MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1485 November, 1994

Parallel simulation of subsonic fluid dynamics on a cluster of workstations

Panayotis A. Skordos
pas@ai.mit.edu

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.

Abstract

An effective approach of simulating fluid dynamics on a cluster of non-dedicated workstations is presented. The approach uses local interaction algorithms, small communication capacity, and automatic migration of parallel processes from busy hosts to free hosts. The approach is well-suited for simulating subsonic flow problems which involve both hydrodynamics and acoustic waves; for example, the flow of air inside wind musical instruments. Typical simulations achieve 80% parallel efficiency (speedup/processors) using 20 HP-Apollo workstations. Detailed measurements of the parallel efficiency of 2D and 3D simulations are presented, and a theoretical model of efficiency is developed which fits closely the measurements. Two numerical methods of fluid dynamics are tested: explicit finite differences, and the lattice Boltzmann method.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097 and by the National Science Foundation under grant number 9001651-MIP.

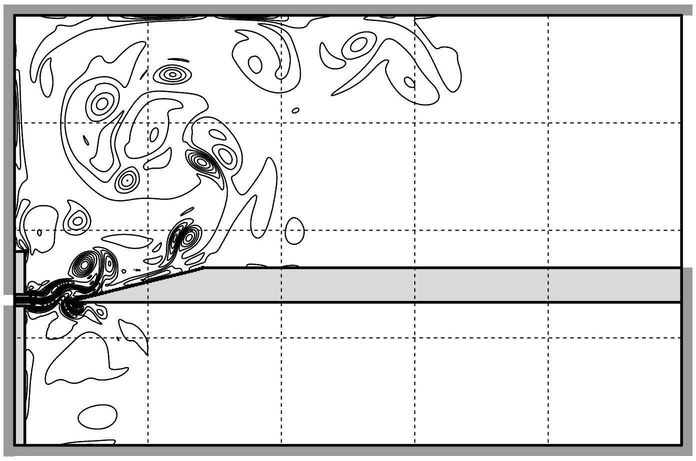

Figure 1: Simulation of a flue pipe using 20 workstations in a (5×4) decomposition.

1 Introduction

An effective approach of simulating fluid dynamics on a cluster of non-dedicated workstations is presented. Concurrency is achieved by decomposing the simulated area into rectangular subregions, and by assigning the subregions to parallel subprocesses. The use of local interaction methods, namely explicit numerical methods, leads to small communication requirements. The parallel subprocesses automatically migrate from busy hosts to free hosts in order to exploit the unused cycles of non-dedicated workstations, and to avoid disturbing the regular users of the workstations. The system is straightforwardly implemented on top of UNIX and TCP/IP communication routines.

Typical simulations achieve 80% parallel efficiency (speedup/processors) using 20 HP-Apollo workstations in a cluster where there are 25 non-dedicated workstations total. Detailed measurements of efficiency in simulating two and three-dimensional flows are presented, and a theoretical model of efficiency is developed which fits closely the measurements. Two numerical methods of fluid dynamics are tested: finite differences and the lattice Boltzmann method. Further, it is shown that the shared-bus Ethernet network is adequate for two-dimensional simulations of fluid dynamics, but limited for three-dimensional ones. It is expected that new technologies in the near future such as Ethernet switches, FDDI and ATM networks will make practical three-dimensional simulations of fluid dynamics on a cluster of workstations.

The present approach is well-suited for simulating subsonic flow problems which involve both hydrodynamics and acoustic waves; for example, the flow of air inside wind musical instruments. This is because such problems favor the use of explicit numerical methods versus implicit ones, as explained below and in section 6. The use of explicit methods is desirable for parallel computing on a cluster of workstations because explicit methods have small communication requirements. Thus, there is a good match between the nature of the problem, the use of explicit methods, and the parallel system.

The choice between explicit and implicit numerical methods is a recurring theme in scientific computing. Explicit methods are local, ideally scalable, and require small integration time steps in order to remain numerically stable. By contrast, implicit methods are challenging to parallelize, have large communication requirements, but they can use much larger integration time steps than explicit methods. Because of these differences between explicit and implicit methods, the decision which method to use depends on the available computer system, and on the requirements of the problem on the integration time step. For instance, the simulation of subsonic flow requires small integration time steps in order to follow the fast-moving acoustic waves. Thus, subsonic flow is a good problem for explicit methods.

1.1 Comparison with other work

The suitability of local interaction algorithms for parallel computing on a cluster of workstations has been demonstrated in previous works, such as [1], [2], and elsewhere. Cap&Strumpen [1] present the PARFORM system and simulate the unsteady heat equation using explicit finite differences. Chase&et al. [2] present the AMBER parallel system, and solve Laplace's equation using Successive Over-Relaxation. The present work emphasizes and clarifies further the importance of local interaction methods for parallel systems with small communication capacity. Furthermore, a real problem of science and engineering is solved using the present approach. The problem is the simulation of subsonic flow with acoustic waves inside wind musical instruments.

In the fluid dynamics community, very little attention has been given to simulations of subsonic flow with acoustic waves. The reason is that such simulations are very compute-intensive, and can be performed only when parallel systems such as the one described here are available. Further, the fluid dynamics community has generally shunned the use of explicit methods until recently because explicit methods require small integration time steps to remain numerically stable. With the increasing availability of parallel systems, explicit methods are slowly attracting more attention. The present work clearly reveals the power of explicit methods in one particular example, and should motivate further work in this direction.

Regarding the experimental measurements of parallel efficiency which are presented in section 7, they are more detailed than in any other reference known to the author, especially for the case of a shared-bus Ethernet network. The model of parallel efficiency which is discussed in section 8 is based on ideas which have been discussed previously, for example in Fox et al. [3] and elsewhere. Here, the model is derived in a clear and direct way, and moreover the predictions of the model are compared against experimental measurements of parallel efficiency.

Regarding the problem of using non-dedicated workstations, the present approach solves the problem by employing automatic process migration from busy hosts to free hosts. An alternative approach that has been used elsewhere is the dynamic allocation of processor workload. In the present context, dynamic allocation means to enlarge and to shrink the subregions which are assigned to each workstation depending on the CPU load of the workstation (Cap&Strumpen [1]). Although this approach is important in various applications (Blumofe&Park [4]), it seems unnecessary for simulating fluid flow problems with static geometry. For such problems, it may be simpler and more effective to use fixed size subregions per processor, and to use automatic migration of processes from busy hosts to free hosts. The latter approach has worked very well in the parallel simulations presented here.

Regarding the design of parallel simulation systems, the present work aims for simplicity. In particular, the special constraints of local interaction problems and static decomposition have guided the design of the parallel system. The automatic migration of processes has been implemented in a straightforward manner because the system is very simple. The availability of a homogeneous cluster of workstations, and a common file system have also simplified the implementation, which is based on UNIX and TCP/IP communication routines. The approach presented here works well for spatially-organized computations which employ a static decomposition and local interaction algorithms.

The approach presented here does not deal with issues such as high-level distributed programming, parallel languages, inhomogeneous clusters, and distributed computing of general problems. Efforts along these directions are the PVM system (Sunderam [5]), the Linda system (Carriero [6]), the packages of (Kohn&Baden [7]) and (Chesshire&Naik [8]) that facilitate a parallel decomposition, the Orca language for distributed computing (Bal&et al. [9]), etc.

1.2 Outline

Section 2 presents some examples of parallel simulations which demonstrate the power of the present approach, and also help to motivate the subsequent sections. Section 3 reviews parallel computing and local interaction problems in general. Sections 4 and 5 describe the implementation of the parallel simulation system, including the automatic migration of processes from busy hosts to free hosts. Section 6 explains the parallelization of numerical methods for fluid dynamics. Finally, sections 7 and 8 present experimental measurements of the performance of the parallel system, and develop a theoretical model of parallel efficiency for local interaction problems which fits well the measured efficiency. Most ideas are discussed as generally as possible within the context of local interaction problems, and the specifics of fluid dynamics are limited to section 2 and section 6.

2 Examples of flow simulations

The parallel system has been successfully applied to simulate the flow of air inside flue pipes of wind musical instruments such as the organ, the recorder, and the flute. This is a phenomenon that involves the interaction between hydrodynamic flow and acoustic waves: When a jet of air impinges a sharp obstacle in the vicinity of a resonant cavity, the jet begins to oscillate strongly, and it produces audible musical tones. The jet oscillations are reenforced by a nonlinear feedback from the acoustic waves to the jet. Similar phenomena occur in human whistling and in voicing of fricative consonants (Shadle [10]). Although sound-producing jets have been studied for more than a hundred years, they remain the subject of active research (Verge94 [11, 12], Hirschberg [13]) because they are very complex.

