Greater computational power is needed for solving Computational Fluid Dynamics (CFD) problems of interest in engineering design. Parallel architecture computers offer the promise of providing orders of magnitude greater computational power. In this paper we quantify that promise by considering an explicit CFD method and analyze the potential parallelism for three different parallel computer architectures. The use of an explicit method gives us a "best case" analysis from the point of view of parallelism, and allows us to uncover potential problems in exploiting significant parallelism. The analysis is validated against experiments on three representative parallel computers. The results allow us to predict the performance of different parallel architectures. In particular, our results show that distributed memory parallel processors offer greater potential speedup. We discuss the import of our model for the development of parallel CFD algorithms and parallel computers. We also discuss our experiences in converting our model code to run on the three different parallel computers.

Computational Fluid Dynamics on Parallel Processors

William D. Gropp† and Edward B. Smith‡

Research Report YALEU/DCS/RR-570
December 1987

This work supported in part by the Air Force Office of Scientific Research under contract AFOSR-86-0098 and in part by United Technologies Research Center. Approved for public release: distribution is unlimited.

†Department of Computer Science, Yale University, New Haven, CT, 06520.
‡United Technologies Research Center, East Hartford, CT, 06108.

Nomenclature

$\overline{(\,)}$	=	value of ( ) after first step
c_v	=	specific heat
e	=	energy
F	=	group of terms defined by equation 2
h_T	=	total enthalpy
H	=	group of terms defined by equation 3
i	=	radial index
j	=	axial index
n	=	time index
p	=	pressure
p_0	=	upstream pressure
Q	=	heat source term
r	=	radial distance
r^-	=	radial distance to $-r$ edge of cell
r^+	=	radial distance to $+r$ edge of cell
r_{ave}	=	average radial distance across cell
R	=	pipe radius
R_{gas}	=	real gas constant
t	=	time
T	=	temperature
T_0	=	constant temperature for Poiseuille flow
$\vec{v}$	=	velocity vector
v_r	=	radial velocity
v_z	=	axial velocity
v_θ	=	azimuthal velocity
V_0	=	centerline velocity
z	=	axial distance
α	=	thermal conductivity
Δr	=	radial step size
Δt	=	time step size
Δz	=	axial step size
κ	=	bulk viscosity
μ	=	viscosity
∇	=	gradient
ρ	=	density

Introduction

Computational Fluid Dynamics (CFD) problems are among the most demanding scientific computing problems in terms of the computational resources they require. Currently, CFD problems are extensively solved on vector supercomputers, primarily Cray and Cyber. While current supercomputers have adequate computational power to solve CFD problems which are three dimensional or unsteady or viscous, extending CFD to problems which are three dimensional and unsteady and viscous requires vastly more computational power.

One example of a computationally intensive CFD problem is modeling the air flow through an airplane's gas turbine engine. The flow through a gas turbine engine is three dimensional, unsteady and viscous. Further the geometry is extremely complex including several turbine stages, several compressor stages and a combustor. Gas turbine engineers must design the entire engine and would simulate the flow through the entire engine if the capability were available. As an example of the computational intensity of CFD calculations, Rai recently took over 100 hours on a Cray X-MP to run a 3-dimensional viscous unsteady model for the air flow past two stator blades and a single rotor blade of an axial flow gas turbine [Rai 87]. Although this run time is far too excessive for the use of such a code in engineering design, only part of the gas turbine was modeled. As the Cray is on of the fastest machines in the world and near the physical limits on the speed of a uni-processor, no existing computer is fast enough for these calculations.

To speed needed calculations up several orders of magnitude faster supercomputers are needed in gas turbine engine design. Computer processors, however, are reaching their physical speed limits. Processor designs appear to be within a single order of magnitude of their speed limits due to physical limitations such as the speed of light and gate switching limits. Therefore needed throughput cannot be achieved with single processor computers.

Parallel processor computers offer the possibility of achieving the throughput needed in CFD. Researchers are now investigating the effectiveness of using parallel computers for the solution of CFD problems [Jame 87,John 87]. The need to understand and exploit the architecture of parallel computers, however, makes it unclear whether we can design parallel algorithms which will achieve the needed throughput for CFD problems.

To test the promise of parallel computers, we consider a model CFD problem and apply it to representatives of several types of parallel computers. This model is based on an explicit difference technique and thus has the greatest opportunities for parallelism (as compared to implicit difference techniques). For each class of parallel computer, we develop a time complexity model and validate it against our model problem. These complexity estimates are analytical estimates of the computer time needed to solve a CFD problem. The complexity estimates are expressed in terms of basic machine and problem parameters, such as floating point and communication speeds and number of grid points. From these estimates, we can estimate the performance of CFD codes on supercomputers utilizing significant parallelism.

Finally, we discuss our experiences in using parallel computers for solving CFD problems.

1 Architectures

A number of different types of parallel architecture computers are available today. These run from very fine grain machines such as vector processors and very long instruction word machines to large numbers of almost independent processors. In this paper, we will consider three architectures which represent an important part of the spectrum of possible parallel computers. These are multiple vector processors, tightly coupled MIMD and loosely coupled MIMD.

Multiple vector processors consist of several vector computers operating in parallel. Exchanges of information and synchronization between the processors is facilitated by special high-speed hardware. Examples are the Cray X-MP, Cray 2, ETA 10, and the Alliant FX/8, that last of which was used in this study.

Tightly and loosely coupled MIMD computers work on essentially independent data in parallel, with differing kinds of access to each other's data.

Tightly coupled MIMD computers are collections of independent processors which are closely connected, usually by shared memory. A parallel program running on this type of machine consists of independent threads of control which may access each other's data and can tightly control each other's operation by manipulating this shared data.

The advantage of the tightly coupled MIMD approach is that there is no single thread of control, and hence no *a-priori* serial execution. There have been a number of machines of this type built by recent startup companies, such as the Encore Multimax 120, used in this study and the Sequent Balance series.

Loosely coupled MIMD machines are collections of independent processors which communicate through some reliable mechanism, but which don't directly share any data. Instead, all interprocessor sharing of data is done by I/O operations, typically the sending and receiving of message packets. This provides a measure of programming safety and reproducibility of results often absent in shared memory (tightly coupled MIMD) machines, since all modifications to "shared" data structures is handled explicitly by the programmer, rather than implicitly through a shared memory access. However, most systems currently on the market have a high overhead associated with interprocessor communication.

Example machines of this type include the various hypercubes from Intel, NCUBE, and others, and some research machines, such as the LCAP system of IBM. The systems themselves vary from a few, fast processors (LCAP) to many slow processors (Intel Hypercube). The Intel Hypercube was used in this study.

2 Finite Difference Algorithms

Finite difference algorithms used in Computational Fluid Dynamics for time accurate calculations can be divided into two major types: explicit and implicit.

