Proceedings of Fourth IEEE International Symposium on High Performance Distributed Computing,
Washington D.C., USA 1995

Parallel Simulation of Subsonic Fluid Dynamics on a Cluster of Workstations

Panayotis A. Skordos

Massachusetts Institute of Technology
545 Technology Square NE43-432, Cambridge, MA 02139 USA
pas@ai.mit.edu

Abstract

An effective approach of simulating subsonic fluid dynamics on a cluster of non-dedicated workstations is presented. The approach is applied to simulate the flow of air in wind instruments. The use of local-interaction methods and coarse-grain decompositions lead to small communication requirements. The automatic migration of processes from busy hosts to free hosts enables the use of non-dedicated workstations. Simulations of 2D flow 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 and compared against the measurements. Two numerical methods of fluid dynamics are tested: explicit finite differences, and the lattice Boltzmann method.

1 Introduction

An effective approach of exploiting a cluster of non-dedicated workstations to simulate subsonic fluid dynamics is presented. Concurrency is achieved by decomposing the problem into rectangular subregions, and by assigning the subregions to parallel subprocesses on different workstations. The use of local-interaction methods and coarse-grain decompositions lead to small communication requirements which can be satisfied on a cluster of workstations. 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 implemented directly on top of UNIX and TCP/IP communication routines.

Simulations of 2D flow achieve 80% parallel efficiency (speedup/processors) using 20 non-dedicated HP-Apollo workstations. Detailed measurements of the parallel efficiency of 2D and 3D simulations are presented, and a theoretical model of parallel efficiency is developed and compared against the measurements. It is shown that the shared-bus Ethernet network is adequate for 2D simulations, but limited for 3D ones.

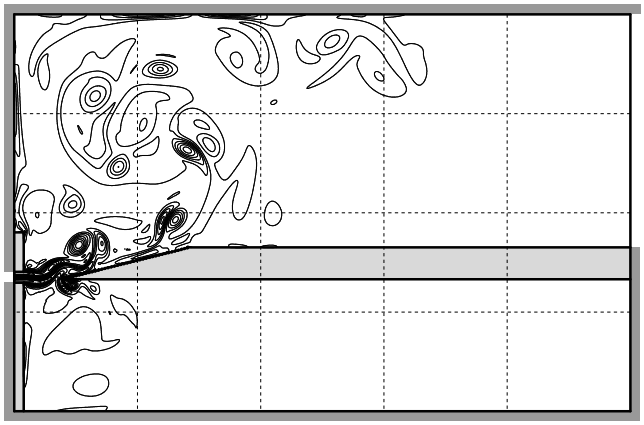

Figure 1: Simulation of a flue pipe using 20 workstations. The dashed lines show the 5x4 decomposition. The grid is 800×500. Equi-vorticity contours are plotted (curl of velocity). The gray areas are walls, and the resonant pipe is located at the bottom. A jet of air enters through an opening on the left wall, and impinges the sharp edge in front of it. Air exits from the simulation through the opening on the right.

The success of the present approach depends critically on the use of local-interaction methods (explicit numerical methods [1]) because the communication capability of the Ethernet network is small. Explicit methods are inherently parallel, have small communication requirements and small cost per integration step, but require small integration time steps for numerical stability. By contrast, implicit methods [1] are challenging to parallelize, have large communication requirements and large cost per integration step, but they can use much larger integration time steps

than explicit methods. Therefore, in general, one has to weigh the advantages and disadvantages of explicit and implicit methods in the context of the particular problem at hand and the available computer system.

In the case of time-dependent subsonic flow, the time steps are usually chosen very small in order to model accurately the acoustic waves. Thus, explicit methods that require small time steps anyway for numerical stability, are well recommended in this case. To give specific numbers for the problem of air flow in wind instruments, the flow speed is 1000 cm/s and the acoustic speed c_s is 30 times larger than that. If we wish to model the passage of acoustic waves at the finest grid resolution Δx, then we must choose the time step $\Delta t \sim \Delta x/c_s$ which is very small and also turns out to be sufficient for the numerical stability of explicit methods for the present problem (see [1] and section 6). On the other hand, if we do not insist on modeling acoustic waves at the finest grid resolution, then we can use larger time steps with implicit methods. However, the time step is still constrained for modeling reasons by the hydrodynamic flow speed and by the high acoustic frequencies such as 20 kHz for wind instruments. Thus, it turns out that the maximum time step of an implicit method is at most $10-30$ times larger than the time step of an explicit method for this problem. If we consider the different costs per integration step, explicit methods may be competitive with implicit methods for this problem even on serial computers. If we also consider parallel computing, explicit methods are a very good choice for this problem.