Using our distributed system we can simulate jets of air inside flue pipes using uniform orthogonal grids

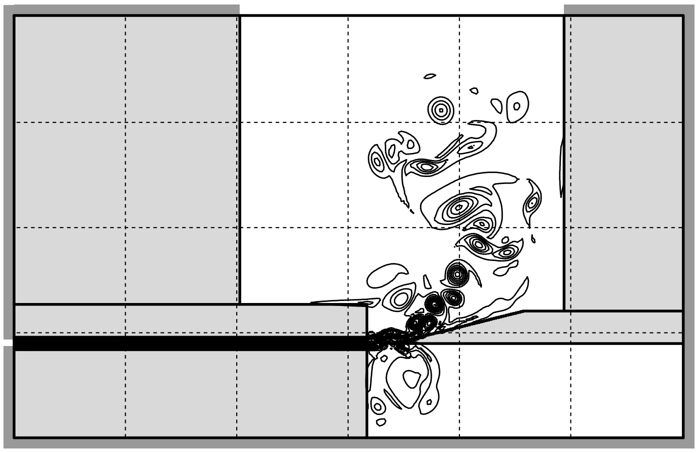

Figure 2: Simulation of a flue pipe using 15 workstations in a (6×4) decomposition with 9 subregions inactive.

as large as 1200×1200 in two dimensions (1.5 million nodes). We typically employ smaller grids, however, such as 800×500 (0.38 million nodes) in order to reduce the computing time. For example, if we divide a 800×500 grid into twenty subregions and assign each subregion to a different HP9000/700 workstation, we can compute 70,000 integration steps in 12 hours of run time. This produces about 12 milliseconds of simulated time, which is long enough to observe the initial response of a flue pipe with a jet of air that oscillates at 1000 cycles per second.

Figure 1 shows a snapshot of a 800×500 simulation of a flue pipe by plotting equi-vorticity contours (the curl of fluid velocity). The decomposition of the two-dimensional space $(5 \times 4) = 20$ is shown as dashed lines superimposed on top of the physical region. The gray areas are walls, and the dark-gray areas are walls that enclose the simulated region and demarcate the inlet and the outlet. The jet of air enters from an opening on the left wall, impinges the sharp edge in front of it, and it eventually exits from the simulation through the opening on the right part of the picture. The resonant pipe is located at the bottom part of the picture.

Figure 2 shows a snapshot of another simulation that uses a slightly different geometry than figure 1. In particular, figure 2 includes a long channel through which the jet of air must pass before impinging the sharp edge. Also, the outlet of the simulation is located at the top of the picture as opposed to the right. This is convenient because the air tends to move upwards after impinging the sharp edge. Overall, figure 2 is a more realistic model of flue pipes than figure 1.

From a computational point of view the geometry of figure 2 is interesting because there are subregions that are entirely gray, i.e. they are entirely solid walls. Consequently, we do not need to assign these subregions to any workstation. Thus, although the decomposition is $(6 \times 4) = 24$, we only employ 15 workstations for this problem. In terms of the number of grid nodes, the full rectangular grid is 1107×700 or 0.7 million nodes, but we only simulate 15/24 of the total nodes or 0.48 million nodes. This example shows that an appropriate decomposition of the problem can reduce the computational effort in some cases, as well as provide opportunities for parallelism. More sophisticated decompositions can be even more economical than ours; however, we prefer to use uniform decompositions and identical-shaped subregions in our current implementation for the sake of simplicity.

We have performed all of the above simulations using the lattice Boltzmann numerical method. We will describe further this method and other issues of fluid dynamics in section 6. Next, we review the basics of local interaction problems, and we describe the implementation of our distributed system. These issues are important for understanding in detail how our system works and why it works well.

3 Local interaction computations

We define a local interaction computation as a set of "parallel nodes" that can be positioned in space so that the nodes interact only with neighboring nodes. For example, figure 3 shows a two-dimensional space of parallel nodes connected with solid lines which represent the local interactions. In this example, the interactions extend to a distance of one neighbor, and have the shape of a star stencil, but other patterns of local interactions are also possible. Figure 4 shows two typical interactions which extend to a distance of one neighbor, a star stencil and a full stencil.

The parallel nodes of a local interaction problem are the finest grain of parallelism that is available in the problem; namely, they are the finest decomposition of the problem into units that can evolve in parallel after communication of information with their neighbors. In practice, the parallel nodes are usually grouped into subregions of nodes, as shown in figure 3 by the dashed lines. Each subregion is assigned to a different proces-

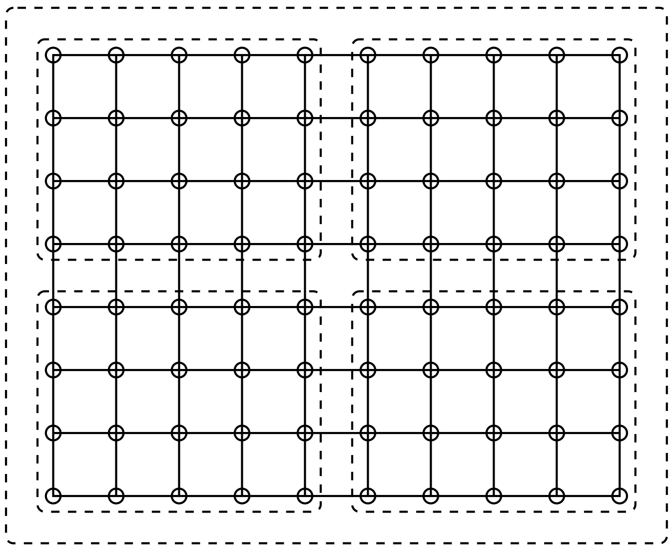

Figure 3: A problem of local interactions in two dimensions, and its decomposition (2×2) into four subregions.

sor, and the problem is solved in parallel by executing the following sequence of steps repeatedly,

- Calculate the new state of the interior of the subregion using the previous history of the interior as well as the current boundary information from the neighboring subregions.
- Communicate boundary information with the neighboring subregions in order to prepare for the next local calculation.

The boundary which is communicated between neighboring subregions is the outer surface of the subregions. Section 4.2 describes a good way of organizing this communication.

Local interaction problems are ideal for parallel computing because the communication is local, and also because the amount of communication relative to computation can be controlled by varying the decomposition. In particular, when each subregion is as small as one node (one processor per node), there is maximum parallelism, and a lot of communication relative to the computation of each processor. As the size of each subregion increases (which is called "coarse-graining"), both the parallelism and the the amount of communication relative to computation decrease. This is because only the surface of a subregion communicates with other subregions. Eventually, when one subregion includes all the nodes in the problem, there is no parallelism and no need for communication anymore. Somewhere between these extremes, we often find a good match between the size of the subregion (the "parallel grain size") and the communication capabilities of the computing system. This is the reason why local interaction problems are very flexible and highly desirable for parallel computing.

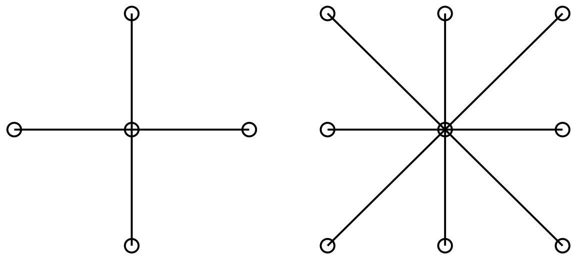

Figure 4: A star stencil and a full stencil represent two typical nearest neighbor local interactions.

4 The distributed system

The design of our system follows the basic ideas of local interaction parallel computing that we discussed above. In this section, we describe an implementation which is based on UNIX and TCP/IP communication routines. Our implementation also exploits the common file system of the workstations.

4.1 The main modules

For the sake of programming modularity, we organize our system into the following four modules:

- The initialization program produces the initial state of the problem to be solved as if there was only one workstation.
- The decomposition program decomposes the initial state into subregions, generates local states for each subregion, and saves them in separate files, called "dump files". These files contain all the information that is needed by a workstation to participate in a distributed computation.
- The job-submit program finds free workstations in the cluster, and begins a parallel subprocess on each workstation. It provides each process with a dump file that specifies one subregion of the problem. The processes execute the same program on different data.

- The monitoring program checks every few minutes whether the parallel processes are progressing correctly. If an unrecoverable error occurs, the distributed simulation is stopped, and a new simulation is started from the last state which is saved automatically every $10-20$ minutes. If a workstation becomes too busy, automatic migration of the affected process takes place, as we explain in section 5.

All of the above programs (initialization, decomposition, submit, and monitoring) are performed by one designated workstation in the cluster. Although it is possible to perform the initialization and the decomposition in a distributed fashion in principle, we have chosen a serial approach for simplicity.

