LBL-23625 c.2

Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

Physics Division

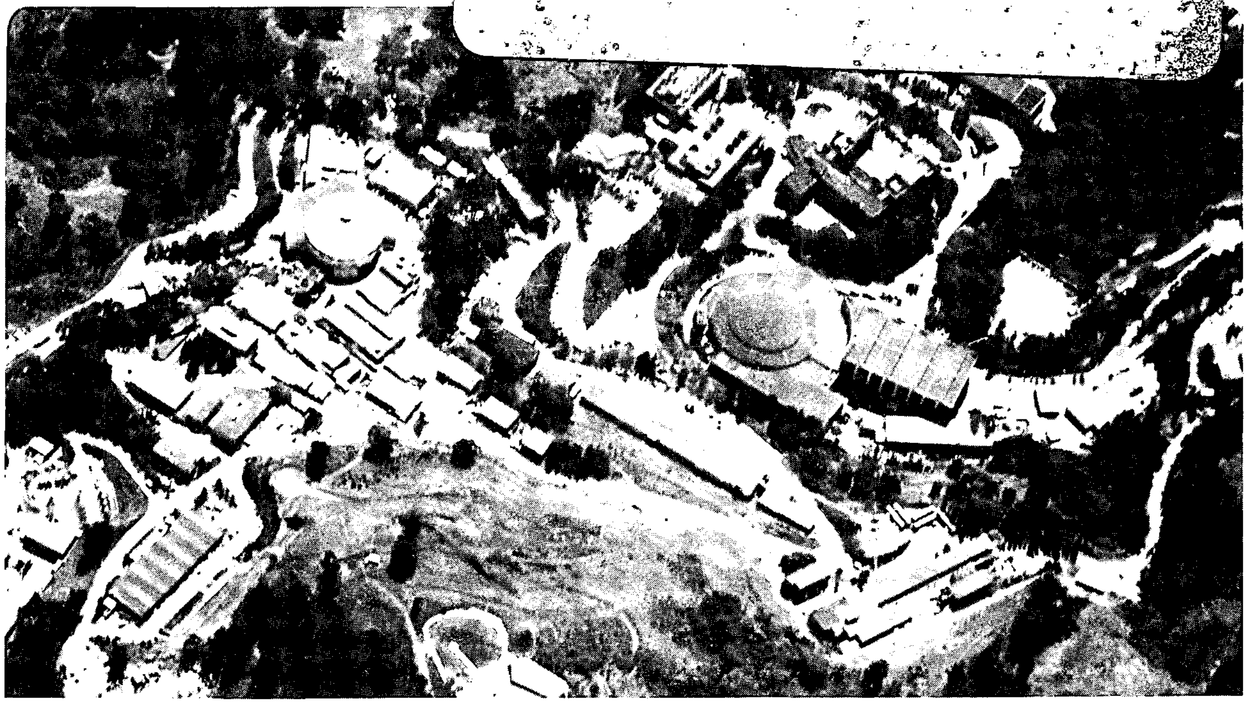
Mathematics Department

RUN-TIME PARTITIONING OF SCIENTIFIC CONTINUUM CALCULATIONS RUNNING ON MULTIPROCESSORS

S.B. Baden

(Ph.D. Thesis)

June 1987



LBL-23625 c.2

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

LBL-23625

RUN-TIME PARTITIONING OF SCIENTIFIC CONTINUUM CALCULATIONS RUNNING ON MULTIPROCESSORS[1]

Scott B. Baden

Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Ph. D. Dissertation,
Computer Science Division
University of California, Berkeley

June 1987

[1] This work was supported in part by a California Fellowship in Microelectronics and in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract DE-AC03-76SF00098.

Run-Time Partitioning
of Scientific Continuum Calculations
Running on Multiprocessors

Scott Benjamin Baden

Run-Time Partitioning of Scientific Continuum Calculations Running on Multiprocessors

Scott Benjamin Baden

Abstract

A wide range of scientific continuum calculations typically concentrate computational effort non-uniformly over localized regions of physical space. We present a run-time partitioning strategy, intended for such methods, that distributes work evenly across a team of processors and that can exploit the spatial localization present in the original computation in order to avoid high overhead costs. We tried out our strategy on Anderson's Method of Local Corrections, a type of vortex method for computational fluid dynamics. Because computational effort follows particles that congregate and disperse irregularly about the domain, this problem is hard to partition in a way that distributes the work evenly among the processors. We ran experiments on 32 processors of an Intel Personal Scientific Computer – a message-passing hypercube multiprocessor – and on 4 processors of a Cray X-MP – a shared-memory vector architecture – and achieved good parallel speedups of 22 and 3.6, respectively. The partitioner may

be implemented as a virtual machine (VM) and made available to the programmer as a library of run-time utilities. The semantics of the VM are insensitive to the application and to the computer architecture on which the VM is implemented. The VM works with ordinary programming languages, incurs modest overhead costs, and requires no special hardware support. It should apply to diverse applications, including finite difference methods, and to diverse architectures without requiring that the application be reprogrammed extensively for each new architecture.

Acknowledgements

I feel privileged to have worked with my advisor, W. Kahan. True to his word, Vel will graduate no student before his time; but, the rewards of meeting Vel's challenges will be with me always. A special word of thanks goes to Phil Colella, who was my fourth committee member. The distinction as fourth member is exceptional, but so is Phil; he made a special effort to show me the wonders of what had once seemed the dark and mysterious side of computer science – computational mathematical physics – while asking very little in return. I have fond memories of our lively discussions at the Cafe Roma, where our friendship was kindled. And yes, many thanks go to Phil's wife, Sue Londerville, for her good spirit.

Others have taken a keen interest in this research and I also wish to extend a warm thanks to them, too. Professors Paul Hilfinger and Beresford Parlett were my official readers, and dutifully waded through the final drafts of my dissertation. Professors Alexandre Chorin and Paul Concus provided constant encouragement and offered a pleasant working environment at Lawrence Berkeley Laboratory. Chris Anderson, a Berkeley graduate and now on the faculty at UCLA, offered insight and moral support, and also provided an initial version of the local corrections code. Cleve Moler arranged for employment at Intel Scientific Computers, where my work was met with great enthusiasm. Cleve's good humor helped brighten the hard work that accompanied this research. I am also grateful to Chris C. Hsiung of Cray Research, Inc. for providing access to the Cray X-MP/416 at the Cray facility in Mendota Heights, Minnesota. Philippe de Forcrand, John L. Larson, and Jeff Nicholson, all of Cray Research, cheerfully advised me on the finer points of the X-MP. Professor Alberto Sangiovanni-Vincentelli generously provided access to the iPSC at Berkeley.

To my office mates here at Berkeley – Ricki Blau, Paul Hansen, and Erling Wold – it's been wonderful sharing a cabin with you during our long cruise together.

To Ann Di Fruscia, thank you for the concern you showed me these past years, and for the graceful dances you performed.

Thank you Kathryn Crabtree, for reminding me to hand this in.

A final word of thanks go to my parents, Ruth Kramer Baden and Howard P. Baden, who watched from afar as I worked my way through to the finish.

This work was supported in part by a California Fellowship in Microelectronics and in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract DE-AC03-76SF00098. Additional support for this work was sponsored by: the National Science Foundation, grant # DMF84-03223; the State of California, California Microelectronics grant # 532429-19900; Intel Scientific Computers, a division of the Intel Corporation; Cray Research Incorporated; and the Defense Advance Research Projects Agency (DoD), Arpa Order No. 4871, Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089. AFOSR helped with travel expenses.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

Introduction

... It took me years to write it,
They were the best years of my life.
It was a beautiful song,
But it ran too long ...
So they cut it down to three-o-five.

–Billy Joel, ''The Entertainer''

1.1. Run-Time Partitioning and Software Portability

Ideally a multiprocessor system would satisfy two conditions: (1) its performance would be linearly proportional to the number of processors in use and (2) programs that run well on it may be written nearly independently of how its processors communicate. In practice, however, work can accumulate on only some of the processors, leaving the rest to sit mostly idle. Unless some kind of run-time partitioning strategy is employed to mitigate such load imbalance, the overall throughput of the system will be proportional not to the number of available processors, but to the small fraction that can be utilized effectively. A large number of run-time partitioning strategies already exist, but the overhead costs they incur can depend heavily upon the system on which they are implemented. Therefore a major difficulty remains: how to construct software that can run on diverse architectures without entailing substantial reprogramming for each implementation. For example, a program that runs well on a shared-memory architecture can look very different from a program implementing the same calculation but optimized for a message-passing architecture with only local memories.

I believe that a good run-time partitioning strategy for load balancing should have three characteristics. First, it should be versatile over some reasonable range of applications, and should not have to be reimplemented for each new combination of multiprocessor system and application. Second, a good strategy should incur modest overhead. A strategy that introduces excessive communication costs, for example, may be unable to improve throughput even if it does mitigate load imbalance; idleness caused by communication latency is no more productive than idleness caused by imbalanced workloads. Third, the low operating costs of the strategy should not depend on excessive hardware support. Owing to physical limitations in circuit packaging, any extra hardware devoted to support for load balancing activity must impinge on the hardware devoted to numerical computation; we would hope that the extra needed hardware would not impinge too much, lest a machine do an excellent job of load balancing at the cost of having the arithmetic run at an unreasonably slow rate, and thereby perform worse than a machine that did a poorer job of load balancing but had faster arithmetic.

This dissertation presents a programming methodology for implementing, on multiprocessors, numerical algorithms exhibiting a *spatial localization property* that will be described shortly. Non-numerical computations, communication-intensive computations, or implementations on distributed multicomputer systems are not within the scope of this research. It is my thesis that a simple set of programming abstractions provided by a virtual machine (VM) can be effective in maintaining balanced workloads and, to some extent, in insulating the programmer from how a particular multiprocessing system handles communication. The abstractions have a low operating cost and require no extraordinary system support, either hardware or software, to operate efficiently. Though my approach requires that the programmer be aware of task decomposition and interprocessor communication activities, I believe that the amount of attention he must pay to them will be modest.

Many scientific continuum calculations use some form of spatial discretization to represent a continuous distribution of data. A specific example would be a method that takes finite differences on a grid to compute the electric field induced by a continuous charge distribution. Such algorithms work by transforming a collection of state variables through a succession of states, beginning with an initial state supplied as input. For an important subset of computational methods, the dependencies between discrete state-variables are localized in space. In this dissertation I formalize these properties in the following way:

(1) The computation consists of two parts: a local part and a relatively inexpensive global part.

(2) In the local part, each variable changes state only under the influence of variables lying within a small neighborhood. If we subdivide space by a fine mesh, then variables interact *locally* only if they they lie in boxes of the mesh whose indices differ by at most some specified small integer.

(3) The time taken to update the state of the variables in the local part can be estimated from the current state by an inexpensive auxiliary computation.

My programming methodology is intended for numerical algorithms that fit this simple model of spatial localization. Specific examples of calculations that fit this algorithmic model include:

- particle methods for fluid dynamics, astrophysics, and plasma physics (as described by Hockney and Eastwood [4]);
- explicit finite difference methods for hyperbolic partial differential equations, including adaptive grid methods (as described by Berger and Bokhari [3]);

Each state-variable is assigned an initial point in problem-space which will often, though not always, have a physical significance. State transformations are localized; though each variable may change state under the influence of the others, it is more strongly influenced by nearby variables than by distant ones, where the notion of nearness is a parameter of the algorithm. More formally, nearby points communicate far more frequently with respect to the computation done on them than do the more distant ones. As a result of this localization property, state transformations divide into two parts: a local part and a relatively inexpensive global part. Consider the local part. If problem-space is subdivided by a fine, regular lattice of boxes, then each variable communicates only with others that lie within a small surrounding square neighborhood of boxes. Variables that are not nearby are locally independent, but may communicate in an unconstrained manner during global computation. State variables are free to move about the lattice as the result of a state-change. The cost of updating a state variable may change from one state transition to the next, and vary non-uniformly over problem-space. The cost depends primarily on the distribution of locally dependent variables and may be predicted with a simple formula involving the current values of those variables.

Numerical algorithms that fit my model apply computational effort non-uniformly over the spatial domain of the problem, and the amount of effort they apply to a point in space is a function of time. Unless dynamically repartitioned when implemented on a multiprocessor, they could cause work to accumulate on some of the processors, with the result that the others would sit mostly idle.

1.2. A Programming Methodology

The user of my strategy for implementing spatially localized numerical algorithms adopts the following programming discipline:

(1) He employs a local-memory execution model for local computation, in which processors communicate by passing messages.

(2) He handles communication during global computation himself.

(3) He calls a few utilities that decompose the computation into rectangular subproblems and handle the local communication across the edges of the subproblems. The utilities' semantics are independent of the application and of the architecture on which the utilities have been implemented.

(4) He writes a few subroutines that depend solely on the application and passes them as arguments to the utilities.

The programmer who adopts such a discipline will be well rewarded by software that is relatively insensitive to changes in the number of processors, to interconnection structure, and to whether or not memory is shared. The parts of his software that pertain to task decomposition and localized communication will be isolated from the numerical parts of the code and confined to subroutines most of which he will use without ever reading. I believe that software modularized in this way will transport much more easily among different architectures than software written in a style in which decomposition and communication activities are woven inextricably within the program.

Computations that are spatially localized usually exhibit spatial coherency; locally interacting variables tend to form dense structures. In constraining computations to work within the framework of a mesh, we can take advantage of spatial coherency in order to reduce the cost of task decomposition and partitioning activities. Because they scatter heavily communicating variables unpredictably among processors, partitioning strategies that do not conserve spatial coherency usually incur high communication and administrative overheads. Variables that have been coherently partitioned need to communicate far less information per unit of computation to keep track of one another than do incoherently partitioned variables. In addition, such information may be communicated *en masse*, rather than a word at a time, to exploit any fast block modes of transfer provided by the underlying architecture. Another reason to exploit spatial coherency is that algorithms exist for partitioning lattices into coherent rectangular regions that can do a reasonable job of balancing workloads at low extra cost.

1.3. Results

To test out my hypothesis I have implemented my load balancing utilities on the Intel Personal Scientific Computer (iPSC), a hypercube-type multiprocessor, and on the Cray X-MP/416, a shared-memory architecture with vector arithmetic capabilities. I applied them to an implementation of Anderson's Method of Local Corrections [1], a two dimensional vortex method for incompressible inviscid flow. This particle method is typical of various problems that are spatially localized–thus appearing well suited to parallel computation– but which are hard to partition because they expend effort that varies non-uniformly both over the spatial domain of the problem and over time.

Parallel speedups of 22 were attained on the iPSC with 32 processors. The overhead of the load balancing utilities was less than 10% including interprocessor communication. This is surprisingly low in light of concerns voiced that the recursive bisection strategy would introduce high communication overhead on message-passing architectures [3,5]. Speedups of 3.6 were attained on the Cray with 4 processors. The overhead of load balancing utilities was less than 5%. Computations vectorized as well as they did on the best uniprocessor implementation, and executed floating point operations at the rate of 250 megaflops/second on 4 processors.

I speculate that the utilities will apply not only to particle methods like the Method of Local corrections but more generally to a diversity of mathematical physics calculations that fit my algorithmic model. Some elliptic problems may also fit this model owing to a local regularity property, noted by Colella, that will be discussed in a forthcoming publication [2].

This dissertation contains 7 chapters. The next chapter reviews past work. Chapter 3 briefly summarizes the important details of the model problem. Chapter 4 presents a set of programming abstractions and then applies them to the model problem, and may be thought of as a programmer's manual for my VM. It also presents an implementation strategy for the VM. Chapters 5 and 6 present experimental results obtained, respectively, from the iPSC and the Cray X-MP, and discuss some implementation details. Chapter 7 presents the major findings of the dissertation.

1.4. References

1. C. R. Anderson, ''A Method of Local Corrections for Computing the Velocity Field Due to a Distribution of Vortex Blobs,'' *J. Comput. Phys. 62*(1986), pp. 111-123.

2. S. B. Baden, M. J. Berger and P. Colella, ''Multitasking Strategies for Adaptive Fluid Dynamics Calculations,'' in preparation, 1987.

3. M. J. Berger and S. Bokhari, ''A Partitioning Strategy for Non-Uniform Problems on Multiprocessors,'' Technical Report 85-55, ICASE, Langley, Va., November 1985.

4. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*, McGraw-Hill, 1981.

5. S. McCormick and D. Quinlan, *Multilevel Load Balancing for Multiprocessors - An Outline*, University of Colorado, Denver, 1986. Grant proposal.

2

Past Work

Я не то что позабыла о былом, нет, я не могла этого забыть, я как-то отдалилась от него.

-- Чингиз Айтматов, Первый Учитель,
перевод с киргизского

... I did not forget the past, no. I couldn't have forgotten it. I sort of drifted away from it.

–Chingiz Aytmatov, "Duishen",
(translation into Russian from the Kirghiz)

2.1. Introduction

Until very recently – the early 1980's – most results concerning run-time partitioning were obtained from simulations. Few studies were conducted on working hardware. Most of these considered simple computations, such as successive overrelaxation (SOR), that use simple regular data structures and are not heavily compute bound. As a result, communication overhead was more troublesome than workload imbalance.

Since run-time partitioning entails assigning pieces of work to processors, the immediate question is how big the pieces should be. Task size, often referred to as task *granularity*, can range from microscopic– on the order of a single machine instruction– to very coarse– a process, say, executing billions of instructions. Oleinick's study [16] of a speech recognition system running on the C.mmp shared-memory multiprocessor gave early empirical evidence that load imbalance can be improved by decreasing the granularity and hence increasing the number of the tasks. Our explanation is fairly simple: the sums of the completion times of tasks can be made more even over the processors.

Oleinick's study also showed that the process of reducing task granularity will eventually reach a point of diminishing returns where the cost of managing the more numerous tasks will exceed any savings that result from having them more evenly balanced. This happens because as tasks shrink in size the fixed overhead in managing them becomes relatively expensive. We can also infer from this that fine-grained tasks do more communication relative to computation. We call this a *surface effect* and explain it in terms of a data dependency graph. Let the nodes of the graph correspond to the variables used in the computation, and the edges connect each pair of variables that interact. The partitioning process embeds a surface into the graph that cuts the edges, and communication overhead is roughly proportional to the number of cuts. As the number of tasks increases, the surface area of the embedded surface increases, and so does the number of cuts. Hence, communication overhead increases. Viewed differently, when surface effects are benign, nearby points are likely to be assigned to the same processor, so frequent communication between them will be inexpensive. Distant points will likely be assigned to different processors between which communication is expensive; but communication between such points turns out to be infrequent for a broad class of mathematical physics calculations of interest here that have the spatial localization property.

Having introduced the important issues, we next present a survey of run-time partitioning strategies. We roughly group the strategies into two categories, according to whether they balance workloads implicitly or explicitly.

2.2. Implicit Load Balancing

In an implicit strategy no attempt is made either to measure workload imbalance or to improve it. Instead, processors obtain work by sampling a pool of tasks. Although tasks may carry varying amounts of work, on average all processors will be equally loaded, so long as the work-pool is sufficiently large. The various implicit strategies differ in the way they construct the work-pool and divide it among the processors.

2.2.1. Non-Traditional Computer Architectures

An ideal multiprocessing system would handle all the details of task decomposition and communication for the user. Dataflow implementations [2,5] are examples of novel ''non-von Neumann'' systems that rely on special architectural and compiler support to distribute work automatically among the processors. See Treleaven's survey [21] for examples of others. Programs in dataflow are graphs of *partially ordered* operations that have no central program counter. Unlike the *totally ordered* operations specified in traditional uniprocessor systems, partially ordered operations can execute in an arbitrary order, so long as the data they need to execute is available. A dataflow machine therefore has the freedom to execute many operations concurrently. Nodes of a data flow graph have a microscopic granularity and represent simple primitive operations like addition or subtraction. For reasons previously stated, each processor will get a fair share of available work but for a price: the cost to administer to the fine grains of work can easily exceed the amount of work the grain does.

Gajski, Padua, and Kuck [9] have characterized the high overhead of dataflow in a different way: a dataflow machine is a very long pipeline; calculations must exhibit extreme levels of parallelism – on the order of several hundred independent instructions – to keep the pipe busy; only certain kinds of calculations in fact do so. As a result of their high overhead costs, dataflow implementations tend to require considerable hardware aimed to reducing the cost of administrative activities. This hardware could be unnecessary were the grains of work coarser, e.g. on the order of subroutines, but then the machine would have to balance workloads explicitly. We discuss one non-traditional architecture that uses an automatic, explicit strategy in section 2.3.1.

In an ideal dataflow multiprocessor, communication overhead as well as administrative overheads would be negligible. In classic dataflow implementations, however, communication is expensive: tasks, which are small and numerous, do a lot of communication relative to the computation they do. Part of the problem is that tasks that share the same processor cannot communicate via processor registers, as would the equivalent machine instructions on traditional architectures. This happens because there is no notion of explicit storage in dataflow: a task's state is bound to a processor only for the duration of task execution and leaves the processor once the task terminates. The information must travel expensively through to higher levels of the memory hierarchy to a special repository where other instructions look for their arguments. Gajski, Padua, and Kuck [9] have cited the lack of explicit storage as one of the major shortcomings of dataflow. Such problems have motivated Hwu, Patt, and Shebanow [17] to propose the HPS restricted data flow microarchitecture with special-purpose hardware support for allowing heavily dependent tasks to communicate through processor registers. Their strategy is loosely based on the Tomasulo algorithm [1] used in the IBM 360/91.

2.2.2. Processor Self-Scheduling

Processor self-scheduling is a popular technique for doing load balancing. Instead of relying on the operating system to make work assignments, each processor obtains work on its own by sampling a shared data structure. A common application is an iterative finite difference method that processes the rows of an array independently of one another, but which must finish with all the rows in one iteration before processing any rows in the next. A shared row counter keeps track of the next available row to be processed. When a processor finishes with a row, it samples and increments the row counter to get a new assignment. The sample and increment code executes as a critical section to ensure mutually exclusive access to the counter. When the counter reaches its limit, processors without work must wait at a barrier synchronization point for the active processors to finish. So long as the number of rows is large compared with the number of processors, the wait time usually won't be noticed.

In processor self-scheduling the programmer is aware of task decomposition activity. On commercial multiprocessor systems such as the Sequent Balance [20] and the Cray X-MP and Cray-2 multiprocessors [4], the programmer annotates his program with special commands that tell the compiler to emit the code required to implement the counters, critical sections, and barrier synchronization used in processor self scheduling.

Processor self-scheduling has also been considered for research multiprocessor systems such as the IBM RP3 [18], the University of Illinois's Cedar [23], and the NYU Ultracomputer [10]. All these systems support shared-memory-based communication. Processor self-scheduling applies primarily to shared-memory architectures. On message-passing architectures both the counters and the program data structures have to shuffle between the processors. This complicates the code and introduces a high communication overhead. Counters must be sent in short messages, which are expensive owing to a high message startup cost. When tasks shuffle between processors they must leave behind forwarding trails for other processors to follow. A processor must send and receive short messages to follow such trails in order to obtain data it needs from tasks that have migrated. And, because several tasks reside on each processor, communication overhead can be further compounded by the difficulty in assigning heavily communicating tasks to the same processor.

Processor self-scheduling can incur a high communication latency on shared-memory architectures because processors must concurrently access the same memory locations to obtain new work assignments. Kumar [13] has recently noted that certain memory access patterns can severly degrade the performance of some kinds of staging networks over which the processors access memory. Even on networks for which such hot spots may not be problematic, the effect of access conflicts– for example, on shared task queues– can be intolerable. In their studies of the MSPLICE circuit simulator running on the BBN Butterfly Multiprocessor, Jacob et al. [11] have found that use of multiple queues can help reduce memory contention delays, but at the cost of increasing the workload imbalance. They ran with up to 99 processors and obtained maximal speedups of about 25 on a machine that had no floating point hardware.

2.2.3. Scatter Decomposition

Fox [8] has advocated a scatter decomposition strategy for the Caltech Hypercube. The technique is currently being tried on diverse applications: a finite element method [15], ray tracing for computer graphics, and matrix decomposition. Unlike most load balancing strategies, the scattered decomposition strategy is static rather than dynamic. It begins by partitioning the domain into a fine lattice of tasks, called templates, that are far more numerous than processors, and then systematically scatters each processor's work assignments throughout the domain. Scattered decomposition for the special case of 2 processors corresponds to a red-black ordering on a checkerboard. The programmer may adjust the size of the templates, and hence the granularity of tasks, in order to trade off load imbalance against communication overhead due to surface effects.

Scatter decomposition works because each processor samples diverse regions of the problem and so obtains a fair cross section of a distribution of various task-sizes. The advantage of scattered decomposition lies in its simplicity: it partitions work statically and into simple shapes. So long as tasks don't interact over distances greater than the spacing of the templates, the underlying communication structure of the computation will involve only nearest neighbors in a mesh. This is a desirable because messages that travel only between nearest neighbors are both less expensive and easier to program than messages that involve intermediate hops. If the templates become too small, however, then message traffic may not always be simple, in which case communication would no longer be restricted to involve only nearest neighbors. In addition, as the tasks become smaller, they will send shorter, more numerous messages.

2.3. Explicit Load Balancing

Explicit load balancing strategies use a work metric as part of a decision-making process for assigning work to processors. The various strategies differ both in the work metric they use and in the process by which they arrive at work assignments. Unlike implicit methods, which split work into many fine-grained tasks, explicit methods tend to split work into relatively small numbers of coarse-grained chunks. As a result, explicit strategies can incur much lower overhead costs than implicit methods. Explicit load balancing strategies come in two varieties: indirect and direct. We discuss two indirect methods– pressure gradient methods and simulated annealing– and two direct methods– each based on recursive bisection.

2.3.1. Indirect Methods

Pressure Gradient Methods. One kind of indirect method computes local gradients in a ''work potential function'' and then shifts work incrementally in the direction of the gradients. To compute a local work gradient, each processor estimates the amount of work assigned to it, exchanges the information with nearest neighbors, and then differences the information. The signs of the resultant numbers identify which processors have excess work to give up and which suffer from a work deficit, and the magnitude gives the intensity of these quantities. Such a strategy has been used in Keller's novel rediflow architecture [12], but also in more traditional implementations. Saltz [19] has used the technique for block iterative methods running on a simulated shared-memory multiprocessor with ''substantial local memory.'' Swensen and Dippé [6] have used it for a three dimensional ray tracing graphics application, used to produce realistic artificial images, on a paper design of a mesh-connected multiprocessor.

Pressure gradient strategies are attractive because of their low overhead costs: (1) they are distributed and can exploit the concurrent resources of the machine; (2) they incur modest amounts of communication since loads shift gradually among processors. However, use of a distributed load balancing strategy complicates the process of deciding which work to shuffle among the processors; decisions must be coordinated to avoid unstable work assignments that would periodically starve some processors while overloading others. Pressure gradient strategies that interleave load balancing activity with computation have the added disadvantage of having to maintain forwarding trails when implemented on message-passing architectures, as with processor self-scheduling.

Simulated Annealing. Simulated annealing is a probabilistic technique that has been successfully applied to various optimization problems, such as VLSI circuit placement, over the past 30 years. Recently Fox [8] and Williams [22] have discussed another application for simulated annealing – dynamic load balancing on multiprocessors. In simulated annealing, the first step is to subdivide the computational domain finely into numerous subproblems and then to initially assign the subproblems randomly to the processors. An annealing algorithm then probabilisticly readjusts the work assignments until all processors have roughly the same amount of work to do. The algorithm is biased to ignore work reassignments that would worsen workload imbalance.

The criteria for selecting desirable reassignments can be expressed as minimizing an objective function describing the energy of a system of particles. We can think of the computational elements in the calculation as charged particles that interact within a ''processor space'' that is distinct from the spatial domain of the problem. Particles repel one another at close distances and will tend to disperse evenly among the processors. There generally exists more than one balanced configuration of particles, though some configurations incur a lower communication overhead than others. If communication overhead were of concern, then the objective function could be augmented by a term that took into account communication costs. This term would consist of long-range attractive forces to prevent work from being moved off a processor if, as a result of the move, communication incurred during the course of computation would increase.

It is not clear how much effort is required to construct the Hamiltonian nor to evaluate it. Indeed, Fox alludes to these problems [8]:

> " ... simulated annealing is a non-trivial undertaking; if we could find a simple method which gave decompositions almost as good, we would be happy".

With simulated annealing, partitions can take on irregular shapes. The software required to handle such shapes can be cumbersome, complicating the application-code. These shapes are also prone to surface effects that increase communication overhead. Though the use of simpler shapes such as rectangles could reduce communication overhead, how severely this would impede the annealer's ability to balance workloads isn't clear.

2.3.2. Direct Methods

In direct methods the computation is mapped onto a regular lattice. The lattice is often rectangular though other tessellations such as hexagons could be used if appropriate. For the case of rectangular tessellations the task is to partition the lattice into rectangular regions, using a work estimate mapping that gives the cost of computing on an arbitrary sublattice. Direct methods for balancing workloads have a major advantage over indirect methods in that tasks they generate tend to form simple convex polygonal shapes without holes, and therefore tend to suffer from only modest surface effects. The reason is that the load balancer knows how to produce the smallest possible number of "nice" shapes (equal to the number of processors) that carry roughly the same amount of work. Load imbalance could probably be lowered if work were split into a swarm of tasks, or if tasks could have ragged shapes, but probably not enough to justify the effort.

We discuss three strategies based on recursive bisection. Recursive bisection works by splitting the domain into two parts that represent roughly equal amounts of work, and then recursively splitting each part until the desired number of subproblems have been rendered. The strategy has been applied to the traveling salesman problem and various problems in computer graphics. More recently, Berger and Bokhari applied it to a two-dimensional adaptive mesh refinement calculation for hyperbolic partial differential equations [3]. Analytic results were discussed, in particular the cost of communication on various kinds of processor interconnection topologies.

McCormick and Quinlan [14] propose a variant of recursive bisection called multi-level load balancing. Their innovation is to implement recursive bisection concurrently and to incorporate communication loading into the work estimation mapping used to determine a fair subdivision of labor. They propose the following 2-stage partitioning process that generalizes to higher-dimensional problems. Assume P processors that form a perfect square. First recursively bisect the domain into $\sqrt{P}$ strips. Then apply the procedure to each strip concurrently, making orthogonal cuts that split each strip into $\sqrt{P}$ pieces.

Fox [7] discusses the Orthogonal Recursive Bisection strategy for decomposing sparse matrix problems. Unlike the first two recursive bisection strategies, ORB partitions space into irregular shapes, and may not apply to applications that work best when partitioned into more simple shapes like rectangles. Fox shows that ORB partitionings incur a favorable communication overhead as compared to optimal partitionings produced by simulated annealing, and for a much lower cost.

2.4. Comparison with Our Approach

We take a direct approach to dynamically balancing workloads that uses recursive bisection to handle task decomposition. Though recursive bisection has already been discussed in the literature, past applications of the strategy have been primarily theoretically oriented, i.e. what kind of communication overhead would be incurred on multiprocessor xyz, and how might it be avoided. In contrast, our results are based on empirical measurements of a substantial calculation running on real multiprocessors.

How recursive bisection works, or how it may be efficiently implemented is relatively unimportant in this study. Any strategy that is fast and accurate, and that can render spatially coherent partitionings would be equally well-suited to the task at hand. We are more concerned with the consequences of using run-time data partitioning to do dynamic load balancing– with communication and programming methodology– than with the mechanics of the partitioning process itself. Our contribution is a programming methodology that allows the user to remain aloof from dynamic task decomposition, and the communication that comes as a side effect. The user of such a methodology can write software that runs on different kinds of multiprocessor systems without necessarily encountering major difficulties in reprogramming for each new system.