With *explicit finite difference algorithms* the value of a variable at the new time is determined from the values of the variables at the old time directly (that is explicitly) without dependence on the values at the new time. Practically speaking this means that the values of the variables can be determined without solving a system of equations. The time step size that can be taken is limited by a time step on the order of the Courant-Freidrichs-Lewy (CFL) criterion (the smallest value of the time for a particle traveling at the local fluid velocity to cross a finite difference grid interval). In many applications the CFL time step is much smaller (often orders of magnitude smaller) than the time step required for accurate calculation of the time variation of the variables.

The model problem studied here uses MacCormack's algorithm [MacC 69] which was chosen for its simplicity and wide familiarity to the CFD community. Major features of MacCormack's algorithm are:

- An explicit difference scheme which is second order accurate in space and first order accurate in time.
- A two step scheme in which the present time values are used to calculate the intermediate values in the first step, and the present time and intermediate values are used to calculate the values at the new time.

Implicit finite difference algorithms differ from explicit finite difference algorithms in that the determination of the values of the variables at the new time depends (implicitly) on the values of the variables at the new time. The advantage of a well-chosen implicit scheme is that numerical stability can be achieved with a time step size much greater than given by the explicit CFL limit [Jame 83].

The price to be paid by most implicit schemes is that the solution of a system of equations is required at each step. Except for the solution of a tightly banded linear system it is often faster to solve the linear system iteratively.

We left implicit schemes for later study in order to address the basic question of whether parallel architecture computers were applicable to solving CFD problems. We intend to study implicit schemes and linear solution techniques in future work. The use of sparse matrix methods in CFD problems is an area of current research on serial computers [Wigt 85] as well as on parallel computers.

3 Model Problem

Descriptions of the physical model problem, its geometry, its mathematical formulation, and the numerical solution method follow.

The *physical model problem* we consider is unsteady compressible viscous flow through an insulated duct. We further assume the flow to be axisymmetric with swirl. A time dependent model problem was chosen for this study since time marching schemes are generally used in CFD for both steady and unsteady problems. A two dimensional problem was chosen both to provide a start at physically reasonable models for combustor flow and to introduce the complexities of multi-dimensional problems. For the model problem studied here the fluid velocities and pressure are known so that the temperature can be obtained by integrating in time the energy equation which is linear in temperature. The density was obtained from the perfect gas law.

The case studied was obtained from the model problem by assuming Poiseuille flow (parabolic velocity profile and linear pressure gradient) and treating the residual terms as a heat source. Poiseuille flow is an exact solution for steady, incompressible, isothermal, viscous pipe flow. We wished to retain the compressible effects in case they affected the parallelization of the code. Poiseuille flow does not exactly satisfy the compressible flow equations. There are residual terms which are collectively treated as a heat source. Thus the solution to the case studied was steady Poiseuille flow and, in particular, constant temperature. This facilitated debugging.

The *geometry* of the model problem is a rectangular domain with a tensor product mesh. The left side is the inflow and the right side is the outflow. The top is the insulated wall, and the bottom is the centerline of the cylinder.

The *mathematical formulation* of the energy equation in cylindrical coordinates is given by equations 1 through 6.

$$\frac{\partial \rho e}{\partial t} + \frac{1}{r}\frac{\partial r F}{\partial r} + \frac{\partial H}{\partial z} = Q \tag{1}$$

where

$$\begin{aligned} F &= \rho v_r h_T - \alpha \frac{\partial T}{\partial r} + \left(\frac{2}{3}\mu - \kappa\right)(\nabla \cdot \vec{v}) v_r \\ &\quad -\mu\left(2v_r \frac{\partial v_r}{\partial r} + v_\theta\left(\frac{\partial v_\theta}{\partial r} - \frac{v_\theta}{r}\right) + v_z\left(\frac{\partial v_r}{\partial z} + \frac{\partial v_z}{\partial r}\right)\right) \end{aligned} \tag{2}$$

and

$$\begin{aligned} H &= \rho v_z h_T - \alpha \frac{\partial T}{\partial z} + \left(\frac{2}{3}\mu - \kappa\right)(\nabla \cdot \vec{v}) v_z \\ &\quad -\mu\left(2v_z \frac{\partial v_z}{\partial z} + v_r\left(\frac{\partial v_z}{\partial r} + \frac{\partial v_r}{\partial z}\right) + v_\theta \frac{\partial v_\theta}{\partial z}\right) \end{aligned} \tag{3}$$

where

$$e = \rho\left(\frac{1}{2}\vec{v} \cdot \vec{v} + c_v T\right)$$

$$
\begin{aligned}
h_T &= e + \frac{p}{\rho} \\
\vec{v} &= (v_r, v_\theta, v_z) \\
\nabla \cdot \vec{v} &= \frac{1}{r}\frac{\partial r v_r}{\partial r} + \frac{\partial v_z}{\partial z}.
\end{aligned} \tag{4}
$$

Q is the heat source term. In the case studied Q was set equal to the right hand side of equation 1 evaluated for Poiseuille flow

$$
\begin{aligned}
\vec{v} &= (0, 0, V_0(1 - r^2/R^2)) \\
p &= p_0 - \frac{4\mu V_0 z}{R}.
\end{aligned} \tag{5}
$$

Density is determined from the equation of state (perfect gas law)

$$
p = \rho R_{gas} T, \tag{6}
$$

and T is the unknown to be solved for. To simplify debugging the solution $T = T_0$ (T_0 a constant) was used.

The *numerical solution method* for the above partial differential equations is by MacCormack's explicit algorithm [MacC 69]. The finite difference form of these equations using MacCormack's algorithm is given by equations 7 and 8.

In the first step of MacCormack's two step difference scheme the derivatives of F and H are calculated using backward differences with central differences for the terms involving second derivatives.

$$
\begin{aligned}
\overline{(\rho e)}_{ij}^{n+1} = (\rho e)_{ij}^n - & \frac{\Delta t^n}{r_{ave_{ij}}} \frac{1}{\Delta r_{ij}} \left(r_{ij}^+ F_{ij}^n - r_{ij}^- F_{i-1j}^n \right) \\
- & \frac{\Delta t^n}{\frac{1}{2}(\Delta z_j + \Delta z_{j-1})} \left(H_{ij}^n - H_{ij-1}^n \right).
\end{aligned} \tag{7}
$$

In the second step of MacCormack's scheme the derivatives of F and H are calculated using forward differences with central differences for the terms involving second derivatives.

$$
(\rho e)_{ij}^{n+1} = \frac{1}{2} \left\{ \begin{array}{ll}
(\rho e)_{ij}^n + \overline{(\rho e)}_{ij}^{n+1} - & \frac{\Delta t^n}{r_{ave_{ij}}} \frac{1}{\Delta r_{ij}} \left(r_{ij}^+ \overline{F}_{i+1j}^{n+1} - r_{ij}^- \overline{F}_{ij}^{n+1} \right) \\
- & \frac{\Delta t^n}{\frac{1}{2}(\Delta z_j + \Delta z_{j-1})} \left(\overline{H}_{ij+1}^{n+1} - \overline{H}_{ij}^{n+1} \right)
\end{array} \right\} \tag{8}
$$

In CFD studies of duct flow typical grid sizes are order of 50 to 100 (radial) by 100 to 1000 (axial), and the CFL limited time steps generally number in the tens of thousands. In this study we used matrices from 50×50 to 250×250 in size and took from 10 to 50 time steps. The matrix sizes were chosen to test effects of memory size and the number of time steps was chosen large enough to get accurate timings.

