data set depends on its geometry.* Consider a data set with layout (:serial, :news, :news) and subgrid size $v_1 \times v_2 \times v_3$. A serial array dimension is always allocated entirely within (never across) processing nodes. Therefore, the *cshift* along the first dimension takes place within each processing node.

Roughly speaking, our internal communication model can be understood as one in which memory moves as a "do loop" process. To make a *cshift* on such a data set along the first dimension, we first perform a memory copying (or memory moving) operation along that dimension. In this operation there is a basic startup cost, C_1, and a memory copying rate, t_M. But we have to repeat the same communication procedure v_2 times along the second dimension for each memory moving along the first dimension. Thus the accumulated time is $T = (C_2 + (C_1 + t_M \times v_1) \times v_2)$, where C_2 is the startup time along the second dimension. Finally, we have to

4. Dimension one, with weight 1000, is to be favored for interprocessor communication over dimension two or dimension three, which gets default weight 1. With such a high weight on the first dimension and no specification of weight on the rest, this layout gives the same subgrid size as does the layout (:serial, :news, :news).

repeat the same memory moving procedure for the first two dimensions a total of v_3 times along the third dimension. Therefore, the overall internal communication time is given by $T = C_0 + (C_2 + (C_1 + t_M \times v_1) \times v_2) \times v_3$, where C_0 is the startup time along the third dimension. Clearly, the startup time for the internal communication is now given by $C_0 + C_1 \times v_2 \times v_3 + C_2 \times v_3$. However, the C_0 term, which does not depend on the subgrid size, is negligible. Therefore, we can simplify the model for the overall internal communication time to

$$T = (C_2 + (C_1 + t_M \times v_1) \times v_2) \times v_3.$$

From a least-square fit of the data in Figure 1, we get $t_M = 0.52 \times 10^{-6}$ sec/per 32-bit word, $C_1 = 1.52 \times 10^{-6}$ sec, and $C_2 = 5.8 \times 10^{-6}$ sec.

MODEL FOR EXTERNAL COMMUNICATION

To develop the model for the external communication, we construct several purely external communication cases. Consider a data set with the layout (:serial, :news, :news) or (1000:news, :news, :news), and move the data along the second or the third dimension with distance equal to the subgrid size of that dimension. This causes the entire chunk of data in a node to move to its nearest node. For example, if we consider a data set of size 128 × 128 × 128 with layout (:serial, :news, :news), its subgrid size is $v_1 \times v_2 \times v_3 = 128 \times 8 \times 4$ for each floating-point node on a 16K CM. To perform communication along the second dimension of distance 8, all the data within a processing node must be moved out to its nearest-neighboring node. This means that there are (128 × 4) × 8 external (Off-Node) communications and no internal communications. In general, the external communication time is given by

$$T = C_{EX} + t_{OFF} \times v_1 \times v_2 \times v_3,$$

where C_{EX} is the startup time for external (Off-Node) communication. From a least-square fit of the timing data, we get $C_{EX} = 24.41 \times 10^{-6}$ sec and an external communication rate $t_{OFF} = 8.736 \times 10^{-6}$ sec/per 32-bit word. In this case, the startup time does not depend on the geometry.

REMARK: For distances of powers of 2 which are larger than the subgrid size of the communication axis, we need only take twice as much time as is needed for the case of communication with distance equal to the subgrid size of the communication axis in view of the binary-reflected Gray code ordering of the off-chip bits in the grid address (see Johnsson, 1987).

COMBINING INTERNAL AND EXTERNAL COMMUNICATION

In this section we combine the results on internal communication with external communication to form a general model of a complete communication model. As an example, consider a data set with layout (:news, :news, :news) and subgrid size $v_1 \times v_2 \times v_3$, so that a single processing node simulates $v_1 \times v_2 \times v_3$ virtual processing nodes. A *cshift* along the second dimension of distance d, with $d < v_2$, consists of both internal and external communication since under the NEWS ordering data are allocated across different processing nodes. Suppose that all virtual processing nodes communicate a distance d to the right along the second axis. There are d layers of data to the rightmost side requiring off-node communication, while the remaining $v_2 - d$ layers only need to do memory moving internally. Thus the total communication time $T(d, 2, v_1, v_2, v_3)$ is the sum of both internal and external communication time because CM-2 cannot perform both internal and external communication instructions simultaneously due to its SIMD architecture. We thus have

$$T(d,2,v_1,v_2,v_3) = T_{internal} + T_{external},$$

$$T_{internal} = (C_2 + (C_1 + t_M \times (v_2 - d)) \times v_1) \times v_3,$$

$$T_{external} = C_{EX} + t_{OFF} \times v_1 \times d \times v_3.$$

We can further decompose the external communication t_{OFF} into two parts. One part is the unit time for memory moving to a temporary buffer, t_M, and the other the unit time for sending out through the external hypercube wire, t_{EX}, i.e., $t_{OFF} = t_M + t_{EX}$. Then we can simplify the formulas so that the total communication time it takes for a data set of subgrid size $v_1 \times v_2 \times v_3$ to perform a *cshift* of distance d to the right along the second axis can be written