Regarding the selection of free workstations, our strategy is to separate all the workstations into two groups: workstations with active users, and workstations with idle users (meaning more than 20 minutes idle time). An idle-user does not necessarily imply an idle workstation because background jobs may be running; however, an idle-user is preferred to an active user. Thus, we first examine the idle-user workstations to see if the fifteen-minute average of the CPU load is below a pre-set value, in which case the workstation is selected. For example, the load must be less than 0.6 where 1.0 means that a full-time process is running on the workstation. After examining the idle-user workstations, we examine the active-user workstations, and we continue the search as long as we need more workstations.

In addition to the above programs (initialization, decomposition, submit, and monitoring), there is also the parallel program which is executed by all the workstations. The parallel program consists of two steps: "compute locally", and "communicate with neighbors". Below we discuss issues relating to communication.

4.2 Communication

The communication between parallel processes synchronizes the processes in an indirect fashion because it encourages the processes to begin each computational cycle together with their neighbors as soon as they receive data from their neighbors. Thus, there is a local near-synchronization which also encourages a global near-synchronization. However, neither local nor global synchronization is guaranteed, and in special circumstances the parallel processes can be several integration time steps apart. This is important when a process migrates from a busy host to a free host, as we explain in section 5 (also see the appendix).

We organize the communication of data between processes by using a well-known programming technique which is called "padding" or "ghost cells" (Fox [3], Camp [14]). Specifically, we pad each subregion with one or more layers of extra nodes on the outside. We use one layer of nodes if the local interaction extends to a distance of one neighbor, and we use more layers if the local interaction extends further. Once we copy the data from one subregion onto the padded area of a neighboring subregion, the boundary values are available locally during the current cycle of the computation. This is a good way to organize the communication of boundary values between neighboring subregions.

In addition, padding leads to programming modularity in the sense that the computation does not need to know anything about the communication of the boundary. As long as we compute within the interior of each subregion, the computation can proceed as if there was no communication at all. Because of this separation between computation and communication, we can develop a parallel program as a straightforward extension of a serial program. In our case, we have developed a fluid dynamics code which can produce either a parallel program or a serial program depending on the settings of a few C-compiler directives. The main differences between the parallel and the serial programs are the padded areas, and a subroutine that communicates the padded areas between processes.

We have implemented a subroutine that communicates the padded areas between processes using "sockets" and the TCP/IP protocol. A socket is an abstraction in the UNIX operating system that provides system calls to send and receive data between UNIX processes on different workstations. A number of different protocols (types of behavior) are available with sockets, and TCP/IP is the simplest one. This is because the TCP/IP protocol guarantees delivery of any messages sent between two processes. Accordingly, the TCP/IP protocol behaves as if there are two first-in-first-out channels for writing data in each direction between two processes. Also, once a TCP/IP channel is opened at startup, it remains open throughout the computation except during migration when it must be re-opened, as we shall see later.

Opening the TCP/IP channel involves a simple handshaking, "I am listening at this port number. I want to talk to you at this port number? Okay, the channel is open." The port numbers are needed to identify uniquely the sender and the recipient of a message so that messages do not get mixed up between different UNIX processes. Further, the port numbers must be known in advance before the TCP/IP channel is opened. Thus, each process must first allocate its port numbers for listening to its neighbors, and then write the port numbers into a shared file. The neighbors must read the shared file before they can connect using TCP/IP.

5 Transparency to other users

Having described the basic operation of our distributed system, we now discuss the issues that arise when sharing the workstations with other users. Specifically, there are two issues to consider: sharing the CPU cycles of each workstation, and sharing the local area network and the file server. First, we describe the sharing of CPU cycles and the automatic migration of processes from busy hosts to free hosts.

5.1 Automatic migration of processes

We distinguish the utilization of a workstation into three basic categories:

- (i) The workstation is idle.
- (ii) The workstation is running an interactive program that requires fast CPU response and few CPU cycles.
- (iii) The workstation is running another full-time process in addition to a parallel subprocess.

In the first two cases, it is appropriate to time-share the workstation with another user. Furthermore, it is possible to make the distributed computation transparent to the regular user of the workstation by assigning a low runtime priority to the parallel subprocesses (UNIX command "nice"). Because the regular user's tasks run at normal priority, they receive the full attention of the processor immediately, and there is no loss of interactiveness. After the user's tasks are serviced, there are enough CPU cycles left for the distributed computation.

In the third case, when a workstation is running another full-time process in addition to a parallel subprocess, the parallel subprocess must migrate to a new host that is free. This is because the parallel process interferes with the regular user, and further, the whole distributed computation slows down because of the busy workstation. Clearly, such a situation must be avoided.

Our distributed system detects the need for migration using the monitoring program that we mentioned in the previous section. The monitoring program checks the CPU load of every workstation via the UNIX command "uptime", and signals a request for migration if the five-minute-average load exceeds a pre-set value, typically 1.5. The intent is to migrate only if a second full-time process is running on the same host, and to avoid migrating too often. In our system there is typically one migration every 45 minutes for a distributed computation that uses 20 workstations from a pool of 25 workstations. Also, each migration lasts about 30 seconds. Thus, the cost of migration is insignificant because the migrations do not happen too often.

During a migration, a precise sequence of events takes place in order for the migration to complete successfully,

- The affected process A receives a signal to migrate.
- All the processes get synchronized.
- Process A saves its state into a dump file, and stops running.
- Process A is restarted on a free host, and the distributed computation continues.

Signals for migration are sent through an interrupt mechanism, "kill -USR2" (see UNIX manual). In this way, both the regular user of a workstation and our monitoring program can request a parallel subprocess to migrate at any time.

The reason for synchronizing all the processes prior to migration, is to simplify the restarting of the processes after the migration has completed. In addition, the synchronization allows more than one process to migrate at the same time if it is desired. In our system, we use a synchronization scheme which instructs all the processes to continue running until a chosen synchronization time step, and then to pause for the migration to take place. The details of the synchronization scheme are described in the appendix.

When all the processes reach the synchronization time step, the processes that need to migrate save their state and exit, while they notify the monitoring program to select free workstations for them. The other parallel processes suspend execution and close their TCP/IP communication channels. When the monitoring program finds free hosts for all the migrating processes, it sends a *CONT* signal to the waiting processes. In response, all the processes re-open their communication channels, and the distributed computation continues normally.

Overall, the migration mechanism is designed to be as simple as possible. In fact, it is equivalent to stopping the computation, saving the entire state on disk, and then restarting; except, we only save the state of the migrating process on disk. In contrast to this simple migration mechanism, we note that process migration in a general computing environment such as a distributed operating system [15] can be a challenging task. In our case the task has been simplified because we can design our processes appropriately to accommodate migration easily.

5.2 Sharing the network and file server

A related issue to sharing the workstations with other users, is the sharing of the network and the file server. A distributed program must be carefully designed to make sure that the system does not monopolize the network and the file server. Abuse of shared resources is very common in today's UNIX operating system because there are no direct mechanisms for controlling or limiting the use of shared resources. Thus, a program such as FTP (file transfer) is free to send many megabytes of data through the network, and to monopolize the network, so that the network appears "frozen" to other users. A distributed program can monopolize the network in a similar way, if it is not designed carefully.

Our distributed system does not monopolize the network because it includes a time delay between successive send-operations, during which the parallel processes are calculating locally. Moreover, the time delay increases with the network traffic because the parallel processes must wait to receive data before they can start the next integration step. Thus, there is an automatic feedback mechanism that slows down the distributed computation, and allows other users to access the network at the same time.

Another situation to consider is when the parallel processes are writing data to the common file system. Specifically, when all the parallel processes save their state on disk at approximately the same time (a couple of megabytes per process), it is very easy to saturate both the network and the file server. In order to avoid this situation, we impose the constraint that the parallel processes must save their state one after the other in an orderly fashion, allowing sufficient time gaps between, so that other programs can use the network and the file system. Thus, a saving operation that would take 30 seconds and monopolize the shared resources, now takes $60 - 90$ seconds but leaves free time slots for other programs to access the shared resources at the same time. Overall, a careful design has made our distributed system mostly transparent to the regular users of the workstations.

6 Fluid dynamics

Having described the overall design of our distributed system, we now turn our attention to the specifics of fluid dynamics. First, we review the equations of fluid dynamics, and then we explain why local interaction methods are appropriate for simulating subsonic flow. Finally, we outline the numerical methods that we use in our system.