The boundary conditions are specified inflow and axially smooth outflow with insulated cylinder walls. The geometry was chosen to keep the code simple for transporting from one parallel machine to another.

4 Time Complexity Model

In order to understand the potential of parallel programming for CFD problems, we need to develop models which will allow us to predict the performance of a parallel computer. Time complexity models are constructed for each of the three classes of parallel computers used in this study. Since the time complexity model is an analytic expression for the computer time needed to solve a problem, based on fundamental parameters of the model problem and the computer, we can use the complexity estimates to predict the performance of new parallel computers and suggest which architectures offer the most potential for CFD problems.

In this section we will first give some basic definitions, then explain superlinear speedup. Then we will give time complexity estimates for simple stencils and apply and extend these estimates to our model problem.

We will denote the time complexity by $T(p,n)$, where p is the number of processors and the problem is on an $n \times n$ mesh. Another useful measure is the speedup, s, defined by $s(p,n) = T(1,n)/T(p,n)$. In a perfectly parallel algorithm with perfect hardware and software, $T(p,n) = T(1,n)/p$. Algorithms, hardware, and software are not, however, perfect and this section is concerned with modeling the "imperfections." We therefore define the computational efficiency as $T(1,n)/(pT(p,n))$. A perfect parallel computer and algorithm has a computational efficiency of 1.

The nature of these imperfections depends on the type of parallel computer. However, they are related to the access to memory. In a loosely-coupled MIMD parallel processor, it is the interprocessor communication time, which can be thought of as access to remote memory, which is important. In a tightly-coupled MIMD parallel processor, it is the access to shared memory, and the establishment of critical regions (access control to memory) which is important. In a multiple vector parallel processor, it is the access to shared memory again and cache contention among the processors which is important. In addition, various "non-parallel" operations may reduce the possible speedup.

4.1 Superlinear speedup

As an example of how memory access can affect the time complexity on parallel computers, we discuss a commonly misunderstood phenomena called superlinear speedup. Superlinear speedup is a speedup higher than the speedup of a "perfect" parallel computer and algorithm—that is, a computational efficiency greater than 1. This effect is real, and it is caused by "non-linear" features in the description of the parallel computer.

In particular, it is evident if the process of breaking the program into a number of parallel pieces produces smaller pieces which can use the hardware more effectively.

An example is a machine where each processor has a memory cache of size C : let the

time to run a problem of size m on p processors be

$$T_m(p) = \frac{m}{p} + g(m,p)$$

where

$$g(m,p) = \begin{cases} m/p & \text{if } m/p > C \\ 0 & \text{otherwise.} \end{cases}$$

The g represents the time to fill the cache from memory if the cache can not hold the entire problem (i.e., $T_m(1) = 2m$) then, as $p \to \infty$,

$$\frac{T_m(1)}{pT_m(p)} \to 2,$$

twice the "perfect" speedup.

Now, if instead we insist that m be "large enough", ie., $m > pC$, then we leave the regime where the problem fits in the cache, and the speed up is linear:

$$\frac{T_m(1)}{pT_m(p)} \to 1.$$

as both p and m go to infinity.

This effect is important in modern parallel computers because the relevant "cache size" has become very large. For example, in a tightly-coupled MIMD shared memory machine like the BBN Butterfly, the cost for accessing non-local memory is roughly three times that of local memory. Thus the entire local memory (several megabytes) can be considered the "cache" for the purposes of this argument. This effect can generate anomalous results for small problems.

4.2 Time complexity estimates for simple stencils

In order to motivate our choice of complexity model, we first develop a model for finite difference schemes with 5 and 9 point stencils for the three parallel architectures used in this study. Using these results, we can construct a model which predicts the time complexity of our model problem. We also discuss some global operations which are present in our model algorithm, and how they are handled on different computer architectures.

We first establish some basic ideas concerning two main classes of parallel programming styles: message passing and shared memory. We first describe these two approaches and then discuss the similarities in terms of both expressiveness and of efficiency.

In a message passing computer, memory access for data on different processors is via inter-processor communication which is handled by explicitly sending a message from one processor to another. There are two major parameters of interest. These are the message startup time α, and the transfer rate r. To make α and r independent of particular hardware implementations, α and r have been nondimensionalized by dividing by the

floating point speed in flops/sec. Thus α is in ops/startup and r is in ops/word. We will denote the floating point computation rate by f flops/sec.

For the tightly coupled computers, memory access is from shared memory to processor and the three parameters of interest are the memory access time, the speed of a floating point operation and the number of simultaneous memory references. The programming constructs used with shared memory machines include *barriers* and *critical regions*. A barrier is a synchronization point which all processors must reach before any of them can proceed. A critical region allows only one process to access and modify data at a time. In a parallel program, barriers may be used, for example, to wait for all processors to reach the end of a time step. A critical region may be used to provide a unique do-loop index to each parallel processor. These are discussed in more detail in, e.g., [Andr 85].

Message passing computers such as the Intel Hypercube and shared memory machines such as the Encore Multimax are often considered two completely different types of parallel computer. In fact, when viewed in the right way, they are very similar. What distinguishes them is the relative cost of referencing remote or shared information. The following table shows the correspondence between the costs of interprocessor communication for the two types of machines.

Message Passing	Shared Memory
I/O startup time	Cache miss overhead and time to establish critical regions and barriers
Transfer rate	Memory transfer rate

Each of these machines have different strengths. The shared memory machines have faster access to shared information, but pay a penalty in terms of higher overheads in the forms of barriers and critical regions. In general, these barriers and critical regions are intrinsic to the multiprocess computation and are therefore unavoidable. Message passing machines have simpler access control but at a higher cost in sharing data. Barriers and critical regions are sometimes used with message passing computers; however, the style of programming normally used with message passing makes them less important.

Consider the cost of computing a step of an explicit PDE on an $n \times n$ (in 2-d) and an $n \times n \times n$ (in 3-d) grid. This step will usually consist of an estimate of the time step size to use, based on CFL estimates, followed by the application of a local stencil. The estimate of the time step requires global information (the solution everywhere); the application of the stencil just local or neighbor information. We describe first the local communication and then the global communication. Since these effects are most noticable in the context of message passing, we describe them in those terms. However, similar considerations apply to shared memory.