$$T(d,2,v_1,v_2,v_3) = T_{Internal\_Memory\_Moving} + T_{External\_Sending},$$

$$T_{Internal\_Memory\_Moving} = (C_2 + (C_1 + t_M \times v_2) \times v_1) \times v_3,$$

$$T_{External\_Sending} = C_{EX} + t_{EX} \times v_1 \times d \times v_3.$$

In general, along the j-th axis, $1 \leq j \leq 3$, the total communication time $T(d, j, v_1, v_2, v_3)$ it takes for a data set of subgrid size $v_1 \times v_2 \times v_3$ to perform a *cshift* of distance $d < v_j$ is given by

$$T(d,j,v_1,v_2,v_3) = T_{Internal\_Memory\_Moving} + T_{External\_Sending},$$

$$T_{Internal\_Memory\_Moving} = (C_2 + (C_1 + t_M \times v_j) \times v_{(j-1)\mathrm{mod}3}) \times v_{(j+1)\mathrm{mod}3},$$

$$T_{External\_Sending} = C_{EX} + t_{EX} \times v_{(j-1)\mathrm{mod}3} \times d \times v_{(j+1)\mathrm{mod}3}.$$

More generally, the communication for an m-dimensional data set is performed in a similar manner. Suppose we want to perform a *cshift* operation along the j-th axis in which data are distributed in the NEWS ordering ($j \leq m$). If the dimension $m > 3$, data of subgrid size $V = v_1 \times v_2 \times \ldots \times v_m$ is considered a three-dimensional set with $V = V_1 \times V_2 \times V_3 = (\Pi_{k=1}^{j-1} v_k) \times v_j \times (\Pi_{k=j+1}^{m} v_k)$. If the dimension $m < 3$, the data is padded out to a three-dimensional set and the communication is performed in the same way as for the three-dimensional data sets. In summary, combining the internal and external communication times, a general formula for the total communication time of a *cshift* of distance d performed along the j-th dimension is given by

$$\begin{aligned} T(d,j,V_1,V_2,V_3) &= (C_2 + (C_1 + t_M \times V_j) \\ &\quad \times V_{(j-1)\mathrm{mod}3}) \times V_{(j+1)\mathrm{mod}3} \\ &\quad + C_{EX} + t_{EX} \times V_{(j-1)\mathrm{mod}3} \\ &\quad \times d \times V_{(j+1)\mathrm{mod}3}, \\ &\quad \text{if } 1 \leq d < V_j; \\ &= C_{EX} + t_{OFF} \times V_1 \times V_2 \times V_3, \\ &\quad \text{if } d = V_j; \\ &= 2 \times (C_{EX} + t_{OFF} \times V_1 \times V_2 \times V_3), \\ &\quad \text{if } d > V_j,\ d = 2^p,\ p \in \mathrm{N}, \end{aligned} \qquad (1)$$

where $C_1 = 1.52 \times 10^{-6}$ sec, $C_2 = 5.8 \times 10^{-6}$ sec, $t_M = 0.52 \times 10^{-6}$ sec per 32-bit word, $C_{EX} = 24.41 \times 10^{-6}$ sec, and $t_{EX} = t_{OFF} - t_M = 8.736 \times 10^{-6} - 0.52 \times 10^{-6} = 8.216 \times 10^{-6}$ sec per 32-bit word.

ACCURACY OF THE COMMUNICATION TIMING MODEL

To test the accuracy of our communication model, we use it to predict the time needed to perform one *cshift* of distance d along the j-th axis. We then compare the predicted communication time with the measured time obtained by averaging over 1000 direct implementations of the *cshift* operation. Here we summarize the main results. We perform *cshift* on the 8K CM-2 with different distances along the *first* axis on a 2D square data with layout (:news, :news). The relative errors are quite uniform with respect to the distance and are all less than 20%. We also perform the *cshift* operation along the *second* axis. Again, the maximum relative errors are less than 20%, but the relative errors are more spread out and centered around zero for different data sizes. If we perform the *cshift* operation along the first and second axis on the 3D data with layout (:news, :news, :news) on the 8K CM-2, the relative errors will still be less than 20%. On a 16K CM-2 for the 3D data, the relative errors are less than 15%.

Timing Model for Floating-Point Operations

In the previous sections, we developed a timing model for grid communication on the current slicewise model of CM-2. In order to predict the time for a finite-difference program we also need a timing model for the arithmetic operations.