The evolution of a flow is described using a set of partial differential equations, known as the Navier Stokes equations (Tritton [16], Batchelor [17], Lamb [18]). These equations can take different forms depending on the specific problem at hand. In our case, the Navier Stokes equations involve three fluid variables ρ, V_x, V_y: the fluid density, and the components of the fluid velocity in the x,y directions respectively. The variables ρ, V_x, V_y are functions of space and time, and the Navier Stokes equations express the rates of change of these variables, as follows,

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho V_x)}{\partial x} + \frac{\partial(\rho V_y)}{\partial y} = 0 \tag{1}$$

$$\frac{\partial V_x}{\partial t} + V_x \frac{\partial V_x}{\partial x} + V_y \frac{\partial V_x}{\partial y} = -\frac{c_s^2}{\rho} \frac{\partial \rho}{\partial x} + \nu \nabla^2 V_x \tag{2}$$

$$\frac{\partial V_y}{\partial t} + V_x \frac{\partial V_y}{\partial x} + V_y \frac{\partial V_y}{\partial y} = -\frac{c_s^2}{\rho} \frac{\partial \rho}{\partial y} + \nu \nabla^2 V_y \tag{3}$$

In the above equations, the symbol ∇^2 is the Laplacian operator $\partial^2/\partial x^2 + \partial^2/\partial y^2$, and the coefficients ν and c_s are constants. ν is the kinematic viscosity of the fluid (a kind of friction), and c_s is the speed of sound. In the case of three-dimensional flow problems, there is another equation for the V_z the component of fluid velocity in the z-direction. Details can be found in any textbook of fluid mechanics.

A flow is simulated by solving the Navier Stokes equations numerically. In particular, a grid of fluid nodes is introduced, which looks very much like the grid of nodes in figure 3. The fluid nodes are discrete locations where the fluid variables density and velocity are calculated at discrete times. A numerical method is used to calculate the future values of density and velocity at each fluid node using the present and the past values of density and velocity at this node, at neighboring nodes, and possibly at distant nodes as well.

A numerical method that employs only neighboring nodes to calculate the future solution, is called an explicit method (or local interaction method), and is ideal for parallel computing. Such a method is also referred to as a "time-marching" method because the present values of each fluid node and its neighbors produce the future value of this fluid node at time $t + \Delta t$, and so on repeatedly, where Δt is the integration time step. By contrast, a numerical method that employs distant nodes to calculate the future solution, is called an implicit method, and is difficult to parallelize. This is because an implicit method computes the solution using a large matrix equation that couples together distant fluid nodes, and leads to complex communication between distant nodes.

There are advantages to both explicit and implicit methods. The obvious advantage of explicit methods is the ease of parallelization. Another issue to consider is that an explicit integration step is much less costly than an implicit integration step. A disadvantage of explicit methods is that they become numerically unstable at large time steps Δt. By contrast, implicit methods can often use much larger integration time steps Δt than explicit methods (Peyret&Taylor [19]). Thus, implicit methods can often compute a solution using fewer time steps than explicit methods. In a practical situation, one has to consider all of the above issues to decide whether implicit or explicit methods are more suitable. Namely, one has to consider the relative cost of an implicit step versus an explicit step, the availability of parallel computing, and the nature of the problem which affects the choice of a small or a large integration time step.

In the case of simulating subsonic flow, the nature of the problem does not allow the use of very large integration time steps Δt. This is because subsonic flow includes two different time-scales – slow-moving hydrodynamic flow and fast-moving acoustic waves – and the latter dominate the choice of integration time step. In particular, the time step Δt must be very small to model accurately the acoustic waves that propagate through the fluid and reflect off obstacles. If Δx is the spacing between neighboring fluid nodes, and c_s is the speed of propagation of acoustic waves, then the product $c_s \Delta t$ must be comparable to Δx in order to have enough resolution to follow the passage and reflection of acoustic waves. Thus, we require the relation,

$$\Delta x \sim c_s \Delta t \tag{4}$$

Because of this requirement, the large time steps of implicit methods are not relevant. Instead, explicit methods are preferable in this case because of their simplicity and ease of parallelization.

In our system, we employ the following two explicit methods: explicit finite differences (Peyret&Taylor [19]), and the recently-developed lattice Boltzmann method (Skordos [20]). The finite difference method is a straightforward discretization of the Navier Stokes equations 1–3. Specifically, the spatial derivatives are discretized using centered differences on a uniform orthogonal grid, and the time derivatives are discretized using forward Euler differences (Peyret&Taylor [19]). For the purpose of improving numerical stability, the density equation 1 is updated using the values of velocity at time $t + \Delta t$. In other words, the velocities values are computed first, and then the density values are computed as a separate step. The precise sequence of computational steps for the finite difference method is as follows,

- Calculate V_x, V_y (inner)
- Communicate: send/recv V_x, V_y (boundary)
- Calculate ρ (inner)
- Communicate: send/recv ρ (boundary)
- Filter ρ, V_x, V_y (inner)

The filter that is included above is crucial for simulating subsonic flow at high Reynolds number (fast moving flow). The fast flow and the interaction between acoustic waves and hydrodynamic flow can lead to slow-growing numerical instabilities. The filter prevents the instabilities by dissipating high spatial frequencies whose wavelength is comparable to the grid mesh size (the distance between neighboring fluid nodes). Our filter is based on a fourth order numerical viscosity (Peyret&Taylor [19]). We use the same filter both for the finite difference method and for the lattice Boltzmann method.

The lattice Boltzmann method is a recently-developed method for simulating subsonic flow, which is competitive with finite differences in terms of numerical accuracy. Because the lattice Boltzmann method is a relaxation type of algorithm, it is somewhat more stable than explicit finite differences. The lattice Boltzmann method uses two kinds of variables to represent the fluid, the traditional fluid variables ρ, V_x, V_y, and another set of variables called populations F_i. During each cycle of the computation, the fluid variables ρ, V_x, V_y are computed from the F_i, and then the ρ, V_x, V_y are used to relax the F_i. Subsequently, the relaxed populations are shifted to the nearest neighbors of each fluid node, and the cycle repeats. The precise sequence of computational steps for the lattice Boltzmann method is as follows,

- Relax F_i (inner)
- Shift F_i (inner)
- Communicate: send/recv F_i (boundary)
- Calculate ρ, V_x, V_y from F_i (inner)
- Filter ρ, V_x, V_y (inner)

More details on the lattice Boltzmann method can be found in Skordos [20].

Regarding the communication of boundary values by the finite difference method (FD) and the lattice Boltzmann method (LB), there are some differences that will become important in the next two sections, when we discuss the performance of our parallel simulation system. The first difference is that FD sends two messages per computational cycle as opposed to LB which sends all the boundary data in one message. This results in slower communication for FD when the messages are small because each message has a significant overhead in a local area network. The second difference is that LB communicates 5 variables (double precision floating-point numbers) per fluid node in three dimensional problems, while FD communicates only 4 variables per fluid node. In two dimensional problems, both methods communicate 3 variables per fluid node.

7 Experimental measurements of performance

The performance of the parallel simulation system is measured when using the finite difference method and the lattice Boltzmann method to simulate a well-known problem in fluid mechanics, Hagen-Poiseuille flow through a rectangular channel (Skordos [20] and Landau&Lifshitz [21, p.51]). The goal of testing two different numerical methods is to examine the performance of the parallel system on two similar, but slightly different parallel algorithms. The question of which numerical method is better for a particular problem is not our main concern here. However, we can say that the two methods produce comparable results for the same resolution in space and time. Moreover, both methods converge quadratically with increased resolution in space to the exact solution of the Hagen-Poiseuille flow problem.

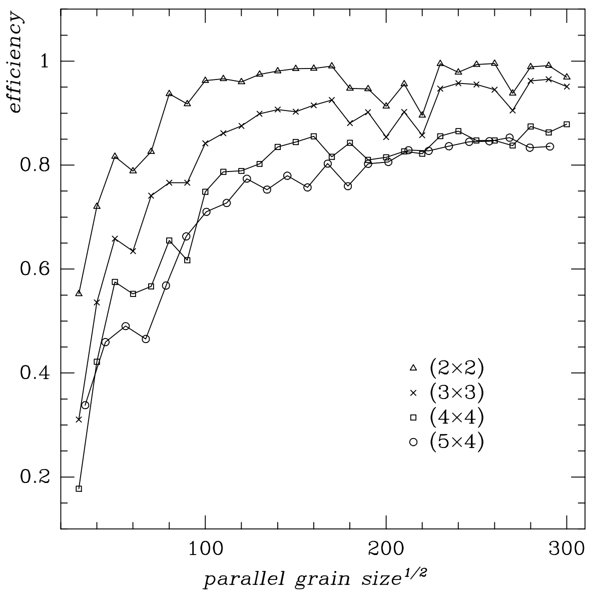

Figure 5: Parallel efficiency in 2D simulations using lattice Boltzmann.