2.5. References

1. D. W. Anderson, F. J. Sparacio and F. M. Tomasulo, ''The IBM system/360 Model 91: Machine Philosophy and Instruction-Handling,'' in *Computer Structures: Principles and Examples.*, D. P. Siewiorek, C. G. Bell and A. Newell (editor), McGraw-Hill, New York, 1982, pp. 276-292. Also appeared in the *IBM Journal*, Vol. 11, Jan. 1967, pp. 8-24..

2. Arvind and R. E. Bryant, ''Design Considerations for a Partial Differential Equation Machine,'' *Scientific Computer Information Exchange Meeting*, Livermore, California, 12-13 September 1979, pp. 94-102.

3. M. J. Berger and S. Bokhari, ''A Partitioning Strategy for Non-Uniform Problems on Multiprocessors,'' Technical Report 85-55, ICASE, Langley, Va., November 1985.

4. *CRAY X-MP Multitasking Programmer's Manual*, Cray Research, Inc., March 1986. Order number SN-0222.

5. J. B. Dennis, G. Gao and K. W. Todd, ''Modelling the Weather with a Data Flow Supercomputer,'' *IEEE Trans. Comput. C-33*,7 (July 1984), pp. 592.

6. M. E. Dippé and J. A. Swensen, ''An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis,'' *SIGGRAPH '84 Conference Proceedings*, Minneapolis, July 1984, pp. 149-158.

7. G. C. Fox, ''Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube,'' C3P-327, Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, California, July 1986.

8. G. C. Fox and S. W. Otto, ''Concurrent Computation and the Theory of Complex Systems,'' C3P-255, Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, California, March 1986. Also CALT-68-1343.

9. D. D. Gajski, D. A. Padua and D. J. Kuck, ''A Second Opinion on Data Flow Machines and Languages,'' *Computer*, February 1982, pp. 58-69.

10. A. Gottlieb, B. D. Lubachevsky and L. Rudolph, ''Coordinating Large Numbers of Processors,'' *Proc. 1981 Intl. Conf. on Parallel Processing*, , pp. 341-349.

11. G. K. Jacob, A. R. Newton and D. O. Pederson, ''An Empirical Analysis of the Performance of A Multiprocessor-Based Circuit Simulator,'' *ACM/IEEE Design Automation Conference*, June 1986.

12. R. M. Keller and C. H. Lin, ''Simulated performance of a Reduction-Based Multiprocessor,'' *Computer*, July 1984, pp. 70.

13. M. Kumar and G. F. Pfister, ''The Onset of Hot Spot Contention,'' *Proc. 1986 Intl. Conf. on Parallel Processing*, August 1986, pp. 28-34.

14. S. McCormick and D. Quinlan, *Multilevel Load Balancing for Multiprocessors - An Outline*, University of Colorado, Denver, 1986. Grant proposal.

15. R. Morison and S. Otto, ''The Scattered Decomposition for Finite Elements,'' C3P-286, Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, California, May 1985.

16. P. N. Oleinick, ''The Implementation and Evaluation of Parallel Algorithms on C.mmp,'' CMU-CS-78-151, Dept. of Computer Science, Carnegie Mellon University, November 1978. Ph. D. Dissertation.

17. Y. Patt, W. Hwu and M. Shebanow, ''HPS, A New Microarchitecture: Rationale and Introduction,'' *Proc. of the 18th Annual Workshop on Microprogramming*, Pacific Grove, California, December 1985.

18. G. F. Pfister and al., ''The IBM RP3 Introduction and Architecture,'' *Proc. 1985 Intl. Conf. on Parallel Processing*, August 1985, pp. 764-771.

19. J. H. Saltz, *Parallel and Adaptive Algorithms for Problems in Scientific and Medical Computing*, Dept. of Computer Science, Duke University, 1985. *Ph. D. Dissertation.*

20. *Balance Guide to Parallel Programming*, Sequent Computer Systems Inc., Beaverton, OR, June 1986. Order Number 1003-42508 Rev. A.

21. P. C. Treleaven, ''Data-Driven and Demand-Driven Computer Architecture,'' *Computing Surveys 14*,1 (March 1982).

22. R. D. Williams, ''Minimisation by Simulated Annealing: Is Detailed Balance Necessary?,'' C3P-354, Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, California, 1986.

23. C. Zhu and P. Yew, ''A Synchronization Scheme and its Applications for Large Multiprocessor Systems,'' *Proc. 4th International Conf. on Distributed Computing Systems*, May 1985, pp. 486-493.

3

A Model Computation

From the beginning it was never anything but chaos: it was a fluid which enveloped me, which I breathed in through the gills.

–Henry Miller, Tropic of Capricorn

3.1. Mathematical Background

The model calculation solves a time dependent, non-linear partial differential equation that arises in fluid mechanics– the *vorticity-stream function formulation* of Euler's equation for two dimensional, incompressible inviscid flow.

Let $\mathbf{u}(\mathbf{x}(t),t)$ be the velocity of the fluid at position $\mathbf{x}(t)$ at time t. Owing to the incompressibility condition, $\nabla \bullet \mathbf{u} = 0$, $\mathbf{u}$ may be expressed in terms of the *stream function* ψ:

$$\mathbf{u} = (u, v) = (-\partial\psi/\partial y \ , \partial\psi/\partial x) \tag{3.1a}$$

where u and v are the x and y components of velocity. Finally define *vorticity*, ω, as the curl of velocity:

$$\omega = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}. \tag{3.1b}$$

Applying (3.1a) to the (3.1b):

$$\omega = -\left[\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} \right] \tag{3.1c}$$

The Euler equations describe the motion of an incompressible inviscid fluid. The *vorticity stream function form* of the equations is:

$$\frac{\partial \omega}{\partial t} + \mathbf{u} dp\, \nabla \omega = \frac{D\omega}{Dt} = 0 \tag{3.2a}$$

$$\omega = -\Delta \psi \text{ in } \Omega, \tag{3.2b}$$

where $\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} dp\, \nabla$ is the material derivative, and $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the two dimensional Laplacian operator. The first equation is a hyperbolic equation, the second elliptic. The flow satisfies free-space boundary conditions: $\mathbf{u} = 0$ at $\mathbf{x} = \infty$. This will be satisfied if

$$\psi \sim C \log(r), \quad C = \frac{1}{2\pi} \int_{\mathbb{R}^2} \omega(\mathbf{r}) d\mathbf{r} \tag{3.2c}$$

The coupled equations (3.2) completely specify the flow. Equation (3.2a) governs the evolution of ω; vorticity is transported by its own velocity field **u**, and it does not decay. Equation (3.2b) tells how to find that velocity field **u** given ω: solve (3.2b) for ψ subject to free-space boundary conditions (3.2c); then differentiate ψ using (3.1a). For a thorough discussion of these equations, see Chorin and Marsden's introductory text on fluid mechanics [6].

3.2. The Calculation

A *vortex blob method* [5] will be used to solve the equations (3.2). It describes the flow of the fluid by computing the motion, over a series of timesteps, of a set of particle-like computation elements called "vortex blobs." The vortices in vortex methods are treated like the particles in astrophysics or plasma physics calculations–they move under mutual interaction. However, they are not particles of fluid but an artifact of a mathematical discretization of the problem–tagged *locations* that carry information about the vorticity of the fluid. Vortex methods have many desirable properties for solving incompressible flow problems; see Chorin's early paper [5] on the vortex blob method for the details, and Leonard's survey [11] for additional information.

The vortex blob method concentrates the vorticity field into discrete patches of vorticity represented by the blobs:

$$\omega(\mathbf{x}(t),t) = \sum_{j=1}^{N} K_\sigma(\mathbf{x}_j(t) - \mathbf{x}(t))\,\omega_j, \tag{3.3}$$

where $\mathbf{x}_j(t)$ is the position of the jth vortex blob at time t, ω_j is a signed constant giving its *strength*, K_σ is called called the vortex *core function*, and σ is the *cutoff radius*. The strength ω_j is the total vorticity collected from a small surrounding neighborhood and concentrated on the vortex blob of radius σ, and must be a constant to satisfy the vorticity evolution equation (3.2a). The core function is determined by accuracy considerations. The one used here is given by Chorin [5]:

$$K_\sigma(\mathbf{x}) = \begin{cases} (2\pi\sigma r)^{-1} & r < \sigma \\ 0 & r \geq \sigma \end{cases} \tag{3.4}$$

where $r = |\mathbf{x}|$. See Beale and Majda [3] or Hald [9] for other kinds of core functions. The vorticity vanishes outside the cutoff radius.

Using (3.2b) it can be shown that the *velocity* field induced by the vorticity distribution of (3.3) may be expressed as:

$$\mathbf{u}(\mathbf{x}(t),t) = \sum_{\substack{j=1 \\ \mathbf{x}_j \neq \mathbf{x}}}^{N} \chi_\sigma(\mathbf{x}_j(t) - \mathbf{x}(t))\,\omega_j, \tag{3.5}$$

where χ_σ is called an *influence function*. This function is like a basis function for the purpose of decomposing the velocity field and it is determined by the choice of core function. The influence function that corresponds to the core function of (3.4) is:

$$\chi_\sigma(\mathbf{x}) = \begin{cases} (-y, x)/2\pi r\sigma & r < \sigma \\ (-y, x)/2\pi r^2 & r \geq \sigma \end{cases} \tag{3.6}$$

where $\mathbf{x} = (x, y)$, $r = |\mathbf{x}| = (x^2+y^2)^{1/2}$.

The vortex method consists of calculating the motion of the centers of a discrete set of vortices whose strengths never change. A system of ordinary differential equations describes the motion of the vortices:

$$\frac{d}{dt}\mathbf{x}_i(t) = \sum_{\substack{j=1 \\ i \neq j}}^{N} \chi_\sigma(\mathbf{x}_j(t) - \mathbf{x}_i(t))\,\omega_j, \; i = 1, \cdots, N, \; i \neq j. \tag{3.7}$$

Equations (3.7) are an approximation to the Lagrangian form of (3.2a); for a derivation see Anderson and Greengard [1]. The differential equations may be discretized in time with a second order Runge-Kutta time integration scheme (Heun's method). Specifically, given the positions of the N vortices after the kth timestep by the vector $\mathbf{X}^k = (\mathbf{x}_1(k\Delta t), \ldots, \mathbf{x}_N(k\Delta t))$, where Δt is the timestep, we are to compute $\mathbf{X}^{k+1}$, the positions at the k+1st timestep, in two steps:

$$\mathrm{Q}_1 = \Delta t\, \mathrm{U}(\mathrm{X}^k; k\Delta t) \tag{3.8a}$$

$$\mathrm{Q}_2 = \Delta t\, \mathrm{U}(\mathrm{X}^k + \mathrm{Q}_1; (k+1)\Delta t) \tag{3.8b}$$

$$\mathrm{X}^{k+1} = (\mathrm{X}^k + \mathrm{Q}_1) + \frac{(\mathrm{Q}_2 - \mathrm{Q}_1)}{2} \tag{3.8c}$$

where Q_1 and Q_2 are N-element displacements, and $\mathrm{U}(\mathrm{X}; k\Delta t)$ is the vector of velocities of the N vortices located at positions X at time $k\Delta t$. In the first step, called the predictor step, the vortices move by the amount Q_1, which has magnitude Δt. In the second step, called the corrector step, the vortices move by a much smaller amount, $\frac{\mathrm{Q}_2 - \mathrm{Q}_1}{2}$, which has magnitude Δt^2. Generally, $\Delta t \ll 1$.

The standard method for evaluating the velocities in (3.7) evaluates all $N(N-1)$ summands, and has a running time of that is quadratic in the number of vortices. The method is expensive for larger problems when the vortices number ten thousand or more. For instance, a single velocity field evaluation of 12848 vortices takes about 1 minute on one processor of a Cray X-MP/416 (Table 3.1). Production runs comprising hundreds of timesteps–with 2 velocity field evaluations per timestep required for second order accuracy in time–would take several hours. The direct method is therefore impractical when $N \gtrsim 10{,}000$.

3.3. Anderson's Method of Local Corrections

Anderson's Method of Local Corrections, abbreviated as the "MLC," may be used to accelerate the computation of (3.7) with reasonable accuracy [2]. It can be ten to fifty times faster than the direct method when the vortices number 10^4 or more. The MLC divides vortex interactions into two components: (1) N-body interactions computed accurately for vortices close enough to one another; (2) long-range interactions approximated by solving a discrete Poisson equation on a finite-difference grid. When they number in the thousands or more, the calculation spends most of its time computing local N-body interactions between nearby vortices. Vortices that are not so close to one another interact indirectly through the relatively inex-

N	Time (sec)
1590	1.77
3180	6.99
6414	28.2
12848	113

Table 3.1. The direct method has an $O(N^2)$ running time. Times are reported for a single timestep of the two-FAV problem used throughout this dissertation. Vortices are distributed uniformly within two circular patches of radius 0.13. The centers are separated by 0.25. Times were measured on a single processor of a Cray X-MP/416.

pensive global finite difference computation. The MLC is much faster than the naive method because it exploits a locality property inherent in the elliptic part of the Euler's equation (3.2b). A logarithmic potential that governs the motion of the vortices diminishes rapidly with increasing distance from a vortex so that distant interactions may effectively be averaged over large distances. The MLC is more accurate than Christiansen's vortex-in-cell method [8], which doesn't treat local interactions specially.

The MLC computes the velocity on a set of vortices in two steps:

(1) Solve Poisson's equation for an approximate velocity field on the finite difference grid and interpolate to the centers of the vortices.

(2) Locally correct the velocity of each vortex by undoing the portion of the approximate velocity due to nearby vortices and substituting in their place local direct interactions.

To simplify matters consider only the the x component of velocity, u; the y component, v, is computed in a similar fashion.

Step 1 calculates $\tilde{u}$, an approximation to u on a grid, in four sub steps. Assume a single vortex centered at the origin. This simple case generalizes to arbitrary collections of vortices by superposition and linearity. The first step divides into 4 parts:

(1.a) Set up the right hand side for Poisson's equation exclusive of Dirichlet boundary conditions.

(1.b) Set up the boundary conditions for (1.a).

(1.c) Solve.

(1.d) Interpolate.

Setting up the right hand side entails smearing the influence of the vortex onto a small square neighborhood of the mesh centered on the box covering the vortex. Specifically, evaluate the spread function $g_D(ih, jh)$ at a discrete set of points on a grid:

$$g_D(ih, jh) = \begin{cases} \Delta^h u(ih, jh) & 0 < |ih| \text{ and } |jh| \leq D \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

where Δ^h is the discrete Laplacian, and u is the x component of the *point* vortex velocity function, modified to avoid self-induced velocities at the origin (the blob velocity function could also be used but is more expensive to compute):

$$\mathbf{u}(x, y) = \begin{cases} (-y, x)/2\pi r^2 & r > 0 \\ 0 & r = 0 \end{cases} \tag{3.10}$$

Computing the Dirichlet boundary conditions for the right hand side entails evaluating the point vortex velocities at the discrete positions on the grid's boundary. This grid is then passed to the Poisson solver. The first step finishes by interpolating the velocity field returned by the solver onto the center of the vortex. We used the Lagrange interpolation formula for complex analytic functions used by Anderson [2].

The second major step of the MLC does the local corrections on the vortices distributed among the bins. Consider correcting the vortices lying in a single bin. These vortices are influenced by others lying in the correction neighborhood, a square region of space of radius C surrounding the bin (Although this square neighborhood is slightly larger than a circular neighborhood of radius C that would be good enough, the extra vortices included there can't hurt the accuracy of the calculation nor slow it down much). The local corrections divide into two parts:

(2.a) Undo the effect of the smearing for all the vortices in the correction neighborhood of that bin.

(2.b) Compute local interactions with those same vortices.

Step (2.a) entails, for each bin, locally reconstructing the velocity field induced by the vortices in the correction neighborhood, and interpolating onto the bin's vortices. These interpolated velocities are subtracted from what has already been accumulated on the vortices to cancel out the locally induced component of the approximate velocity field.

The local interactions in the MLC are computed in much the same way as direct interactions involving charged particles. The MLC requires that a "correction radius" C be chosen by the method's user to distinguish nearby vortices, closer than C, from distant ones. These nearby vortices, once identified at any time, are the ones that participate in the local part of the computation. To speed up the search for nearby vortices, space is customarily subdivided into a few thousand fairly small bins, and then the vortices are sorted into the bins, as shown in Figure 3.1. This technique is discussed in the text [10] on particle-based calculations by Hockney and Eastwood. The local interactions are handled a bin at a time. Convenience dictates setting the correction radius C to a small multiple of the bin width, say 1 or 2. Let C now stand for that multiple. Then, all the vortices influencing bin (i,j) are found in the bins whose indices differ from i and j by integers no bigger than C. These bins form shaded regions in Figure 3.1 where $C = 1$. In practice the bins used in the MLC are much smaller than shown in the figure, so the vortices interact directly only over short distances. The pseudo-code of Figure 3.2 summarizes the MLC.

The MLC runs in time $O(N^2/M^4)+O(NM)+O(f(M))$, where N is the number of vortices, M is the linear dimension of the grid used to solve Poisson's equation, and f is bounded by a polynomial of degree 3. The first term gives the running time for the local interactions, the second for the evaluation of boundary conditions, and the third for the Poisson solver. Usually $N \gg M$, in which case the last two terms may be ignored. This happens because as N increases, the cost of doing the local corrections grows faster than the cost of doing the finite difference computations. This is shown in Table 3.2.

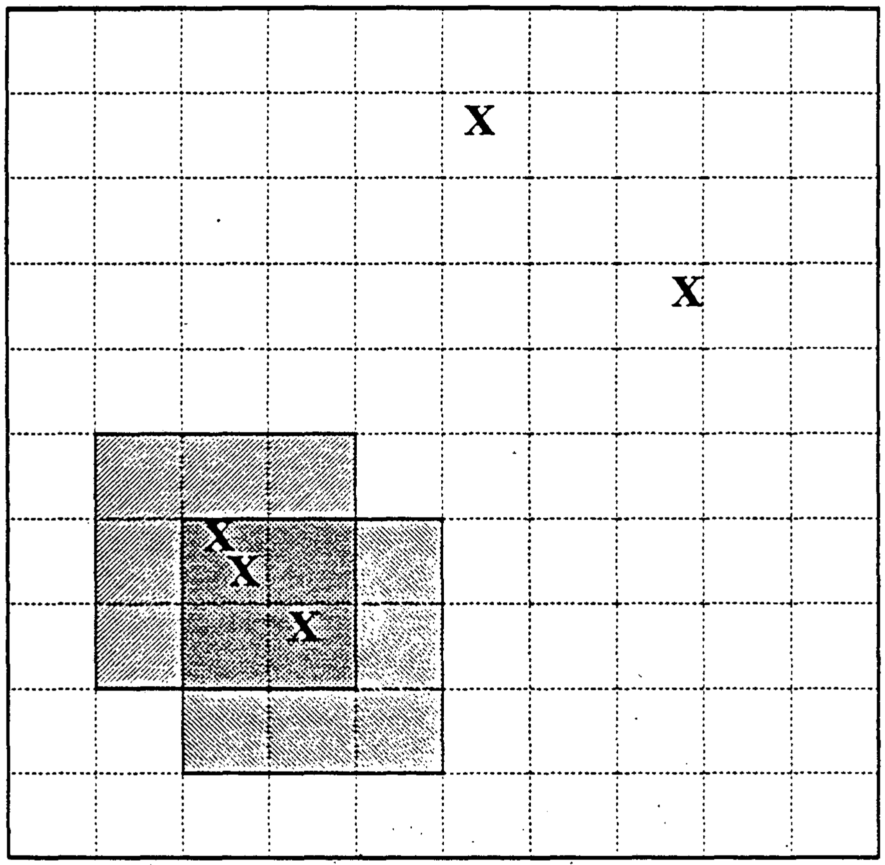


Figure 3.1. Vortices, shown as x's, get sorted into bins, demarcated here by hyphenated lines. Associated with each bin is a pointer to a list of vortices that lie within the bin's region of space. In practice, the bins are much smaller and more numerous than shown here. Each shaded region designates the domain of dependence for the bin at the region's center, with $C = 1$. The region contains all the vortices that influence those in the bin, and includes the bin itself.

N	C	Time (sec)	% Interact	% BCs	% Solver
390	1/15	0.30	25.2	1.48	3.44
796	1/15	0.62	36.2	2.23	1.87
1590	1/15	1.39	51.8	2.96	1.26
3180	1/15	3.49	69.2	3.17	0.69
6414	1/30	6.51	54.8	4.97	1.18
12848	1/30	16.4	70.5	5.04	1.24
25702	1/60	29.3	53.3	7.95	3.85

Table 3.2. A breakdown of the times spent in various phases of the MLC calculation reveals that the local interactions computation predominates when the vortices number in the thousands. For $N > 3180$, we roughly scaled the interaction distance C with the square root of the number of vortices. This reduces the running time of the computation from a quadratic function of N to an almost linear function, and decreases the fraction of time spent computing local interactions. The time column gives the time to compute a single timestep. Measurements were taken from a single processor of a Cray X-MP/22 running version 114g of the CFT compiler (this is a dialect of CFT used at the National Magnetic Fusion Energy Computer Center at the Lawrence Livermore National Laboratory). This Cray runs about 15% slower than the X-MP/416 we used.

3.4. Accuracy and Parameter Selection

The MLC is a subroutine that evaluates the velocity field at the centers of a collection of vortices. Like most particle methods, vortex calculations involve integrating the positions of the vortices with respect to time, i.e. ''pushing'' them over a discrete series of timesteps, doing one or more velocity field evaluations per timestep. As previously mentioned, the time integration scheme we used is accurate to second order, and does two velocity field evaluations per timestep. In addition to computing local interactions, the MLC also does some finite difference computations, including a global calculation to solve Poisson's equation. All finite difference calculations were accurate to fourth order.

A variety of parameters play an important role in determining the speed and the accuracy of the MLC. There is the cutoff radius σ, the finite difference mesh spacing h, the correction radius C, and the spreading radius D. For all the test problems used here vortices are initially set up on a lattice with spacing h_v. The following parameter settings were used for all the runs:

```
--      Compute the velocities induced on a set of vortices sorted into bins
--      Each bin is a pointer to a collection of vortices
--      The calculation uses a finite difference mesh $g_D$ to speed up the computation
--      h is the spacing of the mesh
--      C is the correction distance, measured in mesh boxes, and is chosen by the user
--      Ignore any indices lying outside the domain

--      Compute an approximate velocity field $\bar{u}$ on the grid $g_D$
--      First set up the RHS for the solver
foreach bin $(i,j)$
        foreach vortex $(\mathbf{x},\omega)$ in bin $(i,j)$

--      $\Delta\mathbf{u}$, the Laplacian of the velocity induced by a vortex,
--      vanishes with increasing distance from the vortex
--      We approximate this quantity in two parts: $\Delta^h \bar{u}$ inside a square neighborhood
--      centered on the box covering the vortex, and zero outside.
--      The neighborhood is $2D+1$ boxes on a side.
--      $\chi_{point}(\mathbf{r})$ is the velocity induced at $\mathbf{r}$ by a point vortex of unit strength at the origin
--      $\chi_{blob}(\mathbf{r})$ is the velocity induced at $\mathbf{r}$ by a vortex blob of unit strength at the origin
                $g_D(i,j) := g_D(i,j)+\Delta^h\chi_{point}(\mathbf{y}-\mathbf{x})*\omega$
                where
                  $\mathbf{y}=(-0.5+(i+k)*h,-0.5+(j+l)*h)$,
                  $(k,l)\in[-C-1 \dots 0 \dots C+1]\times[-C-1 \dots 0 \dots C+1]$

--      Compute the boundary conditions
                foreach (i,j) on a boundary point y
                        $g_D(i,j) := g_D(i,j)+\chi_{point}(\mathbf{y}-\mathbf{x})*\omega$

--      Find $\bar{u}$ such that $\Delta\bar{u}=g_D$
        call solver($g_D$,$\bar{u}$)
-- Interpolate
foreach bin $(i,j)$
        foreach vortex $(\mathbf{x},\mathbf{u})$ in bin $(i,j)$
                update $\mathbf{u}$ by interpolating from $\bar{u}$ onto x and adding

-- Local corrections
foreach bin $(i,j)$
        foreach vortex $(\mathbf{x},\mathbf{u})\in$ bin $(i,j)$
                foreach vortex $(\mathbf{y},\omega)\in$ bin $(k,l)$
                where $(k,l)\in[-C+i \,..\, i \,..\, C+i\,]\times[-C+j \,..\, j \,..\, C+j\,]$

--      Subtract off $\bar{u}$ due to nearby vortices
                        update $\mathbf{u}$ by interpolating $\chi_{point}(\bar{\mathbf{x}}-\mathbf{y})*\omega$
                                onto x and subtracting
                        where $\bar{\mathbf{x}}$ ranges over the 5-point stencil
                        of points centered on bin$(k,l)$

--      Compute local interactions
                        update $\mathbf{u}$ by adding $\chi_{blob}(\mathbf{x}-\mathbf{y})*\omega$
```

Figure 3.2. Anderson's Method of Local Corrections computes the velocity field induced at the centers of a collection of vortices.

σ	$h_v^{0.75}$
C	$2h$
D	$2h$

The values of h_v we used varied in magnitude from 10^{-2} to 10^{-3} while h ranged from 1/30 to 1/240. The timestep Δt used in our time integration scheme ranged from 5×10^{-2} to 6.25×10^{-3}. The settings for σ, C, and D Δt seem reasonable in light of other results by Anderson [2], Anderson and Greengard [1], Beale and Majda [4], Perlman [12], and Chorin [7].

The spreading radius D affects the accuracy of the global part of the solution and can be no smaller than σ. It cannot hurt to make D slightly too large, although the cost to set up the right hand size for Poisson's equation increases roughly as D^2. C can be no smaller than σ because the finite difference approximation to u isn't accurate inside the core of the vortex ($r < \sigma$) and must be corrected there. The answers get more accurate with increasing C, but for a price; the cost to do the local corrections varies roughly as the square of C, depending on the local density of vortices.

We did a three parameter study to gain a better understanding of the appropriate settings for some of the parameters. We varied h_v, h, and Δt over runs of a single test problem with a known exact solution, and compared the known solution with that computed by the MLC. The initial vorticity distribution for the test problem is radially symmetric and vanishes outside a circle of radius 0.25 centered about the origin. The vortices are distributed on a uniform mesh of points, with the strength of a vortex at $\mathbf{x} = (x,y)$ given by $4\pi(1-4(x^2+y^2))^7$. The vortices rotate about the origin, with a angular velocity that increases with decreasing distance from the origin. See Perlman [12] for the details. The runs were stopped when the fastest moving vortices had rotated one revolution. The L_2 norm of the error was reported. This error is defined as:

$$h_v \left[\sum_{j=1}^{N} (\text{computed } (x_j) - \text{exact } (x_j))^2 \right]^{1/2} \tag{3.11}$$

Three different values of h_v were used and correspond with N, the number of vortices, as follows:

N	h_v
1005	0.014
4020	0.007
16043	0.0035

The results of the study are presented in Table 3.3, and tell us three things:

(1) $\Delta t / h_v$ must remain constant, i.e. the time step must decrease as the vortices increase in number. The choice of appropriate time step can be inferred by moving across a row and noting when decreasing the timestep doesn't appreciably decrease the error.

(2) The correction radius C scales with $\sigma = h_v^{0.75}$. Accuracy is insensitive to changes in C, so long as $C > \sigma$. This can be inferred by moving down a column and keeping the number of vortices fixed.

(3) Accuracy improves significantly as the vortices are initially spaced more closely, so long as an appropriate timestep has been chosen.

The second result is significant since it tells us that the number of local interactions need not necessarily grow as N^2, if C and hence h are decreased with h_v. This is indeed what has been observed in practice, as shown in Table. 3.4.

N	h	Δt				
		0.1	0.05	0.025	0.0125	0.00625
1005	1/30	9.090×10^{-3}	4.566×10^{-3}	3.497×10^{-3}	3.243×10^{-3}	3.175×10^{-3}
	1/60	9.082×10^{-3}	4.556×10^{-3}	3.488×10^{-3}	3.233×10^{-3}	3.165×10^{-3}
4020	1/30	7.659×10^{-3}	3.648×10^{-3}	1.476×10^{-3}	1.216×10^{-3}	–
	1/60	7.661×10^{-3}	3.650×10^{-3}	1.478×10^{-3}	1.218×10^{-3}	–
	direct	7.658×10^{-3}	3.646×10^{-3}	1.474×10^{-3}	1.214×10^{-3}	–
16043	1/30	7.172×10^{-3}	1.971×10^{-3}	7.479×10^{-4}	4.828×10^{-4}	4.219×10^{-4}
	1/60	7.173×10^{-3}	1.974×10^{-3}	7.500×10^{-4}	4.850×10^{-4}	4.241×10^{-4}
	1/120	7.174×10^{-3}	1.974×10^{-3}	7.506×10^{-4}	4.856×10^{-4}	4.247×10^{-4}

Table 3.3. Results from a three parameter study show the effect of varying certain simulation parameters on the accuracy of the computed solution. For comparison, a result for the direct method is presented for the single case of 4020 vortices; accuracy is not significantly better than it is with the MLC.

N	C	Time/timestep (min)		MLC Speedup
		MLC	Direct	
12848	1/30	0.25	1.9	7.6
25702	1/60	0.48	*7.6*	16
51376	1/60	1.3	*30*	23
102822	1/120	3.2	*122*	55

Table 3.4. The correction distance C scales with the initial spacing of the vortices h_v. Use of this scaling procedure can reduce the quadratic growth in the running time of the calculation to nearly a linear function of N, the number of vortices. Unlike the MLC, the direct method is impractical for the large problems shown here; its running time is quadratic in N. The italicized times for the direct method, when $N > 12848$, were extrapolated from the running times for smaller N that were presented in Table 3.1. For 102822 vortices the MLC is 55 times faster than the naive method. Times are reported for a single timestep of the two-FAV problem and were measured on a single processor of a Cray X-MP/416.

3.5. References

1. C. Anderson and C. Greengard, ''On Vortex Methods,'' *SIAM J. Numer. Anal. 22*,3 (June 1985), pp. 413-440.

2. C. R. Anderson, ''A Method of Local Corrections for Computing the Velocity Field Due to a Distribution of Vortex Blobs,'' *J. Comput. Phys. 62*(1986), pp. 111-123.

3. J. T. Beale and A. Majda, ''The Design and Numerical Analysis of Vortex Methods,'' PAM-48, Center for Pure and Applied Mathematics, University of California, Berkeley, 1981.

4. J. T. Beale and A. Majda, ''Vortex Methods. II: Higher Order Accuracy in 2 and 3 Dimensions,'' *Math. Comput. 39*,159 (July 1982), pp. 29-52.

5. A. J. Chorin, ''Numerical Study of Slightly Viscous Flow,'' *J. Fluid Mech. 57*(1973), pp. 785-796.

6. A. J. Chorin and J. E. Marsden, *A Mathematical Introduction to Fluid Mechanics*, Springer-Verlag, New York, 1979.

7. A. J. Chorin, private communications.

8. J. P. Christiansen, ''Numerical Simulation of Hydrodynamics by the Method of Point Vortices,'' *J. Comput. Phys. 13*(1973), pp. 363-379.

9. O. Hald, ''Convergence of Vortex Methods, II,'' *SIAM J. Numer. Anal 16*(1979), pp. 726-755.

10. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*, McGraw-Hill, 1981.

11. A. Leonard, ''Vortex Methods for Flow Simulation,'' *J. Comput. Phys. 37*(1980), pp. 289-335.

12. M. B. Perlman, "On the Accuracy of Vortex Methods," PAM-192, Center for Pure and Applied Mathematics, University of California, Berkeley, December 1983. Ph. D. Dissertation.