The floating-point operation can be decomposed into three parts: (1) sending data from the common memory area through the Sprint chip without transposition to the FPU; (2) performing the computation in FPU; (3) sending the result through the Sprint chip without transposition back to the associated bit-processors. Therefore, a reasonable floating-point operation performance model consists of a startup time needed to fill the pipeline of the floating-point vector co-processor unit and a linear growth rate once the pipeline is full. This is confirmed by our experiments in which each basic floating-point operation was computed 1,000 times and the measured time averaged for different data sizes. Therefore, we will use a timing model of the form $T(n) = c + \alpha * n$, where $T(n)$ is the time needed to carry out a floating-point operation, α (μ secs/VP) is a pipelined rate for each basic floating-point operation, and n the VP ratio per processing node. By using a least-square fit, we obtain estimates for c and α for several basic floating-point operations (see Table 1). Note that the time required for vector division or scalar division is about 2.5 times that required for vector addition, subtraction, or multiplication.

Since the ratio of the number of floating-point operations to the computational time is $n/(c + \alpha n) = 1/(\alpha + c/n)$, it is clear that a higher VP ratio (n) gives a better Gflops performance of floating-point operations. This is also confirmed by the results of our experiments as shown in Figure 2, where we have plotted the Gflops performance versus the VP ratio for each processor node on an 8K CM-2. The different curves in Figure 2 correspond to the different basic floating-point operations listed in Table 1. The Gflops performance curves overlap for several of the operators since these operators have very similar performance parameters.

Next, we look at more general floating-point operations. The Weitek FPU has one pipe for multiplication and another for addition, so that both operations can be performed in a unit clock cycle. Thus, complicated arithmetic expressions can be performed at a faster rate than the basic operations. As we can see from Table 2, which lists the computational time of several triad vector operations, the time per floating-point operation for these triad operations is almost twice as fast as that for the basic floating-point operations of Table 1.

For more complicated arithmetic expressions, it is thus clear that the execution time is not proportional to the number of arithmetic operations. Figure 3 shows the performance (on an 8K CM) of several commonly used arithmetic expressions for the second-, fourth-, and sixth-order finite-difference schemes for U_x and U_{xx}. For example, the curve at the bottom corresponds to a simple arithmetic expression, $a + \beta(\gamma(u_1 - u_2) + \mu b)$, and the curve on the top to a more complicated

Table 1
Performance Parameters for Basic Floating-Point Operations. *a*, *b* and *x* Are Vectors, While γ ia a Scalar

Operation	c (μsec)	α (μsec)
$x = a + b$	9.9928	1.1555
$a = a + \gamma$	9.5606	0.8233
$x = x - b$	10.1604	1.1557
$a = a - b$	10.3266	1.1212
$a = a - \gamma$	9.5659	0.8233
$x = x * b$	10.1588	1.1557
$a = a * b$	10.3123	1.1212
$a = a * \gamma$	9.6001	0.8233
$x = a/b$	9.4305	2.7644
$a = a/\gamma$	9.3962	2.4302

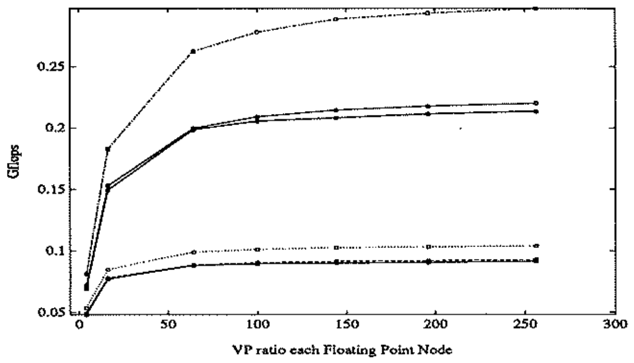

Fig. 2 Some Basic Floating-Point Operations Performance on CM-2.

Table 2

Performance Parameters for Triad Floating-Point Operations. *a, b, c,* and *x* Are Vectors, While γ and β Are Scalars

Operation	c (μsec)	α (μsec)
$a = \gamma * (a + b)$	10.4871	1.3475
$x = \gamma * (a + b)$	10.6963	1.3819
$c = c * (a + b)$	8.8799	1.3885
$x = c * (a + b)$	9.4128	1.4210
$x = c * a + b$	9.2841	1.4996
$c = c * a + b$	8.7737	1.4616
$x = a + \gamma * b$	10.6234	1.2338
$a = a + \gamma * b$	10.2402	1.1962
$a = \beta + \gamma * b$	13.0484	0.9692
$b = \beta + \gamma * b$	11.7802	0.8979
$a = \beta + a * b$	10.6656	1.1977

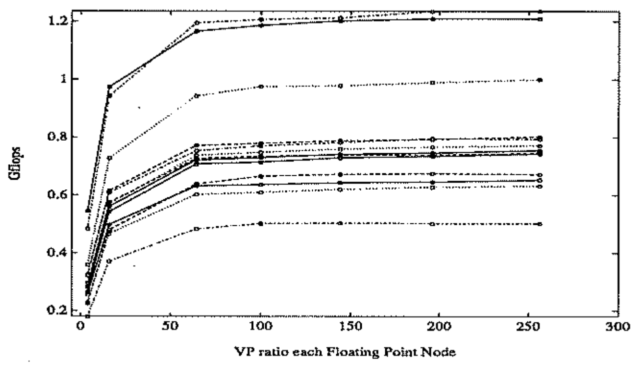

Fig. 3 **Gflops performance of finite difference floating-point operations versus VP ratio on CM-2.**

arithmetic expression, $a + \beta(\gamma(u_1 - u_2) + \mu(v_1 - v_2) + \eta(w_1 - w_2) + \nu b)$, where $a, b, u_1, u_2, v_1, v_2, w_1$, and w_2 are vectors and β, γ, μ, η and ν are scalars. Each curve represents a different arithmetic expression, with the complexity increasing as we move up in the figure. We can see from Figure 3 that the Gflops performance increases as the arithmetic expression becomes longer.