Below we present measurements of the parallel efficiency f, and the speedup S defined as follows,

$$f = \frac{S}{P} = \frac{T_1}{P\,T_p} \tag{5}$$

where T_p is the elapsed time for integrating a problem using P processors, and T_1 is the elapsed time for integrating the same problem using a single processor. We measure the times T_p and T_1 for integrating a problem by averaging over 20 consecutive integration steps, and also by averaging over each processor that participates in the parallel computation. The resulting average is the time interval it takes to perform one integration step. We use the UNIX system call "gettimeofday" to obtain accurate timings. To avoid situations where the Ethernet network is overloaded by a large FTP or something else, we repeat each measurement twice, and select the best performance.

We use twenty-five HP9000/700 workstations that are connected together by a shared-bus Ethernet network. Sixteen of the workstations are 715/50 models, six are 720 models, and three are 710 models. The 715/50 workstations are based on a Risk processor running at 50 MHz, and have an estimated performance of 62 MIPS

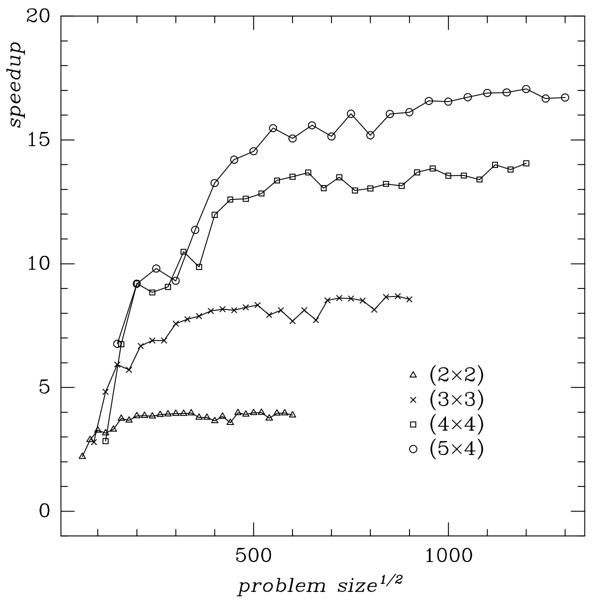

Figure 6: Parallel speedup in 2D simulations using lattice Boltzmann.

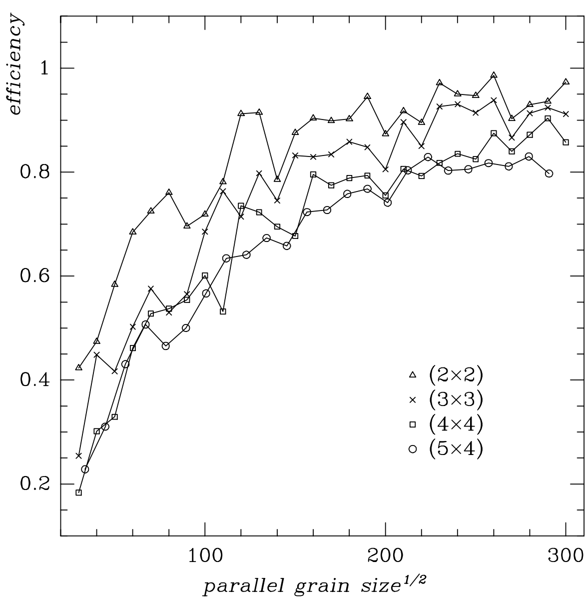

Figure 7: Parallel efficiency in 2D simulations using finite differences.

and 13 MFLOPS, while the 720 and 710 workstations have a slightly lower performance.

For analysis purposes, we define the speed of a workstation as the number of fluid nodes integrated per second, where the number of fluid nodes does not include the padded areas discussed in section 4.2. The table below presents the speed of the workstations for 2D and 3D simulations using the lattice Boltzmann method (LB) and the finite difference method (FD). We have calculated these numbers by averaging over simulations of different size grids that range from 100^2 to 300^2 fluid nodes in 2D, and from 10^3 to 44^3 in 3D. Also, we have normalized the speeds relative to the speed of the 715/50 workstation,

	715/50	710	720
LB 2D	$1.0 \pm .04$	$.84 \pm .02$	$.86 \pm .08$
LB 3D	$.51 \pm .01$	$.40 \pm .01$	$.42 \pm .02$
FD 2D	$1.24 \pm .1$	$1.08 \pm .1$	$1.17 \pm .1$
FD 3D	$1.0 \pm .1$	$.85 \pm .1$	$.94 \pm .1$

The relative speed of 1.0 corresponds to 39132 fluid nodes integrated per second.

In our graphs of parallel speedup and efficiency, we use the the 715/50 workstation to represent the single processor performance. We do not use the performance of the slowest workstation (the 710 model) for normalization purposes because it would over-estimate the performance of our system. In particular, most of the workstations are 715 models, and our strategy is to choose 715 models first before choosing the slightly slower 710 and 720 models. We have tested that the speedup achieved by sixteen workstations, which are all 715 models, does not change if one or two workstations are replaced with 710 models. Thus, it makes sense to normalize our results using the performance of the 715 model.

Figure 5 shows the efficiency as a function of grain size for (2×2), (3×3), (4×4), and (5×4) decompositions (triangles, crosses, squares, circles). On the horizontal axis, we plot the square root of number of nodes N of each subregion. We see that good performance is achieved in two-dimensional simulations when the subregion per processor is larger than 100^2 fluid nodes. In the next section, we present a theoretical model of parallel efficiency that predicts very accurately our experimental results shown in figure 5 and in the other figures also. Figure 6 shows the speedup for the lattice Boltzmann method (LB), and figures 7 and 8 show the efficiency and speedup for the finite difference method (FD).

We notice one difference between the FD and LB efficiency curves: the efficiency decreases more rapidly for FD than LB as the subregion per processor decreases. To understand this difference, we quote a general formula for the parallel efficiency, which is derived in the next section (see equation 12),

$$f \;=\; \left(1 + \frac{T_{com}}{T_{calc}}\right)^{-1} \tag{6}$$

where T_{com} and T_{calc} are the communication and the

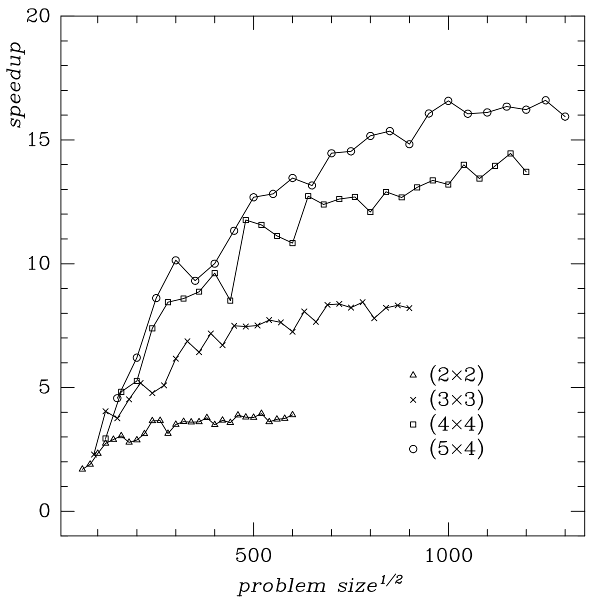

Figure 8: Parallel speedup in 2D simulations using finite differences.

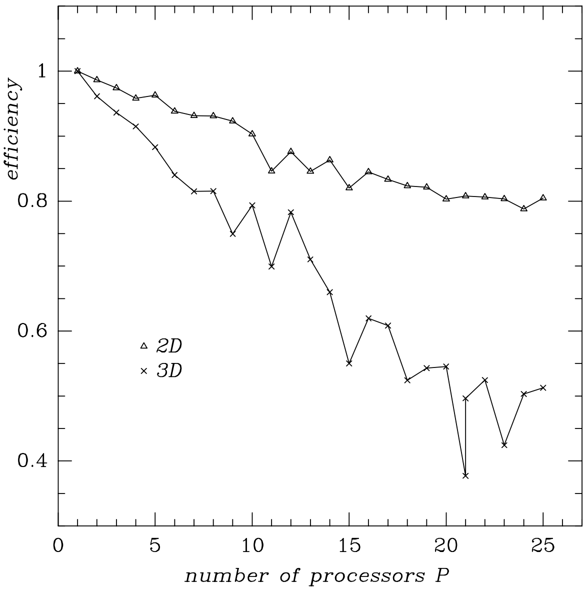

Figure 9: The Ethernet network performs well for 2D simulations (triangles), but poorly for 3D simulations (crosses).

computation time it takes to perform one integration step. We observe that T_{calc} is smaller for FD than LB (see the table of speeds earlier), and moreover that T_{com} becomes larger for FD than LB as the subregion per processor decreases. The latter is true because each message in a local area network incurs an overhead, and FD communicates two messages per integration step as opposed to LB which communicates only one message per integration step (see end of section 6). Because of these differences between FD and LB, the efficiency decreases more rapidly for FD than LB as the subregion per processor decreases.