With the above introduction and clarification on explicit versus implicit numerical methods, we proceed below to describe our distributed computing approach. In section 2 we present examples of distributed simulations, and in section 3 we review 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 7 presents experimental measurements of the performance of our system, and section 8 develops a theoretical model of parallel efficiency of local-interaction problems. Most ideas are discussed as generally as possible within the context of local-interaction problems, and the specifics of fluid dynamics are limited to sections 2 and 6.

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 [2], [3] and elsewhere. Cap&Strumpen [2] present the PARFORM system and simulate the unsteady heat equation using explicit finite differences. Chase&et al. [3] present the AMBER parallel system, and solve Laplace's equation using Successive Over-Relaxation. The present work clarifies further the importance of local-interaction methods for parallel systems with small communication capacity. Furthermore, a real application problem is solved using the present approach: the simulation of subsonic flow with acoustic waves in wind musical instruments.

In the area of fluid dynamics, little attention has been given until recently to simulations of hydrodynamics and acoustic waves because such simulations are very compute-intensive and can be performed only when parallel systems such as the one described here are available. Furthermore, the use of explicit methods has generally been shunned because explicit methods require small integration time steps for numerical stability. With the increasing availability of parallel systems, explicit methods are now attracting more and more attention in all areas of computational fluid dynamics. The present work illustrates the power of explicit methods in one particular area (subsonic compressible flow), and should motivate further work on explicit methods in other areas as well.

Regarding the experimental measurements of parallel efficiency which are presented in section 7, they are more detailed than any other reference known to the author especially for the case of a shared-bus Ethernet network. The model of parallel efficiency which is developed in section 8 is based on ideas which have been discussed previously, for example in Fox et al. [4] and elsewhere. Here, the model is derived in a clear way, and the predictions of the model are compared against experimental measurements of parallel efficiency.

Regarding the problem of using non-dedicated workstations, this problem is handled 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 [2]). Although this approach is important in various applications (Blumofe&Park [5]), it seems unnecessary for simulating fluid flow problems with static geometry and may lead to large overhead. For such problems, it may be more effective to use large fixed-size subregions per processor, and to apply automatic migration of processes from busy hosts to free hosts. This 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 present approach does not address the issues of high-level programming, parallel languages, inhomogeneous clusters of workstations, and distributed computing of general problems. Efforts along these directions are the PVM system [6], the Linda system [7], the packages of Kohn&Baden [8] and Chesshire&Naik [9] that facilitate parallel decomposition, the Orca language for distributed computing [10], etc.

2 Examples of flow simulations

The present distributed system has been applied to simulate the flow of air and the generation of tones in wind instruments such as the recorder and the flute (flue pipes). 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 tones. The oscillations are reenforced by a nonlinear feedback from the acoustic waves to the jet of air. The mechanism of flow-generated sound in wind instruments is a 100-year-old problem whose details are still a subject of active research [11].

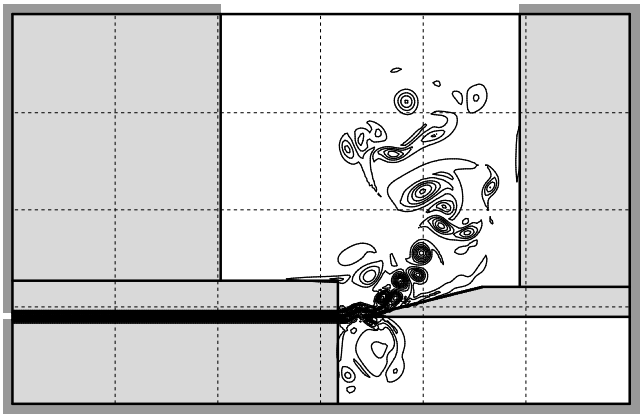

Figure 2: Simulation of a flue pipe using 15 workstations in a 6x4 decomposition with 9 subregions inactive. The air passes through a long channel before impinging the sharp edge. The outlet of the simulation is located at the top of the picture.