In principle, we can develop a timing model for each of the different kinds of arithmetic expressions that may arise in a typical application code. However, this policy may prove to be too cumbersome and problem specific. Instead, we shall take the following approach in order to obtain a simpler model for practical use. We derive an "average" arithmetic operation rate, which we call t_a, by performing several different kinds of combinations of floating-point operations that are similar to the expressions occurring in finite-difference codes, on several data sets of various sizes, and then take the average. This "average" arithmetic operation rate t_a can thus be viewed as the rate for a generic arithmetic operation. It is thus independent of the length of the expressions, the startup time, and the type of the operations (whether scalar operation, scalar-vector operation, or vector-vector operation). This produces a model of the form:

$$\boxed{\begin{array}{l}\text{Total estimate for arithmetic time} = t_a \times n_a, \\ \\ \text{where } n_a \text{ is the total number of arithmetic} \\ \text{operations done on each processing node.}\end{array}} \quad (2)$$

Our experimental results produced the value[5]: $t_a = 3.379 \cdot 10^{-7}$ sec. Note that in this model we have ignored the startup time for arithmetic operations. The numerical experiments we shall now describe show that this time is negligible for arithmetic operations.

Applications

In the previous sections, we developed two timing models, one for grid communication and the other for floating-point operations, for the current slicewise model of

5. This rate was determined with the hardware and software in Fall 1991. The rate of the arithmetic operations is faster now.

CM-2. Given a finite-difference numerical scheme with periodic boundary condition, we can predict its total running time by properly adding the time required for arithmetic operations and the time for communications. To validate our timing model, we predict the performance of high-order finite-difference methods for the 2D Bürgers' system and compare our results with the actual measured times in Chang (1992) and Chang and Kreiss (1992). Our timing model could also be used to predict the general performance trend of high-order finite-difference methods on SIMD machines with architectures similar to CM-2. For this purpose, we also study the sensitivity of the performance to several hardware parameters.

TIMING MODEL FOR DIFFERENCE METHODS

We apply the timing model developed in the previous sections to predict the performance of high-order finite-difference methods for the following 2D viscous Bürgers' equations:

$$\begin{aligned} U_t + UU_x + VU_y &= \epsilon(U_{xx} + U_{yy}), \\ V_t + UV_x + VV_y &= \epsilon(V_{xx} + V_{yy}). \end{aligned} \tag{3}$$

The study of this model problem is motivated by potential applications to high Mach number flow calculation (see Chang, 1992, and Chang and Kreiss, 1992). A p-th order central-difference scheme requires $p/2$ grid point shifts for both sides. Therefore, a higher-order finite-difference scheme takes more time not only because it requires more arithmetic operations but also more communications. Note that if an expression appears several times in a program, the CM compiler recognizes it and executes it only once instead of several times. For example, in the following excerpt from our finite-difference code, *cshift(u, dim = 1, shift = 1)* is computed only once, even though it appears twice explicitly:

```
C  A segment of the code involving the computation for
   Uxx

     f1 = f1 + epsilon *(
   &     s1*(cshift(u,dim=1,shift=1)+cshift(u,dim=1,
          shift=-1))
   &  +s2*(cshift(u,dim=1,shift=2)+cshift(u,dim=1,
          shift=-2))
   &  +s3*u )

C  A segment of the code involving the computation for
   VUx

     f1 = f1 - v * (
   &     c1*(cshift(u,dim=1,shift=1)-cshift(u,dim=1,
          shift=-1))
   &  +c2*(cshift(u,dim=1,shift=2)-cshift(u,dim=1,
          shift=-2))
   &  )
```

In our implementation, we use the fourth-order Runge-Kutta method for the time integration, so that we need to call the spatial difference operator subroutine four times. For a p-th order central-difference scheme, each call to this subroutine requires performing *cshift* once for both variables u and v in both the x- and the y-directions with distance d and $-d$, where $d = 1, \ldots, p/2$. Thus the total communication time for each of the two coordinate directions is the same as that for calling each *cshift* with distance d, $d = 1, \ldots, p/2$, a total of 16 times (four stages in Runge-Kutta times two variables u and v times two shifts $\pm\, d$). By calling CMF_DESCRIBE_ARRAY(), the geometry of an 8K CM-2 is determined as a two-dimensional hypercube of dimension 16×16, and each processing node gets a subgrid of size $v_1 \times v_2 = n/16 \times n/16$. We extend this 2D data set to a three-dimensional set by setting $v_3 = 1$. Finally, by counting the number of arithmetic operations in our code, we get the total number of arithmetic operations done on each processing node in Table 3. Combining our results, we arrive at the following timing model for a single time step of a p-th order finite-difference method for the 2D Bürgers' equations on a 2D $n \times n$ grid on CM-2:

$$\boxed{\begin{aligned} T(n,p) = {} & t_a \times n_a(v_1,v_2,v_3,p) \\ & + 16 \sum_{i=1}^{m=2} \sum_{k=1}^{p/2} T(k,i,n/16,n/16,1), \end{aligned}}$$

where n_a is the total number of arithmetic operations performed on each processing node and $T(k,i,v_1,v_2,v_3)$ is given by Equation (1).

Table 3

Number of Arithmetic Operations, n_a, Required for Several Central Finite-Difference Schemes Ranging from Second to Tenth-Order

Order	Second	Fourth	Sixth	Eighth	Tenth
n_a	$189 \times \Pi_{m=1}^{3} v_m$	$285 \times \Pi_{m=1}^{3} v_m$	$381 \times \Pi_{m=1}^{3} v_m$	$437 \times \Pi_{m=1}^{3} v_m$	$573 \times \Pi_{m=1}^{3} v_m$

Table 4

Predicted Computational Time From Our Timing Model

Size	Second Order	Fourth Order	Sixth Order	Eighth Order	Tenth Order
128 × 128	0.0096	0.0192	0.0310	0.0440	0.0608
256 × 256	0.0286	0.0534	0.0824	0.1121	0.1530
512 × 512	0.0976	0.1715	0.2537	0.3306	0.4435
1024 × 1024	0.3593	0.6067	0.8709	1.0966	1.4498

Table 5

Measured Computational Time for One Time Step

Size	Second Order	Fourth Order	Sixth Order
128 × 128	0.0110	0.0221	0.0346
256 × 256	0.0329	0.0611	0.0911
512 × 512	0.1131	0.1981	0.2804

Table 6

Communication Part of the Timing Model

Size	Second Order	Fourth Order	Sixth Order	Eighth Order	Tenth Order
128 × 128	0.0055	0.0131	0.0228	0.0346	0.0484
256 × 256	0.0123	0.0287	0.0494	0.0743	0.1034
512 × 512	0.0322	0.0729	0.1219	0.1794	0.2452
1024 × 1024	0.0977	0.2122	0.3436	0.4917	0.6567

Note that for $n \geqslant 128$ and order of the difference method less than or equal to 10, we have $d < v_1$ and $d < v_2$, so that in Equation (1) the first case is always true.

In Table 4 we give the predicted times for several high-order finite-difference approximations, ranging from second- to tenth-order. In Table 5 we give the actual measured times from Chang (1992) and Chang and Kreiss (1992). We can see from Table 4 that the ratio of the total times in the second-, fourth-, and sixth-order methods is 1 : 1.76 : 2.60, respectively, in the case of a 512 × 512 grid. This is basically consistent with the measured time in Table 5, where the conquerable ratio is 1 : 1.752 : 2.479. In fact, the maximum relative error in the predicted time is less than 15% for the second-, fourth-, and sixth-order methods for all the data sizes. This indicates that our timing model is reasonably accurate.

In Table 6 and Table 7 we give the breakdown of the communication and arithmetic time. We see that the arithmetic time exceeds the communication time for large data sizes. The communication time is significant only when the data sizes are small. The crossover point depends on the order of the scheme. For example, the arithmetic operation time for the second-order method begins to dominate the total computational time for data sizes larger than 128 × 128. By comparison, the arithmetic operation time for the sixth-order method begins to dominate the total computational time only for data sizes larger than 256 × 256.

PREDICTED PERFORMANCE OF IMPROVED COMMUNICATION STARTUP

Suppose that the communication startup time can be improved by a factor of 10 while the other parameters are kept fixed. For this case we obtain the performance estimates in Table 8. By comparing Table 8 with Table 4, we see that the total times do not decrease very much, while the ratio of the total time in the second-, fourth-, and the sixth-order methods for the 512×512 case is 1 : 1.74 : 2.58, a result which is similar to the original estimates in Table 4. This indicates that the overall performance of the difference methods is not sensitive to the communication startup time. This is to be expected since the startup time is proportional to $V_2 \times V_3$, which is much smaller than the total internal communication time, which is $O(V_1 \times V_2 \times V_3)$.