4

User Abstractions for Run-Time Partitioning

But four young Oysters hurried up,
All eager for the treat...
And this was odd, because, you know,
They hadn't any feet.

Four other Oysters followed them,
And yet another four;
And thick and fast they came at last,
And more, and more, and more...

–Lewis Carroll, Through the Looking-Glass

4.1. Underlying Assumptions

We next discuss a simple set of abstractions for handling run-time partitioning. We first present the user interface, and following that show how to apply our abstractions to a practical example–the MLC. We finish the chapter by giving an implementation strategy for the abstractions. Though we focus on how the utilities work in a particular problem-instance in two dimensions, the ideas presented should also make sense for other localizable calculations of arbitrary dimensionality. We exclude from discussion both low-level optimizations for speeding up arithmetic operations, and the non-localized computation, i.e. the Poisson solver, to which the abstractions do not apply. These will be discussed in chapters 5 and 6, where appropriate.

Our abstractions assume a simple execution model for localized computation, called the *lattice model*. In this model, a computation is viewed as a sequence of state transformations performed on a collection of variables mapped onto a lattice, which we call a `workLattice`. A *workLattice*, Λ, is a uniform collection of sets of variables, indexed by a rectangular subset of $\mathbb{Z}^n$. In the rest of the chapter we will assume that $n = 2$; however, the abstractions we will discuss readily generalize to an arbitrary number of dimensions. Each bin $\Lambda_{i,j}$ abstracts a set of variables. If (i,j) is in the index set of Λ, then $\Lambda_{i,j}$ is the set of variables (or by abuse of notation, their values) at a given bin of the lattice. Each new state of a bin depends solely on it previous state, as well as the previous state of a small square neighborhood of bins. The size of the neighborhood is a run-time parameter C. Let the symbol $\mapsto$ designate data dependence; then $\Lambda_{i,j} \mapsto \Lambda_{k,l}$ (read "$\Lambda_{i,j}$ depends on $\Lambda_{k,l}$") only if $|i-k| \le C$ and $|j-l| \le C$.

To execute the above computation on a multiprocessor, we partition Λ into sublattices, Λ^i, $i = 0, \cdots, P-1$, and assign each sublattice to one of P unique processors. The Λ^i cover all of Λ and are disjoint: $\Lambda = \bigcup_i \Lambda^i$; $\Lambda^i \cap \Lambda^j = \phi \iff i \ne j$. Each processor executes the identical program, called a task, and owns the bins in its assigned sublattice. It stores these bins in its private memory and it alone will be responsible for updating them. To each task's Λ^i there corresponds a small buffer region, Δ^i, used for for handling communication with interacting subproblems. This region is called an *external interaction region* and consists of a collection of bins that lie in a thin rectangular shell surrounding the perimeter of Λ^i (see Fig. 4.1). Since index-set$(\Delta^i) \cap$ index-set$(\Lambda^i) = \phi$, the bins in the external interaction region are not properly owned by the task. The Δ^i serve as a staging area for data owned by all other tasks j such that $\Lambda^i \mapsto \Lambda^j$ (and by virtue of their mutual interaction, $\Lambda^j \mapsto \Lambda^i$), or equivalently, for which index-set$(\Delta^i) \cap$ index-set$(\Lambda^j) \ne \phi$. The thickness of Δ^i is a run-time parameter.

We provide a mechanism for handling the communication among the overlapping regions. The task assigned to partition Λ^i may communicate with that assigned to partition Λ^j by receiving copies of data in Δ^i from Λ^j, where these intersect, and sending copies from Λ^i to Δ^j. These

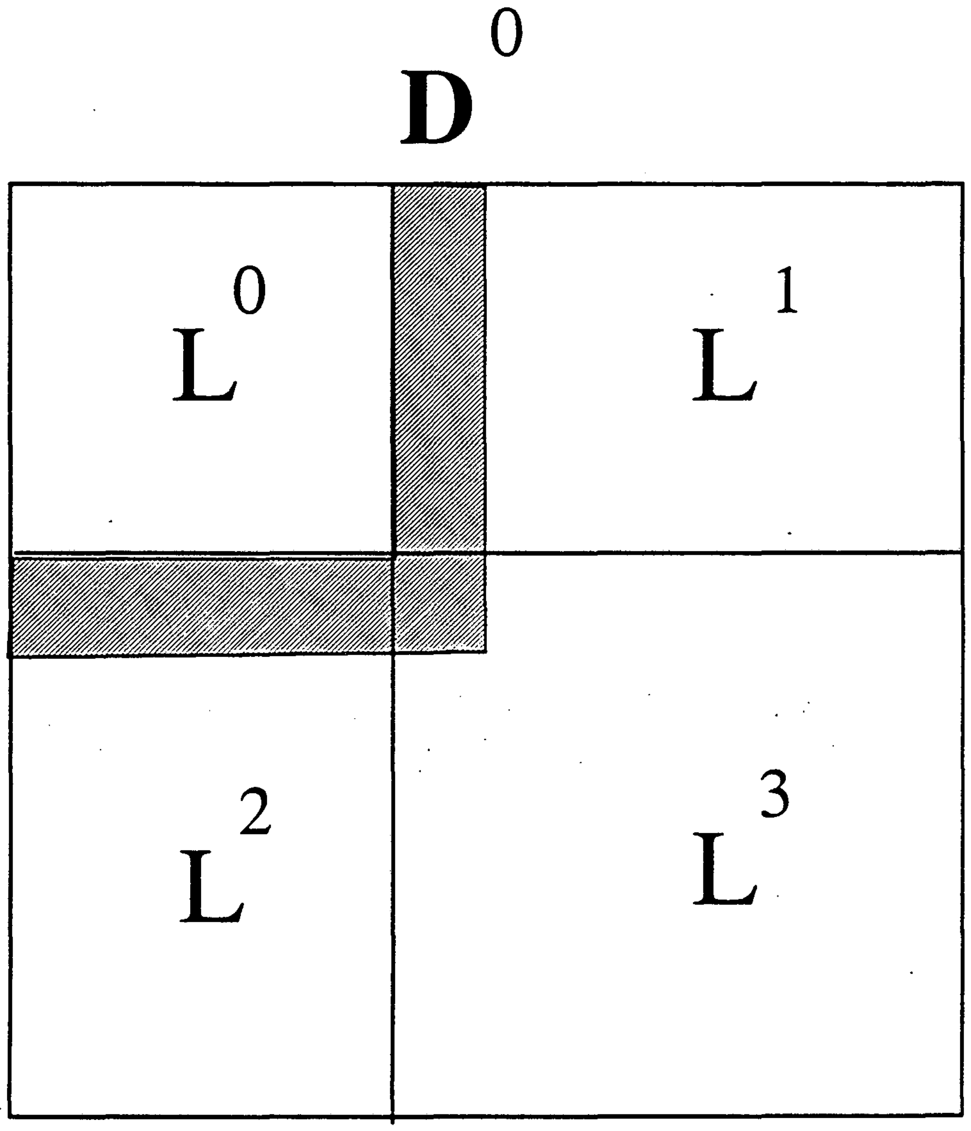


Figure 4.1. Processor 0 is assigned L^0, a subregion of the workLattice L, and an external interaction region D^0.

copies can be thought of as boundary conditions. It is possible, in addition, for the task assigned to Λ^i to modify its own Δ^i, and to transmit this change to neighboring partitions. In the MLC, for example, a vortex may change owners by moving outside of Λ^i and into Δ^i. It will eventually be collected from Δ^i and transmitted to its new owner.

Data transmitted from one task to another can be combined in an arbitrary, application-determined way with data in the receiving bin. In cases where several tasks transmit data to the same bin, the order of receipt is not defined, and the user is responsible for insuring that his algorithm is insensitive to this order.

4.2. A Virtual Machine

We provide the programmer with a set of abstractions for handling the communication and data partitioning activities discussed in the previous section. These abstractions implement a virtual machine (VM)– an abstract local-memory multiprocessor. Since the VM has no shared memory, its processors communicate by passing messages. The semantics of the VM's abstractions are unaffected by the architecture on which implemented; in particular, the kind of communication model employed by underlying layers of hardware and software, i.e. either memory-sharing or message-passing. The user's code will not be completely insulated from a change of architecture, however, since our abstractions apply only to computation that fits the lattice model. But, the user will likely spend less time rewriting software than he would without our abstractions, since he will be primarily concerned with only the parts of the code that the VM leaves up to his discretion. We will discuss such discretionary programming later in this chapter. We next specify the abstractions for handling task decomposition and communication. These are summarized in Figures 4.2 and 4.3.

The VM specifies that in addition to the P tasks executing their own subproblem, called *worker* tasks, there will be a distinguished task called the `boss`. The purpose of the boss is to spawn the worker tasks, to provide them with initial input, and to maintain a small amount of

```
-- Initialize the ''boss:'' establish P as the number of tasks;
-- mapSize as the bounds of the index set of the global lattice;
-- and buffSize (optional) as the maximum length of the VM's message buffers.
-- Spawn P tasks; each will execute the same program ''worker''
-- User is responsible for transmitting initial inputs to the tasks

proc initBoss(P:int,mapSize:tuple,worker:proc(tuple),buffSize:int)

-- thisTask() returns the worker's unique taskIndex assigned by the boss,
-- an int in 0..P-1
func thisTask() → taskIndex

-- numTasks() returns the the total number of worker tasks
func numTasks() → int

--        Types provided by the VM

type partition
-- A partition is a bounding box with two user operations:
--        give upper and lower bound along coordinate k
func limH(k:int,q:partition) → int
func limL(k:int,q:partition) → int

type workMap
-- One operation defined on workMap: set $wm_{i,j}$ to $w$
proc wAssign(w:int,wm:workMap,(i,j):tuple)

--        Partitioner utility
-- Given a workMap for thisTask(), wait for all other tasks initiated
-- by the boss to call partitioner, and return a partition indicating
-- the next work assignment for thisTask().
-- Partition boundaries will move no more than constraint bins in any direction
-- Side effect: clear workMap, establishing the new index set corresponding
-- to the new bounding box
func partitioner(wm:workMap,constraint:int) → partition
```

Figure 4.2. This pseudo-code defines the user interface to the boss, and to task partitioning abstractions. The pseudo-code of Figure 4.3 defines the user interface to local communication abstractions.

supervisory state. The supervisory state is not directly accessible to the workers, but only indirectly through the VM's abstraction mechanisms. The `initBoss` procedure establishes the number of tasks, the common routine to be executed by the tasks, the bounds on Λ's global index set (the lower bounds are always 0), and the maximum length of internal message buffers used by the VM. When `initBoss` returns, the worker tasks will all be executing the same subroutine in parallel with the boss. Each worker may determine the unique identifier assigned to it by the boss by means of the function `thisTask()`, and the total number of worker tasks with `numTasks()`.

The VM defines two abstract data types: `partition` and `workMap`. A `partition` is a rectangular subregion of the Λ and is defined by its bounding box. The user may query the bounds of a partition with the `limL` and `limH` functions. These give the upper and lower bounds, respectively, of the bounding box along a given axis; `limH(0,q) - limL(0,q) + 1`, for example, gives the width of partition `q` in bins, along the 0*th* coordinate. A `workMap` is a mapping from the index set of the Λ^i onto the integers; it gives the cost of updating the state of each bin of Λ^i. There is only one user operation defined on `workMaps`– assignment. Having now described the two data types, and the simple operations defined on them, we next discuss the VM's two utilities for handling run-time partitioning.

4.3. VM Utilities

4.3.1. Mapper

The `Mapper` utility handles the communication among interacting regions of the workLattice. It also enforces a *local barrier* synchronization constraint to ensure correctness. When it encounters a local barrier, each task must communicate with all the tasks it depends on, and will not pass through the barrier until it has finished. Each task will generally encounter the barrier at a different time, according to the fraction of the total work assigned to it; workload assign-

```
-- VM provides abstractions for localized communication

-- Wait for all tasks interacting with thisTask() to call Mapper
-- Transmit data from region within |C| units of boundary of thisTask()
-- to appropriate tasks:
-- If C > 0, region transmitted is within |C| units outside thisPart, else inside
proc mapper(C:int,dataMovingStuff:procPair)

-- The user provides a pair of routines to fill and empty mapper's message buffers
typedef procPair is struct{

        pack:proc(patch:partition,in out buffer:sequence(*) byte,out more:bool)
-- Pack data from within patch into buffer, up to maximum length of byte sequence
-- Assert 'more' if buffer full, but data not exhausted.
-- In this case repeating the call until 'more' clears will convert
-- remaining data into a sequence of byte-streams

        unpack:proc(buffer:sequence(*) byte)
-- When given a sequence of bytes produced by pack, combines them with data
-- that already reside in the memory of thisTask()
        }

-- The message buffers are flexible sequences of bytes
-- They have a maximum length, defined by initBoss()
-- There are two user operations defined on message buffers:
--      writeSeq, used by the user-supplied pack routine; and
--      readSeq, used by the user-supplied unpack routine

        func writeSeq(buff:sequence (*) byte,data:anyType) → bool
-- Appends datum of any type onto the end of buff, returning TRUE unless no room
-- If no room, returns FALSE, leaving buff unchanged

        func readSeq(buff:sequence (*) byte,data:anyType) → bool
-- Reads datum of any type from the head of buff;
-- If successful, returns TRUE and advances the head of buffer marker;
-- else, if not enough bytes left to satisfy request, returns FALSE
-- and leaves the head of buffer marker unchanged
```

Figure 4.3. The VM's abstractions for handling local communication.

ments will rarely be in perfect balance. As a result, some tasks may wait much longer than others to finish communicating, regardless of the amount of information they transmit. C is the first parameter passed to `Mapper`, and it specifies how `Mapper` will handle communication in terms of the global index set of Λ. `Mapper` collects all information within $|C|$ bins of the perimeter of Λ^i and transmits them to the appropriate interacting task(s). For $C > 0$, information is sent from Δ^i to the partitions, Λ^j, of other tasks. If $C < 0$, information is sent from Λ^i to the Δ^j of other tasks. Thus, $|C|$ effectively specifies the thickness, in bins, of the external interaction region. In effect, the sign of C decides whether the Δ^i act as sources or sinks for information transmitted by `Mapper`.

`Mapper` abstracts the communication process in two ways: (1) it specifies communication in terms of regions of the global coordinate system of Λ, without knowing the local memory addresses of the data belonging to the regions; (2) it transmits only sequences of bytes, and knows nothing about how the user represents his data structures. To convert between a byte sequence and his representation of data, the user provides `Mapper` with a second argument, of type `procPair`. A `procPair` consist of two routines: `pack` collects data from the user's data structures and moves them into a byte sequence; `unpack` moves the data in the other direction. These two routines are responsible for transforming between the index set of Λ and the coordinate system of the application.

`Mapper` subdivides the regions of space from which it collects information into a set of disjoint rectangles of type `partition`. `Pack` moves the data in each rectangle coming from the memory of `thisTask()` into an output byte sequence. It will pack up to the maximum allowable length of a byte sequence. If it has more information to supply, `pack` sets a boolean flag passed as an `in out` argument; it will eventually be restarted by `Mapper` and must therefore remember where it has left off. `Unpack` reverses the `pack` operation; it moves a byte sequence of data passed to it into the memory of `thisTask()`, combining the newly-acquired data with existing data, if appropriate. `Unpack` takes neither a `partition` nor a restart flag as arguments since pack will have appended the appropriate geographic information onto the data passed in the byte sequence. Unpack is responsible for allocating any storage it needs. The user is responsible for reclaiming this storage separately.

The VM supplies two operations for dealing with byte sequences, which will be used by pack and unpack. Byte sequences are strings of bytes whose length may dynamically shrink and grow, up to a maximum length established by the boss. Invoking writeSeq(bseq,data) appends a data element of any type to the end of the byte sequence bseq, provided there is room. If there is room, writeSeq returns true, else false. Invoking readSeq(bseq,data) reads the first data element from the beginning of bseq, provided there is one. If there is, writeSeq returns true, else false.

Each call to Mapper may be specified with its own value for C and its own procPair. Though the external interaction region used by each different call will be a logically disjoint entity, the user is free to recycle the storage used by the different interaction regions in order to save memory. However, he may want to implement each distinct interaction region with different data structures. The decision to do so will depend not only on the application, but on the architecture; in particular, on the amount of memory available for computation. We employed such a strategy on the iPSC, on which memory is a scarce resource, but not on the Cray X-MP, where memory is plentiful.

4.3.2. Partitioner

Given a workMap from each task, the partitioner utility subdivides Λ into P partitions (P is part of the global context established by the boss), Λ^i, $i = 0, \ldots, P-1$, and assigns Λ^i to the task with thisTask() $= i$. If Λ^i is empty, the task does nothing. Partitioner requires a global workMap covering all of Λ; but this global structure is hidden from the programmer and each worker computes only the part of the workMap covering its assigned Λ^i. As the result of calling Partitioner, each worker obtains a new bounding-box and a suitably restructured workMap whose index set conforms to the new Λ^i and whose entries are zero. Before he next calls Partitioner, the user must set up the appropriate entries in his local workMap.

Partitioner makes one assumption about the *workMap*, namely that it is roughly set-additive; the total amount of work done in the partitioned problem differs only slightly from the work done on the unpartitioned problem. If the mapping isn't set-additive, Partitioner may not necessarily produce good partitionings. Certain domain decomposition methods for elliptic partial differential equations, for example, fall into this category (see, for example, Keyes and Gropp [7]).

Because the cost of updating each Λ^i may vary with time, Partitioner must be invoked periodically to shift work from the more heavily loaded tasks to the more lightly loaded ones. As a result of such repartitioning, Λ^i changes to $\tilde{\Lambda}^i$, for all i; some data may change owners, and must therefore be transmitted to the correct task by a call to Mapper (see Figure 4.4). The data that $\tilde{\Lambda}^i$ gives up belongs to the region of space specified by $\Lambda^i - \tilde{\Lambda}^i$, where "$-$" is the set differencing operator. Generally, the walls of the partitions can move by arbitrary amounts within the global coordinate space of Λ. Mapper will be able to handle load shifting correctly, i.e. without losing any data, so long as we can guarantee that $\Lambda^i - \tilde{\Lambda}^i \subset \tilde{\Delta}^i$. To ensure that this is true, the user supplies an integer that constrains the amount that Partitioner may move the corners of the partitions from one call to the next. This constraint must be no more than the thickness of the external interaction region specified to Mapper. Though $\tilde{\Delta}^i$ is a superset of the region of space involved in load shifting, we believe that optimizing Mapper to collect only from the necessary parts of $\tilde{\Delta}^i$ would not make much of a difference in overhead.

4.4. Portability Considerations

There are five aspects of an application's implementation that the VM does not specify. These can either depend in an essential way on particular hardware structures or system calls, on

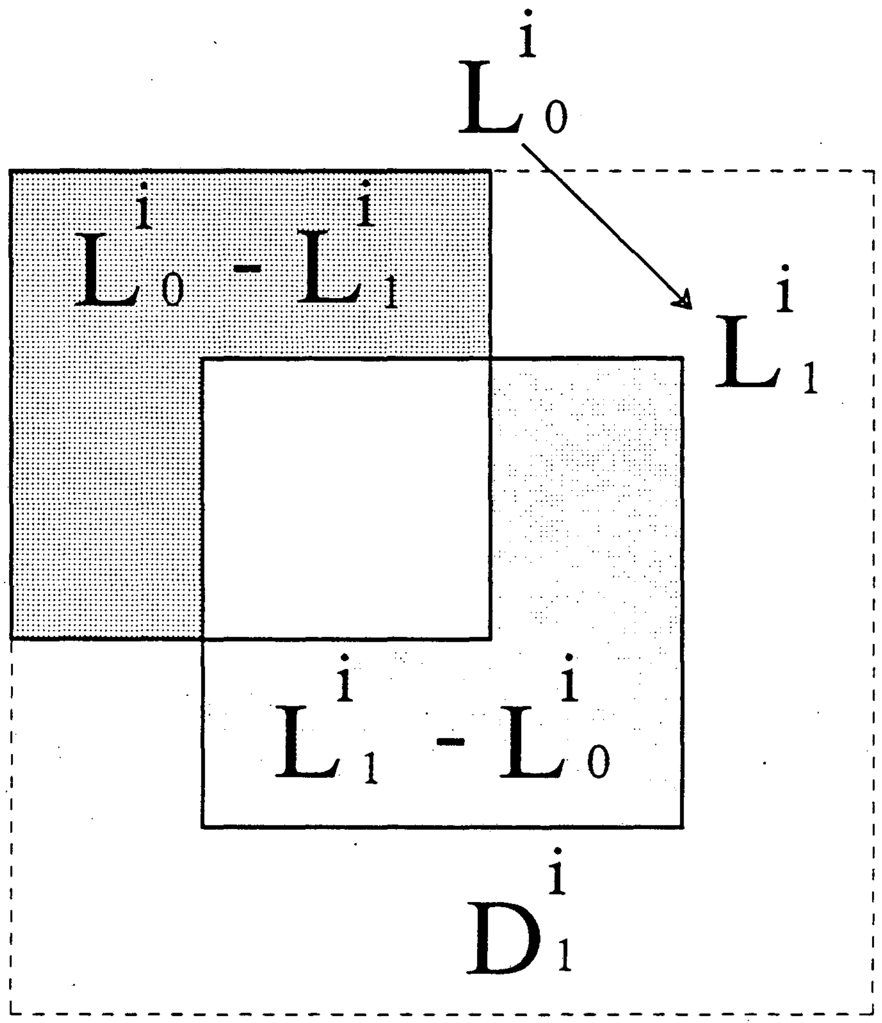


Figure 4.4. As the result of repartitioning, processor i's assigned subproblem changes from L_0^i to L_1^i. The motion of the partition is constrained to ensure that data given up by processor i, $L_0^i - L_1^i$, will be contained in its external interaction region D_1^i.

the application, or may simply be left up to the user's discretion:

(1) Assignment of tasks to processors, and task scheduling.
(2) Communication between the boss and worker tasks.
(3) Storage management.
(4) Communication among workers that cannot be efficiently handled by `Mapper`.
(5) Input/output.

We are primarily concerned about the first three aspects; the other two will be mentioned only briefly in chapters 5 and 6.

The VM specifies that each task will be assigned a unique processor. On many systems, e.g. the Cray X-MP and Intel iPSC, a processor can execute more than one process and so the user is free to have more tasks than there are physical processors on the system. This practice may not prove worthwhile, however, as it can incur a high scheduling overhead, especially within `Mapper`. A more efficient way of executing multiple tasks on a processor would be to have the user to handle scheduling himself, and would require some simple changes to the VM. `ThisTask()` would be modified to return a list of task identifiers, and `Partitioner`, a list of partitions, one for each task. The user would then be responsible for applying his computation to successive elements of the list of partitions. `Mapper`, however, would not have to change.

One aspect of task management that the VM doesn't specify is whether the boss executes on its own processor, or is just scheduled as another task along with the workers. This affects the user's code, and not just the VM, because the user is free to have the boss do useful work on behalf of the workers (for example, distributing initial data or coordinating global parts of the overall computation), a process that entails some communication, and the VM doesn't specify how the boss and workers communicate. On the Cray X-MP, for example, it is convenient to divide the boss's activities into two parts: one part running as a separate task scheduled by the operating system, and another other running as part of a distinguished worker task. A conditional statement selects the distinguished worker that is to impersonate the boss and the boss and workers communicate through shared memory. On the iPSC, on the other hand, there is an additional host processor that may be used to handle the boss's actions. The boss and the workers communicate by passing messages.

The final aspect of what the VM leaves up to the user is storage management. The programmer is free to manage the storage used by the `pack` and `unpack` routines however he wishes. The same holds true for the *workMap* and the skeletal structure of the *workLattice*, i.e. the bins but not the data. In addition, the user is responsible for implementing the necessary conversions between the global index space of Λ and the coordinate system(s) meaningful to his application.

4.5. MLC Implementation

Having introduced the abstractions, we next show how to use them in the MLC. The implementation we discuss will be portable, ignoring the limitations discussed at the end of the last section. We leave the job of specifying the parts of the code that are not portable to succeeding chapters. We begin by asking two questions:

(1) Can the MLC be divided into two parts, one that fits the lattice model of execution (the ''local part''), and a part that does not (the ''global part'')?

(2) Is the global part small enough that it will not become a serial bottleneck on the number of processors we expect to use?

(3) Can the cost of updating each of the bins of the *workLattice* be computing in a simple way?

The answers to these questions are ''yes;'' our abstractions therefore apply to the MLC. The MLC works by pushing a set of particle-like elements at a discrete set of timesteps. Each vortex feels two kinds of influences, local and distant, each of which is handled by a separate computation. The MLC organizes the vortices into a mesh to distinguish the local influences from distant ones. Local influences are computed as N body interactions between nearby pairs of vortices; when there are thousands of vortices, this computation consumes the largest fraction of the MLC's running time. Distant influences are computed separately, by interpolating values from a finite-difference grid. The grid is produced as the result of non-localized computation done by a Poisson solver. The cost of the solver will be small if the vortices are sufficiently numerous.[1] Finally, the cost of updating each bin of vortices can be readily computed by knowing the number of vortices in the bin, and in those nearby.

Because it illustrates all the mechanisms of our VM, we will focus on the local interactions part of the MLC computation. Figure 4.5 shows the boss's code. Figures 4.6, 4.7, and 4.8 show the workers' code, including the calls to the run-time partitioning utilities.

The MLC uses some simple data structures to compute local interactions: the bins used to keep track of vortices, and the vortex-records themselves. Each bin is a `collection` of vortex-records. A `collection` can be thought of as a linked list, but the exact representation is irrelevant. The bins that hold vortices correspond with the bins of the workLattice used by our abstractions. Though each task really needs only a subset of Λ at any one time, we chose to provide each task with a global copy of the *bins* of Λ (though not all the vortices), as well as the `workMap`, and to implement the data structures as dense arrays. This wasted some storage, but also simplified the programming. A more sophisticated implementation would employ sparse data structures instead of dense arrays so that a task would only store the bins of Λ it actually used.

[1] The level of concurrency at which global computation becomes a performance bottleneck depends both on the application, i.e. the relative fraction of time spent in global computation, and the architecture, i.e. the cost of communication relative to computation. The Poisson solver we used was not a performance bottleneck on 32 processors of the Intel iPSC, on which communication is considered expensive. Though it wasn't parallelized on the 4-processor Cray X-MP, the solver was not a performance bottleneck on that machine either.

```
-- Main program: boss reads in simulation parameters,
-- initializes supervisory state and spawns worker tasks with their input
program vortex

        int P, M, maxDeltaX, maxDelPart, corrRad
-- P: the number of worker tasks
-- M: the linear bound of the workLattice (it is assumed to be square)
-- maxDeltaX: maximum amount a vortex may move in one velocity evaluation
-- maxDelPart: constraint on motion of partitions between successive repartitionings
-- corrRad: the maximum distance over which vortices interact directly
-- numerical: a catch-all for various simulation parameters, like the timestep $\Delta t$

--      The MLC's data structures

-- Vortices are organized via the workLattice, an M by M array of bins
-- At any one time the task's partition and external interaction region
-- are kept in a portion of the workLattice and the rest isn't used
-- Each bin of the workLattice is a Collection of records of type vortRec
-- A Collection is a primitive data-type enumerated by the foreach construct
        typedef workLattice: is array(M,M) of Collection(vortRec)

-- vortRec is a user-defined type
        typedef vortRec is struct {

--      position vectors:
                $\vec{x}, \vec{x}_{old}$ : array(2) of real,
--      velocity vectors
                $\vec{u}, \vec{u}_{old}$ : array(2) of real,
--      scalar strength
                $\omega$ : real}

        read P, M, maxDeltaX, maxDelPart, corrRad, numerical
-- Spawn P workers, each executing the external procedure iterate
        call bossInit(P,(M,M),iterate)

-- Transmit the initial input to the workers; this code is supplied
-- by the user, and depends on both the application and architecture
        call workerInit(maxDeltaX,maxDelPart,corrRad,numerical)
end vortex
```

Figure 4.5. The main program reads the input, initializes the boss, spawns off the worker tasks and provides them with their initial input. Also shown are the data types defined by the MLC. The routine that initializes the workers isn't shown. Figures 4.6, 4.7, and 4.8 show the pseudo-code for the workers' computation.

```
-- Worker procedure
-- Main timing loop does two velocity evaluations per timestep
-- Calls the MLC to do the velocity evaluations
-- MLC exists as two routines: one handles sorting, other evaluates velocities
proc iterate(input:tuple)

-- These routines are used by mapper, and may not be portable
-- The code is simple and will not be shown
        external proc pack, unpack

        workLattice bins

        int maxDeltaX, maxDelPart, corrRad
-- Obtain input from the boss, this code is not portable
        call recvInput(maxDeltaX,maxDelPart,corrRad,numerical)

-- Set up various data, including an initial balanced assignment of vortices
-- Set thisPart to the initial bounding-box of the task's partition $i$
-- Fill in only the part of the workLattice corresponding to thisPart;
-- In particular, leave the external interaction region empty
        call taskInit(bins,numerical,numTasks(),thisPart)
        do while $t < t_{max}$
--      Call the MLC to do the first velocity evaluation, and then push the vortices
                call push(bins,1,maxDeltaX,maxDelPart,corrRad,numerical)
--      Second velocity evaluation and push phase
                call push(bins,2,maxDeltaX,maxDelPart,corrRad,numerical)
                $t \leftarrow t + \Delta t$
        end while
end iterate
```

Figure 4.6. The boss spawns off this worker procedure to each task. The work is done by the procedure called "push," defined in Figure 4.7. The code that receives the input from the user has been omitted, as has the code that sets out the initial distribution of vortices.

There are three simple rules for determining where to insert calls to the `Partitioner` and `Mapper` utilities:

(1) Find a convenient stopping place in the computation where the data may be repartitioned, i.e. between timesteps, and insert a call to `Partitioner`.

(2) Locate *local barrier synchronization points*, which are essentially the boundaries between state transitions, as discussed in § 4.1. At these points, a task's assignment of data can become inconsistent with the logical partitioning established by `Partitioner`, or copies of data obtained by a previous call to `Mapper` can become inconsistent with the originals. Insert a call to `Mapper` at each barrier synchronization point.

The next two subsections discuss each in turn.

4.5.1. Data Partitioning

Figure 4.9a shows a simple way to divide up the workLattice among 16 processors: split the bins uniformly into a regular pattern of box-like subproblems. This strategy, however, would underutilize the processors; only 4 of 16 would be given much work to do. The trouble is that the vortices distribute themselves unevenly so that the completion time for a subproblem may not be proportional to its area. Figure 4.9b shows a better way to split up the problem that compensates for the uneven distribution of vortices over the domain. This adaptive decomposition generates somewhat irregularly sized subproblems that all complete in roughly the same time, and it diminishes the running time of the computation by a factor of four. This is the kind of partitioning rendered by `Partitioner`.