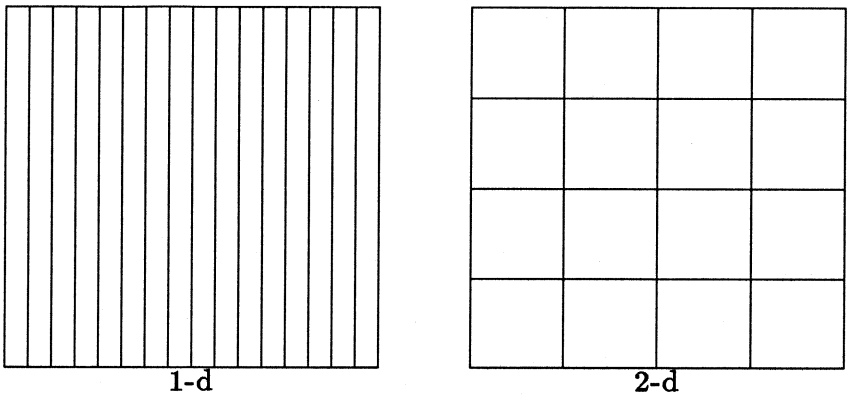

Figure 1: Two different decompositions of a 2-d domain. Each outlined area is given to a different processor.

The first step in any parallel program is the division of work among the processors. Divide this grid among the processors by slicing the problem domain by planes in 1, 2, or all 3 cartesian directions. Call the resulting decomposition 1-d, 2-d, or 3-d slices. An example of 1-d and 2-d slices for a 2-d domain is shown in Figure 1.

In each case, it is the length of the boundary of the slice which determines the communication cost. For example, in 2-d where the domain is sliced in one direction, the boundaries between processors are n mesh-points long, so that the effort for one step is

$$\frac{n^2}{p} + 2(\alpha + rn).$$

(All times are expressed in nondimensional units of floating point operations.) The "2" comes from sending the left internal boundary to the left neighbor and the right internal boundary to the right neighbor. While in principle these operations could go on simultaneously, in practice they require essentially exclusive use of the processor and memory bus. For slices in two directions, there are now either 4 neighbors (for a 5-point stencil) or 8 neighbors (for a 9-point stencil), and the cost becomes

$$\frac{n^2}{p} + 4\left(\alpha + r\frac{n}{\sqrt{p}}\right) + 4\left(\alpha_2 + r\right).$$

Here α_2 to denotes the cost of a two hop link and is only present in the 9-point case. Similar analysis can be carried out for other cases; they are summarized in Table 1.

From these formulas it is clear that once the problem is large enough (n large), the communication costs become negligible. However, in practice the problems are neither enormous nor the constants the same size. Thus, the communication terms, though asymp-

slicing	dim	Time Estimate
1-d	2-d	$\frac{n^2}{p} + 2(\alpha + rn)$
2-d	2-d	$\frac{n^2}{p} + 4\left(\alpha + r\frac{n}{\sqrt{p}}\right) + 4\left(\alpha_2 + r\right)$
1-d	3-d	$\frac{n^3}{p} + 2(\alpha + rn^2)$
2-d	3-d	$\frac{n^3}{p} + 4\left(\alpha + r\frac{n^2}{\sqrt{p}}\right) + 4\left(\alpha_2 + rn\right)$
3-d	3-d	$\frac{n^3}{p} + 6\left(\alpha + r\frac{n^2}{\sqrt[3]{p^2}}\right) + 12\left(\alpha_2 + r\frac{n}{\sqrt[3]{p}}\right) + 8\left(\alpha_3 + r\right)$

Table 1: Times for various decompositions of the domain, for both 2 and 3-d domains. α_2 is the cost for a two-hop link and α_3 the cost for a three hop link. The terms containing them are not present for 5 or 7 point stencils.

totically negligible, can be of dominant importance. For example, take

$$\begin{aligned} p &= 64 \\ n &= 256 \\ f &= 1\ \mu\text{seconds/mesh-point} \\ f\alpha &= 3\ \text{milliseconds} \\ fr &= 1\ \mu\text{seconds/word.} \end{aligned}$$

(Recall that α and r are expressed in terms of the floating point computation rate f). This represents a fast node (over 1 megaflop) with moderate communication speeds and has at at least 2 megabytes of memory. An existing machine with similar parameters is the Intel VX hypercube. In this case, in 3-d with 1-d slicing and a 5-point stencil and using the time estimates in Table 1, the communication time is roughly 0.12 seconds and the computation time is roughly 0.24 seconds. The communication is taking *one third* of the total time. Further, this is assuming that the message requires only one startup, despite its large size (n^2 words). If we also require all messages to be less than 16384 bytes, this adds another 0.1 seconds of startup time, making communication almost *half* of the total time.

The analysis so far has been adequate for centered single step schemes such as Lax-Wendroff or leapfrog. However, many schemes use combinations of one-sided stencils. The MacCormick scheme used in our model problem is one such scheme. In this case, we simply consider each access separately. Specifically, sharing information at a boundary with another processor takes time

$$\alpha + rn. \tag{9}$$

One problem with considering each access separately is that there may be synchronization delays caused by different processors finishing at different times. We will ignore these effects for now; however, they are likely to become more important in large scale MIMD parallelism.

A certain amount of global communication is necessary in these algorithms as well. For example, the choice of time step size requires the global value of the maximum velocity. In addition, the values of interest in the computation include integrals along the boundary, which may need to be computed across processors. These operations fall into the general category of *reductions*: taking data from many places and combining it to produce a single result.

Reductions are often done using a fast distribution algorithm such as those in [Saad 85], with the operation (maximum or sum) inserted into the distribution. For example, a tree-based distribution algorithm such as those described in [Saad 85] is fast on both shared memory and hypercube/message passing computers. In this case, the time to reduce p items is

$$(\alpha + r + f) \log p$$

assuming simultaneous send/receives. One potential problem here is possible round-off error; to avoid that, we reduce upward using the tree to a single node, then distribute that single value downward to all nodes, again using the tree. The time is then

$$(2(\alpha + r) + f) \log p. \tag{10}$$

When using a small number of processors and a shared-memory or complete interconnect, it is often easier for a single processor to compute the reduction, then make that value available to all processors. In this case, the time is pf plus the time to establish two barriers, one before the reduction to insure that the data is ready, and one after it to insure that the result is ready.

4.3 Time complexity estimates for the model problem

Our model problem is more complicated than a simple 5-point explicit scheme because it is multistep and each step is not centered. However, we can break it down into

- floating point work
- local data exchanges
- global data exchanges.

This form is suitable to a program where the parallelism is expressed explicitly. In our particular implementation, we can write the total computation time per step as

$$T = F_1 \frac{n^2}{p} + F_2 \frac{n}{p_y} + 8\,\text{“global reduce”} + 6\,(\text{“send in y”} + \text{“send in x”}) \tag{11}$$

where F_1 is the floating point time per mesh point for operations on the whole grid, F_2 is the floating point time per mesh point for operations along one boundary, the grid is $n \times n$, p is the number of processors, and p_y is the number of processors in the y direction.

Using equations 9 and 10 as the communication model, equation 11 becomes

$$T = F_1\frac{n^2}{p} + F_2\frac{n}{p_y} + 8(\alpha + r)\log p + 6\left(\alpha + r\frac{n}{p_x}\right) + 6\left(\alpha + r\frac{n}{p_y}\right). \tag{12}$$

The memory access terms this equation are the direct communication terms involving α and r.