PREDICTED PERFORMANCE OF IMPROVED COMMUNICATION RATE

We consider three cases. We first reduce the external communication parameter t_{EX} by a factor of 10, obtaining the results shown in Table 9. Observe that the total time does, in fact, decrease, but not by much. We also see that the reduction in time is in the ratio 1 : 3 : 6 for the second-, fourth-, and sixth-order methods, respectively, since the external communication time is proportional to the amount of data moving outside the processing node and does not depend on the geometry. For example, to approximate U_x with periodic boundary condition by a second-order central-difference scheme, we need to do distance 1 and distance -1 communications. In the CM Fortran code, we do

$$s_1 \times (cshift(u, 1, 1) - cshift(u, 1, -1)),$$

which requires one column, located at the right boundary of a 2D data set in one processing node, to move out to its right neighbor, while a second column, located at the left boundary, must move out to its left neighbor. Thus, a total of two columns must move out. Analogously, for the fourth-order scheme, we need to do distance 1, –1, 2, and –2 communications. In the CM Fortran code, we do

$$s_1 \times (cshift(u, 1, 1) - cshift(u, 1, -1)) + s_2 \times (cshift(u, 1, 2) - cshift(u, 1, -2)),$$

which requires two columns of a 2D data set in one processing node to be communicated to another processing node at distance 1 away, and four columns to be communicated to another processing node at distance 2

Table 7
Arithmetic Part of the Timing Model

Size	Second Order	Fourth Order	Sixth Order	Eighth Order	Tenth Order
128 × 128	0.0041	0.0062	0.0082	0.0095	0.0124
256 × 256	0.0163	0.0247	0.0330	0.0378	0.0496
512 × 512	0.0654	0.0986	0.1318	0.1512	0.1983
1024 × 1024	0.2616	0.3945	0.5273	0.6049	0.7931

Table 8
Overhead Times Are 10 Times Faster

Size	Second Order	Fourth Order	Sixth Order	Eighth Order	Tenth Order
128 × 128	0.0082	0.0165	0.0269	0.0385	0.0539
256 × 256	0.0259	0.0480	0.0742	0.1012	0.1394
512 × 512	0.0922	0.1607	0.2376	0.3090	0.4166
1024 × 1024	0.3486	0.5852	0.8387	1.0536	1.3962

Table 9
External Communication Is 10 Times Faster

Size	Second Order	Fourth Order	Sixth Order	Eighth Order	Tenth Order
128 × 128	0.0076	0.0132	0.0189	0.0239	0.0306
256 × 256	0.0246	0.0413	0.0582	0.0719	0.0926
512 × 512	0.0896	0.1473	0.2054	0.2501	0.3227
1024 × 1024	0.3432	0.5584	0.7743	0.9356	1.2083

Table 10
Internal Communication Is 10 Times Faster

Size	Second Order	Fourth Order	Sixth Order	Eighth Order	Tenth Order
128 × 128	0.0087	0.0177	0.0288	0.0414	0.0578
256 × 256	0.0250	0.0464	0.0723	0.0992	0.1374
512 × 512	0.0828	0.1422	0.2106	0.2740	0.3740
1024 × 1024	0.2989	0.4869	0.6926	0.8608	1.1575

Table 11
Both Internal and External Communication Are 10 Times Faster

Size	Second Order	Fourth Order	Sixth Order	Eighth Order	Tenth Order
128 × 128	0.0067	0.0116	0.0168	0.0212	0.0277
256 × 256	0.0210	0.0344	0.0482	0.0589	0.0770
512 × 512	0.0747	0.1181	0.1623	0.1935	0.2533
1024 × 1024	0.2828	0.4386	0.5960	0.6998	0.9160

Table 12
Floating-Point Operation Is 10 Times Faster

Size	Second Order	Fourth Order	Sixth Order	Eighth Order	Tenth Order
128 × 128	0.0060	0.0139	0.0239	0.0358	0.0501
256 × 256	0.0144	0.0321	0.0540	0.0797	0.1103
512 × 512	0.0407	0.0865	0.1406	0.2017	0.2740
1024 × 1024	0.1316	0.2670	0.4191	0.5824	0.7735

away. Thus, there are a total of six columns to communicate. Therefore, the external communication for the fourth-order approximation of U_x is three times that for the second-order approximation. Similarly, we can deduce the reduced time for the sixth-order approximation of U_x. Therefore, the reduced times of the external communication for the second-, fourth- and sixth-order difference operators are roughly in the ratio 1 : 3 : 6. The sixth-order method saves more time when we reduce the external communication parameter, but the actual amount saved is not substantial.

The second case is to improve the internal communication parameter t_M by a factor of 10. The results are given in Table 10. The total times do not decrease by much. The ratio of the total time in the second-, fourth-, and sixth-order schemes for the 512 × 512 case is 1 : 1.71 : 2.54, which is similar to the ratio 1 : 1.76 : 2.6 for the unchanged case in Table 4. We also see that the ratio of the time saved in the three schemes is about 1 : 2 : 3. This can be explained as follows. Once the NEWS communication is required, data have to be moved in the internal memory of a processing node, or must be copied to some temporary buffer for external communication (see the timing model of Equation (1)). Therefore, all the data have to be moved in the memory. Since the ratio of the number of vectors involved in the second-, fourth-, and sixth-order schemes is 2 : 4 : 6, the ratio of the saved times is about 1 : 2 : 3.