Since vortices move, the partitioning cannot be left fixed for all time; work must be periodically reapportioned as shown in Figure 4.10. If the work were not redistributed, then some processors would become overloaded while others would only stand and wait. Some work must be shifted from the more heavily loaded processor(s) to the more lightly loaded one(s) in order to advance the latest completion time. Owing to a time step constraint, vortices to jump by only

```
-- This code implements the MLC and includes the calls to the VM's utilities
proc push(bins:workLattice,(evalNumber,maxDeltaX,maxDelPart,corrRad):int,numerical)
-- Place each vortex into the correct bin, as determined by $\vec{x}$
-- Error if any move maxDeltaX bins beyond thisPart
        call sort(bins,thisPart,maxDeltaX)
-- Migrate any vortices that have changed owners since the last velocity evaluation
        call Mapper(maxDeltaX,(pack,unpack))
-- Reclaim all the workLattice outside thisPart
        call trim(thisPart,maxDeltaX)

-- Repartition space according to the predicted time to push the vortices in each bin
-- Vortices move only slightly between velocity evaluations; repartition only every timestep
        if (evalNumber = 1) then
-- First produce the workMap
                call makeWorkMap(wm,thisPart,bins,corrRad)
-- Partitioner returns the new bounding box
-- No corner of a bounding box may move more than maxDelPart bins in any direction
                thisPart := Partitioner(wm,maxDelPart)
-- Migrate any vortices that have changed owners as the result of repartitioning
                call Mapper(maxDelPart,(pack,unpack))
-- Remove the originals
                call trim(thisPart,maxDelPart)
        end if

-- Send copies of vortices lying just inside your partition that others depend on
        call Mapper(−corrRad,(pack,unpack))

--      compute velocities
        call vorvel(thisPart,bins)
-- Part of the second order time integration scheme
        call timeIntegration(evalNumber,thisPart,bins)
-- Done with the external interaction region
        call trim(thisPart,corrRad)
end push
```

Figure 4.7. Inserting the calls to the VM's utilities into the *push* procedure which handles time integration and uses the MLC to evaluate velocities. The MLC executes as two subroutines. One routine, called *vorvel*, does the velocity computation. The other sorts the vortices into their correct bins. Sorting is necessary as some vortices may have changed bins since the last velocity evaluation. The work associated with time integration is negligible compared with that done in the velocity evaluations and may be ignored; see chapter 3 for additional details. Certain code has been omitted: the workMap computation, data packing and unpacking done for Mapper, and storage reclamation. The code that *push* calls to compute local interactions appears in Figure 4.8.

small amounts between time steps; repartitioning need therefore be done every few, say 4 or 8, timesteps.

The number of local corrections done on each bin is a good measure of work for the `workMap` that `Partitioner` uses to partition bins into subproblems. The number of local interactions done on the vortices in a single bin is easy to compute and is the product of two quantities: the number of vortices in the bin, and the number of vortices in that bin and neighboring bins found within the correction distance. This can be expressed as in terms of a work-mapping:

$$workEst(i,j) = \mathbf{card}(i,j)\left[\sum_{|k|,|l| \le C} \mathbf{card}(i+k,j+l)\right] \tag{4.5}$$

where "card(i,j)" is the number of vortices in workLattice(i,j), and i and j range over the index-set, `thisPart`, of `thisTask()`, i.e. `limL(0,thisPart)` $\le i \le$ `limH(0,thisPart)`, and `limL(1,thisPart)` $\le j \le$ `limH(1,thisPart)`. The code is simple and will not be shown.

After calling `Partitioner` each worker now has the bounding box of the region of the workLattice assigned to it. With the aid of the `limL()` and `limH()` functions, the user can construct loop bounds for ranging over the appropriate bins of Λ.

Because `Partitioner` may not split bins of the workLattice, each bin represents an indivisible unit of work. In the case of the MLC, a less restricted scheme that allowed internal boundaries to pass through the bins would impose additional coding overhead, and appears to confer no advantage. An easier way to improve the workload imbalance is to refine the workLattice. At some point, however, a point of diminishing returns is reached where the cost of manipulating additional bins outweighs the benefits of having the work more evenly balanced.

The pseudo-code of Figure 4.8 does not specify the order in which vortices' influences are accumulated. For obvious implementations of the `foreach` construct and the `unpack` routine, we may expect this order to be non-deterministic, depending on the relative speeds of the processors, for example. However, as required according to the discussion in § 4.1, our algorithm is relatively insensitive to these orderings, and results of multiple runs may be expected to agree within roundoff.

One measure of effectiveness for a partitioning strategy is the amount of duplicate computation it introduces along the perimeter of each partition. There is no redundant computation in the local interactions computation shown in Figure 4.8. In our iPSC implementation, however, there was some redundant computation in other localized parts of the calculation that are not shown in the pseudo-code; in particular within the part of the code that locally corrected the far-

```
-- Compute local interactions
proc vorvel(thisPart:partition,
            workLattice:array(M,M) of collection(vortRec),
            C:int)
-- Visit each bin in the index set of thisPart
foreach $(i,j) \in [limL(0,thisPart) .. limH(0,thisPart)] \times [limL(1,thisPart) .. limH(1,thisPart)]$

-- Compute local interactions against each vortex in the bin
        foreach vortex at $\vec{x}$ with velocity $\vec{u} \in$ workLattice $(i,j)$

-- Influencing vortices are found in a neighborhood of bins surrounding the vortex
-- The correction radius gives the extent of a vortex's influence, in bins
                $\vec{u} := 0$
                foreach vortex at $\vec{y}$ with strength $\omega \in$ workLattice $(k,l)$
                where $(k,l) \in [-C+i .. i .. C+i] \times [-C+j .. j .. C+j]$
-- $\chi(\vec{r})$ is the velocity induced at $\vec{r}$ by a unit-strength vortex blob at the origin
                        update $\vec{u}$ by adding $\chi_{blob}(\vec{x}-\vec{y})*\omega$
end vorvel
```

Figure 4.8. This code computes the local interactions.

field velocities computed by the global solver computation. There was no duplication at all in the Cray X-MP implementation, in which the computations were organized differently to take advantage of the Cray's vector-mode hardware.

4.5.2. Local Communication

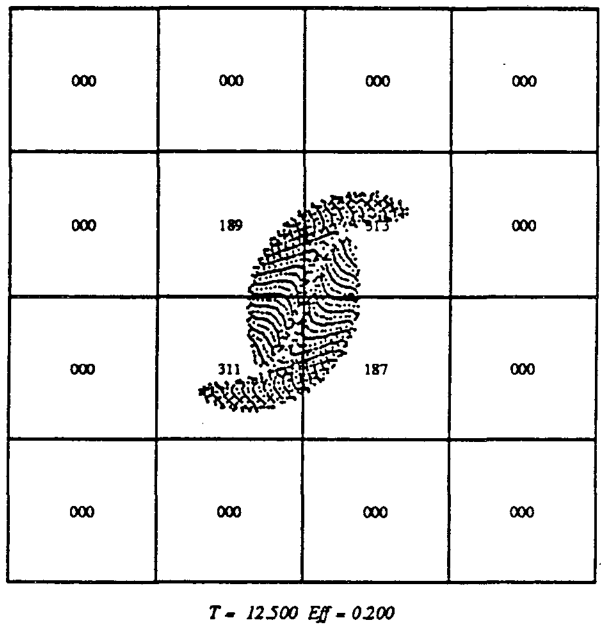
We used three calls to `Mapper` to handle local communication in the MLC. Setting up the calls entails:

(1) Choosing the appropriate sign and magnitude of C.
(2) Writing the pack and unpack routines, and handling storage reclamation.

Part (2) generally requires re-implementation for each new system.

Choice of C. Generally the user should make C no larger than necessary, since the cost of mapping increases with C. The value of C depends on how `Mapper` will be used. `Mapper` performs 2 functions in the MLC: (1) it issues copies of vortices to allow each processor to obtain boundary conditions; (2) it migrates vortices that change owers, either as the result of their own motion, repartitioning, or both (see Figure 4.11). The proper value for C in the first call equals the negative of the correction radius $-corrRad$, since all copies must originate from inside some task's Λ^i. If the magnitude of C is too small, the answers will be incorrect since some of the vortices a task needs will not become visible through the external interaction region.

When `Mapper` migrates vortices, C will be positive. There are two separate calls; one to handle migration as the result of sorting, the other to handle migration as the result of repartitioning. For the first call, C should be chosen to be somewhat larger than the maximum amount that vortices can move between timesteps. This number will generally be known to the user, and can be determined through experimentation. We found that a good value was 1 or 2. Whatever this value of C is, it must also be passed to the `sort` routine, and `sort` must check that no



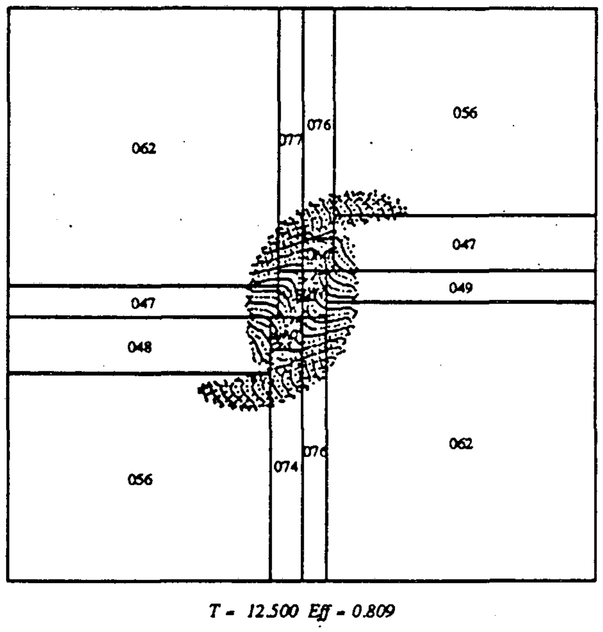


Figure 4.9. Partitions with (a) equal areas and (b) equal amounts of equal work. The labels give each partition's share of the workload normalized to 1000 units of total work. If loads were perfectly balanced, then each subproblem would get 062 units of work. The calculation began with 2 finite area vortices with radius 0.120 and centers separated by 0.25. Each patch had 795 vortices, shown as dots, placed evenly on lattice points spaced 7.5×10^{-3} units apart. Some vortices have been omitted for the purpose of clarification.

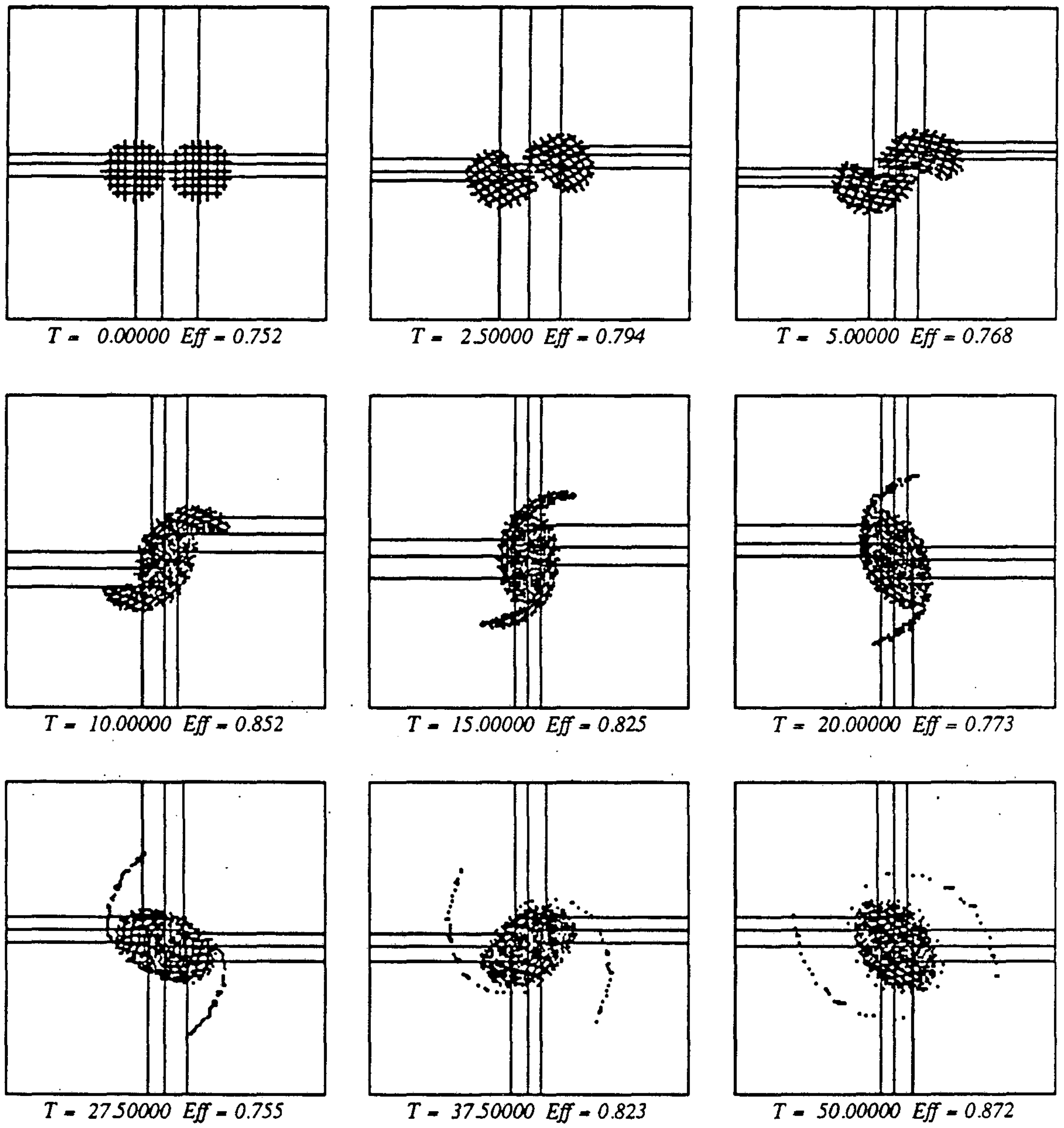


Figure 4.10. The distribution of vortices changes with time, so the work must be periodically repartitioned. This series of snapshots was taken from the same calculation used to produce Fig. 4.9b. Some vortices have been omitted for the purpose of clarification.

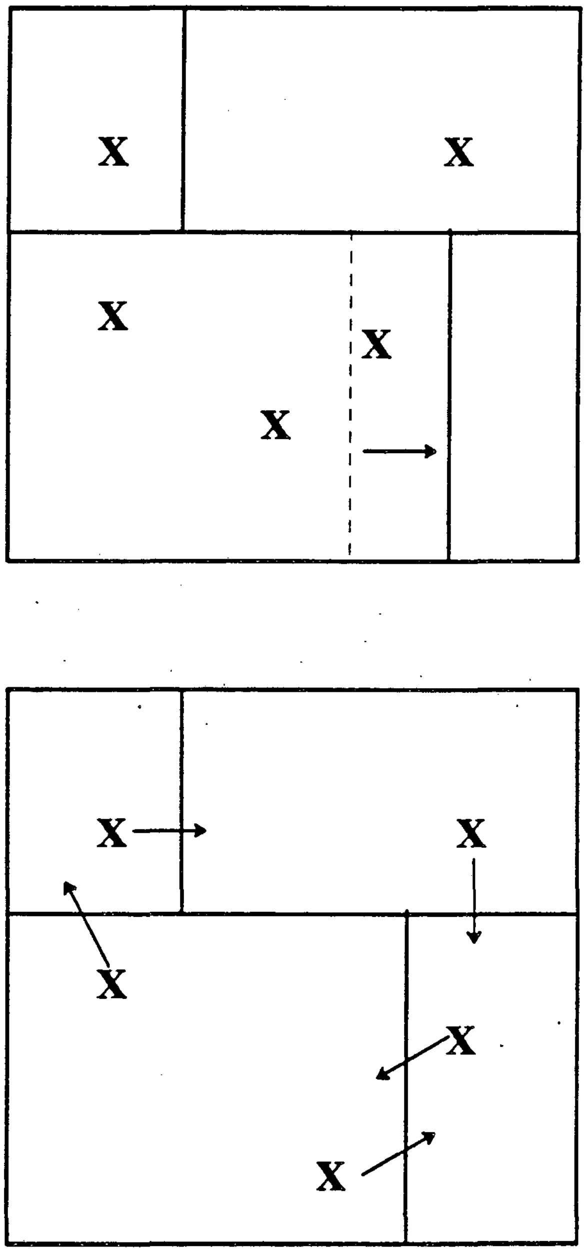


Figure 4.11. Vortices must be migrated when (a) boundaries slide past vortices, (b) vortices slide past boundaries, or both.

vortex moves further than C bins outside the partition of thisTask(); otherwise some vortices could become lost to `Mapper` and results would be incorrect. The value of C for the final call to `Mapper` must be no smaller than the repartitioning constraint the user passes to `Partitioner`. A good value is 2 or 3.

Pack and unpack routines. The `pack` and `unpack` routines implement the gather and scatter operations familiar to users of vector-type architectures like the Cray. The code is simple and will not be shown. Because the programmer is free to allocate different kinds of data structures for the different uses of the external interaction region, each call to `Mapper` can have its own set of `pack` and `unpack` routines. This was done on the iPSC, for example, in the interest of saving memory, but was unnecessary on the Cray X-MP, which had plentiful memory. In the iPSC implementation we also managed the different kinds of data structures out of separate heaps and so had to modify the numerical inner loops of the code to treat indices lying outside Λ^i specially. However, if we used a single heap, the inner loops would not have been modified.

In order to satisfy correctness requirements, the external interaction region must be cleared out before it can be reused by `Mapper`. Owing to limitations on memory, disused vortex-records must then be recycled. We managed our own heap of vortex records, in both the iPSC and Cray X-MP implementations, and therefore handled reclamation ourselves. Reclamation consisted of visiting all the bins of the external interaction region and reclaiming any vortices found there.

4.6. Implementation Strategy for the Virtual Machine

In this section we give an implementation strategy for the VM. Roughly speaking, the strategy we describe here is the same as that we used for the Intel iPSC and the Cray X-MP implementations. We will mention how the two implementations differ in chapters 5 and 6.

4.6.1. Partitioner

`Partitioner` works with two data types: `partition` and `workMap`. We chose to implement `partition` as a 4-element vector; the bounds query operations are merely array subscripting operations. We implemented `workMap` as a doubly-subscripted array; the one user operation, `wAssign`, is simply an assignment statement to a subscripted variable.

Prior to calling `Partitioner`, each task has only a subset of the `workMap` in its private memory. Since `Partitioner` needs the entire map, it may have to accumulate the local pieces into a single global data structure. The implementor is free to do whatever is convenient. On the iPSC, for example, we used a spanning tree utility, described by Moler and Scott [8] to combine the pieces. On the Cray, we implemented the `workMap` as a single large array in shared memory. When `Partitioner` returns, the boss has a copy of all P subproblems for the internal use of the VM.

We used a recursive bisection algorithm to partition the workLattice. The algorithm is a useful abstraction for determining a fair partitioning of work across a team of processors since it carries no knowledge about either the application or machine architecture. Any partitioning algorithm that rendered rectangles, however, and that was as fast as recursive bisection, would have sufficed. The recursive bisection algorithm has been around for some time and has been used by Warnock [9] for hidden surface removal in computer graphics; by Karp for the traveling-salesman problem [6]; by Dippé and Swensen [3], and Dippé and Wold [4] for realistic image rendering in computer graphics; and by Berger and Bokhari [1] for partitioning hyperbolic differential equations across multiprocessors. In two dimensions, the strategy is to cut an area of interest into two rectangles that represent equal amounts of work, or as nearly equal as possible, and then to apply the procedure recursively to each part; see Figure 4.12. This simple procedure generalizes trivially any number of dimensions. The two-dimensional algorithm is shown in Fig. 4.13.

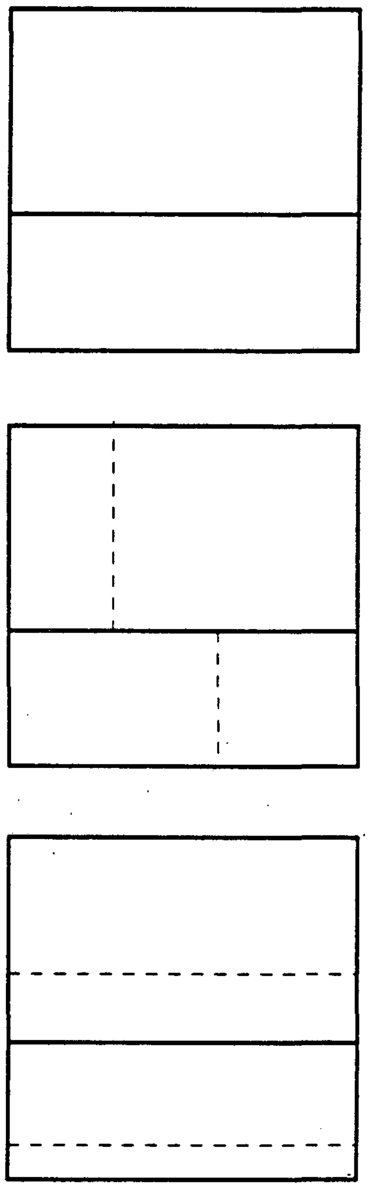


Figure 4.12. Recursive bisection into 4 subproblems: (a) first split the problem into two parts that complete in roughly the same time, (b) apply the procedure recursively to each part. To obtain boxes, the algorithm alternates the direction of the cut from one level of recursion to the next; this has the effect of diminishing the length of the longest internal boundary every other level of recursion. To obtain strips (c) the algorithm makes cuts in one direction only.

Our philosophy for doing load balancing is to do it cyclicly and as often as possible. We make no provision for balancing only when load imbalance exceeds a user-specified threshold value, as others have advocated. However, given that `Partitioner` has all the necessary information to measure load imbalance– the work estimate mapping–thresholding could be installed without difficulty.

In our discussion we have specifically ignored certain kinds of partitionings. First, we assume that the number of subproblems that `Partitioner` can render remains constant over time. We have also assumed that in the course of repartitioning, each processor can be identified with a volume of space that shrinks, grows, and translates by no more than a given small amounts. As a result, workloads will shift gradually among processors. In particular, no cut may change directions (from horizontal to vertical or vice-versa) from one call of `Partitioner` to the next. Consider, for example, the simplest two-processor case in which the single cut changes direction, as shown in Figure 4.14. Roughly half the work gets redistributed. Such a massive work redistribution can incur a communication overhead that overwhelms any savings that result from improving the workload balance. We therefore disallow it. We have also assumed that all processors run at the same rate. A simple change to the algorithm would accommodate processors that ran at different speeds, so long as the speeds didn't vary too quickly over time.

4.6.2. Mapper

We may specify the internal behavior of the utility in terms of a message-passing communication model. The use of this model facilities the discussion, though the implementor is free to design `Mapper` however he wishes. Let us characterize the message passing model, perhaps simplistically, by two primitive operations `send` and `receive` (see, for example, the *iPSC User's Guide* [5] for the details of how message passing works in practice). Invoking `send(buffer,id)` sends the message in `buffer` to the processor designated by `id`.

```
-- Recursively bisect a region of the workLattice into a list of sublattices
-- of roughly equal weights.
-- The weight of a sublattice is defined as the sum of the weights of its bins
-- A workMap, provided as input, weights each bin of the lattice and is used
-- in placing the cuts. The algorithm may return some empty partitions
-- if too many workMap entries are zero.

proc rba(q:partition,dir:int,P:int,
        maxDelta:int,wm:workMap) returns partList
-- q              is the region of interest
-- dir=0 or 1     cut in the horizontal or vertical direction
-- P              number of desired sublattices
-- maxDelta       the maximum distance that any partition is allowed to move
--                relative to the cuts generated by the previous call
-- wm             the workMap

-- Internal state maintained by the algorithm to enforce the constraint
-- on the motion of a partition's corners.
        oldCuts array (*) of int

-- If you asked for only 1 sublattice, return the original lattice
-- partList() converts from type partition to type partList
        if (P = 1) return partList(q)

        P1 := P div 2
        P2 := P - P1

-- Split into two parts, with no cut moving more than maxDelta units
-- relative to oldCuts; in particular no cut may switch directions
-- The ratio of the weights of the 2 parts will be as close as possible to P1/P2
-- In particular, if P is a power of 2, split into two equally-weighted parts
-- For a horizontal cut return the left sublattice
-- For a vertical cut return the upper sublattice
-- Align the cut along direction 'dir'; if that fails, and this
-- is the first time through the Partitioner, try the other direction

        leftPart := split(q,dir,P1/P2,wm,oldCuts)

-- Recurse; if you got back only one sublattice from split,
-- any remaining sublattices will be empty
-- Split returns an empty partition if its input is also empty
-- The append operator joins together two partLists
-- The infix '—' operator forms set differences on proper subsets

        return append(rba(leftPart,1-dir,P1,wm),
                      rba(q—leftPart,1-dir,P2,wm))
end rba
```

Figure 4.13. The recursive bisection algorithm.

Invoking `receive(buffer,id)` allows an incoming message into `buffer` and sets `id` to identify the processor, if any, that sent the message. Message buffers are flexible byte sequences, as discussed earlier in the chapter.

`Mapper` knows how to handle communication in terms of the global coordinate system of the workLattice. It uses two routines–`pack` and `unpack`–whose internal behavior it knows nothing about to make the necessary conversions to the local coordinate system of each task's private address space. Mapping divides into two activities, an influence action and a dependence action. The influence action uses `pack` to collect variables from the the designated regions of Λ into byte sequences, and sends the information to any processor needing them. The dependence action invokes `unpack` to copy incoming information from message buffers into newly-allocated storage assigned by the user. The influence and dependence actions execute concurrently.

`Pack` provides `Mapper` with a stream of data, though the stream may have to be broken into pieces owing to the finite length of `Mapper'.s` message buffers. So although `Mapper` doesn't know how `pack` and `unpack` work, it does know that they may have to be restarted. `Mapper` expects `pack` to set a flag signaling that more data is to come. `Unpack` doesn't use a flag, since it relies on `pack` to furnish the appropriate restart information in the message-buffer.

4.7. Summary

We have outlined a simple approach to writing numerical software for multiprocessors. It relies on using a virtual machine whose semantics are unaffected by the architecture on which it is implemented. The VM is not universal or complete, however; it applies only to localized parts of a computation, and leaves some programming details up to the discretion of the user, and to the particulars of the system he is using. The user must divide the data and computation for the local part of the problem into bins of a regular rectangular mesh, must supply work

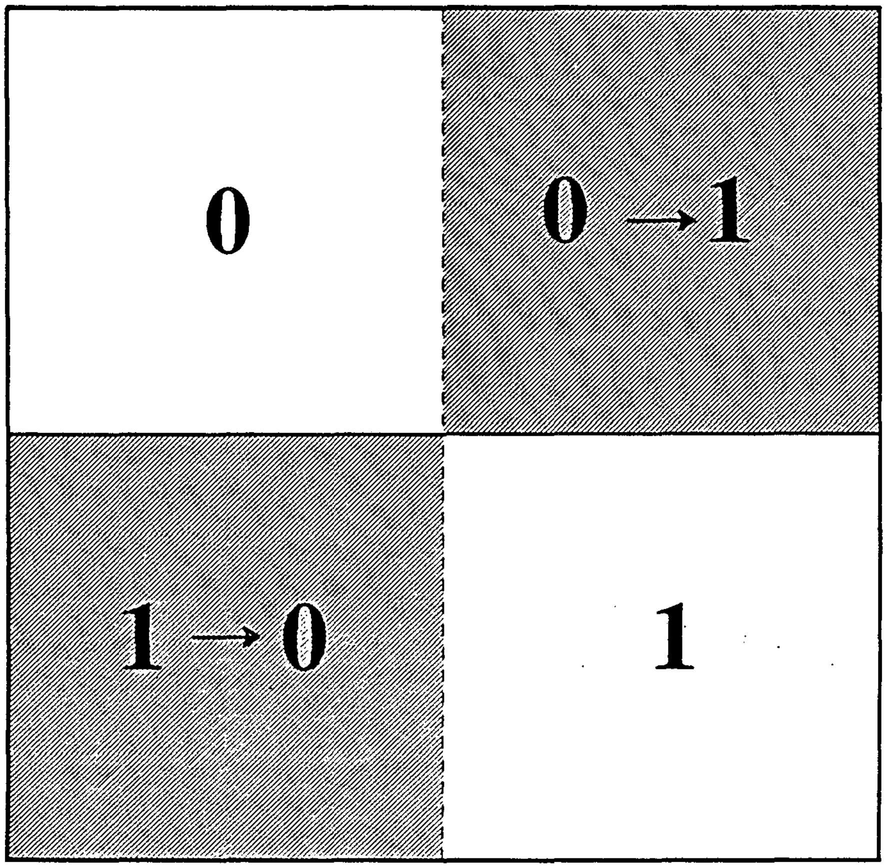


Figure 4.14. Partitioner does not allow cuts to change direction, as shown here, since such cuts would result in massive work redistribution that would incur a high communication overhead. We show the simplest case with two processors in which the work in the shaded regions migrates to the other processor.

estimates for the computation in each bin, and must supply routines for converting these data to and from byte streams. The VM will assign bins to tasks in order to even the workload and will allow each task to access and communicate the necessary boundary data.

Our VM is not universally suitable even for local computations. In chaotic algorithms (cf. Chazan and Miranker [2]), the state transitions are, in effect, far too numerous to perform efficiently using `Mapper`. However, there is a wide class of well-behaved local computations for which the VM should work well, from which we have selected the MLC as an example.

The purpose of using the MLC as a model computation is that it is a good model for other computations in mathematical physics. The localized computation supported by our `Mapper` utility is an essential activity that arises not only in the MLC and in other particle methods, but also in finite-difference methods. In adaptive mesh refinement algorithms (AMR), for instance, systems of locally refined grids interact across grid interfaces, as shown in Figure 4.15. Boundary conditions must propagate across the interfaces regardless of whether the computation has been implemented on a multiprocessor. AMR is also interesting because partitioning the data structures introduces some redundant computation along newly-generated grid-boundaries. If the grids were not split, then these values would be computed only once. Unlike the MLC, this problem is intrinsic to the AMR algorithm, and not just to the implementation.

Our VM has two attractive attributes that together contribute to the writing of simpler code, in the local part of the computation: (1) it hides all the details of interprocessor communication from the programmer; (2) it doesn't need to know how the application's data structures are represented. Application-dependent code and system-dependent code need never become heavily intertwined; were the code transported to a new machine, the parts that would have to change to accommodate a different communication model are restricted mostly to code the programmer never sees.