The present parallel system can simulate the flow of air inside flue pipes using uniform orthogonal grids as large as 1200×1200 in two dimensions (1.5 million nodes) and even larger. We typically employ smaller grids, however, in order to reduce the computing time. For example, if we divide a 800×500 grid (0.38 million nodes) 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 and a jet of air that oscillates at 1000 cycles per second.

Figures 1 and 2 show snapshots from simulations of flue pipes. The geometry of figure 2 is particularly 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.

3 Local-interaction computations

Here we review local-interaction problems in general. We define a local-interaction problem 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 4-star (cross) stencil, but other patterns of local interactions are also possible.

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 however, the parallel nodes are usually grouped into subregions of nodes, as shown in figure 3 by the dashed lines. The subregions are assigned to different processors, and the problem is solved in parallel as follows,

- 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 subre-

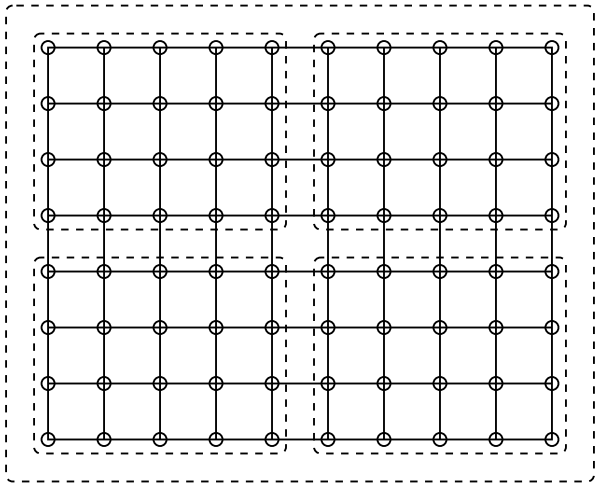

Figure 3: A problem of local interactions in two dimensions, and its decomposition 2x2 into four subregions.

gions. Thus, the amount of communication relative to computation is proportional to the surface-to-volume ratio of the subregions. This is very important because by coarse-graining the subregions (increasing their size), we can reduce the time spent on communication versus computation and achieve good processor utilization. This is the reason why local-interaction problems are very flexible and highly desirable for parallel computing.

4 The distributed system

The implementation of the distributed system is now described. It is based on UNIX and TCP/IP communication routines, and it also exploits the common file system of the workstations. For the sake of programming modularity, the system is organized 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 (the 5-minute average CPU load must be less than 0.5 where 1.0 means there is one process running full-time), 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 20 minutes. If a workstation becomes too busy, automatic migration of the affected process takes place.

All of the above programs (initialization, decomposition, submit, and monitoring) are executed by one designated workstation in the cluster. In addition to the above programs, there is the parallel program that is executed by all the workstations and consists of two parts: "compute locally", and "communicate with neighbors". Below, we discuss the communication.

4.1 Communication

The communication between parallel processes synchronizes the processes because it encourages the processes to begin each computational cycle together with their neighbors as soon as they receive data from their neighbors. Also, two neighbors are always less than one time step apart because a process must receive the new boundary data from its neighbors before it can proceed to the next integration step. Our measurements show that during normal execution, all the parallel processes are very close in time, starting and ending each computational cycle almost at the same time.

The communication of data between processes is organized using a well-known programming technique which is called "padding" or "ghost cells" (Fox [4], Camp [12]). Specifically, we pad each subregion with one or more layers of extra nodes on the outside depending on how far the local-interaction rule extends to (for example, one layer in the present fluid flow simulations). 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.

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, we can develop a parallel code for fluid dynamics by extending a serial code with a few subroutines that communicate the padded areas between neighboring subregions.

Our communication subroutines are implemented using sockets (see UNIX manual) and the TCP/IP protocol which provides first-in-first-out channels for writing data in each direction between two processes on different workstations.

Finally, it should be noted that the current system does not overlap communication with computation. A new implementation of the system is in progress which overlaps communication with computation, and the results of the new system will be described in a forthcoming article.

5 Transparency to other users

We now discuss the issues that arise when sharing the workstations with other users. The utilization of a workstation can be distinguished 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 process.

In the first two cases, it is appropriate to time-share the workstation with another user. We 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 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. However, when a workstation is running another full-time process in addition to a parallel process, the parallel process must migrate to a new host that is free. This is because the parallel process interferes with the regular user, and further, the distributed computation slows down because of one busy workstation. Clearly, such a situation must be avoided.