We remark that although the sixth-order method represents a considerable improvement over the unchanged case, the fourth-order method is still more favorable, since even if the communication time is made negligibly small, the operation time required by the fourth-order method is still less than that required by the sixth-order method. Results from Chang (1992) and Chang and Kreiss (1992) indicate that the fourth-order method can achieve roughly the same error tolerance as the sixth-order method with the same number of grid points.

Finally, simultaneously reducing the internal and the external communication parameters t_{EX} and t_M by a factor of 10 gives us the results in Table 11. The total times decrease more but the ratio of the total times in the second-, fourth-, and sixth-order methods for the

512×512 case is 1 : 1.58 : 2.17 which is still not too far away from the original ratio of 1 : 1.76 : 2.60.

We conclude that the fourth-order method remains the best method if we improve the communication times, because the floating-point operation time is dominant for the large grids.

PREDICTED PERFORMANCE OF IMPROVED FLOATING-POINT OPERATION TIME

Next, we look at what happens when we improve the floating-point operation parameter t_a by a factor of 10. The predicted performance of CM-2 is given in Table 12. As expected, the total times in this case decrease much more than in all the previous cases we have considered. The reduced times are proportional to the total number of floating-point operations, 1.89 : 2.85 : 3.81. This shouldn't be surprising because the floating-point operation time is dominant over the communication time for larger VP ratios.

One interesting observation is that when we scale down the arithmetic parameter by a factor of 10, the second-order scheme with twice as many grid points in each dimension becomes comparable with the sixth-order method in terms of performance. However, the second-order scheme requires four times as much memory space in 2D calculations. Therefore, it is still preferable to use the fourth-order scheme, which only needs as many grid points as the sixth-order scheme to achieve the same error tolerance and is also faster than the second-order scheme with four times as many grid points.

ASYMPTOTIC BEHAVIOR FOR VARIOUS IMPROVED TIMING PARAMETERS

We tested our model by systematically reducing these parameters, that is, the communication startup times (C_1, C_2, and C_{EX}), external communication time (t_{EX}) and internal communication time (t_M), and floating-point operation time (t_a), by a factor of 5, 10, 30, 60 and 100, respectively. We illustrate this behavior in Figure 4 for the fourth-order finite-difference method as an example. In this figure, we vary all the communication parameters (C_1, C_2, C_{EX}, t_{EX}, and t_M) by the same factor while keeping t_a fixed. Each curve reaches an asymptote very quickly. In fact, there is very little change for all the data sizes after the communication parameter has been reduced by a factor of 30 or more. In the extreme case of zero total communication time, the total computational time is governed only by the arithmetic operation time. This comparison result is the same as on a sequential machine. But as we know from Table 7, the second-order method with twice as many grid points in each dimension is more expensive than the sixth-order method. So the fourth-order method is preferable since both the fourth- and the sixth-order methods can achieve the same error (a few percent) with the same number of grid points. The behavior which results from reducing the arithmetic operation parameter is very similar. This we illustrate in Figure 5 for the fourth-order finite-difference method. In this figure, we keep all the communication parameters fixed and vary the arithmetic parameter t_a. Again, each curve reaches an asymptote as the arithmetic parameter is reduced. In the extreme case of zero arithmetic time, the total computational time is governed by the communication time. From Table 6, we know that the execution time required for the second-order method with twice as many grid points in each dimension is comparable with the execution time in the sixth-order method. Therefore, the simple second-order method with four times as many grid points is competitive with the sixth-order method in that it achieves the same error tolerance so long as memory storage is not a concern. As before, the fourth-order method is still the best since it can achieve roughly the same error tolerance with the same number of grid points as the sixth-order method, and is also faster than the second-order method with twice as many grid points in each dimension.

Concluding Remarks

We developed a timing model for the slicewise model on the CM-2. We also verified its accuracy and applied it to predict the performance of high-order finite-difference schemes for the 2D Bürgers' equations. This model should be easily extendable to other architectures that are similar to CM-2.

Our study indicates that the fourth-order scheme gives the best overall performance on the current CM-2

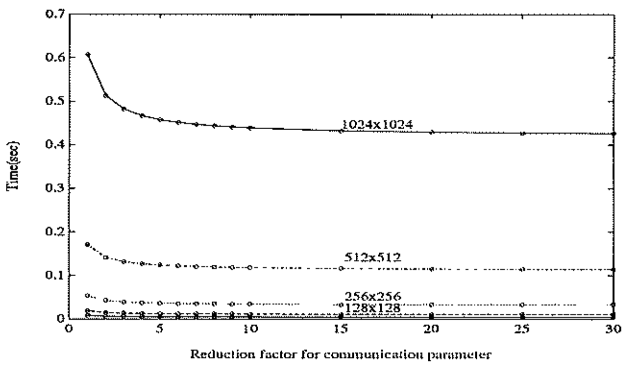

Fig. 4 Total time versus reduction factor for fourth-order method.