Next, we compare the efficiency of three-dimensional simulations versus two-dimensional ones, using the lattice Boltzmann method. Figure 9 plots the efficiency of 2D and 3D simulations as a function of the number of processors P. Here, we simulate a problem which grows linearly with the number of processors P, and is decomposed as $(P \times 1)$ in 2D, and as $(P \times 1 \times 1)$ in 3D. The subregion per processor is held fixed at 120^2 nodes in 2D, and 25^3 nodes in 3D, which are comparable sizes, equal to about $14,500$ fluid nodes per processor. We see that the efficiency remains high in 2D (triangles), and decreases quickly in 3D (crosses) as the number of processors increases. This is because the total traffic through the shared-bus network increases in proportion to the number of processors, and this affects T_{com} in equation 6 as we shall see in more detail in the next section. Also, we note that 3D requires much more data to be communicated per step than 2D. Thus, T_{com} increases faster for 3D than 2D, and the efficiency drops faster in the case of 3D simulations.

Another way of examining the efficiency of 3D simulations is shown in figures 10 and 11. Figure 10 plots the efficiency against the size of the subregion for different decompositions $(2 \times 2 \times 2)$, $(3 \times 2 \times 2)$, etc. We can see that the efficiency is rather poor. Figure 11 plots the speedup against the total size of the problem. We can see that the speedup does not improve when finer decompositions are employed because the network is the bottleneck of the computation.

The results shown in figures 10 and 11 have been obtained using the lattice Boltzmann method. The parallel efficiency of the finite difference method (FD) in 3D simulations is even worse than the lattice Boltzmann method (LB), and is not shown here. The FD efficiency is worse than LB because the FD computes twice as fast as LB per integration step (see earlier table of speeds), which makes the ratio T_{com}/T_{calc} larger for FD than LB, and leads to lower efficiency according to equation 6.

We note that in our system the low efficiency of 3D simulations is accompanied by frequent network errors because of excessive network traffic. In particular, the TCP/IP protocol fails to deliver messages after excessive retransmissions. Both the low efficiency, and the network

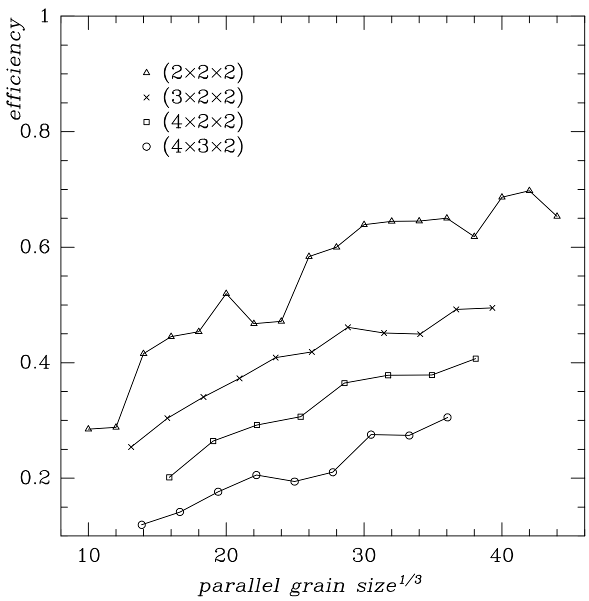

Figure 10: Parallel efficiency in 3D simulations using the lattice Boltzmann method.

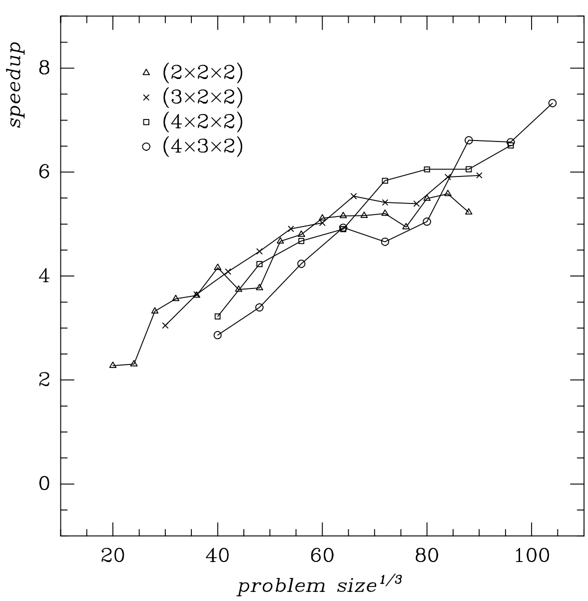

Figure 11: Parallel speedup in 3D simulations using the lattice Boltzmann method.

errors indicate the need for a faster network, or dedicated connections between neighboring processors in order to perform 3D simulations efficiently.

8 Theoretical analysis of parallel efficiency

In order to understand better the experimental results of the previous section, we develop a theoretical model of the parallel efficiency of local interaction problems. In particular, we derive a formula for the parallel efficiency in terms of the parallel grain size (the size of the subregion that is assigned to each processor), the speed of the processors, and the speed of the communication network. Our analysis is based on two assumptions: (i) the computation is completely parallelizable, and (ii) the communication does not overlap in time with the computation. The first assumption is valid for local interaction problems, and the second assumption is valid for the distributed system that we have implemented. The extension of our analysis to situations where communication and computation overlap in time is straightforward as we shall see afterwards.

We first examine the relationship between the efficiency and the processor utilization. We define the efficiency f as the speedup S divided by the number of processors P. Further, we define the speedup S as the ratio T_1/T_p of the total time it takes to solve a problem using one processor, denoted T_1, divided by the total time it takes to solve the same problem using P processors, denoted T_p. In other words, we have the following expression,

$$f \;=\; \frac{S}{P} \;=\; \frac{T_1}{P\,T_p} \tag{7}$$

We define the processor utilization g as the fraction of time spent for computing, denoted T_{calc}, divided by the total time spent for solving a problem which includes both computing and waiting for communication to complete. Also, we use the simplifying assumption that the communication and the computation do not overlap in time, so that we define T_{com} as the time spent for communication without any computation occurring during this time. Thus, we have the following expression,

$$g \;=\; \frac{T_{calc}}{T_{calc} + T_{com}} \;=\; \left(1 + \frac{T_{com}}{T_{calc}}\right)^{-1} \tag{8}$$

To compare f and g, we note that the values of both f and g range between the following limits,

$$\begin{aligned} 0 \le g \le 1 \\ 0 \le f \le 1 \end{aligned} \tag{9}$$

for the worst case and the best case respectively. We expect that high utilization g corresponds to high parallel efficiency f; however, this depends on the problem that we are trying to compute in parallel.

In the special case of a problem that is completely parallelizable, the processor utilization g is exactly equal to the parallel efficiency f. To show this, we use the following relation as the definition of a problem being completely parallelizable,

$$T_{calc} = \frac{T_1}{P} \tag{10}$$

Then, we also use the assumption that communication and computation do not overlap in time, so that we can obtain a second relation,

$$(T_{calc} + T_{com}) = T_p \tag{11}$$

By substituting equations 10 and 11 into equation 7, and comparing with equation 8, we arrive at the desired result that the parallel efficiency is exactly equal to the processor utilization,

$$f = g = \left(1 + \frac{T_{com}}{T_{calc}}\right)^{-1} \tag{12}$$

We have derived the above equation under the assumption that communication and computation do not overlap in time. If this assumption is violated in a practical situation, then the communication time T_{com} should be replaced with a smaller time interval, the effective communication time. This modification does not change the conclusion $f = g$, it simply gives higher values of efficiency and utilization.

To proceed further, we need to find how the ratio T_{com}/T_{calc} depends on the size of the subregion. First, we observe that T_{calc} is proportional to the size of the subregion. If N is the size of the subregion (the number of parallel nodes that constitute one subregion), we write,

$$T_{calc} = \frac{N}{U_{calc}} \tag{13}$$

where U_{calc} is a constant, the computational speed of the processors for the specific problem at hand. In a similar way, we seek to find a formula for the communication time T_{com} in terms of the size of the subregion that is assigned to each processor. As a first model, we write the following simple expression,

$$T_{com} = \frac{N_c}{U_{com}} \tag{14}$$

where N_c is the number of communicating nodes in each subregion, namely the outer surface of each subregion. The factor U_{com} represents the speed of the communication network.