The distributed system detects the need for migration using the monitoring program 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. During 24 hours, there is about one migration every 45 minutes on-the-average for a distributed computation that uses 20 workstations from a pool of 25 workstations. Each migration lasts between $10 - 30$ seconds which is slow (this will be improved in a future implementation), but the migrations do not happen often. A typical migration is as follows,

- Process A receives the signal to migrate.
- All the processes get synchronized.
- Process A saves its state into a file and exits.
- 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 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.

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 amounts to stopping the computation, saving the state of the migrating process on disk, and then restarting. It is straightforward because the processes are programmed to deal with the migration themselves. By contrast, process migration in a general computing environment such as a distributed operating system [13] can be a very challenging task.

5.1 Sharing the network

A related issue to sharing the workstations with other users, is sharing the network and the file server. 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 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.

6 Numerical algorithms

In our simulations of subsonic flow, we solve numerically the compressible Navier Stokes equations (in the adiabatic form [14]). Three fluid variables are involved: the fluid density ρ, and the components of the fluid velocity V_x, V_y in the x,y directions respectively. We employ a uniform grid of fluid nodes 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. We use explicit numerical methods (explicit finite differences and the lattice Boltzmann method) to calculate the future values of density and velocity at each fluid node using the present values of density and velocity at this node and at neighboring nodes. The finite difference method [1] has the following form:

- 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)

For numerical stability reasons, the density is updated using the values of velocity at time $t + \Delta t$; namely, the velocity is computed first, and then the density is computed as a separate step. A filter is also included (fourth-order numerical viscosity [15, 16, 1]) in order to mitigate the nonlinear instabilities of compressible flow at high Reynolds number. The same filter is used 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 [17, 15] and has slightly better stability properties. 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)

Regarding the communication of boundary values for the finite difference method (FD) and the lattice Boltzmann method (LB), there are some differences that should be noted. 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 Performance measurements

We measure the performance of the distributed system when using the finite difference method and the lattice Boltzmann method to simulate Hagen-Poiseuille flow through a rectangular channel [17]. The goal of testing both methods is to examine the performance of the parallel system on two similar, but slightly different parallel algorithms. It should be noted that the two methods produce comparable results for the same resolution, and that both methods converge quadratically with increased spatial resolution to the exact solution of the Hagen-Poiseuille flow problem.

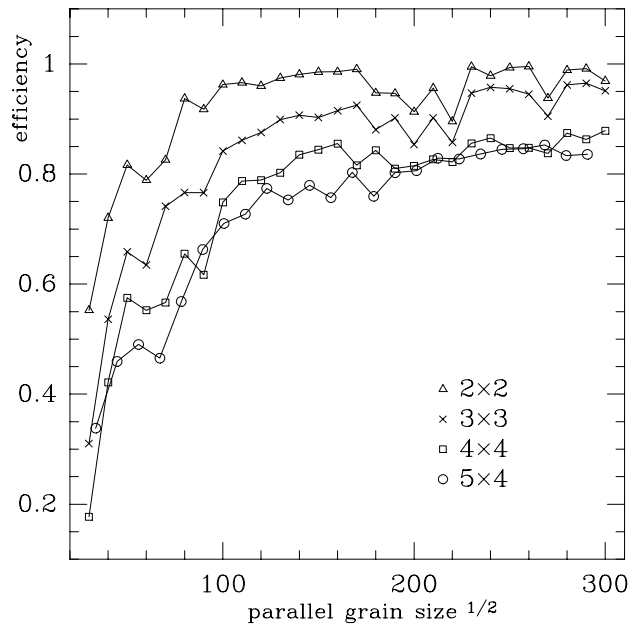

Figure 4: Parallel efficiency in 2D simulations using the lattice Boltzmann method and 2x2, 3x3, 4x4, 5x4 decompositions (triangles, crosses, squares, circles). The horizontal axis plots the square root of the number of nodes per subregion.