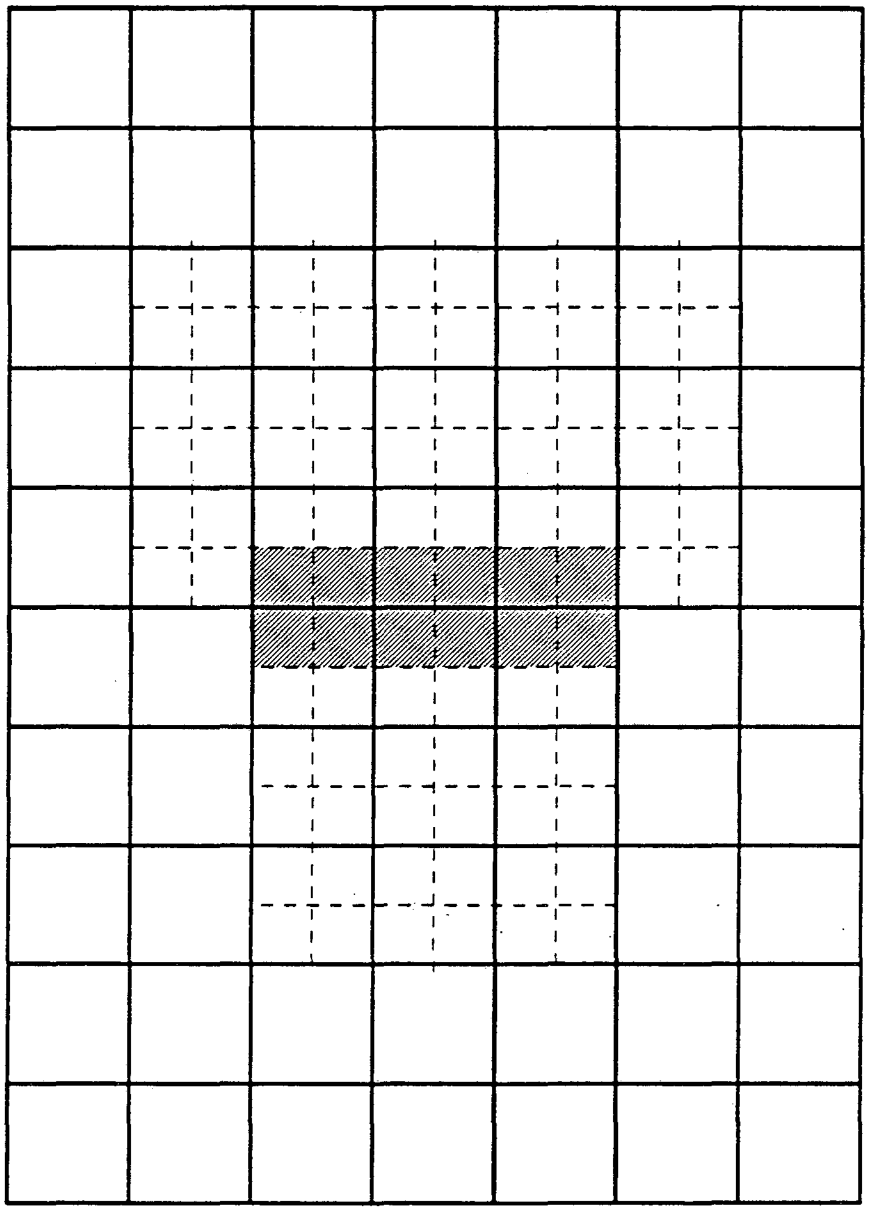


Figure 4.15. In adaptive mesh algorithms, boundary conditions must propagate across grid interfaces for numerical reasons. This activity is identical to the mapping process used by our virtual machine. We have shaded an interface shared by two grids at the same level of refinement. Some interfaces have not been shown; they connect grids at different levels of refinement.

4.8. References

1. M. J. Berger and S. Bokhari, "A Partitioning Strategy for Non-Uniform Problems on Multiprocessors," Technical Report 85-55, ICASE, Langley, Va., November 1985.

2. D. Chazan and W. L. Miranker, "Chaotic Relaxation," *Linear Algebra Appl. 2*(1969), pp. 199-222.

3. M. E. Dippé and J. A. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *SIGGRAPH '84 Conference Proceedings*, Minneapolis, July 1984, pp. 149-158.

4. M. A. Z. Dippé and E. H. Wold, "Antialiasing Through Stochastic Sampling," *Computer Graphics 19*,3 (July 1985), pp. 69-78, ACM. Also appeared in the SIGGRAPH '85 Conference Proceedings.

5. *iPSC User's Guide*, Intel Corporation, Beaverton, Oregon, October 1985. Order Number: 175455-003.

6. R. M. Karp, "Probabilistic Analysis of Partitioning Algorithms for the Traveling-Salesman Problem in the Plane," *Math. of Operations Res. 2*,3 (August 1977), pp. 209-224.

7. D. E. Keyes and W. D. Gropp, "A Comparison of Domain Decomposition Techniques for Elliptic Partial Differential Equations and their Parallel Implementation," YALEU/DCS/RR-448, Yale Univ., Dept. of Computer Science, December 1985.

8. C. Moler and D. S. Scott, *Communication Utilities for the iPSC*, Intel Scientific Computers, Portland, OR, August 1986. Technical Report.

9. J. E. Warnock, "A Hidden Surface Algorithm for Computer Generated Halftone Pictures," TR 4-15, University of Utah, Computer Science Dept., June 1969.

5

iPSC Implementation

My own interests are in using computers as God intended – to do arithmetic.

–Cleve Moler

5.1. Overview

From the standpoint of this dissertation, traditional MIMD multiprocessor architectures differ primarily according to how they implement interprocessor communication, and come in roughly two varieties: message-passing or shared-memory architectures[1]. In this chapter we evaluate our VM on an architecture of the first kind: the Intel Personal Scientific Computer (iPSC), manufactured by Intel Scientific Computers of Beaverton, Oregon. In chapter 6 we will consider a shared-memory architecture. We will not compare the cost-effectiveness of the iPSC with other systems; that is out of the scope of this research. Instead, we will show that the performance of an iPSC system running our VM can scale reasonably well with the number of processors in use. The chapter begins with a description of the iPSC system. We will then pose a

[1] The IBM RP3 [8] is a notable exception. The user may divide the RP3's physical memory address space into two parts: a global part, shared by all the processors, and a local part, divided equally among the processors into private sections. When there is no local part, the RP3 operates as a shared-memory architecture, when there is no local part the machine operates as a message-passing architecture. In between these two extreme settings, the RP3 operates in a "hybrid" mode.

series of questions, answering each question with the aid of computational experiments conducted with the Method of Local Corrections.

5.2. The iPSC

The iPSC is a hypercube-type multiprocessor inspired by the Caltech cosmic cube [10]. An iPSC system may be configured with 32, 64, or 128 processor nodes that communicate by sending messages over the hypercube interconnection network shown in Figure 5.1. Nodes may communicate with a host processor but may not communicate with the outside world in any other way. The 32 processor model d5 we used is nominally a 1 megaflop machine and has been observed by Moler [6] to deliver 0.8 megaflops on Gaussian elimination. A single processor node delivers about 0.033 megaflops in double precision for the *daxpy* operation used in LINPACK and the BLAS [2]:

```
for i = 1 to n do
        y[i] = y[i] + a*x[i]
end
```

Each node consists of the following off-the-shelf VLSI parts: an Intel 80286 central processor and 80287 arithmetic co-processor each running at 8 megahertz[2]; 512 kilobytes of local memory, of which about 300 kilobytes are accessible to the user; and eight ethernet communication ports. Seven of the eight ports provide dedicated links to up to 7 nearest neighbors. All of the links are used in the model *d*7 with 128 processors; the model *d*5, with 32 processors, uses only five of seven. The eighth port communicates with the host via a multiplexed ethernet channel. The host is an Intel 286-310 system with 4 megabytes of memory, 30 megabytes of

[2] The 80286 overlaps indexing with the floating point arithmetic done on the 80287.

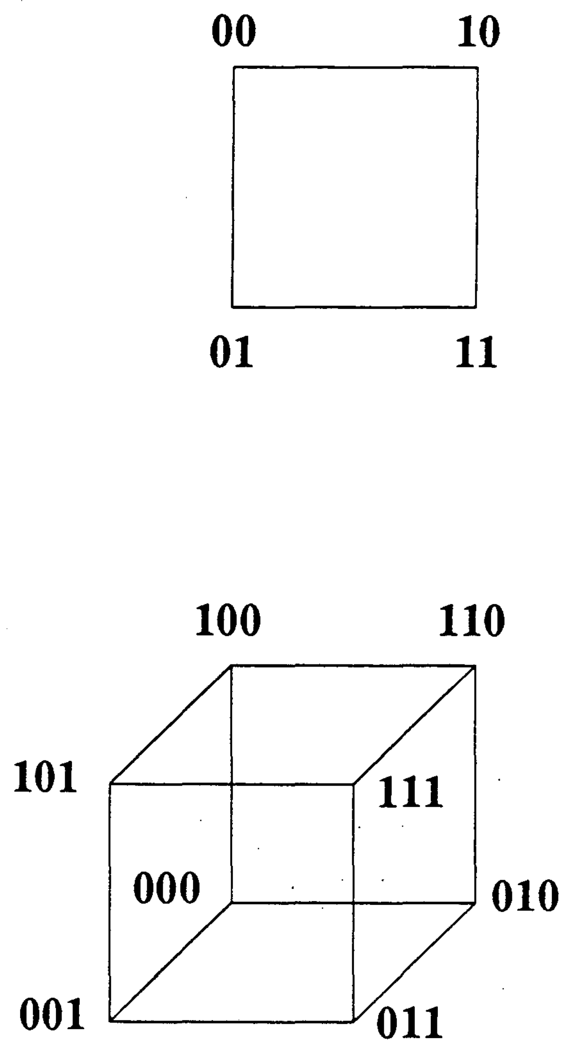


Figure 5.1. A two-dimensional hypercube, or 2-cube (upper), and a 3-cube (lower). A hypercube of dimension d, called a d-cube, has 2^d nodes. Each node communicates with d nearest neighbors over dedicated links, and with nodes with which it does not share a link by making intermediate hops. If the nodes are labeled with integers from 0 to 2^d-1, then the number of hops between two nodes equals one less than the Hamming distance between them. The Hamming distance is defined as the number of places in which the bit strings of the labels differ. The maximum distance between two nodes is d and follows from the inductive property of hypercubes. A 3-cube, for example, may be inductively constructed from two 2-cubes by connecting each pair of corresponding nodes. This process may be used to construct higher dimensional hypercubes. The inverse operation, tearing, may be used to partition hypercubes.

disk storage, and two ethernet ports. One port connects the host to the multiplexed ethernet channels and the other is used for diagnostics. Both host and nodes run a modified version of the Intel XENIX-286 operating system. The iPSC is a standalone system; though several users may share the host, only one at a time may use the hypercube processor array.

The processing nodes of a hypercube multiprocessor may be thought of as forming the 2^d corners of a d-dimensional hypercube, or *d-cube* for short. Figure 5.1 shows a 2-cube with 4 nodes, and a 3-cube with 8. Each node communicates directly with d nearest neighbors, and with any node that is not a neighbor via intermediate nodes. On the iPSC, however, as on most other hypercubes, the user sees a fully connected network; the operating system routes messages transparently through intermediate nodes. There can be at most $d-1$ hops involving intermediate nodes, so the worst-case communications delay varies as the logarithm of the number of nodes. In the absence of any other message traffic, nearest processors in the hypercube interconnection network communicate at a rate that ranges from 160 kilobytes/sec for a 1 kilobyte message to 286 kilobytes/sec for the maximum-sized message of 16 kilobytes. The following expression gives the time to send a message of n bytes:

$$T_{setup} + T_{packet} \times (\lceil n/1024 \rceil - 1) + T_{transmit} \times n \tag{5.1}$$

where $T_{startup}$ is a fixed *setup time*, T_{packet} is the *packetization cost* charged to every packet after the first, and $T_{transmit}$ is the transmission time per byte (One way of interpreting message latency (5.1) is as half the time for a message to make a round trip between two nearest neighbors). For release 2.1 of the operating system that we used, these times are[3]:

$T_{startup}$	=	5.0 milliseconds
T_{packet}	=	2.6 milliseconds
$T_{transmit}$	=	0.87 *microseconds*

[3] Release 2.1 is now obsolete and the the startup time has been reduced to about 1 millisecond.

The 5 millisecond startup cost dominates the cost of transmitting short messages of up to 1 kilobyte.

Because iPSC uses a store-and-forward method for implementing message-hops, each intermediate node must buffer an incoming message in its memory and then read out the message upon retransmission. The cost of forwarding a message would therefore be expected to be proportional to the number of hops. Rudell [9], however, has shown that owing in part to the high cost of starting up a message, forwarded messages go faster than expected; a 5 hops message, for example, takes only about 2.5 times longer to reach its destination than does a similar message passed to a nearest neighbor, regardless of its length.

Message latency increases in the presence of other message traffic. Because the memory used to buffer messages cannot keep up with more than one ethernet communication channel at a time (the CPU also competes for memory bandwidth), messages that arrive at a node at the same time will be processed sequentially. Message latency under such conditions is unpredictable; the receiving processor will force all but one of the senders to retransmit at a later time. A node may also reject incoming messages if it runs out of internal buffers used by the operating system to hold messages. Until it has disposed of a sufficient number of buffered message packets, the node will force the senders to keep retrying. This is obviously a severe problem when many messages are going through intermediate nodes, and is prone to deadlock. We did not observe a hard deadlock, however, though we did observe transient deadlock, lasting for *seconds* or even *minutes*. The long delays occurred when two communicating processors, each with a full memory, backed off at the same rate, and continued to do so until their clocks grew sufficiently out of phase[4].

There are only 16 system calls for the iPSC: blocking and non-blocking message-passing primitives (discussed below), a 60 Hz timer, and some calls to manage communication channels.

[4] This problem has since been fixed in the latest release of the node operating system.

The *iPSC User's Guide* [3] offers a more complete discussion than can be given here. Timing measurements can be taken from the nodes only (the host has no timer) and may be resolved down to about 33 milliseconds.

In simple terms, the iPSC provides two message-passing primitives: `send` and `recv`. `Send(buffer,length,type,destination)` passes `length` bytes of information in the message `buffer` to the `destination` processor, and tags the message with the specified `type`. `Recv(buffer,length,type,sender,count)` filters out messages of the prescribed `type` only and ignores all others. It places the incoming message into `buffer`, writes the identity of the processor that sent the message into the `sender` argument, and writes the length of the received message into the `count` argument. The `length` argument gives the maximum size message that `recv` can accept; longer incoming messages will be truncated to that length.

`Send` and `recv` come in two flavors: non-blocking and blocking. Non-blocking message-passing is an asynchronous activity that can proceed in parallel with computation; a return from `send` or `recv` does not necessarily signal completion of the action. The message buffer must not be used again until the `status` routine determines it is free for re-use; otherwise, unpredictable behavior may result. The blocking versions of `send` and `recv` have procedure-call semantics; a return from `sendw` (''send wait'') indicates that the buffer is free for re-use, although the message may not yet have been received at its destination. A return from `recvw` (''recv wait'') indicates that the message has been received. Messages arriving at a node from a particular source will be received in FIFO order, i.e. in the order sent. With multiple senders the FIFO ordering will be preserved within each group of messages coming from a single source.

5.3. Implementation

5.3.1. Overview

Because we set them up in much the same way as we did in chapter 4, we will not discuss the calls to the VM's utilities in great detail. We will, however, say a few things about how we implemented the VM. All software was written in FORTRAN 77 and compiled using the Intel *ftn286* compiler. Two programs were written: one for the host and the other for the nodes. The host executed as the boss task. It did all the I/O on behalf of the nodes, such as reading in simulation parameters, and it also ran the `Partitioner` utility. The nodes executed as the worker tasks; all executed the same program and handled the numerical parts of the computation. Each node processor recorded its own timing information about the various phases that make up each time step of the calculation. At the end of a run the host collected these data from the nodes and wrote them to a file. A separate program reduced the raw data to report the times spent communicating, doing localized computation, solving Poisson's equation, doing task partitioning, and so on.

5.3.2. Virtual Machine Implementation

We chose to run the boss on the host, in order to conserve node memory. Because of that, `Partitioner` was obliged to communicate with the nodes both to obtain a copy of the work estimate mapping and to return the table of subproblems it had rendered. The cost of this communication, however, was unnoticeable.

Mapping, as discussed heretofore, divides into two concurrent activities: the dependence action and the influence action. Though the iPSC nodes may be multiprogrammed, we elected not to implement the influence and dependence actions as parallel processes, but to achieve a similar effect by interleaving the two actions. This roughly balances incoming and outgoing message traffic to help avoid transient deadlocks that won't lock up the code permanently but which could slow it down. However, per our previous comments, we still experienced some problems with the nodes temporarily locking up.

The iPSC implementation of `Mapper` diverged from the specification in one way: the user-supplied `pack`, `unpack`, and `reclaim` routines were hardwired into `Mapper` and were not passed to it as subroutine-arguments. This was done in the interest of expediting a rapid prototype of the code and was never changed. The code could be modularized in such a way that the user-routines could be passed to `Mapper`, without introducing unreasonably high overhead costs. The additional overhead would come from having to call a subroutine passed as an argument, whose name was not known at compile-time, instead of calling a subroutine whose name was known statically. This overhead, however, is slight: the number of messages that `Mapper` will send is proportional to the number of calls to the external subroutines; each message costs at least 5 milliseconds; a subroutine call, including argument transfer, costs tens of *microseconds*.

5.3.3. Global Computation and Communication

To handle global computation and communication we relied on various utilities supplied by others. One utility is a fast 9-point Poisson solver, written by Cleve Moler of Intel Scientific Computers. For P processors and an $M \times M$ finite difference mesh, the solver uses a direct solution method involving $O(M^3/P)$ arithmetic and $O(M^2/P)$ storage per processor[5]. The solver gets called twice during each velocity evaluation, once for each component of velocity. Because of the relatively high communication overhead of the solver (see Figure 5.2), we found that computing each velocity component on $P/2$ processors, and in parallel with the other, was faster than computing the two components serially, but on P processors.

[5] The method is based on knowing the eigenvalues and eigenvectors of the 9-point difference operator for the Laplacian. The dimensions of the grid need not be a power of two, unlike an FFT method. The code for the solver appears in the Appendix.

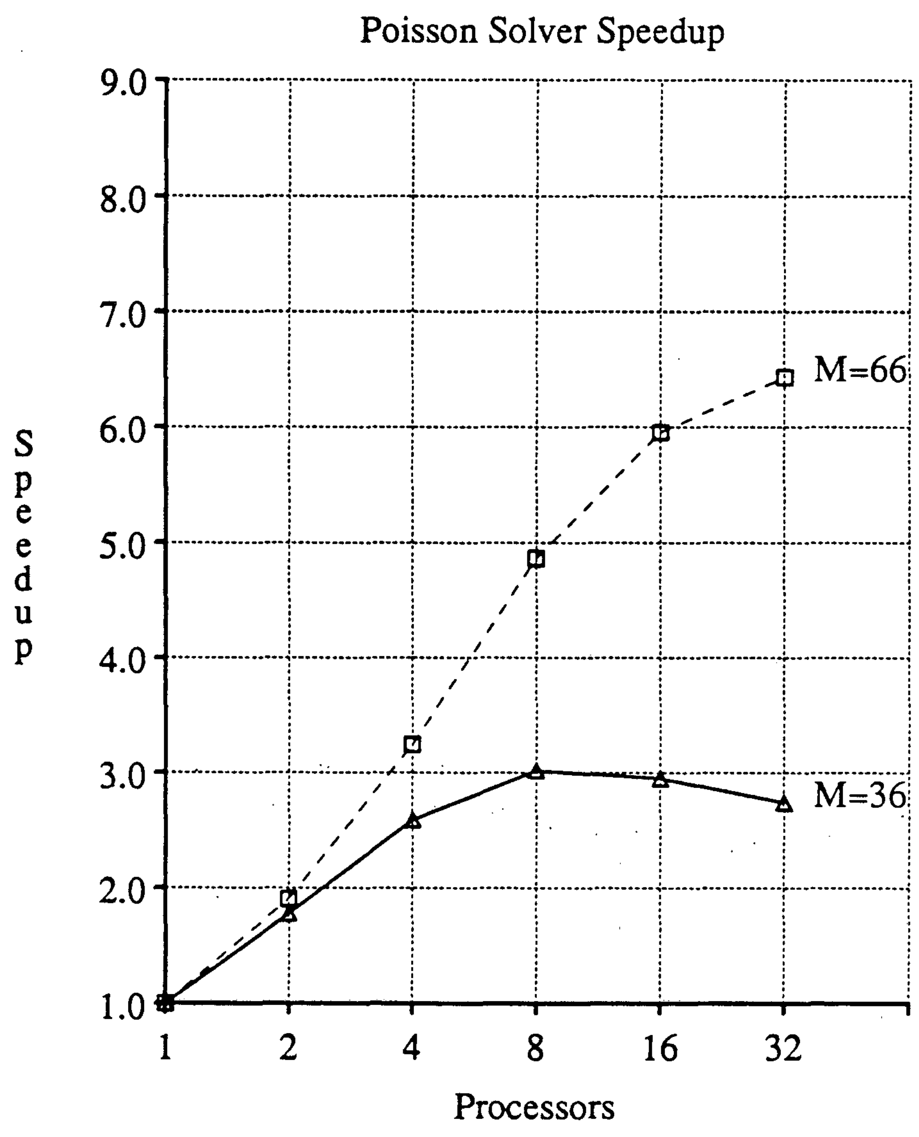


Figure 5.2. For the 36×36 finite-difference mesh we used in our calculation, the solver runs fastest on 8 processors. Beyond that point, the increased cost of communication overwhelms any savings in computation time. The cost of communication decreases relative to computation as the size of the mesh increases. For a 66×66 mesh, for example, the maximum achievable speedup is roughly twice what it is for the smaller problem.

The solver assumes that the input resides in the memory of a designated node and returns its result to that same node. Each node, however, contributes only a local portion of the right hand side in the parallel MLC algorithm. The local right hand sides must somehow be combined into a single global right hand side. In addition, each node must also obtain a copy of the solver's result. To handle these activities we relied on communication utilities, discussed by Moler and Scott [7], that use a spanning tree communication structure. There were two routines: (1) `gsum`, that accumulates like-size arrays stored on different processors into a single array of that size stored on a designated root processor; and (2) `gsend`, that broadcasts an array from the root processor to all other processors.

5.3.4. Arithmetic

All arithmetic was done using 8-byte double-precision numbers. The program used three major data structures that were duplicated on all the processors. The major data structures used were: three 42×42 finite-difference meshes; three 84×84 arrays of 2-byte integers used for the bins and to do work estimation; and vortex-records, each describing a single vortex. Each bin is a pointer to a list of vortex records, each consisting of 154 bytes of information: 2 real-valued position vectors; 2 real-valued velocity vectors; real-valued vortex *strength*, analogous to an electrostatic charge; 5 complex-valued interpolation coefficients; and a 2-byte integer pointer that links vortices into the bins. To economize the iPSC's scarce memory a short form of the vortex record was also used to hold copies of vortices obtained by `Mapper` from other processors. Since the only information needed about such vortices is position and strength, the short vortex record is 26 bytes long and consists of 2 real-valued position vectors, 1 real-valued strength, and an integer link. For each of the 32 processors there were many more short-form vortex-records than long-form ones; 1500 of the former and 250 of the latter. The major data structures therefore consumed 162 kilobytes of each node's memory. In addition, the `Mapper` utility used two message buffers that were 10080 bytes long each, leaving about 120 kilobytes that contained mostly code.

5.4. Evaluation Strategy

The ideal way to evaluate an implementation would be to take account of adapting the mathematical algorithm to the hardware, but this lies beyond the scope of our research. As a result of this restriction, we will assume that the cost of non-parallelizable computation, or computation that does not parallelize well, cannot be significantly changed. The task at hand is therefore to find out how well our VM can utilize the iPSC's hardware, given that only a certain fraction of computation parallelizes well, and communication has a certain cost relative to computation. We will not be concerned with the cost-performance of our implementation compared to implementations on other architectures. What we are interested in is relative cost-performance, i.e. how well performance scales with the number of processors used. We will pose a series of questions, and answer each question using the appropriate measurements taken from runs of the MLC.

All results were obtained from runs involving the finite area vortex (FAV) problem of Figure 4.10: the vortices were positioned on a lattice of points confined to two circular patches placed symmetrically about the origin. The patches had a radius of 0.12 units and their centers were 0.25 units apart. N, the number of vortices, was kept proportional to P, the number of processors; this is consistent with the expectation that ever-larger problems may be handled as computational resources increase. Each processor was initially assigned about 100 vortices regardless of the size of the problem though, as a result of the motion of the vortices and of the dynamically changing partitionings of the problem, the exact number fluctuated with time. Experiments were run on 4, 8, 16, and 32 processors but not on 1 or 2; the problems that 1 or 2 processors could accommodate – about 100 or 200 vortices – are too small to overcome the fixed overhead costs of the finite difference part of the calculation. Worthwhile problems should have at least several hundred vortices and more likely several thousand.

A calculation involving 3180 vortices ran at a rate of 3.5 minutes per time step on 32 processors. Table 5.1 compares the average execution time per timestep for various values of N. The timestep Δt was fixed at 0.05, and all runs lasted 64 timesteps or, equivalently, 3.2 units of simulation time. We did not scale the timestep Δt with N, as suggested by results of the 3 parameter study presented in chapter 3. We would liked to have decreased Δt to 0.025 for the largest problem, but doing so would have increased the running time of the calculation to an unreasonable amount of time – about 11 hours – so we had to settle for a larger timestep.

The computation took place within the unit box, extended in all directions by the correction radius. We ran with a 36×36 finite difference mesh. The mesh spacing h was fixed at 1/30, and the correction radius C was set at $2h$. Owing to insufficient memory, we were unable to scale h and hence C with the initial spacing of the vortices. Given more memory we would have decreased h from 1/30 to 1/60 for $N = 3180$ vortices. However, we did have sufficient memory to reduce the spacing of the bins to one-half that of the finite difference meshes, i.e. 4 bins lay under each finite difference panel. This was done because `Partitioner` can do a better job of balancing workloads as the bins become smaller and more numerous. If there were sufficient memory to refine the finite difference mesh, however, the bins would have the same size as the finite difference mesh-boxes, and would not have to be made any finer.

N	P	iPSC time (minutes)
386	4	1.3
796	8	1.6
1586	16	2.2
3180	32	3.5

Table 5.1. The average running time per timestep on the iPSC (in minutes) averaged over 64 timesteps. N is the number of vortices and P the number of processors. The loads were rebalanced every other timestep or every fourth velocity evaluation – each timestep did two velocity evaluations. The computation time is roughly proportional to $P^{1.7}$.

5.5. The Experiments

The first question we ask is: how well does performance scale with P ? To answer this we compute parallel efficiency, a familiar performance metric for evaluating multiprocessor implementations. Kuck [5, p. 33] defines η_P as the efficiency with P processors:

$$\eta_P = \frac{T_1/P}{T_P} \tag{5.2}$$

where T_P is the time to complete on P processors. T_1 is the time taken on a uniprocessor. For this special case of $P = 1$, various overheads that would be incurred on a multiprocessor, such as communication, are non-existent.[6] By definition $\eta_1 = 1$.

Exclusive of the Poisson solver, the size of the problem increases with the number of processors, and so T_1 cannot be measured directly. A problem with 3180 vortices, for example, would take about 100 hours to complete on 1 processor. We must therefore rely on indirect means of measuring T_1. T_1 comprises two parts: T_1^{pois}, the time spent solving Poisson's equation, and T_1^{Local}, localized computation split up by `Partitioner`. T_1^{pois} can be measured directly on a uniprocessor, because the size of the problem passed to the solver doesn't change with the number of processors. T_1^{Local} is the part that cannot be measured directly. But, since `Partitioner` roughly conserves the total amount of work that would be done in a uniprocessor computation (this is true because the work-estimate mapping is almost set-additive), T_1^{Local} can be reasonably approximated by summing up the completion times for all P processors:

$$T_1^{Local} = \sum_{i=0}^{P-1} T_P^{Local}(\Lambda_i), \tag{5.3}$$

[6] The iPSC implementation of the MLC was written to work on any number of available processors and no attempt was made to optimize it for the special case of one processor. However, the spurious overheads thus incurred on a uniprocessor run are slight enough that they can be ignored.

where $T_P^{Local}(\Lambda_i)$ is the time to complete the localized part of subproblem Λ_i on P processors.

Using equations (5.2-3), we determined the parallel efficiency from the timing information reported by each node. As shown in the final column of Table 5.2 or, equivalently, in the top and bottom curves of Figure 5.3, η_P ranges from 90% with 4 processors to 74% with 32. This is quite good; if efficiency were 100%, the program would run at most only 35% ($(1-\eta_P^{-1})\times 100\%$) faster.

The next question we ask is: why does η_P fall short of 100% efficiency? Generally there are three reasons why: (1) some computations must run on only one processor and act as serial bottlenecks; (2) processors must spend some of their time communicating; (3) loads are not perfectly balanced. We next measure the cost of these three factors.

A different measure of efficiency than the one used before, that ignores all but serial bottlenecks, is called *maximum theoretical efficiency*, $\bar{\eta}_P$. This is the efficiency that could be attained under ideal conditions of instantaneous communication and perfectly balanced workloads. It therefore establishes an upper bound on efficiency, since we assume that serial bottlenecks cannot be substantially reduced except by a change of algorithm.[7] $\bar{\eta}_P$ is defined as

$$\bar{\eta}_P = \frac{(T_1^{compute} + T^{partition})/P}{(T_1^{compute}/P + T^{partition})} \tag{5.4}$$

where $T^{partition}$ is the time spent partitioning, and

$$T_1^{compute} = T_1^{local} + T_1^{pois} \tag{5.5}$$

The denominator of (5.4) reflects perfect parallelization of the computational work, but no parallelization of `Partitioner`.

[7] The cost of the serial bottleneck generally depends on the architecture as well as the algorithm. However, the subject of adapting algorithms to architectures is not within the scope of this research. We assume that the parts of the computation that execute serially have been appropriately minimized.

As shown in Table 5.2, (or equivalently in Figure 5.3) maximum theoretical efficiency is never less than 98%; our implementation of the MLC parallelizes well and includes only a small amount of computation that must be done on a single processor – `Partitioner`. Furthermore, the cost of this non-parallelizable work, relative to that of numerical computation, is relatively insensitive to the number of processors in use. The observed efficiency, by comparison, decreases much more sharply than the maximum theoretical efficiency as the number of processors increases. Load imbalance is the major difficulty here; communication and other overhead are comparatively benign. This can be seen by looking at the idealized efficiency successively degraded by the various sources of overhead, shown in Table 5.2 and plotted in Figure 5.3. Two new measures of efficiency divide the difference between the idealized and observed efficiency into three parts: the upper part represents efficiency losses due to communication overhead, except what was incurred in the solver, the middle part represents efficiency losses incurred by the Poisson solver; the lower part represents the losses due load imbalance. The total

N	P	Idealized Efficiency			Observed	
		Max Theor	With Comm	Comm + Solve	η_P	S_P
386	4	0.990	0.961	0.944	0.904	3.6
796	8	0.988	0.946	0.922	0.849	6.8
1586	16	0.988	0.938	0.914	0.787	13
3180	32	0.990	0.941	0.918	0.738	24

Table 5.2. Parallel efficiency and speedup figures. N, the number of vortices varies linearly with P, the number of processors. Columns 3 to 5 give three measures of idealized efficiency. "Max Theor" assumes that loads are perfectly balanced and ignores all communication overhead. It assumes that partitioning is work that could not be parallelized. "With Comm" includes the cost of communication that was ignored in the previous figure, and finally, "Comm + Solve" adds in the efficiency loss that results from imperfect parallelization of the solver. Columns 6 and 7 give the observed efficiency, η_P, and speedup, S_P, that include all overhead costs. By definition $\eta_P = S_P/P$. The differences between adjacent columns in columns 3 to 6 give the efficiency losses due to three factors: communication overhead (including spanning tree communication), solver overhead, and load imbalance, respectively. The losses incurred by the first two factors increase much more slowly with the number of processors than do the losses due to load imbalance.