For analysis purposes, we want to know exactly how N_c varies with the size of the subregion N. We consider the geometry of a subregion in two dimensions. We can see that the boundary of a subregion is one power smaller than the volume expressed in terms of the number of nodes. For example, if we consider square subregions of size L^2 nodes, the enclosing boundary contains $4L$ nodes, and the ratio of communicating nodes to the total number of nodes per subregion can be as large as $4/L$. In general, we have the following relations,

$$N_c = m\, N^{1/2} \tag{15}$$

$$N_c = m\, N^{2/3} \tag{16}$$

in two and three dimensions respectively, where the constant m depends on the geometry of the decomposition. For example, if the decomposition of a problem is $(P \times 1)$, then $m = 2$ because each subregion communicates with its left and right neighbors only. The following table gives m for a few decompositions which we use in our performance measurements in section 7,

	$P \times 1$	2×2	3×3	4×4	5×4
m	2	2	3	4	4

If we introduce the above formulas for N_c and m into equation 12, we obtain the following expressions for the parallel efficiency of a local interaction problem in two and three dimensions respectively,

$$f = \left(1 + N^{-1/2}\, \frac{m\, U_{calc}}{U_{com}}\right)^{-1} \tag{17}$$

$$f = \left(1 + N^{-1/3}\, \frac{m\, U_{calc}}{U_{com}}\right)^{-1} \tag{18}$$

The above equations show that if N is sufficiently large compared to the term mU_{com}/U_{calc}, then we can achieve high parallel efficiency.

A few comments are in order. First, we must remember that in practice we can not increase arbitrarily the size of the subregion per processor in order to achieve high efficiency. This is because the computation may take too long to complete, and because the memory of each workstation is limited. In our present system, each workstation has maximum memory 32 megabytes, and a large part of this memory is taken by other programs, and other users. A practical upper limit of how much memory we can use per workstation is 15 megabytes, which corresponds to 300^2 fluid nodes in 2D simulations and 40^3 fluid nodes in 3D simulations.

In 2D simulations the upper limit of 300^2 fluid nodes per subregion is large enough to achieve high efficiency. As we saw in figure 5, high efficiency is achieved when the subregion per processor is larger than 100^2 fluid nodes. By contrast, in 3D simulations the upper limit of 40^3 fluid nodes per subregion is too small to achieve high

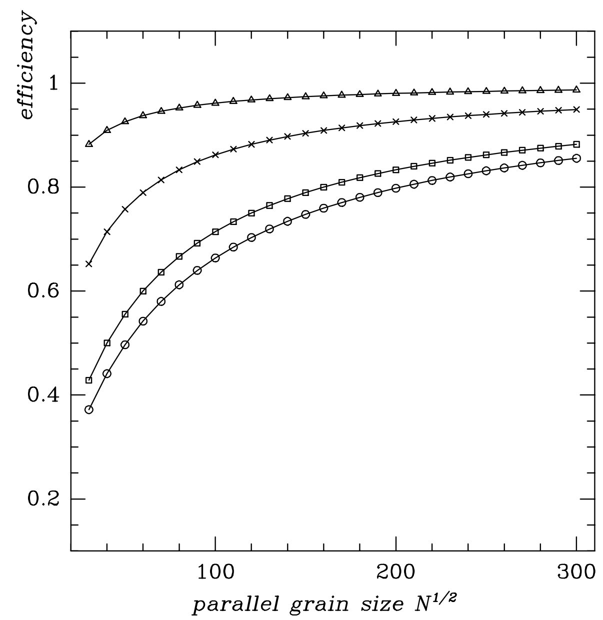

Figure 12: Theoretical model of parallel efficiency for two-dimensional subregions of size N.

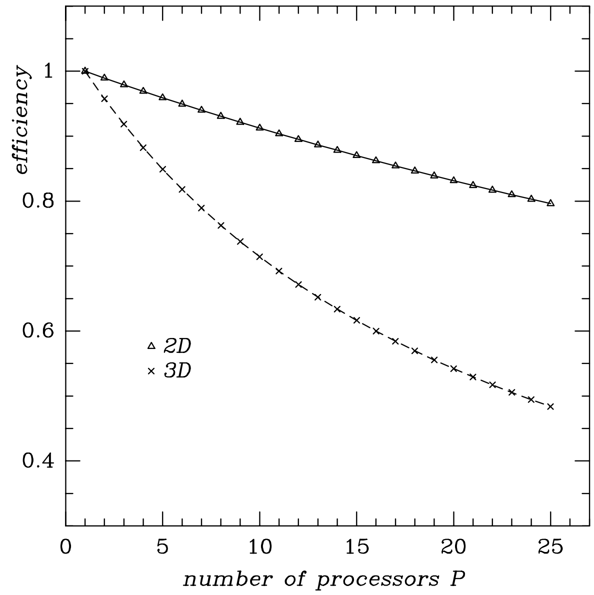

Figure 13: Theoretical model of parallel efficiency which assumes that the communication time increases linearly with the number of processors.

efficiency. Further, the efficiency depends on the size of the subregion as $N^{-1/3}$ in 3D versus $N^{-1/2}$ in 2D, as we can see from equations 17 and 18. This means that the size of the subregion N must increase much faster in 3D than in 2D to achieve similar improvements in efficiency. Because of this fact, achieving high efficiency in 3D simulations is much more difficult than in 2D simulations.

Having described the basic idea behind our model of parallel efficiency, we now discuss a small improvement of our model. We observe that in the case of a shared-bus network the communication time T_{com} must depend on the number of processors that are using the network. In particular, if we assume that all the processors access the shared-bus network at the same time, then the communication time T_{com} must increase linearly with the number of processors. Based on this assumption, we rewrite equation 14 for T_{com} as follows,

$$T_{com} \; = \; \frac{m\, N^{1/2}\, (P-1)}{V_{com}} \qquad (19)$$

for the case of two dimensional problems. The constant V_{com} is the speed of communication when there are only two processors sharing the network. Using the new expression for T_{com}, the equation of parallel efficiency in two dimensions becomes as follows,

$$f \; = \; \left(1 \; + \; N^{-1/2}\, (P-1)\, \frac{m\, U_{calc}}{V_{com}}\right)^{-1} \qquad (20)$$

To verify our model, below we compare the efficiency that is predicted by our model against the experimentally measured efficiency of section 7.

Figure 12 plots the efficiency f versus $N^{1/2}$ according to formula 20, using $U_{calc}/V_{com} \; = 2/3$. The four curves marked with triangle, cross, square, circle correspond to different numbers of processors $P = 4, 9, 16, 20$ and also different values of $m = 2, 3, 4, 4$ which depends on the geometry of the decomposition as we explained earlier. A comparison between the predicted efficiency shown in figure 12 and the experimentally measured efficiency shown in figure 5 reveals good agreement when the subregion per processor is larger than $N > 100^2$. However, for small subregions, $N < 100^2$, the predicted efficiency is too high compared to the experimental efficiency. The reason for this is that messages in a local area network have a large overhead which becomes important when the messages are small, namely, when the subregion per processor is smaller than $N < 100^2$ fluid nodes. The overhead of small messages leads to a smaller communication speed V_{com}, and a corresponding decrease of efficiency f. We have not attempted to model the overhead of small messages here.

Another way of examining the validity of equation equation 20 is to plot the efficiency f versus the number of processors P while keeping all other parameters constant. In figure 13, we plot the efficiency of 2D simulations according to equation 20 using $N = 125^2$. We set $U_{calc}/V_{com} \; = 2/3$ as we did in figure 12, and we set $m = 2$ because each subregion communicates with its left and right neighbors only. For comparison purposes, we also plot the efficiency of 3D simulations, using $N = 25^3$ and $m = 2$. The computational speed is half as large in 3D than in 2D, and the communication of each fluid node in 3D requires 5/3 as much data as in 2D. Taking these numbers into account, we can write the following expression for the parallel efficiency of 3D simulations,

$$f = \left(1 + \frac{5}{6}\, N^{-1/3}\,(P-1)\,\frac{m\,U_{calc}}{V_{com}}\right)^{-1} \tag{21}$$

where the factor 5/6 arises because we use the 2D values of U_{calc} and V_{com} which give $U_{calc}/V_{com} = 2/3$.

We now compare the predicted efficiency shown in figure 13 against the experimentally measured efficiency shown in figure 9. We can see that there is good agreement. Also, the overhead of small messages, which we mentioned earlier, does not affect the predicted efficiency in this case because the subregion per processor is large, $N = 125^2$ in 2D, and 25^3 in 3D. Overall, we find reasonable agreement between the theoretical model and the experimental measurements of parallel efficiency. The model can be improved further, if desired, by employing more sophisticated expressions for the communication time T_{com} in equation 19 which describes the behavior of the shared-bus Ethernet network.

9 Conclusion

A promising approach of simulating fluid dynamics on a cluster of non-dedicated workstations has been presented. The approach is particularly good for simulating subsonic flow which involves both hydrodynamics and acoustic waves. A parallel simulation system has been developed and applied to solve a real problem, the simulation of air flow inside wind musical instruments.