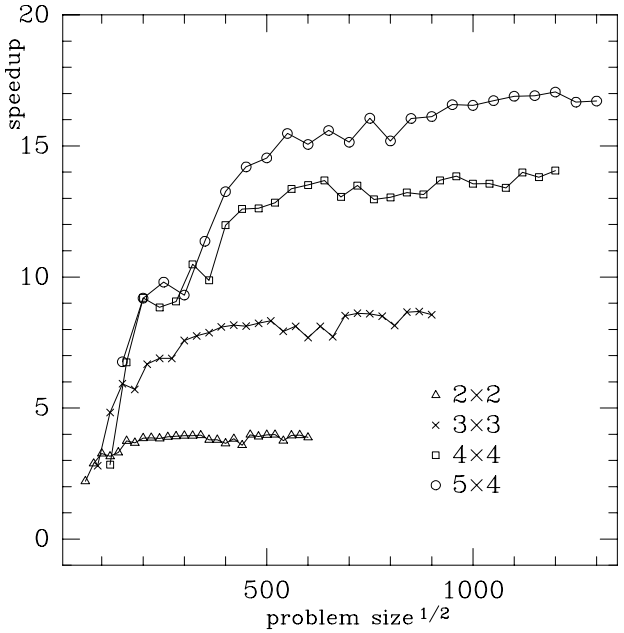

Figure 5: Parallel speedup in 2D simulations using lattice Boltzmann. The horizontal axis plots the square root of the total number of nodes in the problem.

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

$$f = \frac{S}{P} = \frac{T_1}{P\,T_p} \qquad (1)$$

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 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 4.1. 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

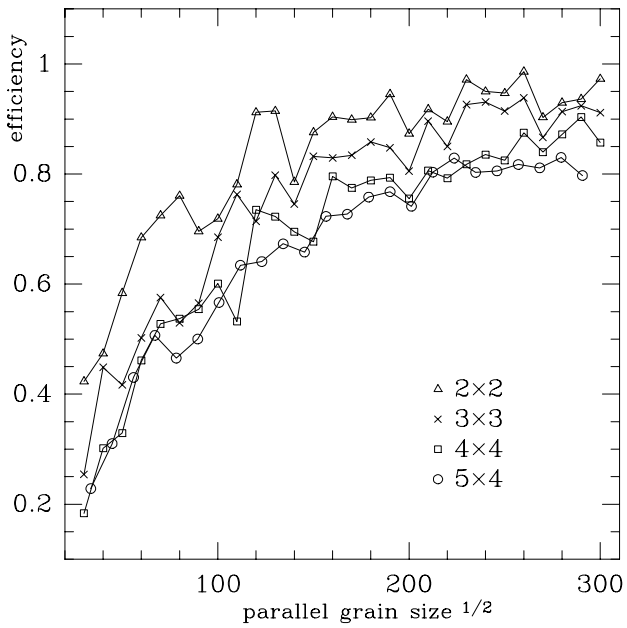

Figure 6: Parallel efficiency in 2D simulations using finite differences.

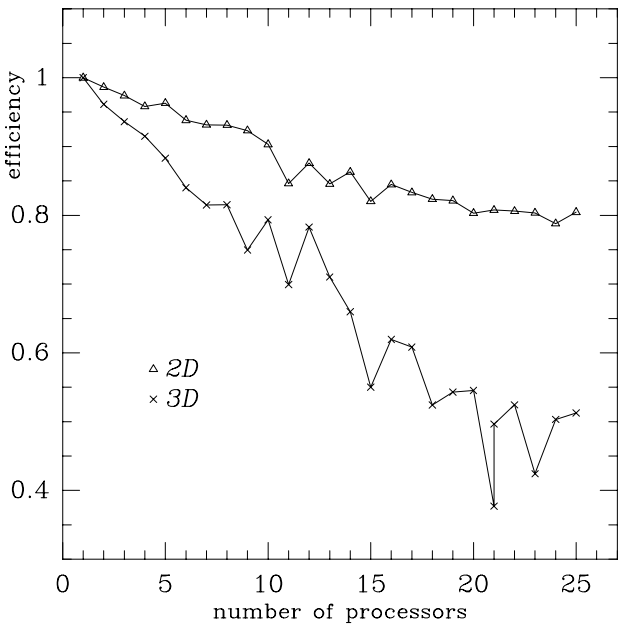

Figure 7: Parallel efficiency of 2D and 3D simulations (triangles, crosses) as a function of the number of processors on a shared-bus network. The lattice Boltzmann method is used. The problem grows linearly with the number of processors P and is decomposed as Px1 in 2D and as Px1x1 in 3D. The subregion per processor is held fixed at 120^2 nodes in 2D, and 25^3 nodes in 3D, which are comparable sizes and equal to about $14,500$ fluid nodes per processor.