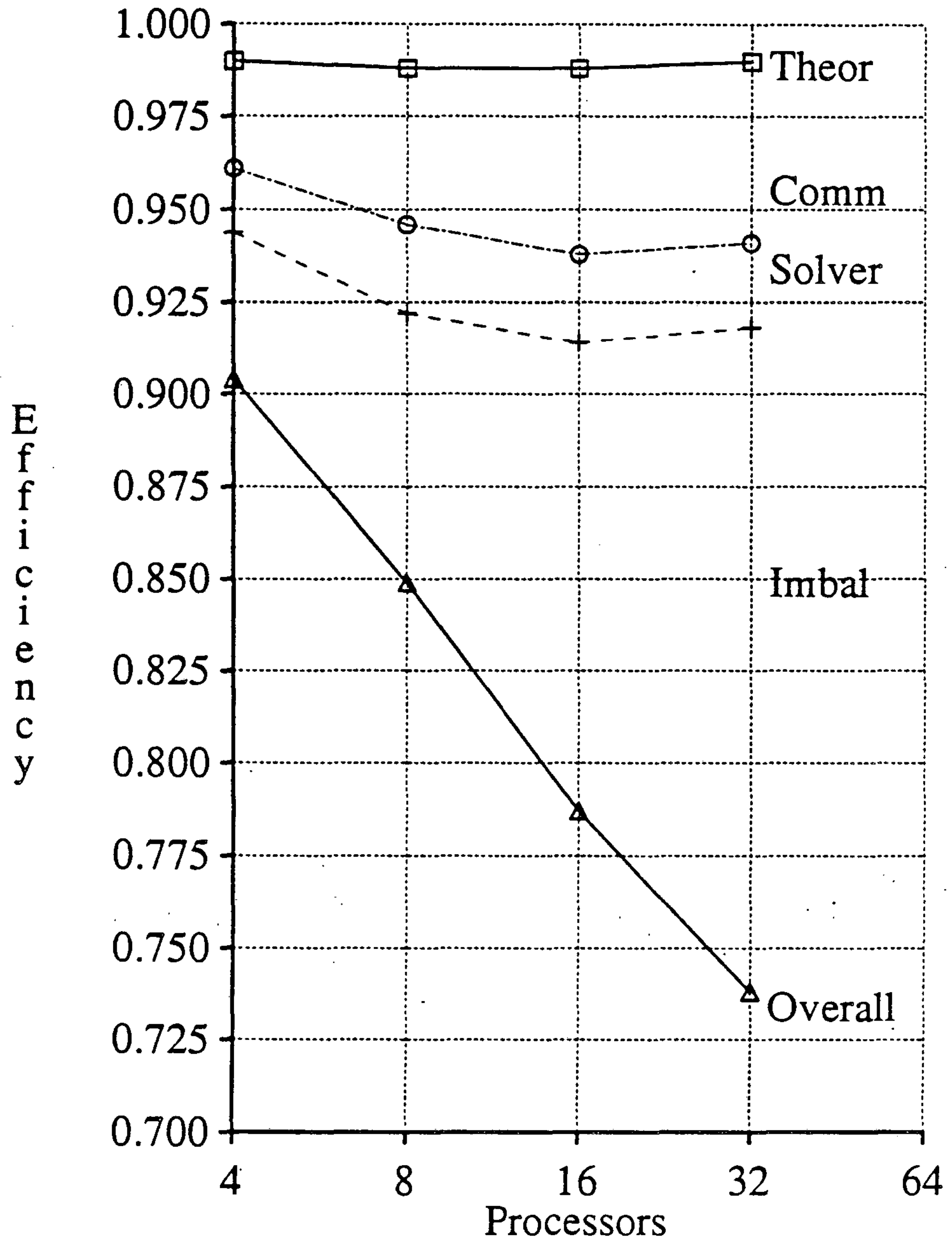


Figure 5.3. Parallel efficiency decreases as the number of processors increases. The top curve gives the maximum theoretical efficiency that would be attained under ideal conditions. The bottom curve gives the efficiency observed for all phases of the computation. The two curves in the middle divide the efficiency losses represented by the gap between the upper and lower curves into 3 regions, corresponding to losses due to communication, the Poisson Solver, and to load imbalance, respectively. Loads were balanced every other time step, or every fourth velocity evaluation.

communication overhead increases gently with the number of processors and the solver overhead decreases gently, while load imbalance increases sharply. We will next explain the solver and other overheads, and return to the problem of load imbalance later.

We identify three sources of efficiency loss: (1) poorly-parallelizing computation – the Poisson solver, (2) partitioning, and (3) communication. The solver we used does not parallelize well because it communicates frequently with respect to computation. However, the fraction of time spent in the solver depends both on the numerical algorithm and its implementation, topics which we have explicitly excluded from discussion. We therefore are more interested in the solver's overall cost, relative to the entire computation, rather a breakdown of its overhead costs. This is consistent with the specification of the VM: the VM does not apply to a computation like the solver, but merely accepts it as global computation that does not parallelize well. We can determine how throughput is affected by a change of solver simply by measuring the solver's execution time and then applying Amdahl's law. The solver overhead shrinks with P because the while the amount of localized computation grows roughly as $P^{1.7}$, the size of the solver's computation remains fixed. This is shown in Table 5.3.

N	P	Computation		Overhead			
		Local	Pois	total =	partition +	mapping +	global
386	4	0.83	0.13	0.040	0.012	0.003	0.026
796	8	0.88	0.07	0.050	0.012	0.008	0.030
1586	16	0.91	0.04	0.053	0.010	0.014	0.029
3180	32	0.92	0.02	0.046	0.008	0.016	0.022

Table 5.3. This table gives the fraction of time spent in localized computation (''Local''), the Poisson Solver (''Pois''), and in various overheads. ''Total'', is the total fraction of time spent in computation that would not be done on a uniprocessor and divides into three subtotals: ''partition,'' Partitioner overhead; ''mapping'' Mapper overhead, and ''global,'' the overhead of doing global summations and broadcasts. The fractions don't quite add up to 1.0 owing to a slight uncertainty in the measurement technique.

Exclusive of the Poisson solver, communication overhead ranged from 2.9% on 4 processors to 4.3% on 16. Communication serves two purposes: (1) to do mapping and (2) to manipulate the finite difference grids used by the solver. The first activity was less expensive than the second; it incurred a communication overhead that ranged from 0.3% to 1.6% of the total execution time. This overhead was low because the partitions have simple shapes and because `Mapper` amortizes the iPSC's high message startup cost over the sending of several vortices in one message. As the number of processors increases, the cost of global communication increases less sharply than than the local communication done in `Mapper`. This happens because the cost of spanning tree communication is $O(log(P))$, but the cost of local communication done by `Mapper` is roughly $O(P)$ (see Table 5.4).

Surprisingly, `Mapper's` communication structure is not especially localized on the hypercube interconnection network, despite the fact that the MLC is considered a spatially localized computation. With 32 processors, for instance, each processor communicates on average with 12 others. Table 5.4 shows that the number of communicating ''neighbors'' increases roughly with the number of processors. This reflects the tendency of the smallest of the partitions to become much smaller in area than their respective external interaction regions as the number of processors increases. Were we able to scale C with h_v, the initial spacing of the vortices (or more precisely, with $\sigma = h_v^{0.75}$), however, the external interaction regions would scale with the size of the smallest partition and so would the number of vortices that interacted across a partition boundary. Communication costs and storage overhead would grow much more slowly then if C were not scaled. This is shown in the Table 5.5.

One overhead remains to be discussed–`Partitioner`. Partitioning overhead was never more than 1.2%, including both the time to compute a workload estimate and to do recursive bisection. The work estimate computation parallelizes and hence is not a serial bottleneck. The recursive bisection algorithm is fast, doing only integer arithmetic and running in time that is

p	Neighbors		Packets		Messages	
	max	avg	max	avg	max	avg
4	3	2.6	7	6	6	5
8	5	3.8	15	12	10	8
16	11	7.4	30	23	20	15
32	18	12	53	39	31	24

Table 5.4. During a velocity evaluation each processor sends packets to a substantial number of other processors, called neighbors. "Max neighbors" is the maximum number of processors that any one processor had to communicate during a single velocity evaluation, and "avg neighbors," the average number. "max packets" and "avg packets" give the maximum and average number of 1024-byte packets sent out; "max messages" and "avg messages" give the maximum and average number of messages sent per processor. The message buffers were 10080 bytes long. The correction distance C was 1/15.

C	Neighbors		Vortices		Total Work ($\times 10^7$ Interactions)
	Max	Avg	Total	Copies	
1/15	15	10	1071	982	14
1/30	8	5.7	555	469	4.2

Table 5.5. Halving the local interaction radius halves the number of vortices that interact across partition boundaries during a velocity evaluation, and hence reduces both the message traffic and the amount of storage that must be reserved for vortex-copies. The total number of local interactions is also reduced by a factor of three. Message traffic is measured in terms of the number of interacting neighbors (columns 2 and 3) and the numbers of vortices communicated (columns 4 and 5). Column 2 gives of the maximum number of tasks that any one had to communicate with, column 3 the average number of interacting tasks per task. Column 4 gives the maximum number of vortices that any task stored, including those the task owned. Column 5 gives the number of vortex copies out of that maximum total. These data were obtained from traces of 3180-vortex runs done on the Cray X-MP; they were run on the Cray since the iPSC lacked sufficient memory to store a refined finite-difference mesh.

proportional to the logarithm of the number of processors. The total time spent in the communication and load balancing utilities was therefore never greater than 2.4% (This figure is the maximum *combined* utility overhead observed on any single number of processors. The previous overheads were reported separately and occurred on different numbers of processors).

Having considered the overhead costs and the cost of the solver, we next consider the major source of efficiency loss: load imbalance. We ask the following question: was load-imbalance the result of our using a poor work estimate mapping, the restriction on allowable cuts made by `Partitioner`, or a combination of the two? The answer is the latter, as we now show. We can eliminate the first cause by a simple experiment. We compare the observed efficiency with an efficiency measure based on the number of local corrections as the unit of time, which is the efficiency predicted by `Partitioner's` the work estimate mapping. The two efficiency measures agreed to within 5%, as shown in Figure 5.4.

Since the work estimate mapping used by `Partitioner` appears to give an accurate prediction of processor loading, we suspect that the cause of load imbalance lies in the restrictions on how `Partitioner` can make cuts. In particular, cuts may not subdivide bins into which vortices get sorted. A way to get around this problem is to reduce the size of the bins. To show how this helps, we ran a simulation experiment on the Cray (the iPSC had insufficient memory). As in the previous experiment, we measure the efficiency predicted by `Partitioner`, using the number of local corrections as the measure of time. Table 5.6 shows that efficiency increases as the bins get smaller.

What the table does not show is that the process of refining the bins may not be carried on indefinitely. Eventually, the cost of manipulating still more bins will exceed any savings due to

bin-size	Efficiency
1	0.528
1/4	0.734
1/16	0.869
1/64	0.918

Table 5.6. Efficiency loss due to load imbalance can be reduced by making the bins smaller, increasing the spatial resolution of the partitioning process. These runs are with 32 processors, and were obtained from simulations on the Cray (we did not have sufficient memory to run on the iPSC). The completion time for a task was measured in terms of the number of local corrections computed. For the iPSC runs we used bins that were 1/4 the size of the Poisson solver mesh-boxes. This bin-size was optimal for our implementation in the sense that either increasing or decreasing the size of the bins would lower efficiency. Load imbalance diminishes with the size of the bins, but the cost of doing load balancing increases as the bins become smaller and more numerous.

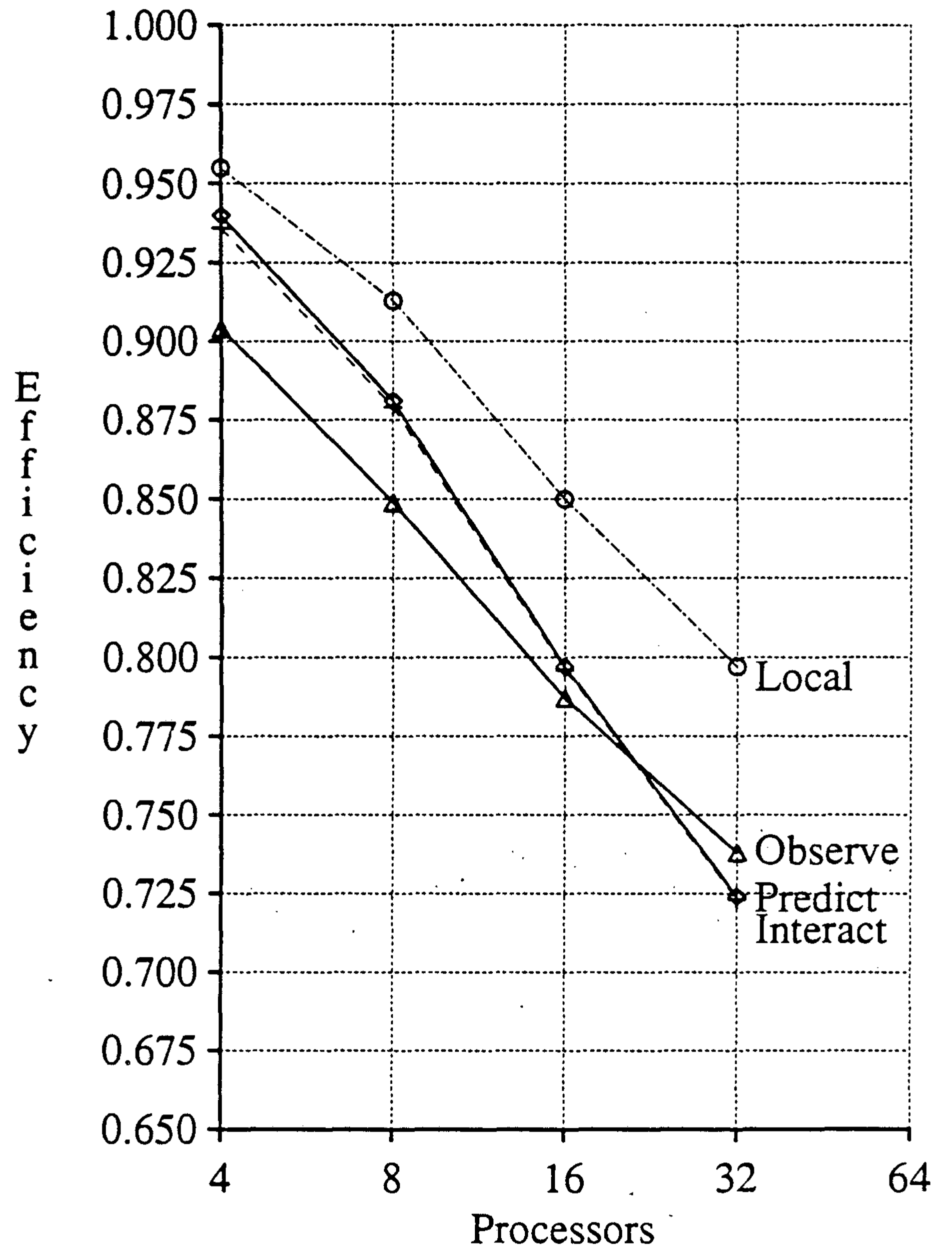


Figure 5.4. The efficiency predicted by partitioner agrees reasonably well with the actual efficiency observed for the entire calculation – "observe." The unit of work for the predicted efficiency is the local interaction. Not surprisingly, the predicted efficiency agrees almost exactly with the actual efficiency observed for the local interaction part of the computation – "interact." We also show the efficiency for the localized computation, that includes local interactions as well as some other computations. The localized efficiency is always higher than the observed efficiency since it ignores the overhead costs included in the latter.

a reduction in load imbalance. We observed this behavior on a extended-memory model of the iPSC with 4 megabytes of memory per node, and found that little was gained by refining the bins any further than was possible on the standard-memory model iPSC. For our particular implementation of the MLC, then, the level of load imbalance we observed was optimal in the sense that reducing it would not substantially improve throughput. However, because many bins were empty, we could reduce the optimal bin-size, and hence increase throughput, by implementing a sparse bin data structure.

Having now established the various sources of efficiency loss, and in particular the dominating effect of load imbalance, we next ask the following question: how often must workloads be rebalanced? In the first experiment, we balance workloads on the initial timestep only, i.e. do static load balancing, and then observe how efficiency varies as a function of time (see Figure 5.5). Efficiency decreases steadily, but gently with time. Clearly loads must be periodically rebalanced, but not on every timestep.

Though load balancing incurs modest overhead for the MLC, and may be done on every timestep, the overhead may be high enough in some calculations to justify an increase in repartitioning frequency. The optimal repartitioning frequency will have the property that either decreasing or increasing it will increase the running time of the computation. This implies that increasing the frequency will improve load imbalance at the expense of load balancer overhead and decreasing it will produce the opposite effects. For the MLC we found that the optimal repartitioning frequency depended on the number of processors but was somewhere around 2 to 4 timesteps for $P \geq 8$, as shown in Table 5.7 and the equivalent plots of Figures 5.6-7.

5.6. Discussion

Processor idleness due to misbalanced workloads is the major performance bottleneck for the Method of Local Corrections running under our VM on the iPSC. Workload imbalance is

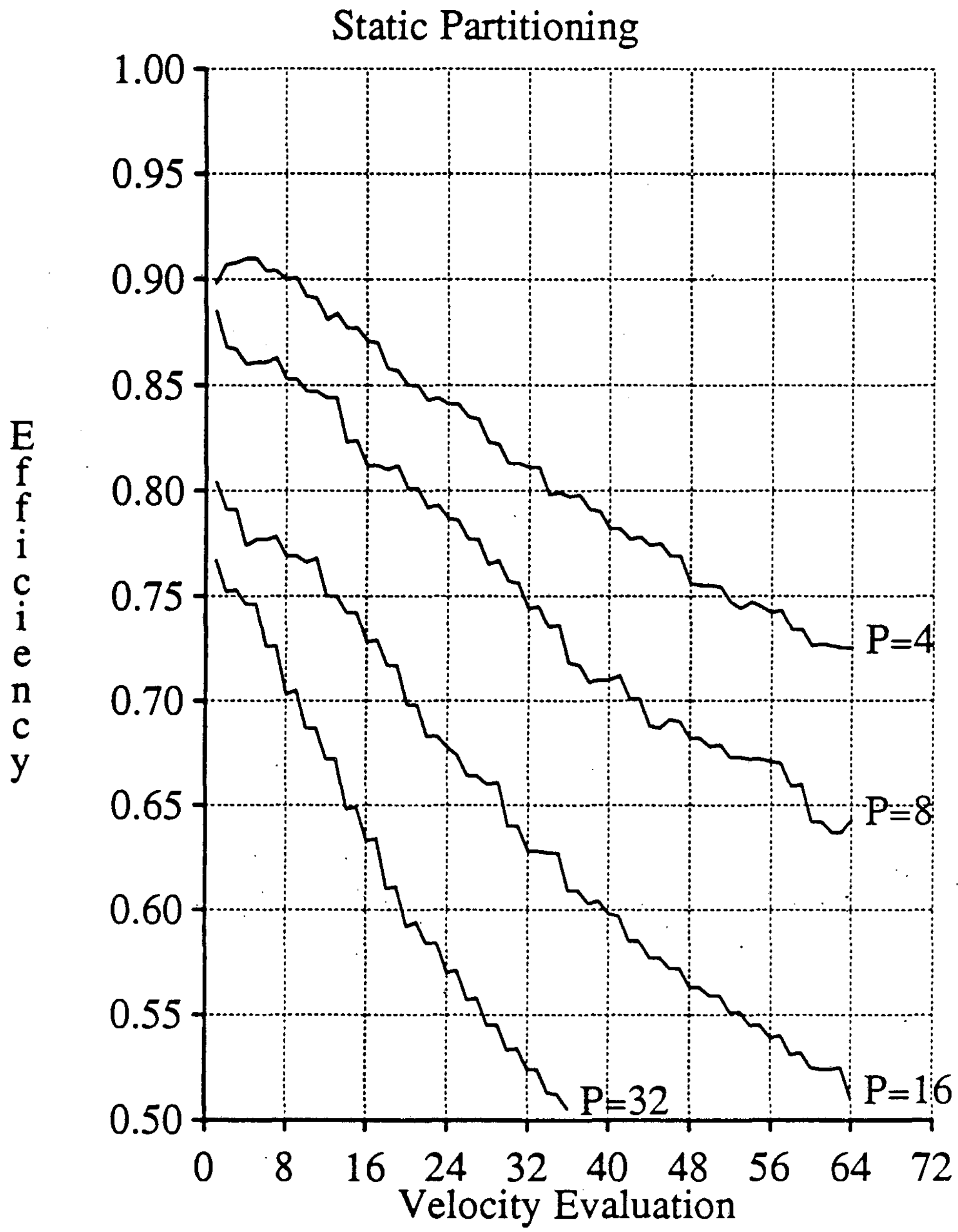


Figure 5.5. With static load balancing, the loads will drift gradually out of balance. Each curve plots efficiency against time for either 4, 8, 16, or 32 processors. Efficiency is reported for each of the two velocity evaluations done each timestep. The run with 32 processors ran out of memory after only 18 timesteps; the loads had drifted so far out of balance that some processors lacked sufficient memory to hold an excessively large number of vortices. The staircasing effect on the curves comes as the result of vortices jumping by much smaller amounts in the second velocity evaluation of time integration than in the first. Load imbalances therefore drift more gradually out of balance during even-numbered velocity evaluations than in odd-numbered ones.

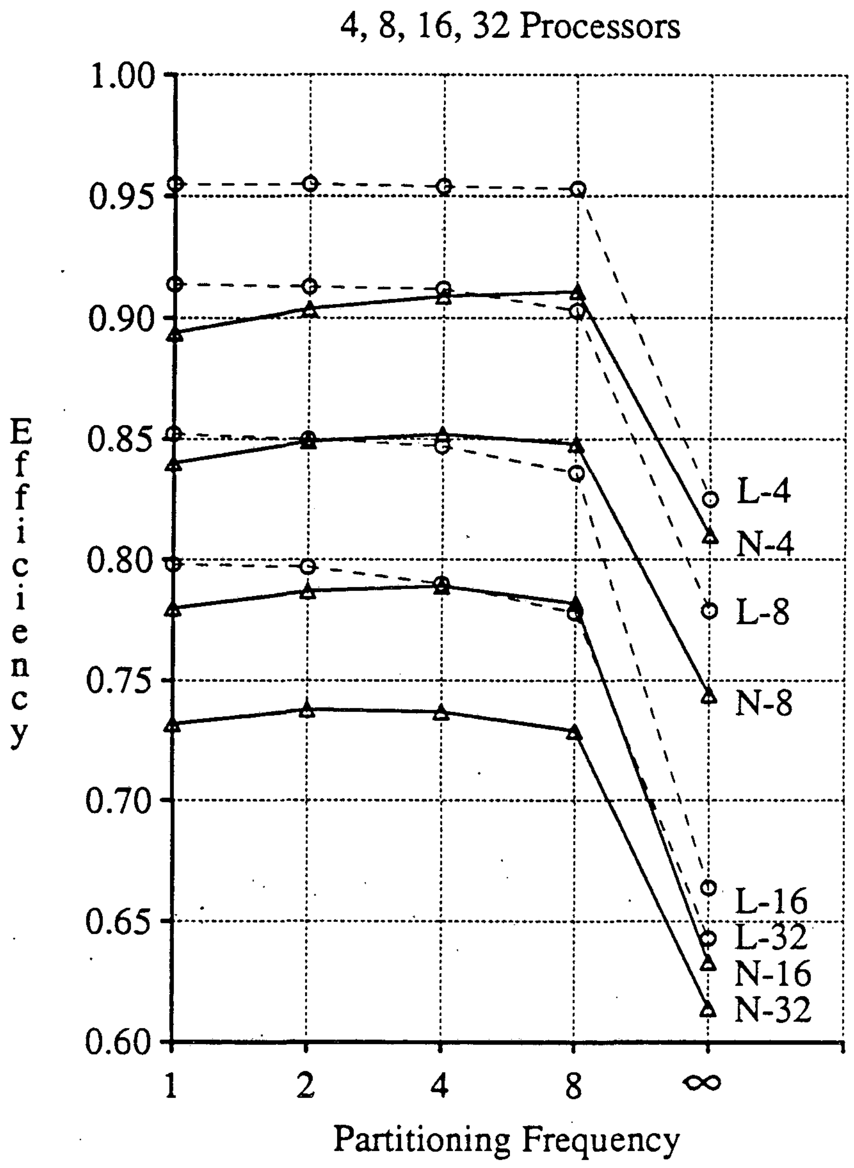


Figure 5.6. Each family of curves plots the efficiency achieved with a different number of processors, as a function of repartitioning frequency. Repartitioning frequency varies from 1 to 8 timesteps. An infinite repartitioning frequency corresponds to static partitioning. The hyphenated lines plot the efficiency for the local part, labeled L, and the solid lines, labeled N, the net efficiency for the entire calculation. The labels on the curves also give the number of processors. Except for 4 processors, the net efficiency first increases – the savings due to improving the workload balance exceeds the overhead – and then decreases as the overhead begins to dominate. The optimal repartitioning frequencies correspond to the places where the solid curves level off. The efficiency in the local part doesn't include the cost of doing load balancing and tends to decrease as the repartitioning frequency increases.

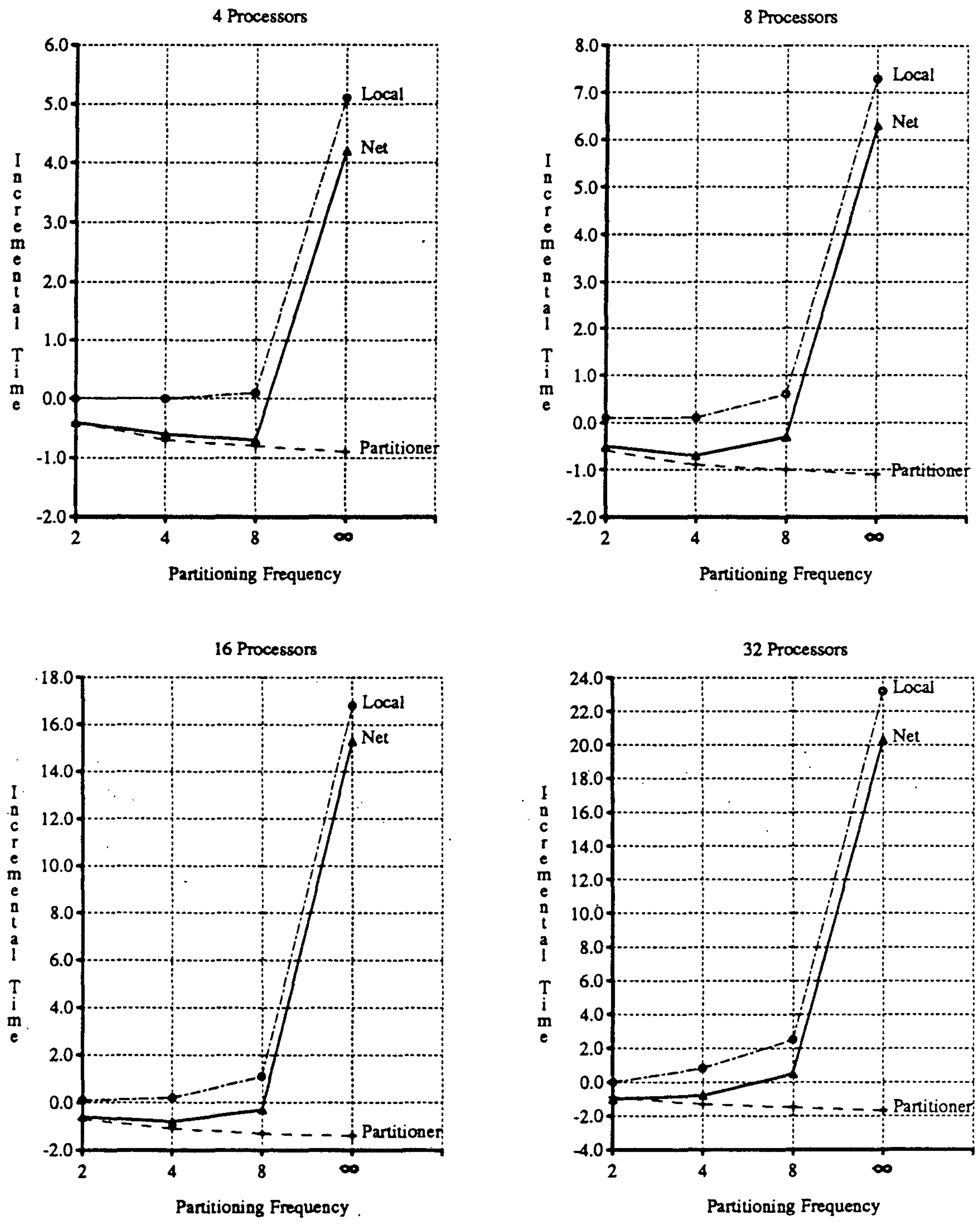


Figure 5.7. Each of the four plots show, for a single value of P, how increasing the number of velocity evaluations between repartitionings affects the additional time spent partitioning and doing localized computation. The net effect on the entire computation is also shown. Net negative times indicate an improvement. The final data point on each curve corresponds to static partitioning

Processors	Part Freq	Time			Efficiency	
		Overall	Local	Partitioner	Overall	Local
4	1	39.2	32.1	0.9	0.894	0.955
	2	-0.4	+0.0	-0.4	0.904	0.955
	4	-0.6	+0.0	-0.7	0.909	0.954
	8	-0.7	+0.1	-0.8	0.911	0.953
	∞	+4.2	+5.1	-0.9	0.810	0.825
8	1	47.4	41.1	1.1	0.840	0.914
	2	-0.5	+0.1	-0.6	0.849	0.913
	4	-0.7	+0.1	-0.9	0.852	0.912
	8	-0.3	+0.6	-1.0	0.848	0.903
	∞	+6.3	+7.3	-1.1	0.744	0.779
16	1	67.2	60.2	1.4	0.780	0.852
	2	-0.6	+0.1	-0.7	0.787	0.850
	4	-0.8	+0.2	-1.1	0.789	0.847
	8	-0.3	+1.1	-1.3	0.782	0.836
	∞	+15.3	+16.8	-1.4	0.633	0.664
32	1	105.9	96.5	1.7	0.732	0.798
	2	-1.0	+0.0	-0.9	0.738	0.797
	4	-0.8	+0.8	-1.3	0.737	0.790
	8	+0.5	+2.5	-1.5	0.729	0.778
	∞	+20.3	+23.2	-1.7	0.614	0.643

Table 5.7. The optimal partitioning frequency increases with the number of processors. Repartitioning frequencies were varied from once every time step down to once at the initial timestep only, i.e. infinite repartitioning frequency. Times are reported for the overall computation and for two interesting components: local computation time and partitioning time. For the repartitioning rate of 1 timestep, time is reported in absolute seconds. For lower repartitioning rates, the unit of time is delta seconds taken against the absolute times. The optimal repartitioning rate corresponds to the largest negative delta value in the overall time column, i.e. for P=8, the optimal rate is 4. The last two columns of the table give the parallel efficiency for the overall computation and for the parallelizable part.

reasonable, however, given that we achieved better than 70% efficiency on 32 processors; a version of the program that ran at 100% efficiency would run only 35% faster. Owing to memory limitations we could not run on a model d6 iPSC with 64 processors or a model d7 with 128; in order to use these machines we would have to exploit the inherent sparseness of various arrays used in the MLC calculation. Sparse data structures would also be essential even on 32 processors, in order to scale the simulation parameters properly with the size of the problem. Doing so would allow us to reduce the numerical operation count of the computation. We have already begun to work out the details of sparse data structure representations for a three-dimensional vortex code presently under development [1].

Surprisingly, the high message latency time of the iPSC appeared to have very little impact on the running time of the calculation, even though communication is not as localized on the hypercube interconnection network as it is in physical problem-space. Our VM incurred a low communication overhead because it can transmit data in bulk rather than an element at a time. This was facilitated by restricting the partitions to have simple shapes. Because the VM does not localize communication on the iPSC's interconnection network, and because it concentrates communication into brief instants of time, it can severly stress the iPSC's communication subsystem. The operating system must cope with the peak loading by carefully controlling the flow of disruptive messages.