For this local communication model to be applicable, it must be possible to map a grid onto the parallel processor in such a way as to make adjacent slices adjacent in the parallel processor. We will assume that any 2 or 3-d grid of interest can be efficiently embedded in the parallel processor. Such embeddings are easy on hypercubes; alternatively, everything we say here applies equally well to a mesh connected processor of the correct dimension.

For a shared memory machine with sufficient memory bandwidth, the figures are similar, except the global reduction is done differently, and the overhead of data sharing is slightly different. In this case, the results are

$$T = F_1\frac{n^2}{p} + F_2\frac{n}{p_y} + 16\text{“barriers for reduce”} + 6\text{“barriers for data”}. \tag{13}$$

Here, the “barriers” are synchronization points in the code. Depending on the implementation, these can be order p or $\log p$. Further, it is possible to reduce the number of these by using barriers with values [Eise 87].

On a tightly coupled multiple vector processor such as the Alliant FX/8 the work estimates are slightly different. In these machines, “close coupling” of the processors allows the rapid transfer of information from one processor to another, and extremely fast synchronization. On the negative side, since there is no explicit parallelism, the programmer may be dependent on the Fortran compiler to generate parallel code and it is possible for there to be substantial amounts of non-parallel code generated. Further, any shared memory machine suffers from a potential bottleneck in memory access; depending on cache or register utilization and the exact pattern of operations chosen by the compiler, the performance may exhibit anything from superlinear speedup to a plateau in performance.

We can model this bottleneck in shared resources such as memory as follows. Let the computation consist of two parts: one which is arbitrarily parallel, and one which uses the shared resource. For example, consider a machine with p processors in which p floating point operations may be done in parallel. However, only k processors may use the shared memory at any time. Then the time to do a computation has the form

$$T(p) = \frac{a}{p} + \frac{b}{\min(p,k)}. \tag{14}$$

For $k = 1$, this is the well-known Amdahl's law. The speedup predicted for such a machine as p gets large $(p > k)$ is

$$s(p) = \frac{a+b}{\frac{a}{p} + \frac{b}{\min(p,k)}} \to \frac{(a+b)kp}{ak + bp}. \tag{15}$$

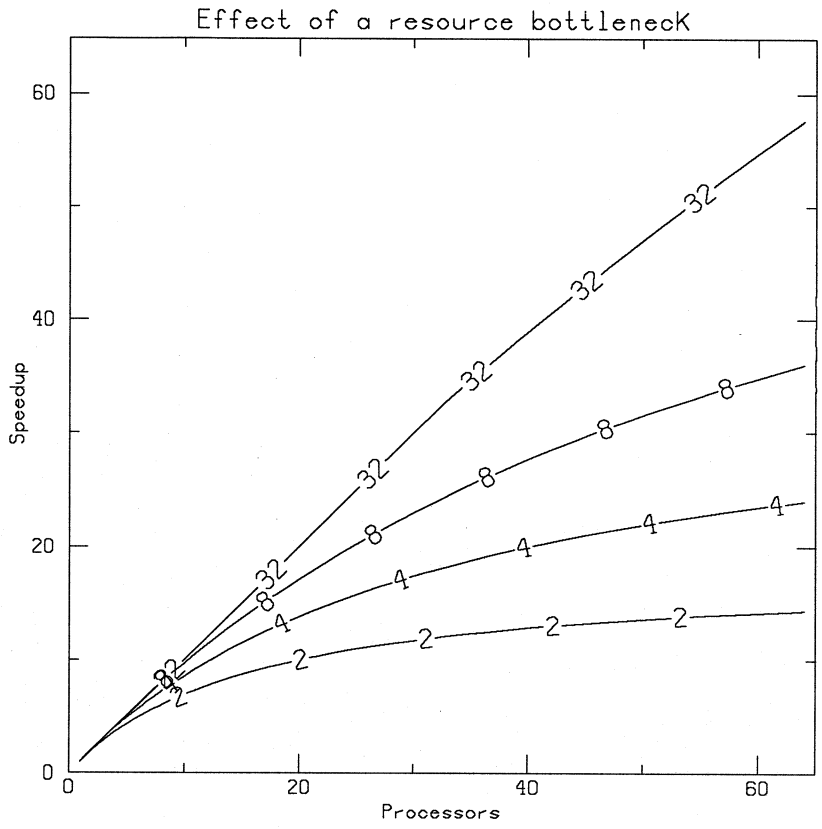

Figure 2: Graph of equation 15 for $a = 8$ and $b = 1$, for various values of k (k is labeled on the lines). The ratio $a/b = 8$ is similar to the ratios we observed in our experiments.

As $p \to \infty$, the speedup tends to $(a + b)k/b$ or

$$s \to \left(1 + \frac{a}{b}\right) k. \tag{16}$$

Figure 2 shows the behavior of s for different values of k. Note the approach to the asymptotic value of s, which for the example values of a and b is $9k$.

This "bottleneck" effect is present in both multiple vector processors and in tightly-coupled shared memory computers. Jordan [Jord 87] discusses these effects in more detail.

We have not considered all possible effects. One important effect is *load balancing.* On distributed memory machines, the current approach is to divide the problem statically among the processors. This is a large grain division of work. If the computational work is not equally divided among the processors, there will be additional costs caused by some processors going idle while others continue to work. However, this effect diminishes as the size of the problem relative to the number of processors increases.

5 Experimental Results

In this section we show the results of computing our model problem on several different types of architectures.

We ran our model problem on three different parallel computers. An Intel iPSC Hypercube, an Encore Multimax 120, and an Alliant FX/8. The Intel Hypercube is a loosely coupled MIMD, the Encore is a tightly coupled MIMD, and the Alliant is a multiple vector processor. Each of these represents a different class of parallel computer.

Figures 3, 4, and 5 show the results of our experiments. We have computed fits to these experiments using the models developed in Section 4. These fits were obtained by a non-linear least-squares fit to the timing data, after scaling the timing data by n^2/p.

5.1 Intel iPSC Hypercube

The Intel iPSC Hypercube is an example of a loosely coupled MIMD computer. The processors are connected in a hypercube architecture, and communicate by sending messages over a dedicated ethernet link (one per processor-pair). Each processor has 0.5 megabytes of memory, and on our machine, there is an additional 4.0 megabytes of memory on an attached board.

Figure 3 presents the results of these experiments. Each computation represents 10 time steps over the grid. A fit to the data using equation 12 gives

$$T = 0.066\frac{n^2}{p} + 0.014\frac{n}{p_y} + 0.861 \log p + 0.0302(n_x + n_y) + 0.868, \tag{17}$$

where $n_x = n/p_x$ and $n_y = n/p_y$.

In Figure 3, we see the effects of the uneven distribution of work among the processors in the stepped increase in the speedup figures. Each processor has some fixed number of "slices", each of which has roughly $n/p_x \times n/p_y$ mesh points. If p_x or p_y do not divide n, then some processors have $\lceil n/p_x \rceil \times \lceil n/p_y \rceil$ and some have $\lfloor n/p_x \rfloor \times \lfloor n/p_y \rfloor$. The relative difference is approximately (assuming both p_x and p_y don't divide n)

$$\frac{p_x}{n} + \frac{p_y}{n}.$$

As the number of processors approaches n, the difference can be very large. The points in p where there are jumps in the speedup for the 1-d decomposition are just those values of p which divide n, as can be seen in Figure 3. Note that the hypercube interconnections are not of major importance; a 2-d mesh interconnection would give almost the same results.