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 4 plots the efficiency as a function of parallel grain size (the size of the subregion that is assigned to each processor) in 2D simulations using the lattice Boltzmann method. We see that good performance is achieved when the subregion per processor is larger than 100^2 fluid nodes. Figure 5 shows the speedup for the lattice Boltzmann method (LB), and figure 6 shows the efficiency 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 parallel grain size decreases. To understand this difference, we quote a general formula for the parallel efficiency, which is derived in the next section (see equation 8),

$$f = \left(1 + \frac{T_{com}}{T_{calc}}\right)^{-1} \qquad (2)$$

where T_{com} and T_{calc} are the communication and the computation time it takes to perform one integration step. It turns out that T_{calc} is smaller for FD than LB, and that T_{com} becomes larger for FD than LB as the parallel grain size 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. Because of these differences between FD and LB, the efficiency decreases more rapidly for FD than LB as the parallel grain size decreases.

Next, we compare the efficiency of 2D versus 3D simulations by plotting the efficiency as a function of the number of processors (see figure 7). We see that the efficiency remains high in 2D, and decreases quickly in 3D as the number of processors increases. This is because 3D requires much more data to be communicated per step than 2D, and the total traffic through the shared-bus network increases in proportion to the number of processors. Thus, T_{com} increases faster for 3D than 2D, and the efficiency decreases faster in the case of 3D simulations.

Another way of examining the efficiency of 3D simulations is shown in figures 8 and 9. Figure 8 plots the efficiency against the parallel grain size for different decompositions 2x2x2, 2x2x2, etc. We can see that the efficiency is rather poor. Figure 9 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

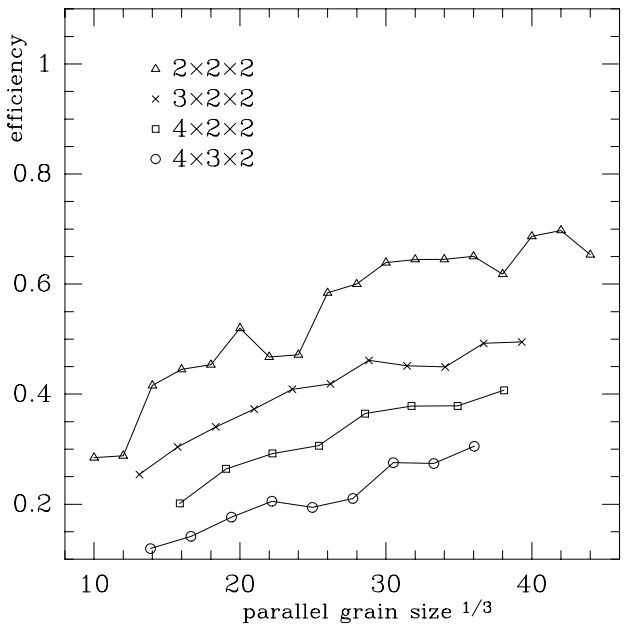

Figure 8: Parallel efficiency in 3D simulations using the lattice Boltzmann method. The horizontal axis plots the cubic root of the number of nodes per subregion.

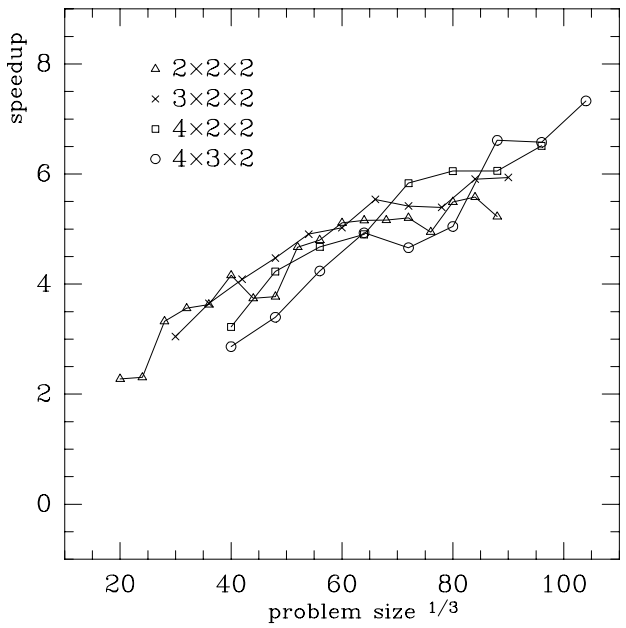

Figure 9: Parallel speedup in 3D simulations using the lattice Boltzmann method.

bottleneck of the computation.