The system achieves concurrency by decomposing the flow problem into subregions, and by assigning the subregions to parallel subprocesses on different workstations. The use of explicit numerical methods leads to minimum communication requirements. The parallel processes automatically migrate from busy hosts to free hosts in order to exploit the unused cycles of non-dedicated workstations, and to avoid disturbing the regular users. Typical simulations achieve 80% parallel efficiency (speedup/processors) using 20 HP-Apollo workstations.

Detailed measurements of the parallel efficiency of 2D and 3D simulations have been presented, and a theoretical model of efficiency has been developed which fits closely the measurements. The measurements show that a shared-bus Ethernet network with 10Mbps peak bandwidth (megabits per second) is sufficient for two-dimensional simulations of subsonic flow, but is limited for three-dimensional simulations. It is expected that the use of new technologies in the near future such as Ethernet switches, FDDI and ATM networks will make practical three-dimensional simulations of subsonic flow on a cluster of workstations.

Acknowledgments

The author would like to thank Jacob Katzenelson, and Hal Abelson for useful criticisms on earlier versions of this paper. The author would also like to thank all the members of the project on Mathematics and Computation at MIT for generously allowing the use of their workstations.

Appendix

The appendix describes certain aspects of our distributed system that are not vital for a general reading, but are useful to someone who is interested in implementing a distributed system similar to ours.

A Un-synchronization of processes

The synchronization between parallel processes that we discussed in section 4.2 can be violated in situations such as the following. Let us suppose that process A stops execution after communicating its data for integration step N. The nearest neighbor B can integrate up to step $N+1$ and then stop. Process B can not integrate any further without receiving data for integration step $N+1$ from process A. However, the next to nearest neighbor can integrate up to step $N+2$, and so on. If we consider a two-dimensional decomposition $(J \times K)$ of a problem, the largest difference in integration step between two processes is ΔN,

$$\Delta N = max(J, K) - 1 \tag{22}$$

assuming that neighbors depend on each other along the diagonal direction (this corresponds to a full stencil of local interactions as shown in figure 4). If neighbors depend on each other along the horizontal and vertical directions only (this is the star stencil of figure 4), then the largest difference in integration step between two processes becomes,

$$\Delta N = (J-1) + (K-1) \tag{23}$$

These worst cases of un-synchronization are important when we consider the migration of processes because a precise global synchronization is required then, as is explained in section 5.

B Synchronization algorithm

The synchronization algorithm that is used during process migration (see section 5) is as follows. First, a synchronization request is sent to all the processes by means of a UNIX interrupt. In response to the request, every process writes the current integration time step into a shared file (using file locking semaphores, and append mode). Then, every process examines the shared file to find the largest integration time step $T_{\max}$ among all the processes. Further, every process chooses $(T_{\max} + 1)$ to be the upcoming synchronization time step, and continues running until it reaches this time step. It is important that all the processes can reach the synchronization time step, and that no process continues past the synchronization time step.

The above algorithm finds the smallest synchronization time step that is possible at any given time, so that a pending migration can take place as soon as possible.

C Order of communication

A minor efficiency issue with regard to TCP/IP communication (see section 4.2) is the order in which the neighboring processes communicate with each other. One way is for each parallel process to communicate with its neighbors on a first-come-first-served basis. An alternative way is to impose a strict ordering on the way the processes communicate with each other. For example, we consider a one-dimensional decomposition $(J \times 1)$ of a problem with non-periodic outer-boundaries where each process receives data from its left neighbor before it can send data to its right neighbor. Then, the leftmost process No. 1 will access the network first, and the nearest-neighbor process No. 2 will access the network second, and so on. The intent of such ordering is to pipeline the messages through the shared-bus network in a strict fashion in an attempt to improve performance. However, it does not work very well if one process is delayed because all the other processes are delayed also. Small delays are inevitable in time-sharing UNIX systems, and strict ordering amplifies them to global delays. By contrast, asynchronous first-come-first-served communication allows the computation to proceed in those processes that are not delayed, and better performance is achieved overall. In our system we implement first-come-first-served communication using the "select" system call of sockets (see UNIX manual).

D Other communication mechanisms

In section 4.2 we described the communication mechanism of our system which is based on the TCP/IP protocol and sockets. Apart from the TCP/IP protocol, another protocol that is popular in distributed systems is the UDP/IP protocol, also known as datagrams. The UDP/IP protocol is similar to TCP/IP with one major difference: There is no guaranteed delivery of messages. Thus, the distributed program must check that messages are delivered, and resend messages if necessary, which is a considerable effort. However, the benefit is that the distributed program has more control of the communication. For example, a distributed program could take advantage of knowing the special properties of its own communication to achieve better results than the TCP/IP standard. Also, another advantage is robustness in the case of network errors that occur under very high network traffic. For example, when TCP/IP fails, it is hard to know which messages need to be resent. In UDP/IP the distributed program controls precisely which data is sent and when, so that the failure problem is handled directly. Despite these advantages of UDP/IP over TCP/IP, we have chosen to work with TCP/IP because of its simplicity.

E Performance bugs to avoid

In section 7 we presented measurements of the performance of our workstations. Here, we note that the performance of the HP9000/700 Apollo workstations can degrade dramatically at certain grid sizes by a factor of two or more, but there is an easy way to fix the problem. The loss of performance occurs when the length of the arrays in the program is a near multiple of 4096 bytes which is also the virtual-memory page size. This suggests that the loss of performance is related to the prefetching algorithm of the CPU cache of the HP9000/700 computers. To avoid the loss of performance, we lengthen our arrays with 200-300 bytes when their length is a near multiple of 4096. This modification eliminates the loss of performance.

References

[1] C. H. Cap and V. Strumpen, "Efficient parallel computing in distributed workstation environments," *Parallel Computing*, vol. 19, no. 11, pp. 1221–1234, 1993.

[2] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield, "The Amber system: Parallel programming on a network of multiprocessors," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 5, pp. 147–158, 1989.

[3] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, vol. 1. Prentice-Hall Inc., 1988.

[4] R. Blumofe and D. Park, "Scheduling large-scale parallel computations on networks of workstations," in *Proceedings Of High Performance Distributed Computing 94, San Franscisco, California*, pp. 96–105, 1994.

[5] V. S. Sunderam, "A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, December 1990.

[6] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook, *Adaptive Parallelism with Piranha.* Report No. YALEU/DCS/RR-954, Department of Computer Science, Yale University, February 1993.

[7] S. Kohn and S. Baden, *A robust parallel programming model for dynamic non-uniform scientific computations.* Report CS94-354, University of California, San Diego, 1994.

[8] G. Chesshire and V. Naik, "An environment for parallel and distributed computation with application to overlapping grids," *IBM Journal Research and Development*, vol. 38, no. 3, pp. 285–300, May 1994.

[9] H. Bal, F. Kaashoek, and A. Tanenbaum, "Orca: A languate for parallel programming of distributed systems," *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190–205, March 1992.

[10] C. H. Shadle, *The Acoustics of Fricative Consonants.* Department of Electrical Engineering and Computer Science MIT, Ph.D. Dissertation, 1985.

[11] M. Verge, B. Fabre, W. Mahu, A. Hirschberg, R. van Hassel, and A. Wijnands, "Jet formation and jet velocity fluctuations in a flue organ pipe," *Journal of Acoustical Society of America*, vol. 95, no. 2, pp. 1119–1132, February 1994.

[12] M. Verge, R. Caussè, B. Fabre, A. Hirschberg, and A. van Steenbergen, "Jet oscillations and jet drive in recorder-like instruments," *accepted for publication in Acta Acustica*, 1994.

[13] A. Hirschberg, *Wind Instruments.* Eindhoven Institute of Technology, Report R-1290-D, 1994.

[14] W. Camp, S. Plimpton, B. Hendrickson, and R. Leland, "Massively parallel methods for engineering and science problems," *Communications of the ACM*, vol. 37, no. 4, pp. 31–41, April 1994.

[15] F. Douglis, *Transparent Process Migration in the Sprite Operating System.* Report No. UCB/CSD 90/598, Computer Science Division (EECS), University of California Berkeley, September 1990.

[16] D. Tritton, *Physical Fluid Dynamics.* Oxford Science Publications, Second Edition, 1988.

[17] G. Batchelor, *An Introduction to Fluid Dynamics.* Cambridge University Press, 1967.

[18] H. Lamb, *Hydrodynamics.* Sixth Edition, Dover Publications, N.Y., 1932,1945.

[19] R. Peyret and T. D. Taylor, *Computational Methods For Fluid Flow.* Springer-Verlag, New York, N.Y., 1990.

[20] P. Skordos, "Initial and boundary conditions for the lattice Boltzmann method," *Physical Review E*, vol. 48, no. 6, pp. 4823–4842, December 1993.

[21] L. Landau and E. Lifshitz, *Fluid Mechanics, 2nd Edition.* Pergamon Press, New York, NY, 1989.