5.2 Encore Multimax

The Encore Multimax is an example of a shared memory MIMD computer. There are two processors per board, along with a cache memory. These boards are connected to

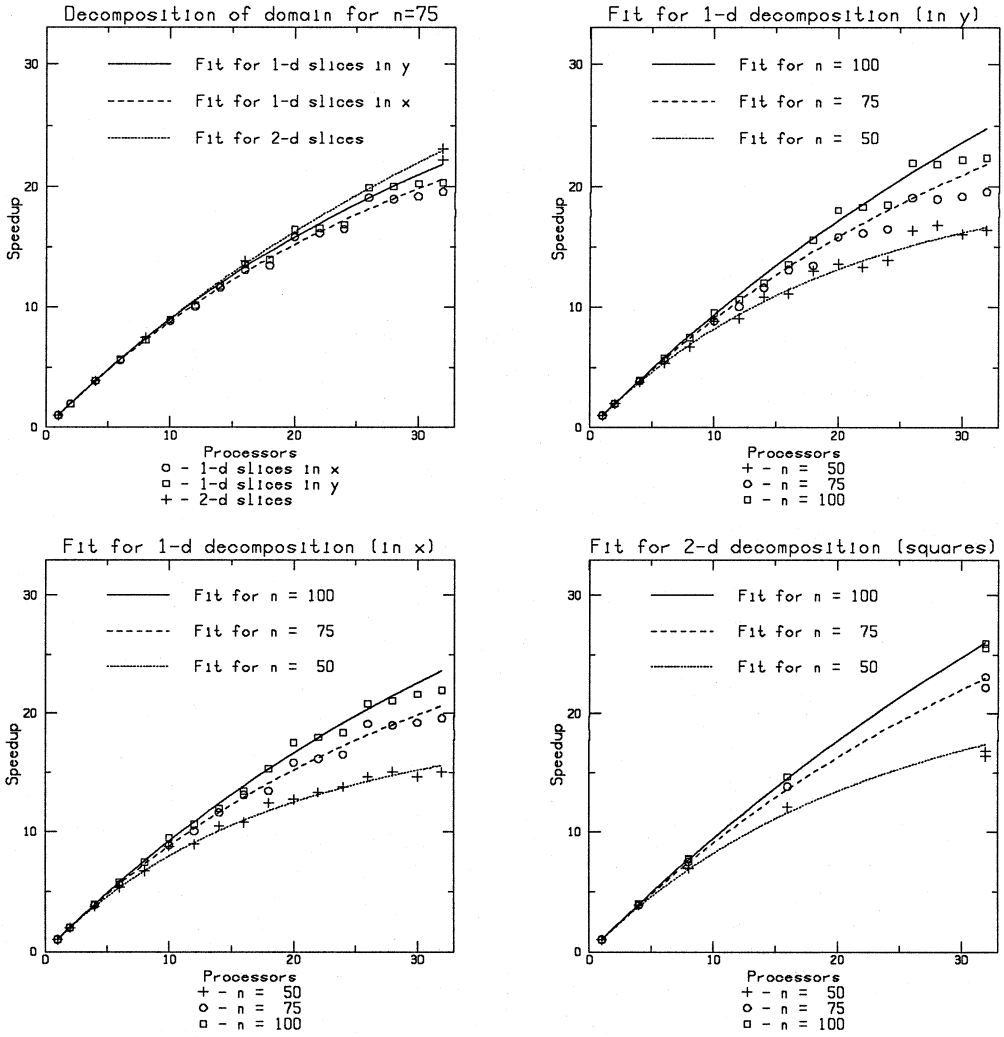

Figure 3: Speed up figures for the Intel iPSC hypercube. The graphs show fits as a function of decomposition at a fixed problem size, and as a function of problem size for fixed decomposition. The same fit parameters were used in all graphs, and are drawn as curves.

a very fast bus, on which the shared memory resides. The computer runs Unix™ and provides a time-sharing environment. One problem when doing timing measurements in a multi-user environment is that any parallel job is competing for processors with other users, mailer daemons, etc. Timings are thus somewhat erratic. Further, they don't reflect synchronization problems well, since idle processes won't use CPU time, and "real time" measurements are even more inaccurate on a time-sharing system. Thus, our results shown in Figure 4 are at best approximate. However, they do match our speedup estimate very well with parameters in equation 18, which is a combination of equation 13 and equation 14.

$$T = 0.059\frac{n^2}{p} + 0.0080\frac{n^2}{\min(p, 10)} + 0.08\frac{n}{p_y} + 0.29p. \tag{18}$$

These parameters give a very good fit to our experimental data as shown in Figure 4.

Note that since the onset of the "bottleneck" term is so high, this may be due more to the availability of processors than to limitations in the hardware. In fact, the Encore bus is very fast relative to the speed of the processors, and we do not expect to see a significant degradation for the number of processors available.

5.3 Alliant FX/8

The Alliant FX/8 is an example of a multiple vector processor computer. It has up to eight "computational elements", each of which is a vector processor. These processors share two high-speed caches, which in turn have a high speed channel to memory. It is possible for the processors to exceed speed of the cache.

Computations were done for 51 time steps and mesh sizes of $n = 50$, 100, 150, 200, and 250 on an eight processor Alliant FX/8. Figure 5 shows the speedups observed for various problem sizes compared to a fit against Equation 14. The fit used is

$$T = 0.00645\frac{n^2}{p} + 0.00085\frac{n^2}{\min(p, k)},$$

where $k = 2.45$. The fit is insensitive to k in the range $2 \leq k \leq 3$.

The apparent super-linear speedup for two processors shown by the curves in Figure 5 is an artifact of the curve fit, caused by the rapid turn-down in speedup at three processors.