The results shown in figures 8 and 9 have been obtained using the lattice Boltzmann method. The parallel efficiency of the finite difference method (FD) in 3D simulations is 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 2.

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

8 Theoretical analysis

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. The 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.

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,

$$f = \frac{S}{P} = \frac{T_1}{P\,T_p} \tag{3}$$

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,

$$g = \frac{T_{calc}}{T_{calc} + T_{com}} = \left(1 + \frac{T_{com}}{T_{calc}}\right)^{-1} \quad (4)$$

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

$$\begin{array}{l} 0 \leq g \leq 1 \\ 0 \leq f \leq 1 \end{array} \quad (5)$$

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,

$$T_{calc} = \frac{T_1}{P} \quad (6)$$

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

$$(T_{calc} + T_{com}) = T_p \quad (7)$$

By substituting equations 6 and 7 into equation 3, and comparing with equation 4, we arrive at the desired result that the parallel efficiency is exactly equal to the processor utilization,

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

The above equation has been derived under the assumption that communication and computation do not overlap in time. If this assumption is violated, 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 can write,

$$T_{calc} = \frac{N}{U_{calc}} \quad (9)$$

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,

$$T_{com} = \frac{N_c}{U_{com}} \quad (10)$$

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 N of the subregion. 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,

$$N_c = m\, N^{1/2} \quad (11)$$

$$N_c = m\, N^{2/3} \quad (12)$$

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 Px1, then $m = 2$ because each subregion communicates with its left and right neighbors only. The following table gives m for different decompositions which are used in the performance measurements of 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 8, we obtain the following expressions for the parallel efficiency of a local-interaction problem in two and three dimensions respectively,

$$f = \left(1 + N^{-1/2}\, \frac{m\, U_{calc}}{U_{com}}\right)^{-1} \quad (13)$$

$$f = \left(1 + N^{-1/3}\, \frac{m\, U_{calc}}{U_{com}}\right)^{-1} \quad (14)$$

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

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 the present system, each workstation has maximum memory 48 megabytes, and a large part of this memory is taken by other programs, and other users. In the measurements of section 7, we consider simulations that take up to 15 megabytes per workstation, corresponding to 300^2 fluid nodes in 2D simulations and 40^3 fluid nodes in 3D simulations.

In 2D simulations, the size of 300^2 fluid nodes per subregion is large enough to achieve high efficiency. As we saw in figure 4, 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 efficiency. Further, the efficiency depends on the size of the subregion as $N^{-1/3}$ in 3D versus $N^{-1/2}$ in 2D, as can be seen from equations 13 and 14. 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.

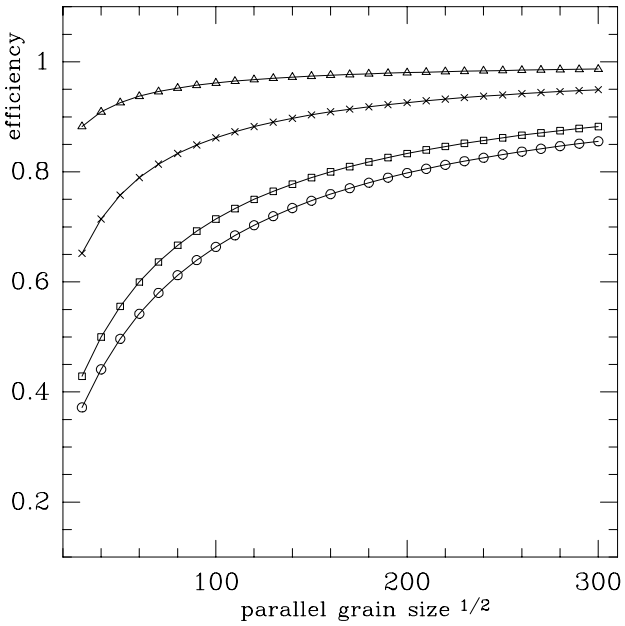

Figure 10: Theoretical model of parallel efficiency for two-dimensional subregions of size N.