The reason why our VM did a good job of avoiding load imbalance is that the simple work estimate mapping that `Partitioner` uses appears to be a good metric for dividing up work fairly; the efficiency predicted by `Partitioner` agreed to within 5% of what was observed. `Partitioner` is surprisingly effective, considering all the constraints on the way it may partition the work. Since it is recursive it can render only a subset of all possible partitionings into rectangles; the partitioning of Figure 5.8, for instance, cannot be achieved by the recursive bisection strategy. So far, the simple partitionings appear adequate. Although the rectangular geometry of the partitions further restricts the way that the `Partitioner` can split up work, the benefits of using more complicated shapes such as general tetrahedra would probably not be worth the trouble as the data structures used to represent the partitions would be difficult to manipulate.

Recently Intel Scientific Computers has announced a high performance option of the iPSC, called the iPSC-VX [4]. Arithmetic runs about 100 times faster on the VX than on the iPSC we used, but communication is only about 4 times faster. With the improved memory utilization made possible by sparse data structures, an iPSC-VX system could tackle large problems with, say, ten thousand vortices. Communication overhead, and not load imbalance, would be the

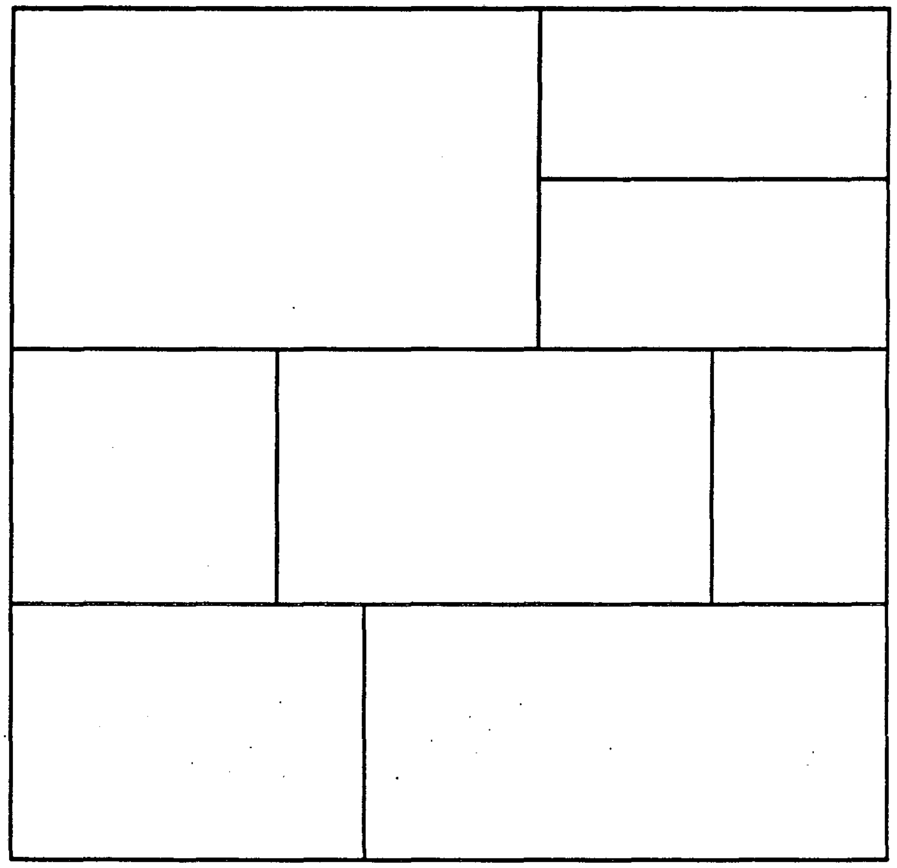


Figure 5.8. The recursive bisection strategy cannot render this partitioning.

dominant performance bottleneck, since communication on the iPSC-VX is much more expensive relative to computation than it is on the iPSC we used. We found that communication consumes roughly 5% to 10% of the running time of our implementation of the MLC running on the "original" iPSC. A simple application of Amdahl's law reveals that the iPSC-VX's fast vector units could speed up our MLC computation by a factor of 10 or 20.

5.7. References

1. S. B. Baden, T. Buttke and P. Colella, "A Fast Adaptive Vortex Method in Three Dimensions," In preparation, 1987.

2. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK User's Guide*, SIAM Publications, Philadelphia, 1978.

3. *iPSC User's Guide*, Intel Corporation, Beaverton, Oregon, October 1985. Order Number: 175455-003.

4. *iPSC-VX Product Summary*, Intel Corporation, Beaverton, Oregon, 1986. Preliminary.

5. D. J. Kuck, *The Structure of Computers and Computations, Vol. 1*, John Wiley & Sons, New York, 1978.

6. C. B. Moler, "Matrix Computation on Distributed Memory Multiprocessors," *Proc. 2nd Conf. on Hypercube Multiprocessors*, Knoxville, TN, September 1986. To be also published by SIAM.

7. C. Moler and D. S. Scott, *Communication Utilities for the iPSC*, Intel Scientific Computers, Portland, OR, August 1986. Technical Report.

8. G. F. Pfister and al., "The IBM RP3 Introduction and Architecture," *Proc. 1985 Intl. Conf. on Parallel Processing*, August 1985, pp. 764-771.

9. R. Rudell, *Parallel Processing Efficiency on the Hypercube*, Computer Science Div., University of California, Berkeley, Berkeley, California, Fall 1985. CS252 class project.

10. C. L. Seitz, ''The Cosmic Cube,'' *Comm. ACM 28*,1 (January 1985), pp. 22-33.

6

Cray X-MP Implementation and Evaluation

I've miles
And miles
Of files ...
to suit our great computer.

Graeme Edge (Moody Blues), In the Beginning

6.1. Overview

To test the hypothesis that our run-time partitioning utilities can work on diverse architectures, we implemented them on a shared-memory machine: the Cray X-MP, manufactured by Cray Research, Inc. The X-MP differs from the iPSC in three major ways: (1) the processors communicate via shared memory instead of messages; (2) there are fewer of them – at most four; and (3) individually they are quite powerful – they provide fast vector mode arithmetic and each is capable of computing at a sustained rate of 100 megaflops or more. Despite these differences, however, we were able to implement the Method of Local Corrections using a local-memory execution model and to use our run-time partitioning utilities in the same way as they were on the iPSC. We were also able to handle global communication with utilities that had the same semantics as they did on the iPSC.

The X-MP code did differ from the iPSC's in two important ways. The most noticeable difference was that many of the inner loops in the numerical portion of the code had to be rewritten so that they could run in vector mode. However, the problem of getting a program to vectorize runs orthogonal to the one of how to handle run-time partitioning. The only significant difference, therefore, was the second one: how the user sets up parallel tasks to run on the Cray, in particular, the boss task. Despite this difference, we believe that our VM could help the user save considerable time in porting his code between such diverse architectures as the X-MP and iPSC.

The chapter begins with a description of the X-MP system, both hardware and software. Next, we describe our implementation of the MLC and of the VM. Finally, we present the results of some experiments we ran on the X-MP.

6.2. The Cray X-MP Multiprocessing System

The X-MP is a shared-memory architecture with up to four processors and 16 million words of main memory (1 word = 64 bits). We used this largest model, known as the X-MP/416. The definitive source of information about the X-MP is the hardware reference manual published by Cray Research, Inc. [3]. A more accessible document is the pamphlet by S. Chen et. al. made available by Cray [1].

The Cray X-MP processing engine is a descendent of the Cray-1 [6, 7]. It has a very fast cycle time – as fast as 8.5 nanoseconds in some models – and multiple arithmetic functional units that can work in parallel. Some of the functional units are pipelined and operate in a very fast vector mode of computation for FORTRAN-style DO loops. Independent scalar operations can also be pipelined. Put in round figures, a loop that vectorizes runs ten times faster in vector mode than in scalar mode. But not all loops can vectorize and, as the fraction of time spent in vector mode decreases, the maximum useful speedup attributable to vector mode also decreases. To see why, compare two programs; the first spends 95% of its time in vector mode, the second

50%. If the time spent in vector mode could be reduced to zero with infinitely fast vector hardware, then the first program would run twenty times faster than before, but the second only twice as fast. The maximum useful vector execution rate is therefore ten times greater for the first program than for the second. This phenomenon is known as Amdahl's law:

$$S(R, f_v) = ((1-f_v)+(f_v/R))^{-1} \tag{6.1}$$

where S is the reduction in execution time, the speedup, f_v is the fraction of time in vector mode, and R the ratio of vector mode execution rate to the scalar mode execution rate.

Another way of stating Amdahl's law is that speeding up the vector rate beyond a certain point will not decrease the running time of a program unless the program can be coerced to spend less time out of vector mode. In general R is a function of both the length of the pipeline and the vector length, n. For short vectors R will be small, since the time to fill the pipeline will be high compared with the time to pass through it. As $n \rightarrow \infty$, R approaches the maximum asymptotic rate of Q, where Q is the number of stages in the pipeline. Hockney and Jesshope [5] define $n_{1/2}$ as the value of n required to attain one-half the maximum rate. The Cray's adder, for example, has six pipeline stages, so $n_{1/2} = 6$. This is considered to be quite low; the CDC Cyber 205 has an $n_{1/2}$ of about 100 [5].

What makes the Cray so attractive is that it is a *balanced* architecture: not only does it perform well on short vectors, but can also execute scalar operations reasonably fast. There are two reasons why this is so: (1) scalar mode is not that much slower than vector mode (2) independent scalar operations may be pipelined. This is important because operations on scalars and short vectors, as opposed to long vectors, ultimately limit performance.

6.2.1. Arithmetic and Registers

The Cray is a register-based architecture, with 64-bit words. Memory referencing instructions load and store registers. All other instructions operate only on registers, of which there are several kinds: 8 vector (V) registers, each 64 elements long and 64 bits wide; 8 address (A) registers, each 24 bits wide; 8 scalar (S) registers, each 64 bits wide; and B and T register-sets, each with 64 registers, to back up the A and S registers, respectively. The backup registers may be used as a small software-managed cache memory; the X-MP has no other local storage. There are also some special control registers, a real-time clock, and a set of shared registers that provide a fast path for communicating small amounts of information between processors. These shared resources will be ignored for the moment. The X-MP also provides a hardware performance monitor for counting floating point operations, dynamic instruction frequencies, and so on.

The CPU provides the usual arithmetic, logical, and register transfer operations.[1] A complex of 14 functional units implements the various arithmetic and logical operations. Certain combinations may be active simultaneously to increase the throughput of the engine. The integer adder, for instance, may work in parallel with the floating point adder so that the CPU may overlap loop control with floating point arithmetic. Moreover, some functional units may be hooked together into a pipeline by a process called *chaining*. Since memory read and write operations are also chainable, the Cray may overlap memory transfers with arithmetic. Many of the Cray's instructions come in two varieties: vector-mode instructions and scalar mode instructions. A vector-mode instruction repeats the same operation over up to 64 sets of arguments. One argument is always stored in a vector register; the other, if it exists, may be in either a vector register or a scalar register. In the case of a scalar argument, the scalar value is used repeatedly for each element of the vector argument (There are also triadic instructions that take a third argument stored in a mask register to be discussed below).

[1] One exception is division: the Cray has no divide instruction. Instead it has a reciprocal approximation instruction. The approximate reciprocal is refined through one Newton-type iteration to provide division to near machine accuracy.

Scalar mode instructions can also be pipelined though they may not keep the pipeline as busy as vector instructions. The problem is that certain scalar operations may have to wait until any pending operations have completed. Because this can include memory transfers, and because such transfers go at one tenth the rate in scalar mode than in vector mode, the pipeline delays on scalar operations can be expensive.

The Cray provides vector comparison instructions for generating a bit-string of comparison flags, held in a 64-bit mask register, and vector merge instructions that use the mask to control the merging of two vector register arguments. For example, if Z is assigned the result of "merge (X,Y,M)," for data vectors X and Y and mask vector M, then $Z_i = X_i$ if M_i is set, and Y_i otherwise.

6.2.2. Memory

Memory is organized into 4 sections of 16 banks each. Each processor has 4 memory access ports to main memory: two for reading, one for writing, and one for I/O. All four ports can be simultaneously active. Each port of each CPU has a private connection to each section of memory. Memory referencing instructions reserve ports for the duration of a memory transfer; transfer times are unpredictable since the flow of the data can be disrupted by access conflicts between ports on one or more processors. Access conflicts occur as the result of a bank conflict– simultaneous accesses to the same bank by more than one port from within a CPU or between CPU's– or a section conflict– simultaneous accesses to the same section over two or more ports of a single one CPU. Different CPUs may access the same section of memory, however, so long as there isn't a bank conflict. In addition, a CPU can wait on a bank that is already occupied with a pending memory request; this is called a bank busy conflict. Each section of memory monitors incoming address traffic and blocks incoming references whenever the path is busy. Since this mechanism cannot detect concurrent reads and writes to a critical section of memory, the programmer is responsible for employing an appropriate software mechanism to avoid hazardous race conditions, e.g. lock out critical sections.

Memory transfers go fastest in block transfer mode, either when loading a vector register or when loading a backup register set. The cost of an n-word block transfer is $(17 + n)$ CPs for vector registers, $(16 + n)$ CPs for backup registers. The maximum transfer rate is 3 words/CP per CPU with the two read and the single write port simultaneously active upon three transfers. Fast block-mode transfers need not be confined to arrays of consecutive words so long as their addresses are in an arithmetic progression; rows or diagonals of arrays may be accessed as quickly as columns. However, consecutive accesses to the same bank must be separated by at least 4 CPs – otherwise the bank will be busy with the last request. This problem can be avoided in software by dimensioning arrays with bounds that are relatively prime to 64. The cost of transferring one word to the scalar or address registers is 14 CPs. The X-MP also provides hardware gather and scatter instructions for speeding up non-contiguous memory transfers involving doubly nested array subscripts; but, our code did not invoke these, and they will be discussed no further[2].

6.2.3. Shared Registers

The X-MP provides 5 sets of shared registers. Each set has 32 (1 bit) shared semaphore registers, 8 (64 bit) shared scalar registers and 8 (24-bit) shared address registers. A processor that wishes to share a register set does so by loading its private cluster register with the appropriate set number. Processors in the same cluster may then communicate address and scalar information, one data element at a time, by loading and storing the shared registers. Members of a cluster may also synchronize themselves through a semaphore register by means of an atomic test-and-set instruction. This instruction implements a binary semaphore; it samples and then sets a designated shared semaphore register. If the semaphore was already set, the test-and-set

[2] This happened for historical reasons: the MLC code was initially developed on a Cray-1 that had no gather-scatter hardware, so we had to implement gather-scatter in software.

waits until the register clears. To guarantee the atomicity of the operation, no processor may execute the test-and-set on a semaphore register until any pending test-and-set completes. If all processors in a cluster are waiting on the same semaphore, then the machine issues a deadlock interrupt. An interrupt handler may then decide on an appropriate course of action such as job cancellation.

6.2.4. X-MP System Software

All software was written in Cray X-MP FORTRAN [2]. This is a superset of FORTRAN 77 which Cray Research, Inc. has extended to improve the vectorizability of code and to deal with certain aspects of concurrency. Several new Cray FORTRAN library routines are provided in support of these activities and are described both in the FORTRAN manual and in the Cray X-MP Multitasking User's Guide [4]. We used Version 1.14 of the CFT compiler, dated October 8, 1985, and Version 2.2 of the loader (segldr), dated December 1, 1986, and ran under version 1.16BF1 of COS, the Cray Operating System. Since COS is a batch operating system, and therefore non-interactive, jobs were submitted to the Cray mainframe from an interactive front-end processor.

The X-MP may be run in either timeshared mode or dedicated mode. Jobs in timeshared mode compete for processors to the extent that no job has control over how many CPUs it will get. Since different jobs can interfere with one another either by contending for memory or by issuing interrupts, the running time of a job is not generally predictable. In dedicated mode jobs may be run one at a time. Interference from other jobs is virtually non-existent, except for certain system activities, and a job will have all the CPUs at its disposal. Various front-end processors may be disconnected, to avoid costly I/O interrupts. Timeshared mode was used to debug the code, since dedicated mode time is dear. But dedicated mode was used to make all timing measurements since only then can execution times be accurately reproduced.

Using the Cray's real time clock, times can be measured accurately down to the tens of microseconds. The *flowtrace* option of the CFT compiler produces code that uses this clock to generate a profile of the fraction of time the program spent in each subroutine. In addition, the *floptrace* facility can be used to provide useful information about floating point and memory operation counts obtained from the X-MP's hardware performance monitor. The mechanisms impose an overhead cost on each subroutine call. Floptrace, for instance, introduces a 300 to 600 CP overhead. Unfortunately, flowtrace and floptrace work only for uniprocessor runs since they would be confused by multitasking primitives that exit through the job scheduler. No timing breakdowns could be obtained directly on multiple processors, and indirect means had to be used instead (e.g., floptrace on uniprocessor runs).

The Cray X-MP multitasking library provides subroutines for creating and administering parallel tasks. Calling `tskstart` with an external subroutine as argument spawns an independent task that begins executing the subroutine. The call to `tskstart` is asynchronous since a return doesn't indicate that the task has completed; the user invokes the `tskwait` routine to determine when a spawned task has finished. Owing to the high overhead incurred by these routines, the user will usually want to call `tskstart` only at the beginning of his program, and `tskwait` only at the end. The first call to `tskstart` a program makes costs roughly 1,500,000 CPs, while each additional call costs 25,000 CPs. `Tskwait` costs roughly 25,000 CPs for each call. The prudent user will want to employ a less expensive mechanism provided by the multitasking library for synchronizing at a finer level of granularity than is feasible with `tskstart` and `tskwait`.

One inexpensive synchronization mechanism is the binary semaphore; it incurs no more than a 1500 CP overhead, excluding any waiting time. Binary semaphores may therefore be used to synchronize at the level of millions of instructions, but never within tight inner loops.[3]

[3] The multitasking library also provides an event mechanism that we did not use but which could just as easily have sufficed.

Two routines, called `lockon` and `lockoff`, implement binary semaphores, called *locks*. These may be used, for example, to lock out critical sections of code or to construct higher level synchronization mechanisms such as barrier synchronization. Both lock primitives take a standard FORTRAN integer variable, the lock, as argument. `Lockon` performs a test-and-set operation: it samples the lock, waiting until the lock clears, then sets the lock. `Lockoff` clears the lock, possibly enabling a task that was waiting on the lock. The testing and setting, and the clearing of locks are atomic operations.

The lock mechanism is implemented in software; a test-and-set instruction is used to guarantee exclusive access to the lock variable, but the variable itself is a word in memory and not a bit from a hardware semaphore register. When `lockon` encounters a cleared lock, it returns at a total cost of 200 CPs, with the lock set. When `lockon` encounters a set lock, the task executing it enters a blocked state. `Lockon` will then periodically poll the lock. To avoid tying up the CPU and memory on non-productive polling activity, however, blocked tasks will not be allowed to idle for very long – typically a few thousand CPs – at which point they enter a wait state. A block tasked enters the wait state by exiting `lockon` through the multitasking library's scheduler, which then tries to pre-empt the waiting task with one that is ready to execute. The cost of exiting through the library scheduler is roughly 1500 CPs. Under certain conditions, the COS scheduler may have to intervene; this will increase the overhead of `lockon` to tens of thousands of CPs (For additional details, see the *Multitasking User's Guide*). `Lock-off` always exits through the library scheduler, taking only 200 CPs to complete if no tasks are waiting on the newly-unlocked lock variable. The time increases to 1500 CPs if any task becomes unblocked as a result of the lock variable becoming clear, since tasks must be moved among job queues and perhaps moved to an available processor.

Code-reentrancy is novel feature of Cray X-MP FORTRAN and is required to permit the sharing of code. In earlier versions of Cray FORTRAN for uniprocessors, local variables were allocated statically. In a multi-tasked environment static allocation could cause programs to behave incorrectly, since all the tasks would share the same set of local variables. In X-MP FORTRAN each subroutine call dynamically allocates a private set of local variables on a stack. This can cause some old software to behave incorrectly since local variables, which were once static, are no longer defined at the beginning of a subroutine call, nor do they persist from one call to the next. There is no way to have static read-write local storage; the user must work around the problem in software. In particular, the `save` statment cannot be used, for although the saved variables are visible to only the routine executing the `save`, they are also visible to all tasks executing the routine.

To avoid conflicts involving common blocks there is a new form of common block called `task common`. `Task common` is a version of common that is global to a single task only. In other words, task common is memory shared among all subroutines in a single task unlike standard common, which is memory shared among all subroutines in all tasks. Task common is dynamically allocated as the result of a `tskstart` and initially it is undefined.

6.3. Implementation

We next discuss how we implemented the Method of Local Corrections on the Cray X-MP, using our VM. Since the semantics of the VM's utilities are the same regardless of the architecture on which run, we will not be concerned with setting up the calls to the utilities; that has already been discussed in sufficient detail in chapter 4. Instead, we will focus on the details of the implementation that are left unspecified by the VM. We will begin with a discussion of how we parallelized the the MLC. Next, we will discuss the implementation of the VM and of the global communication utilities that were also used in the iPSC implementation. We also discuss how we coerced the tight inner loops of the MLC to vectorize. This is done for completeness only, as the problem of how to vectorize the code runs orthogonal to the problem of how to parallelize it.

6.3.1. Parallelization

We show how we parallelized the MLC using the Cray X-MP FORTRAN extensions, the multitasking library, and our run-time partitioning utilities. There are six aspects to parallelizing the MLC code:

(1) Determine which parts of the code, if parallelized, could substantially reduce the running time of the calculation, and which parts can run serially, i.e. on just on the one boss task.

(2) Determine which common blocks should be global and which should be local to a task.

(3) Insert calls to the multitasking library routines.

(4) Implement data partitioning, i.e. modify loop bounds (already discussed in chapter 4).

(5) Insert calls to the `Partitioner` and `Mapper` utilities (already discussed in chapter 4).

(6) Handle global communication.

Serialization. The MLC spent over 90% of its time executing as multiple tasks. The solver consumed the major part of the time spent executing on one processor; program initialization and task partitioning consumed the remaining parts, and may be ignored, since they accounted for less 0.01% of the total running time of the computation (This does not include the cost of constructing `partitioner's` work estimate mapping, which does parallelize. That will be considered separately). Though the Poisson solver could have been parallelized, doing so would not have saved significant time on a machine with only four processors. A calculation with 12848 vortices that used a 64×64 solver mesh spent only 0.6% of its time in the solver. Amdahl's law tells us that the effect of running the Poisson solver on just one processor instead of four is to reduce the ideal speedup of 4.00 (achieved when all computations parallelize and all overheads are non-existent) to 3.92, where in equation (6.1) we replace R by $P = 4$ and f_v by the fraction of time spent in the solver (0.006). Thus, parallelizing the solver perfectly on four processors would speed up the whole calculation by at most 2%.

Common blocks. As on the iPSC, the major data structures were duplicated for each task in local memory: the vortices, the bins, and most of the finite difference grids were stored in `task common`. Each task reserved local storage for two grids (one for each component of velocity) and three copies of the bin structure. These structures were 132 elements on a side and consumed a total of 87 kilowords of memory per task. There were also four grids stored in global common, and shared by all tasks. Two held the solver's global right hand side, the other two tables of trigonometric functions that were internal to the solver, for a total of 70 kilobytes of storage regardless of the number of tasks.

To enable the code to vectorize, each task also had 10 gather/scatter vectors, each 4096 elements long, for an additional storage overhead of 41 kilowords per task.

The storage assigned to the vortex-records overwhelmed what was consumed by the grids and the gather/scatter vectors. Together all tasks could hold as many as 136,000 vortices. With 12 words per vortex, this came to 1.6 million words of memory for all tasks. Each task statically allocated space for 136,000/P vortices, where P was the number of tasks.[4] In our implementation the boss executes partly as a $P+1$st task, and so all the task common blocks were duplicated on this task, too. As a result, the total number of vortex-records allocated by all the tasks actually decreases with the number of tasks, since the extra bit allocated by the boss decreases with P. The total amount of memory reserved for all the major data structures was therefore 3.6, 3.0, and 2.8 megawords for 1, 2, and 4 processors respectively[5]. The *debug* utility indicated that each task consumed an additional 100 kilobytes of stack space and that all tasks shared 400 kilobytes of heap storage. In total, the running program consumed 4.1, 3.6, and 3.6 megawords for 1, 2, and 4 processors. Using the load map, we determined that the shared code consumed only

[4] Storage was allocated statically by compiling different versions of the code, each intended for a different number of processors. Each version of the code differed only in a compile-time constant specified by the user. This would have been unnecessary with dynamic storage allocation.

[5] With dynamic memory allocation, the duplicate set of task common blocks would vanish before the workers were spawned, and the total storage costs would increase with the number of tasks, and drop to 1.8, 2.0, and 2.2 megawords, for 1, 2, and 4 tasks respectively.

43 kilobytes of storage, divided roughly equally among the MLC code and the system libraries.

Though we used some global storage in our X-MP implementation, we abided by the VM's requirement that each task execute on its own set of data during local computation. We restricted our use of global common in two ways that did not violate the spirit of our VM's local-memory execution model: (1) to handle global communication, that lies out of the scope of the VM; and (2) to have the boss set up read-only data for the workers, e.g. simulation parameters.

Multitasking library calls. The calculation starts from a single thread of control, the boss. The boss reads all the simulation parameters into global common, and then sets up the initial partitioning of space based on the initial configuration of vorticity. It calculates the work estimate without actually setting down any vortices since it does not have enough memory to store them all. This is the only time that the work estimate is constructed this way; subsequently, the workers assume the responsibility and each examines the vortices assigned to it to determine its contribution to the work estimate. Next, the boss calls the `tskstart` routine within a loop to spawn off P worker tasks:

```
for i = 0, P-1
        call tskstart(idtask(1,i),iterate)
end for
```

where `idtask` is a task descriptor, and `iterate` is the subroutine that the newly-created worker calls upon creation.

We elected to multiplex some of the boss's activities, i.e. partitioning, as part of a distinguished worker task, number 0. Conditional statements were used to examine a task's id to decide whether the task is to execute the boss's code. The worker tasks view the simulation parameters and subproblem descriptions as read-only data since only the boss may change these data, and then only when all workers are inactive.[6]

Since the calls to `tskstart` are asynchronous, a return does not indicate completion of the worker tasks. The part of the boss that spawned the workers must therefore wait for each to finish by repeatedly calling the `tskwait` subroutine, once for each task:

```
for i = 0, P-1
   call tskwait(idtask(1,i))
end for
```

All the workers have finished when control drops out of the loop. This is very coarse-grained synchronization. We used barrier synchronization to coordinate tasks at a finer level of granularity than could be accomplished with `tskstart` and `tskwait`. This routine was generously provided by John L. Larson of Cray Research, Inc. in Chippewa Falls, Wisconsin, and uses locks to implement the barrier.

In barrier synchronization, no task may pass through the barrier until all have arrived. We used barrier synchronization to interleave the boss's activities with those of the distinguished worker task. The part of the code that handles task partitioning, for example, is one place where such synchronization is needed. Before it can invoke `Partitioner`, the worker task impersonating the boss must wait for the others to finish computing their contribution to the global work estimate mapping. Otherwise, `Partitioner` could try and balance workloads using an incomplete work estimate. The remaining tasks must then wait for the boss to finish, since their activities depend on the new work assignments returned by `Partitioner`. A conditional statement selects worker task 0 to execute the boss's code. The other tasks immediately reach a barrier and will wait until the boss finishes with his work and reaches the barrier, too. All are then free to continue. This is shown in the following code fragment:

[6] To improve the portability of the code, we would want to implement these data structures in task common and use the global broadcast utility to give each worker its own copy. However, this falls out of the VM's jurisdiction.

```
compute the work estimate
synchronize at the barrier
if this is task #0 call partitioner
synchronize at the barrier
```

Though Larson's barrier synchronization routine is reasonably efficient, it should not be used too often, i.e. within tight inner loops. The Cray Multitasking User's Guide [4] alludes to this:

> "Multitasking does not reduce the CPU cycles necessary to execute the program. In fact, multitasking introduces an overhead that increases CPU time; therefore, the number of calls to the Multitasking Library must be minimized. You should exploit parallelism at the highest level possible to reduce the overhead "

The overhead of Larson's barrier synchronization primitive is about $3000P$ CPs, where P is the number of tasks. So long as a task does roughly 10^6 floating point operations between barriers, then the overhead of synchronization will be reasonable; assuming that a flop costs 2CPs, the overhead of barrier synchronization is roughly 0.5% on four processors.

Global Communication. We duplicated the global summation and global broadcast routines used in the iPSC implementation to manipulate the Poisson solver's global finite difference grid. The semantics of these routines were the same as on the iPSC, though the implementations were different.

`Gsum` combines an array duplicated in each task's task common into a single array stored in global common. Gsum uses locks to execute as a critical section. Once inside the critical section, each task sums its local common block into an accumulator stored in global common. In addition, two barrier synchronization points bracket the beginning and end of the routine to satisfy two correctness constraints: (1) no task may enter `gsum`, and hence modify the global accumulator, until all other tasks that have not yet entered gsum have finished using the accumulator's contents; (2) no task may use the global accumulator until all have updated it with their contribution. `Gsend`, the global broadcast operation, generally accompanies a gsum. Its effect is to copy a global common array into arrays stored in task common. `Gsend` begins with a barrier synchronization point to ensure that no task will try and copy an array that is still being produced. No task, for example, must try and copy the solver's result until the boss finishes computing it.

A possible disadvantage with using `gsum` and `gsend` is that they can incur a memory overhead that is proportional to the number of tasks, unless some attempt is made to exploit the inherent sparseness of the local grids. In our implementation of the MLC, even though a task uses only parts of the local mesh it stores an entire copy. It really only needs a subset of the mesh covering the task's assigned region of space (extended slightly to include a surrounding external interaction region) and a one-dimensional region covering the physical boundary of the problem. Because memory was so plentiful on the X-MP/416, we were not compelled to take advantage of the sparseness of the mesh.

There are two alternatives to using gsum/gsend that would obviate the need for maintaining local copies of the finite difference grids, and require only a single global grid. Both incur a much higher overhead than using gsum and gsend, as we now show. One way is to serialize the local finite difference computations that set up the right hand side for the solver. Amdahl's law tells us that this strategy would slow down the entire computation by 30% to 45% on 4 processors, since the local finite difference computations can account for up to 10% to 15% of the total running time of the uniprocessor computation. When the `gsum` and `gsend` utilities are used, the local finite difference work can be done concurrently; the serial work gets concentrated into brief phases of communication and computation. On four processors, for example, the overhead of `gsum` is roughly 6×10^4 CPs; there are three barrier synchronization points, and a total of 8 calls to `lockon` and `lockoff`. Gsum does impose some extra computational overhead to serially add up the arrays, and gsend must copy arrays, but all this would cost roughly 7×10^4 CPs at worst[7] and is small compared with the work done to produce the local right hand sides (roughly 3×10^7 CPs).

An alternative to serializing the local computations is to use locks to serialize accesses to the global array. To update a mesh box of the global accumulator, a task sets a lock, performs the fetch-add-replace, then unlocks the lock. The overhead of the such a scheme, however, would be intolerable; at least 400 CPs to invoke the lock primitives compared to only 2CP to do the addition. Empirical measurements have shown that for the finite area vortex problem we have used throughout this dissertation, using 12848 vortices and a 64×64 solver mesh, the calculation does roughly 10^5 fetch-add-replace operations per velocity evaluation per component of velocity. The total cost of synchronization is therefore at best 4×10^7 CPs. This is about 500 times more expensive than using `gsum` and `gsend`. `gsum` and `gsend` are inexpensive because the number of times they call `lockon` and `lockoff` is proportional to P, not to the number of accesses to the shared array.

6.3.2. VM Implementation

In chapter 4 we showed how to invoke the utilities of our VM and set up their argument lists. We therefore will not repeat that discussion, but move directly to the VM implementation. We consider only `Mapper`, since the implementation of `Partitioner` was the same as that of chapter 4.