The data in Figure 5 show a dropoff below linear speedup as the number of processors increases. This dropoff is due in part to a memory bottleneck. The memory bottleneck occurs when many processors require the same data from memory. In the Alliant FX/8 each of the four quadrants of the fast cache memory can be accessed by any two computational elements (processors) via a crossbar interconnect. When more than two computational elements try to access the same data there is "cache contention" which is a form of memory bottleneck. In addition cache and main memory are updated via hardware to maintain memory system consistency. Thus if more cache memories were added there would be

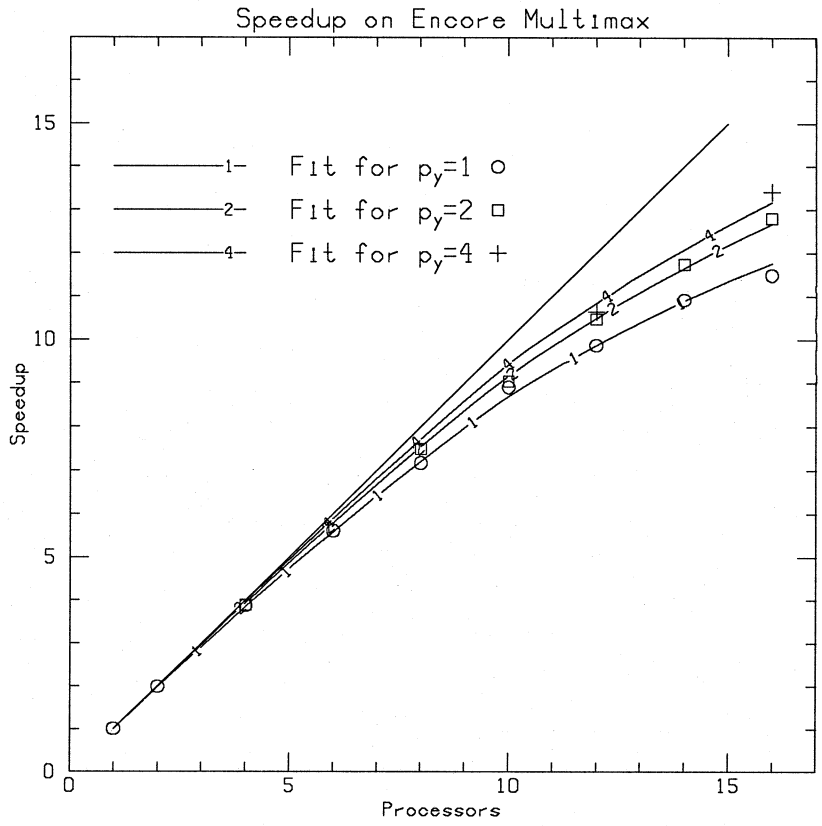

Figure 4: Speedup figures for the Encore Multimax. The fit uses $T = 589.5/p + 79.5/\min(p, 10) + 8.22/p_y + .29p$. The problem size is $n = 100$. The line labels are the values of p_y used for that curve. The solid line represents a perfect speedup. Note the sharper deviation from perfect speedup at roughly 10 processors.

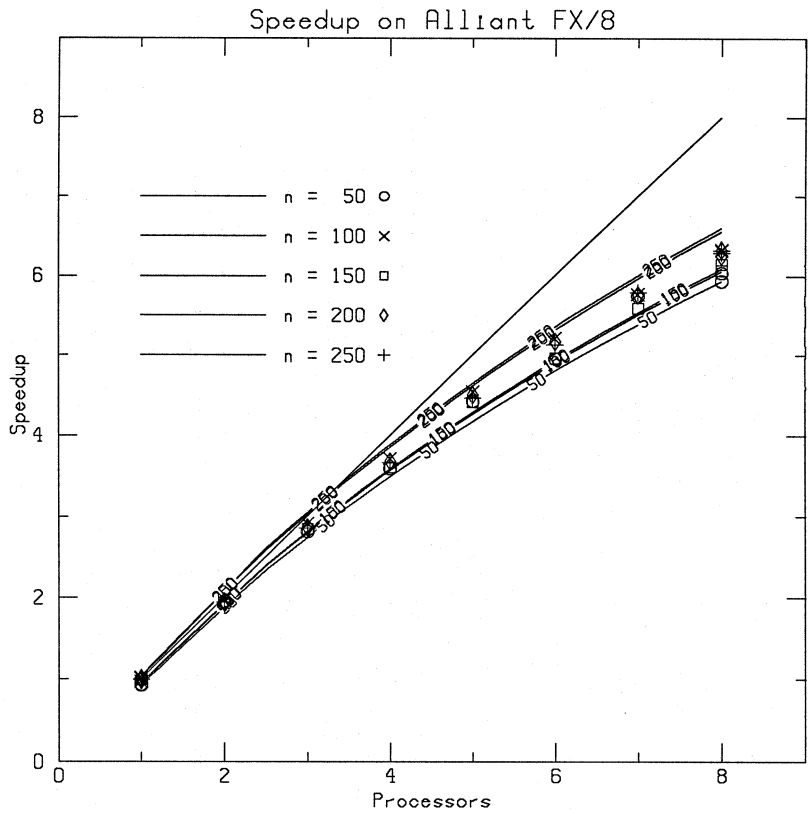

Figure 5: Results for the Alliant FX/8. The fit uses $T = 0.00645n^2/p + 0.00085n^2/\min(p, 2.45)$. The speedup is almost independent of the problem size. The solid line represents a perfect speedup; note the deviation from this line by both the experimental results and the theory at around three processors.

a memory bottleneck as cache and main memories are updated to maintain consistency. Another part is due to some of the program running in serial (using only a single processor); this is due in part to the Fortran compiler on the Alliant, in a deliberate trade-off between ease of use and maximum parallelism.

Memory bottleneck will occur in any shared memory parallel computer. Since memory bottleneck will cause a falloff in the speedup curve below linear speedup, it is a very significant limitation on shared memory parallel computers for massive parallelism.

5.4 Summary of experimental results

The agreement between the experiments and the complexity estimates is very good. The complexity estimates capture the important contributions to the computation time. Thus we can use the complexity results to predict performance for differing values of the parameters, including the number of processors.

In the case of a distributed memory parallel computer, such as the Intel Hypercube, the time estimate for the computation has the form shown in equation 12. For simplicity, we will consider the case $p_x = p_y = \sqrt{p}$. Then the estimate can be simplified to

$$T = F_1 \frac{n^2}{p} + F_2 \frac{n}{\sqrt{p}} + 8(\alpha + r) \log p + 12 \left(\alpha + r \frac{n}{\sqrt{p}} \right).$$

Because of the $\log p$ term there is a value of p beyond which adding additional processors actually slows the computation down. The maximum speed up is achieved when $dT/dp = 0$, and is at

$$\sqrt{p} = n \frac{\frac{1}{2}F_2 + 6r + \sqrt{(\frac{1}{2}F_2 + 6r)^2 + 32F_1(\alpha + r)}}{16(\alpha + r)}.$$

From this we can see that the the optimal number of processors depends on n. It can be shown that, as $n \to \infty$ and with this optimal p, the speedup tends to

$$\frac{An^2}{\log n}.$$

Thus, the speedup for the distributed memory parallel computer (e.g., Intel Hypercube) increases without bound as the problem size and number of processors grow.

For a shared memory parallel computer, such as the Alliant FX/8 or Encore Multimax, the time estimate for the computation has the form shown in equation 14. (Note that the shape of the Encore and Alliant results are very similar; the terms corresponding to equation 14 in both cases are the dominant terms.) The predicted speedup is independent of the problem size, as we observed experimentally. The parameters determined by our experiments give a maximum possible speedup of roughly 20 for the Alliant and roughly 80 for the Encore. Of course, the number of processors available on both the Alliant and the Encore were picked to match the available memory bandwidths. Other technologies

Machine	f_1	f_2	c_1	c_2	c_3
(a)	0.066	0.014	0.861	0.868	0.0302
(b)	0.00066	0.00014	0.861	0.868	0.0302
(c)	0.0132	0.0028	0.215	0.217	0.0302
(d)	0.0033	0.0007	0.215	0.217	0.0302
(e)	0.00066	0.00014	0.215	0.217	0.0302
(f)	0.00132	0.00028	0.0172	0.0176	0.00755

Table 2: Parameters for some possible distributed memory parallel processors.

and designs would allow larger limits. The limits themselves are, however, intrinsic in any shared resource design. Thus, because of the bottlenecks in the shared memory, even with an infinite number of processors, the maximum speedup of a shared memory parallel computer is limited.