Having described the basics of the model of parallel efficiency, we now discuss a small improvement of the 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 10 for T_{com} as follows,

$$T_{com} = \frac{m\, N^{1/2}\, (P-1)}{V_{com}} \qquad (15)$$

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,

$$f = \left(1 + N^{-1/2}\,(P-1)\,\frac{m\, U_{calc}}{V_{com}}\right)^{-1} \qquad (16)$$

This model is tested below by comparing the efficiency which is predicted by the model against the experimentally measured efficiency of section 7.

Figure 10 plots the efficiency f versus $N^{1/2}$ according to formula 16, 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 explained earlier. A comparison between the predicted efficiency shown in figure 10 and the experimentally measured efficiency shown in figure 4 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 equation 16 is to plot the efficiency f versus the number of processors P while keeping all other parameters constant. This is done in figure 11 using $N = 125^2$. We set $U_{calc}/V_{com} = 2/3$ as we did in figure 10, 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

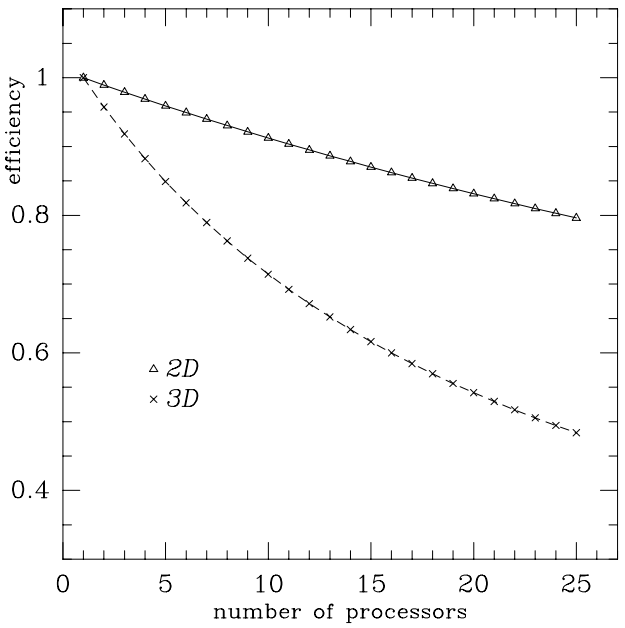

Figure 11: Theoretical model of parallel efficiency which assumes that the communication time increases linearly with the number of processors.

these numbers into account, we can write the following expression for the parallel efficiency of 3D simulations,

$$f = \left(1 + \frac{5}{6} N^{-1/3} (P-1) \frac{m U_{calc}}{V_{com}}\right)^{-1} \qquad (17)$$

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

The theoretical efficiency of figure 11 is to be compared against the experimentally measured efficiency of figure 7. We note that the overhead of small messages, 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, there is reasonable agreement between the theoretical model and the experimental measurement of parallel efficiency. The model can be improved further by employing more sophisticated expressions for the communication time T_{com} in equation 15 which describes the behavior of the shared-bus Ethernet network.

9 Conclusion

An effective approach of simulating subsonic fluid dynamics on a cluster of non-dedicated workstations has been presented. The approach is well-suited for problems that favor the use of explicit methods: for example, time-dependent flow with acoustic waves. A parallel distributed system has been developed and applied to simulate the flow of air in wind instruments. The system achieves concurrency by decomposing the flow problem into coarse-grain subregions, and by assigning the subregions to parallel subprocesses on different workstations. The use of explicit methods and coarse-grain decompositions lead to small 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.

Simulations of 2D flow 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 which show that a shared-bus Ethernet network with 10Mbps peak bandwidth is sufficient for 2D simulations, but is limited for 3D simulations. Finally, a theoretical model of efficiency has been developed and compared against the measurements.

Acknowledgements

The author thanks the members of the MAC project at MIT for allowing the shared-use of their workstations. The work is supported by ARPA contract N00014-92-J-4097 and NSF grant 9001651-MIP.

References

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

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

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] P. Skordos, *Modeling flue pipes: subsonic flow, lattice Boltzmann, and parallel distributed computers.* Department of Electrical Engineering and Computer Science, MIT, Ph.D. Dissertation, January 1995.

[16] P. Skordos and G. Sussman, "Comparison between subsonic flow simulation and physical measurements of flue pipes," in *Proceedings of ISMA 95, International Symposium on Musical Acoustics, Le Normont, France*, July, 1995.

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