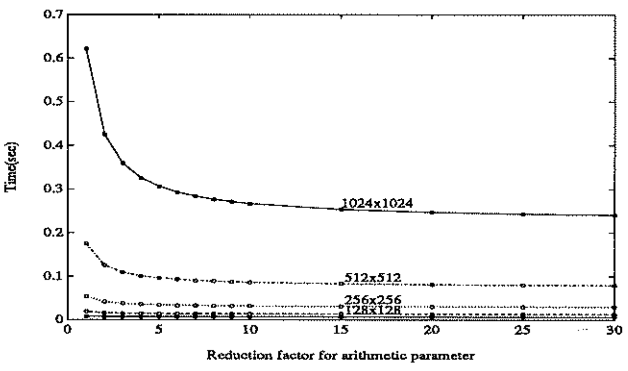

Fig. 5 Total time versus reduction factor for fourth-order method.

for the 2D Bürgers' system based on considerations of computational efficiency and memory requirements. This conclusion remains valid even when we reduce the communication cost or improve the arithmetic operation speed by a factor of 10.

We also found that under the current CM-2 architecture communication dominates the arithmetic operation when the data size is small. But for large data sizes the arithmetic operation tends to dominate the calculation. Only when the arithmetic speed is improved substantially can a significant speed-up be achieved in the overall computational performance.

Our study shows that if the arithmetic operation can be substantially improved, the second-order method becomes very competitive with the higher-order methods so far as computational efficiency is concerned. However, the price we pay is to use more grid points (e.g., four times as many as required for a sixth-order scheme in 2D calculations). When memory storage is a concern, a higher-order method is still more preferable. On the other hand, a lower-order method is easier to implement and more flexible for mesh refinement and treatment of boundary conditions. These considerations may lead to the choice of a lower-order method.

ACKNOWLEDGMENT

Both authors were supported in part by the Office of Naval Research under contracts N00014-90-J-1695, the Department of Energy under contract DE-FG03-87ER25037, the National Science Foundation under contracts ASC 9003002 and BBS-87-14206, and the Army Research Office under contract DAAL03-91-G-0150 and subcontract DAAL03-91-C-0047. The second author was also supported in part by the Office of Naval Research under contracts ONR-N0014-92-J-1890, and the National Science Foundation under contracts ASC 92-01266.

We would also like to thank M. Bromley, S. Duggirala, A. Greenburg, and K. Mathur of Thinking Machines Corporation for their assistance.

BIOGRAPHIES

Yu-Chung Chang received her Ph.D. degree in Applied Mathematics from the University of California, Los Angeles in 1992. From 1992 to 1993, she was a postdoctoral fellow at the Courant Institute, New York University. She is currently a postdoctoral fellow at Caltech. Her research interests have focused on computational fluid dynamics and parallel computation.

Tony F. Chan, a native of Hong Kong, received his B.S. and M.S. degrees from the California Institute of Technology in 1973 and his Ph.D. in Computer Science from Stanford University in 1978. After a postdoctoral fellowship at Caltech, he joined the Department of Computer Science at Yale University as an assistant professor and since 1986 he has been Professor of Mathematics at the University of California at Los Angeles. His main research interests are parallel numerical algorithms, iterative PDE algorithms, and numerical linear algebra. He is a council member of SIAM and an editor of SIAM Review, SIAM Journal of Scientific Computing, and several other journals.

REFERENCES

Chang, Y. C., and Kreiss, H.-O. 1992. Comparison of finite difference and the pseudo-spectral approximations for hyperbolic equations (preprint).

Chang, Y. C. 1992. Comparison of finite difference approximation and the pseudo-spectral approximations for hyperbolic equations, and implementation analysis on parallel computer CM-2. Ph.D. Thesis. CAM Report 9202, UCLA.

Hockney, R. W., and Jesshope, C. R. 1981. *Parallel Computers.* Bristol, U.K.: Adam Hilger Ltd.

Johnsson, L. S. 1987. Communication efficient basic linear algebra computations on hypercube architecture. *J. Parallel and Distrib. Comput.* 4:133–172.

Leiss, E. L., and Lee, K. H. 1991. A CM-2 implementation of 2D migration in the pressure of anisotropy. In *Expanded Abstracts with Biographies, 61st Annual International SEG Meeting, SEG,* Houston, Texas.

Levit, C. 1988. Grid communication on the connection machine: Analysis, performance, and improvements. In *Proc. Conf. on Scientific Applications of the Connection Machine,* edited by H. D. Simon. NASA Ames Research Center, California. World Scientific Publishers, pp. 316–332.

Pozo, R. 1991. Performance modeling of parallel architectures for scientific computing. Ph.D. diss. Dept. of Computer Science, University of Colorado, Boulder.

Thinking Machines Corporation, 1991. *CM Fortran Optimization Notes: Slicewise Model, Version 1.0.* Thinking Machines Corporation, Cambridge, Massachusetts.

Thinking Machines Corporation, 1991. *CM Fortran Reference Manual, Version 1.0.* Thinking Machines Corporation, Cambridge, Massachusetts.