Comparing these results, we see that the distributed memory computers offer more potential speedup than shared memory computers as the number of available processors increases.

6 Predictions and Observations

As an example of the predictive use of our complexity estimates, we can estimate the performance of a distributed memory parallel processor, starting with our results for the Intel Hypercube. The relevant formula is equation 17, written as

$$T = f_1 \frac{n^2}{p} + f_2 \frac{n}{p_y} + c_1 \log p + c_3(n_x + n_y) + c_2.$$

Terms f_1 and f_2 depend only on the floating point speed. c_1 and c_2 depend on the communication startup times, and c_3 on the communication transfer rate (c_1 also depends on floating point speed and transfer rate; however, startup times will usually dominate c_1).

Table 2 shows possible parameters for some distributed memory hypercubes. All of these estimates are drawn from machines which are either available today or planned for the near future. Machine (a) is the Intel hypercube used in our tests. Machine (b) represents a machine with 4 megaflop vector boards. Machine (c) is a more balanced machine, with both floating point and communication startup times reduced through a faster processor (such as the Intel 80386). Machine (d) is (c), but with four times as fast floating point. This might come from using a non-vector/pipelined fast floating point chip. Machine (e) is (c), using the vector boards from (b). Finally, Machine (f) uses high speed combined communications and computation hardware, such as in the INMOS Transputer chips.

		a		b		c		d		e		f	
p	n	T	S	T	S	T	S	T	S	T	S	T	S
32	250	137	30	9.1	4.5	30	28	10	20	5.3	7.8	3.4	25
64	250	73	57	8.6	4.8	16	50	6.6	31	4	10	1.9	44
128	250	41	101	8.6	4.8	9.6	86	4.7	44	3.4	12	1.1	74
32	500	527	31	16	11	110	30	32	25	12	14	12	28
64	500	268	61	12	13	57	58	18	45	7.9	21	6.2	53
128	500	139	119	11	15	30	109	11	76	5.7	30	3.4	97
256	500	75	221	10	16	17	196	7	117	4.5	37	1.9	171

Table 3: Predicted times and speedups for various distributed memory parallel processors. Times are in the column T and speedups are in the column S.

In Table 3 we show the predicted performance of a number of possible distributed memory parallel computers described in Table 2.

The relatively poor showing of machines (b) and (e) is due to the large communication terms, and suggests changes to both the algorithm (to reduce data exchanges) and the hardware (to make communication speeds comparable to floating point speeds). The figures for Machine (f) show that if communication speeds (particularly startups, the "α" term) are increased, very respectable speedups can be obtained, even for large number of processors. Similar calculations can be performed for proposed shared memory machines.

It is also interesting to compare the amount of programming work needed to run our experiments. In no case was much effort required to modify the original serial code to run on any of the parallel machines. The easiest to run was the Alliant, since the Fortran compiler on the Alliant automatically makes reasonably good use of the parallelism. Neither the Encore nor the Intel machines have such compilers, and each required a small amount of special coding. In addition, the Alliant required some special coding to improve the performance. The overall efforts were similar.

7 Conclusions

We have shown that significant parallelism is available at least for explicit methods for CFD. Our complexity estimates, validated by our experiments, show speedups limited only by the problem size for distributed memory architectures. For shared memory architectures, bottlenecks at the shared resource constrain the available speedups.

In actual practice, our results show that the communications costs associated with distributed memory parallel computers can seriously degrade the performance of these systems. Commercial shared memory computers such as the Alliant FX/8 have achieved a better balance of system components, at the (recognized) cost of limited parallelism. Since only massive parallelism has the promise of orders of magnitude increases in speed,

we hope that our results and ones like them will stimulate both algorithm and hardware designers to overcome these practical problems.

In this paper we have considered shared memory and distributed memory parallel computers. In reality, actual computers are rarely so easily categorized. For example, a computer presenting the programmer with a shared memory model but implemented in hardware with several levels of distributed memory (a hierarchical memory) offers an intermediate form of parallel computer. The significance of our results is that, for significant parallelism, use of any shared resource that presents a bottleneck must be extremely limited. The success of the distributed memory code shows that explicit CFD codes can be easily written in such a form which avoids these bottlenecks.

Acknowledgements

The authors wish to thank James D. Wilson of AFOSR, Michael J. Werle and Joseph R. Caspar of UTRC, and Martin H. Schultz of Yale University, for their comments and suggestions.

References

[Andr 85] André, F., D. Herman, and J.-P. Verjus, Synchronization of Parallel Programs, The MIT Press, Cambridge, 1985.

[Eise 87] Eisenstat, S., private communication, 1987.

[Jame 83] Jameson, A.: Transonic Flow Calculations. Department of Mechanical and Aerospace Engineering, MAE Report 1651, Princeton University, July 1983.

[Jame 87] Jameson, A.: Successes and Challenges in Computational Aerodynamics. AIAA paper 87–1184, presented at AIAA 8th Computational Fluid Dynamics Conference, Honolulu, Hawaii, June 9–11, 1987.

[John 87] Johnson, G. M.: Parallel Processing in Fluid Dynamics. ISC Technical Report 87003, Institute for Scientific Computing, Fort Collins, Colorado, paper presented at ASME Fluids Engineering Conference, Cincinnati, Ohio, June 14–18, 1987.

[Jord 87] Jordan, H. F.: Interpreting Parallel Processor Performance Measurements. Siam Journal on Scientific and Statistical Computing, Volume 8, No. 2, pages s220–s226.

[MacC 69] MacCormack, R. W.: The Effect of Viscosity in Hypervelocity Impact Cratering. AIAA Paper No. 69–354, presented at AIAA Hypervelocity Impact Conference, Cincinnati, Ohio, April 30–May 2, 1969.

[Rai 87] Rai, M. M.: Unsteady Three-Dimensional Navier-Stokes Simulations of Turbine Rotor-Stator Interaction. AIAA paper 87–2058, presented at 23rd. Joint AIAA/ASME Propulsion Conference, San Diego, California, June 29–July 2, 1987.

[Saad 85] Saad, Y. and M. H. Schultz: Topological Properties of Hypercubes. Yale University Department of Computer Science Research Report Yale/DCS/RR–389, June, 1985.

[Wigt 85] Wigton, L. B., N. J. Yu and D. P. Young: GMRES Acceleration of Computational Fluid Dynamics Codes: AIAA paper 85–1494, presented at AIAA 7th Computational Fluid Dynamics Conference, Cincinnati, Ohio, July 15–17, 1985.