Mapper was implemented in much the same way as on the iPSC, except that we first had to implement the message-passing primitives `send` and `recv` on top of the X-MP's global memory, using shared mailboxes. `Send(type,buffer,length,destId)` sends the first `length` words of `buffer` to the task with id `destId` and tags the message with `type`. When `send` returns, the buffer is free for re-use.

[7] Assuming that a fetch-add-store operation costs 2CPs, that the global broadcast routine copies data at the rate of 2CPs per word, and that copying must be done serially owing to memory contention.

Recv(type,buffer,length,cnt,srcId) admits the first length words of the least recently sent message of type type into buffer, and stores the length of the message in cnt. If cnt eq -1, there is no such message pending; otherwise, the first cnt words of buffer contain the message, and srcId contains the id of the task that sent the message. The user is responsible for determining how to cope with messages that have not yet arrived; recv is non-blocking. As with the iPSC, messages of a single type passing between a single source and destination arrive in FIFO order.

Each processor has its own set of mailboxes, one box for each of the allowed types 0..MAXTYPE-1. *Mailbox(p,t)* is the head of a list of messages of type *t* pending for task *p*. The mailbox resides in global common and may be read or written by any task. Messages are broken into packets of length PACKLEN words and the packets of a single message are threaded together into a list. Messages can be of any length so long as there is sufficient available storage for packets. There is enough storage for MAXPACK packets. The parameters MAXTYPE, PACKLEN, and MAXPACK are selected at compile-time and the values we used were:

MAXPACK	=	64
MAXTYPE	=	8
PACKLEN	=	2048

i.e. 128 kilowords of storage were reserved for message packets.

Send appends a message onto the end of the list of messages attached to the appropriate mailbox. This operation must execute as a critical section since other tasks must not disturb the mailbox until the message has been appended. Recv examines the head of the appropriate mailbox within a critical section. If it has ascertained that a message is pending, it moves the data from the message packets into the message buffer and reclaims the packet-storage. Mapper used two message buffers for each task, an input buffer and an output buffer. The buffers we used were 16 kilowords long. The total storage devoted to Mapper was therefore less than 160 kilowords, which is small compared with the memory used in the numerical portions of the calculation. In particular, `mapper's` storage overhead was comparable to that of the gather/scatter operations, which used 40960 words of buffer storage per task, or about 160 kilowords for 4 tasks.

6.3.3. Vectorization

The incentive of having do loops executing in vector mode had a major impact on the design of the numerical portion of the MLC code. The task of getting the code to vectorize, however, is completely independent of any concerns regarding code parallelization. Two changes had to be made to certain inner loops, to allow the compiler to vectorize them.

(1) Vortices stored as linked lists must be *gathered* into vectors before being used within inner loops and the results *scattered* to memory afterwards.

(2) Within tight inner loops both branches of a conditional statement must be executed and the desired result selected with a Cray FORTRAN intrinsic merge function that vectorizes.

Gather/Scatter. Owing to the high cost of following pointers on the Cray, linked lists should first be gathered into contiguous vectors before they can be used within a tight inner loop. Later, the result of the computation will be scattered from the adjacent locations of the array into the non-adjacent locations of the linked list. The effectiveness of gather and scatter operations is contingent on their being done infrequently relative to numerical operations. We implemented gather and scatter operations in software; the code is simple and will not be shown.

Conditionals. Operation counts can be a misleading timing metric on the Cray since operation times are sensitive to how well the Cray's pipelined functional units are kept filled. Even if they execute more arithmetic operations, calculations that have been modified to utilize the pipeline more effectively often run faster than the original computation that executed fewer arithmetic operations. Decisions, for example, are often very expensive within tight inner loops, since the flow of data through the pipe can be disrupted. Under certain conditions some inner loops with embedded conditional statments can vectorize without change. Others will have to be rewritten to evaluate both branches of the loop and then to use the CFT intrinsic vector merge function to select the desired results. Figure 6.1 shows two versions of a loop with the same semantics; one vectorizes and the other does not.

The merge function is fast because it executes vector compare and vector merge instructions that are fully pipelined. The net effect, therefore, of using the merge function is to replace a conditional *statement* that does not vectorize by a conditional *expression* that does. In making the conversion, however, we may have introduced spurious illegal operations, such as division by zero, whose results will never be used but which could abort the program. Such erroneous operations must be somehow be avoided in software or rendered harmless. We chose the first method, called ''protection,'' which employs an intrinsic function like `max`.[8]

6.4. Evaluation

We evaluated our VM by taking measurements of full-scale runs of the Method of Local Corrections.

6.4.1. Simulation Parameter Selection

The evaluation on the X-MP differed in four major ways from what was done on the iPSC:

[8] The use of max would be unnecessary, resulting in a faster running code, by suppressing the Cray's floating point mode flag while the loop executed. Upon division by zero the machine would return a distinguished value, without raising an exception. The error-value would later be filtered out by the merge function, so that the the spurious error would disappear without a trace. However, suppressing the mode flag will suppress *all* exceptions, including ones that are not spurious, and may lead to incorrect results if not used carefully.

```
-- This loop may not vectorize, owing to the embedded  conditional statement
-- t, x, and y are n-element arrays
-- if x[i]^2 + y[i]^2 is non-zero, then it it must be no smaller than
-- a small constant eps, where eps << 1

        for i = 1, n
                r[i] := x[i]**2 + y[i]**2
                if (r[i] ≥ 1.0) then
                        t[i] := x[i]/r[i]
                else if (r[i] > 0) then
                        t[i] := x[i]/sqrt(r[i])
                end if
        end for


-- This loop vectorizes;
-- the conditional statement has been replaced by a conditional expression

        for i = 1, n
                r[i] := x[i]**2 + y[i]**2

-- If r[i] was zero, rNew  will be assigned a small distinguished value,
-- smaller than eps, which r[i] can never take on
-- This protects against a possible division by zero below
-- The compiler allocates a vector temporary for rNew to allow the statement to vectorize
                rNew := amax(r[i],eps/1e6)

-- cvmgp is a vectorizing intrinsic merge function
-- merge(x,y,t) <=> if (t ≥ 0) → x, else y
                u = cvmgp(x[i]/rNew,x/sqrt(rNew),rNew−1.0)

-- This rejects the result where r[i] = 0, for if rNew < eps,
-- then r[i] must have been zero
                t[i] = cvmgp(u,0,rNew−eps)
        end for
```

Figure 6.1. The merge function replaces a non-vectorizing conditional statment by a vectorizing conditional expression. The max function protects against any spurious division by zero. Merge functions come in different flavors; the one we used, cvmgp, bases its decision on whether or not the third argument is positive. If the third argument is positive, then cvmgp returns the first argument, else the second.

(1) Larger-scale problems with up to 2.5×10^4 vortices could be run.

(2) The local interaction radius C could be scaled with the problem-size.

(3) Uniprocessor running times could be measured directly.

(4) Indirect means had to be used to account for the various sources of efficiency loss.

The first three differences came as the result of the Cray's high throughput rate, its plentiful memory, and the fact that it has only a few processors. The final difference is a consequence of the uncertainty in measuring times on computations with multiple threads of control. We used the same test problem for the Cray X-MP as for the iPSC: initially two Finite Area Vortices (FAVs) of radius 0.12 with centers separated by 0.25. However, we allowed many more vortices on the Cray. We ran with two different problem sizes – 12848 and 25702 vortices – and used the following simulation parameters:

N	h_v	h	C	Δt
12848	2.6516×10^{-3}	1/60	1/30	0.0125
25702	1.8750×10^{-3}	1/120	1/60	0.0125

The time step Δt was chosen in accordance with the results of the 3 parameter study of chapter 3. The correction distance C, however, had to be made somewhat larger to avoid certain overhead costs that increase as C decreases. To decrease C, the finite difference mesh spacing h must also decrease. But the cost of setting up the finite difference computation is $O\left[\frac{1}{h}+\frac{1}{h^2}\right]$, where the first term corresponds with setting up Dirichlet boundary conditions, the second with setting up the right hand side. As C decreases, then, the number of local interactions computed decreases, but the cost of the finite difference computation increases. In addition, decreasing h also decreases bin size and hence the number of vortices in each bin. Since vector length equals the cardinality of a bin for most parts of the local computation, decreasing the size of the bins can slow the calculation down, even if the total number of arithmetic computations performed is reduced. This is shown in Table 6.1. For the run with 12848 vortices, for instance, decreasing C from 1/30 to 1/60 didn't speed up the calculation but actually slowed it down slightly. The reason why was that although the total number of floating point operations was reduced roughly by a factor of two, so was the rate at which those operations got done; the average vector length decreased from 160 to 40 elements, as determined from trace-files.

6.4.2. Computational results

Except as noted, all the runs were done during two four-hour blocks of dedicated time. At the end of the simulation the positions of all the vortices were written to a trace-file using formatted I/O. We later compared the results obtained from different numbers of processors but with the same number of vortices. In all cases the results agreed to printing precision (5 decimal

N	C	Local Interactions		Finite Differences		Overall	
		Time (sec)	Mflops per sec	Time (sec)	Mflops per sec	Time (sec)	$\times10^9$ flops
12848	1/15	67.4	89	3.02	84	72.0	6.30
12848	**1/30**	**28.2**	**69**	**5.37**	**68**	**36.1**	**2.33**
12848	1/60	19.1	34	13.0	47	38.5	1.35
25702	1/60	95.1	78	8.85	79	108	8.19
25702	**1/120**	**45.8**	**50**	**17.9**	**63**	**71.6**	**3.50**
25702	1/240	36.9	23	49.8	42	112	3.50

Table 6.1. The optimal values used for the correction radius C have the property that increasing or decreasing them increases the overall running time of the computation. The optimal values, shown in bold-face, are larger than the minimum allowable values, immediately below the bold-face, owing to an increase in the cost of finite difference computations. Two effects contribute here: the amount of work increases and the rate at which work can be done decreases. Using the floptrace facility we were able to measure the running time and the execution rate for the two dominant parts of the computation: local interactions and finite difference computations (finite difference computation ignores the time spent in the Poisson solver). We also report the running time and the floating point operation count for the entire computation. Though these measurements were taken from runs that lasted just two timesteps, we have evidence that the operation count fluctuates only slightly as a function of time, perhaps no more than 10% from the time-averaged value.

digits).

The first question we ask is how well can our VM utilize the X-MP's processors? As on the iPSC, we used parallel efficiency as the figure of merit. Unlike the iPSC, however, we were able to measure T_1 directly, since we ran on at most 4 processors. We ran the two instances of the 2-FAV problem on $P = 1, 2, 4$ processors. To conserve scarce dedicated computer time, we started the runs at an advanced stage of the simulation, using a snapshot file that had been generated from a simulation done on another X-MP, where batch time was more plentiful. The snapshots were taken at 10.0 units of simulated time; this is an interesting point in the simulation where the patches have begun to entrain, as shown in the plots of Fig. 4.10. We ran the 12848-vortex run for 400 timesteps, the 25702-vortex run for 240 timesteps. Load balancing was done every time step. Table 6.2 gives the timings and the parallel speedup and efficiency for the various runs. Efficiency was never less than 89%; under ideal conditions of 100% efficiency the programs would run only 12% faster. Parallel efficiency was quite good. Nevertheless we would like to know what prevented us from achieving 100% efficiency.

There are six causes of efficiency loss, some of which are interrelated, and none of which are present in a uniprocessor run:

(1) Load imbalance. The `Partitioner` utility cannot do a perfect job of distributing the work among the processors.

(2) Serial bottlenecks. The Poisson solver and parts of the program initialization ran on just one processor.

(3) Memory contention. Concurrent access to the same memory bank or to the same word in memory can increase memory access time.

(4) `Mapper` overhead. `Mapper` passes messages by moving blocks of data through memory. In turn, the MLC program must move vortices out of their linked lists into the blocks, and back again. `Mapper` also invokes various synchronization primitives that incur an overhead cost.

(5) `Partitioner` overhead. This entails producing a work estimate mapping and invoking the recursive bisection algorithm.

(6) Global communication overhead. There is an overhead associated with accumulating and broadcasting finite difference arrays. Calls to Larson's barrier synchronization primitive were used within these communication routines, as well as in various other places within the MLC program.

We have ranked these sources of overhead in what we believe to be decreasing order of cost. Since *flowtrace* does not apply to multitasked programs, we had to resort to indirect means of determining the various overheads. We consider only the case of 4 processors; our measurement techniques are too imprecise to account for the slight overheads incurred on 2 processors. Our technique is to account for the various overheads individually by computing an ideal efficiency that ignores all overhead costs except the one in question, and then to multiply the all ideal efficiency measures to arrive at a guess of overall efficiency. This efficiency will then be compared with what was observed. This procedure is a bit imprecise; we had no way of judging how some overheads correlated, e.g. how the construction of a local memory on top of shared memory affects memory contention. We speculate that our methodology will give a pessimistic guess.

Load imbalance. We estimate load imbalance by tallying the number of local corrections computed by each task, and using this as a measure of time to compute the parallel efficiency. We modified the code to compute this estimate of load imbalance, and ran for just 20 timesteps— running for hundreds of timesteps would have been too expensive, and we believe that our efficiency estimate would not have changed significantly. Table 6.3 shows that the estimated efficiency was 0.965 for 12848 vortices, for example.

Serial bottlenecks. To measure the cost of computations that ran as one task, we measured a uniprocessor version of the program running under flowtrace. Using the profiling information we determined the fraction of the time spent in non-parallelized computations and then applied

N	P	Time (sec)	S_P	η_P
12848	1	5999	1.00	1.000
12848	2	3081	1.95	0.973
12848	4	1651	3.63	0.908
25702	1	7032	1.00	1.000
25702	4	1970	3.57	0.892

Table 6.2. Timings, parallel efficiency and speedup for the X-MP runs. All runs began at 10.0 units of simulated time. The runs with 12848 vortices ran for 400 timesteps, the larger run for 240 timesteps. We did not run the larger problem on 2 processors since we needed the CPU time for other experiments.

N	T_1	T_4	Speedup	Efficiency
12848	1.981×10^8	5.129×10^7	3.86	0.965
25702	2.209×10^8	5.646×10^7	3.91	0.978

Table 6.3. We estimated the load imbalance on 4 processors by computing an efficiency measure based on the local correction as a unit of time. Efficiency is idealized in that it ignores all effects other than load imbalance. T_1 is defined as the total amount of time consumed by each task. T_4 is defined as the sums of the times on the most heavily loaded task for each timestep. Generally the most heavily loaded task varies with time. The runs lasted just 20 timesteps.

N	h	Time fraction	Efficiency
12848	1/60	0.006	0.982
25702	1/120	0.015	0.957

Table 6.4. The idealized efficiency for serial bottlenecks on four processors is high because only a small fraction of time was spent in the solver. The table also shows h, the finite difference mesh spacing used by the solver.

Amdahl's law to get an idealized efficiency that measured only the serial bottlenecks. The Poisson solver accounted for nearly all non-parallelized computation. The two other parts the computation that did not parallelize well – program initialization and vortex tracing – consumed an insignificant amount of time. Both did a lot of I/O. Table 6.4 gives the Poisson solver time per timestep for the two values of h used, as well as the idealized efficiency measure that considers only the cost of serial bottlenecks.

Memory contention. To gain a rough estimate of memory contention we executed the operation "x := x + 0.01" 10^5 times within an inner loop that was in turn repeated $30720/P$ times, where P was the number of tasks. This test program executed 3.07×10^9 floating point operations regardless of the number of tasks and loads were perfectly balanced. The program ran for about 9 seconds on four processors. Because the only calls to the multitasking primitives where made at the beginning and the end of the program, (`tskstart` and `tskwait`), memory contention could be the only plausible cause of performance loss. Table 6.5 gives the running times and the ideal speedup and efficiency.

Mapping overhead. To measure the overhead due to `Mapper`, we constructed a shared-memory version of the MLC-code, in which the vortices and the bins resided in global common, and `Mapper` was therefore not used. We then compared the running time of this version to the local memory version running with `Mapper`. We had enough computer time to do one full-

Processors	Seconds	Speedup	Efficiency
1	36.9696	1.00	1.000
2	18.5256	2.00	0.998
4	9.4485	3.91	0.978

Table 6.5. Measurement of an ideal efficiency measure that ignores all overheads except memory contention. This test program accesses memory twice per floating point operation, which is three times the rate at which the MLC program accesses memory. Where therefore expect that the memory contention for the test to establish an upper bound on the memory contention for the MLC.

length run; we ran with 12848 vortices on 4 processors for the full 400 timesteps. Surprisingly, the net effect of running with `Mapper` was found to speed up the computation by 0.3%. This savings is too small to affect our analysis and presumably is due to a reduction in memory bank conflicts. We can't say by how much, however, since we can't separate `Mapper's` overhead from the overhead due to memory contention.

Partitioning overhead. As previously mentioned, flowtrace revealed that the `Partitioner` utility consumed insignificant time, so its costs may be ignored. However, we ignored the cost of computing `partitioner's` work estimate mapping in that analysis. Using flowtrace, we found that the fraction of time spent producing the work estimate mapping on a uniprocessor was 0.22% for 12848 vortices and 0.18% for 25702. However, this computation parallelizes, and, assuming that loads were about as well balanced as they were in the remaining part of the computation, the overhead of work estimation would drop to less than 0.05% on 4 processors, which is too small to be of concern.

Global communication. Per our previous analysis, the overhead of the global communication utilities was 1% of the cost of setting up the right hand side for the Poisson solver. However, *that* operation accounted for never more than 3% of the entire computation, so the cost of global communication can effectively be ignored. We also lump the cost of Larson's barrier synchronization routine with global communication. There were six barrier synchronization points in each timestep, used to interleave the boss task with one of the worker tasks. Each synchronization cost roughly 72,000 CPs ignoring wait time due to load imbalance. By comparison, the cost of arithmetic computation was roughly 10^9 flops per timestep. If in a worst case analysis we assume that 1 flop executes in 1 CP, then global communication adds only about 0.01% to the computation's running time and may be ignored.

We next multiply the various ideal efficiencies to arrive at a guess of the overall efficiency. With 12848 vortices running on 4 processors, we predict 92.8% efficiency which agrees reasonably well with was measured – 90.8%. With 25702 vortices we predict 91.5%, which again is close to what we observed – 89.2%. Given the rough nature of our analysis, there are many possible sources for error. We have ignored the costs of I/O and of producing the work estimate. If we include the cost of these operations, then the predicted efficiency is 91.8% for the 12848 vortex run, and agrees even more closely with what was measured[9].

6.5. Conclusions

We have successfully applied our programming methodology to a very high-performance architecture with vector arithmetic capabilities. We achieved speedups in excess of 3.5 on 4 processors. The code was fully vectorized and ran at a rate of 200 to 250 megaflops/sec on the four processors. The overhead of our communication and load balancing utilities was negligible; indeed, use of the local communication utility actually sped up the computation slightly. The two major sources of inefficiency were load imbalance and the Poisson solver that ran on just one processor. Parallelizing the solver, however, would not significantly improve throughput on a machine with only 4 processors, though if we had a Cray-like multiprocessor with 8 or 16 CPUs, we would have a much strong incentive to parallelize the solver. A simple strategy for supporting global communication and synchronization works reasonably well; the strategy is compatible with what was done on the iPSC and incurs a modest overhead.

6.6. References

1. S. S. Chen, C. C. Hsiung, J. L. Larson and E. R. Somdahl, "CRAY X-MP: A Multiprocessor Supercomputer," in *Vector and Parallel Processors: Architecture, Applications, and Performance Evaluation*, M. Ginsberg (editor), North Holland. To be published..

[9] The 12848 vortex problem ran in 5999 seconds on a uniprocessor, and in 1651 seconds on 4 CPUs. The times spent initializing and tracing were 4.29 and 6.01 seconds, respectively, on the uniprocessor run, and 10.28, 6.49 seconds on the 4-processor run. The 4-processor run also incurred an additional 3.54 seconds overhead incurred by workload estimation.

2. *Cray-1 FORTRAN (CFT) Reference Manual (2240009)*, Cray Research, Inc., Mendota Heights, MN, 1978.

3. *Cray X-MP Hardware Reference Manual*, Cray Research, Inc., 1986. Order number HR-0097.

4. *CRAY X-MP Multitasking Programmer's Manual*, Cray Research, Inc., March 1986. Order number SN-0222.

5. R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Adam Hilger, Bristol, 1981.

6. R. M. Russell, ''The CRAY-1 Computer System,'' *Comm. ACM 21*,1 (January 1978).

7. R. L. Sites, ''An Analysis of the Cray-1,'' *Proc. of the 5th Symp. on Computer Architecture*, 1978, pp. 101-106.

7

Discussion and Conclusions

... you spend a good piece of your life gripping a baseball and in the end it turns out that it was the other way around all the time.

–Jim Bouton, baseball pitcher

7.1. What Has Been Accomplished

We have presented a programming methodology that allows its user to cope with run-time partitioning reasonably well without having to pay attention to all the low-level details. We have tried out the methodology on a substantial numerical computation –Anderson's Method of Local Corrections (MLC), a vortex method for modeling the flow of an incompressible ideal fluid–and have obtained good parallel speedups on two diverse architectures–the Cray X-MP and the Intel iPSC.

Our programming methodology is intended for calculations that fit a simple model of spatial locality that we believe applies not only to the MLC but also to many other scientific and engineering calculations that arise in such areas as:

- fluid mechanics,
- plasma physics,
- astrophysics,
- structural engineering,
- electronic circuit simulation, and
- computer graphics.

Our approach is to provide the user with a virtual machine (VM) consisting of software utilities for handling task decomposition and communication activities. The VM makes no special requirements of either programming language or computer architecture. Its semantics are thoroughly decoupled from the application and from the underlying layers of hardware and software that handle communication. The VM is not universal, however; it applies to only the localized part of a computation, and is intended for systems with tens of processors, though it may still be more generally effective for bigger systems so long as the amount of non-localized work to be done isn't too great. Nevertheless, we believe that the VM will help diminish the user's preoccupation with such details as whether he is using a message-passing or a shared-memory architecture.

Because our VM applies to communication with only a certain kind of localized structure, the programmer must to deal with other kinds of communication himself. We believe, however, that the user may be able to rely on someone else's software for handling some of the more commonly-occurring communication structures to which our VM does not apply. Communication structures that arise in array-based computations, like the Poisson solver we used in the MLC, are fairly well understood, and could be handled by a standard application library for multiprocessors. Indeed, the spanning-tree communication utilities we used on the iPSC were also implemented with the same semantics on the Cray X-MP.

Our VM relies on a redundant storage scheme to cache information a processor does not properly own. Though this does reduce the amount of memory otherwise available to the computation, the overhead will be be reasonable for localized computations. For calculations with strong long-range coupling, however, the storage overhead would be unacceptable. If some numerical scheme could be found to weaken the long-range coupling, then the need to store large amounts of information redundantly could be avoided. The Method of Local Corrections, for instance, provides the means of approximating N-body interactions without entailing massive global computation. This is possible because of the logarithmic potential governing the motion of the vortices: distant interactions can be computed by means of a fast Poisson solver involving a small number of degrees of freedom; direct interactions will be computed only among nearby vortices. Although such methods are attractive from the standpoint of their localized communication structure, they can also have a lower operation count than globally strongly-coupled methods, even on uniprocessors on which the MLC, for instance, can be substantially faster than the direct N-body method. Thus, the need to localize a computation for the purpose of parallelization may also be beneficial because it encourages the user to seek a numerical method that is more efficient regardless of parallelization.

Though computations running under our VM will exhibit physically localized data dependencies, this does not necessarily mean that the VM will localize their communication structures within processor space. In running the MLC on iPSC, for example, each processor was observed to communicate with many others during a timestep. The reason why this is so is that our VM partitions work into different-sized tasks that do not have a fixed connectivity with respect to neighboring tasks, and because some tasks are so small that others may still interact even if they are not nearest neighbors. Such a communication structure stressed the iPSC's communication subsystem; processors occasionally became clogged with messages, resulting in a severe degradation in the message-passing time. This had the undersirable side effect of consuming storage that would otherwise be available to the user, since the operating system had to reserve additional storage for messages that pile up at a node.

Experimental results from our research suggest, to a certain extent, that there is an optimal range of problem sizes N for each particular value of P, the number of processors. In particular,

N must be varied concomitantly with P. On the iPSC, for example, the largest problem we ran would not fit into the memory of a single processor. If N is too small for a given value of P, the cost of administrative overheads and of computations that don't parallelize well will become unreasonable. In MLC, the number of vortices must be kept sufficiently large to prevent the Poisson solver from consuming too large a fraction of the computational effort, and becoming a performance bottleneck. On the Cray X-MP, we were obliged to run with at least 10,000 vortices on four CPUs, since the solver ran on just one CPU.

Our results also suggest that as the number of processors increases, load imbalance has a tendency to increase, unless the spatial resolution available for task subdivision also increases. For the MLC, this meant keeping track of the vortices on an ever-finer mesh. There is a cost, however, associated with improving the resolution of task subdivision, since the number of bins in the *workLattice* will increase. The added cost of manipulating and storing more bins places an effective limit on how well loads can be balanced, or alternatively how many processors may effectively utilized; improvements in the workload imbalance may actually degrade throughput, or there may not be enough memory to store the additional bins. We ran into both these problems on the iPSC; however, if we had exploited the sparseness of the MLC's *workLattice* mesh, we could probably have improved efficiency somewhat.

There may be other more compelling reasons for not wanting to avoid load imbalance entirely because, long before that good point is reached, other performance bottlenecks like non-parallelizable computation may well become more significant. As load imbalance decreases, the total running time of the computation decreases and those bottlenecks consume an ever-increasing fraction of that time (Amdahl's law). On 32 processors of the iPSC, for example, reducing load imbalance to zero would speed up the computation by only about 30%.

The aforementioned problems with non-parallelizable computation and load balancing overhead bound the number of processors P_{crit} that may be effectively used to speed up a computation. The performance of the communication subsystem and the total amount of memory available to the user are also important factors in determining P_{crit}. One important measure is the ratio, R, of the maximum *sustainable* rates of communication and of computation. The maximum sustainable communication rate is important because our VM breaks a calculation into relatively long phases of computation, interspersed by relatively short but concentrated bursts of communication during which the communication subsystem is under maximum stress. Communication can be thought of as a kind of serial bottleneck; as R decreases, so does P_{crit}.

7.2. Comparison with Other Approaches

Our approach to run-time partitioning has two characteristics: (1) it partitions work at a coarse level of granularity; and (2) it shuffles around work only at well-defined times, and then only when computation is suspended. These characteristics allow it to exploit spatial localization and avoid high overhead costs even on message-passing architectures. One drawback of our approach is that we require its user to discipline communication somewhat and to predict the work density distribution of his computation. In light of their beneficial effect on performance, we believe these requirements will turn out to be as reasonable in other applications as they were for our model problem.

By and large, two approaches to run-time partitioning compete with ours: dataflow, and processor self-scheduling. We focus on these approaches because they have received much more attention than others. Both, and especially dataflow, have motivated a number of architectural innovations. Most experience with practical dynamically partitioned scientific applications appears to be with processor self-scheduling. In contrast to our approach, dataflow and processor-self scheduling use fine-grained partitionings to diminish load imbalance, and they assign work to processors on demand. A major difficulty that remains is how to exploit spatial locality without increasing load imbalance, in order to avoid high communication and administrative overheads. The problems are much more severe for dataflow than for self-scheduling because dataflow's tasks have a much finer granularity; the work a task does can be overwhelmed by the cost to manage it.

The principal advantage that dataflow has over our approach is that in theory the user would hardly know that he was using a multiprocessor. We say ''in theory'' because current dataflow implementations must somehow overcome substantial performance bottlenecks to compete with the cost/performance of traditional computer architectures. We believe, however, that even if the obstacles were overcome, a traditional system running our VM would be far simpler and less expensive build than its dataflow counterpart.

Processor self-scheduling is applicable to a wide variety of mathematical-physics problems. The major problem with it is that contention on the shared data structures used to control task dispatchment tends to increase with the number of processors, and can become a serious performance bottleneck. Furthermore, on vector architectures vector lengths can be extremely sensitive to the number of tasks [5]; some computations may actually run fastest on just one processor. If these problems were overcome, however, processor self-scheduling would probably still apply primarily to shared-memory architectures, and be of only limited use on message-passing architectures. The reason why is that the small quanta of shared data used to control task dispersal must constantly shuffle among the processors, and doing so is expensive owing to a high message startup cost.

Though our VM doesn't balance workloads perfectly, and though the user must provide it with a cost function appropriate for his application, we believe that the benefits from these compromises far outweigh their costs. Computations can vectorize as well as they would in a uniprocessor implementation. Given a prediction of computational effort, the VM is able to adjust the assignment of work by redistributing it among tasks without having to break the tasks into ever smaller pieces, and can therefore use coarse-grained coherent partitionings that are known to incur low overhead costs. If the solution changes slowly enough, as it did in the MLC, loads shift gradually among processors according to local changes in the density of computational effort, and the imbalance of loading builds up gradually enough that intermittent rather than continual re-balancing works well enough. The VM can partition work semi-statically, at convenient stopping points in the computation, rather than continually interleave work-redistribution with computation. This is desirable since it reduces the amount of shared information that would otherwise be manipulated expensively during computation. The single global work queues used in processor self-scheduling, for example, are effectively replaced by private balanced queues, one for each processor.

7.3. Future Work

We think that our VM should apply to a class of problems that is far more general than the single model problem discussed here. We have already begun collaborating with Berger and Colella [2] on how best to partition adaptive grid methods. These computations are interesting because they represent the solution at different levels of description using a hierarchy of subproblems. We have also implemented a subset of our VM – the partitioner utility – on ten processors of the Sequent Balance 8000, a shared-memory multiprocessor, and applied it to G. K. Jacob's PSPLICE3 circuit simulation program [4]. The installation required only a few changes to the PSPLICE3 code, and preliminary results indicate that performance is comparable to what it was when an old self-scheduling algorithm was used. We are developing the remaining parts of the VM in order to facilitate the installation of the PSPLICE3 code onto a hypercube architecture that uses message-based communication.

We expect that our VM would have to be enhanced to run on architectures that support high levels of parallelism, such as the N-Cube Corporation's N-CUBE with up to 1024 processors. As the number of processors increases, so does the incentive to reduce communication overhead by a prudent assignment of tasks to processors. Our current strategy is to assign tasks in any convenient way. A better way would be to assign the more heavily communicating tasks in a way that best utilized the communication links with the greatest bandwidth. Bokhari [3] has referred to this as the processor mapping problem. However, the semantics of our VM would not change with the inclusion of processor mapping.

We are pursuing additional questions concerning how well our approach generalizes to various kinds of calculations. We are collaborating with Buttke and Colella [1] on a three-dimensional vortex method. The computation will be applied to vortex breakdown, a problem exhibiting spatial workload distributions that vary wildly with time. Also of interest are heterogeneous problems that exhibit different physics in different regions of the problem and which must be treated by different numerical methods. Boundary layer problems involving viscous fluid flows are one example.

7.4. References

1. S. B. Baden, T. Buttke and P. Colella, "A Fast Adaptive Vortex Method in Three Dimensions," In preparation, 1987.
2. S. B. Baden, M. J. Berger and P. Colella, "Multitasking Strategies for Adaptive Fluid Dynamics Calculations," in preparation, 1987.
3. S. Bokhari, "On the Mapping Problem," *IEEE Trans. Comput. C-30*,3 (1981).
4. G. K. Jacob, private communications.
5. J. Nicholson, private communications.

Appendix A

This appendix contains the source listings for the software written during the course of this research. The listings have been photographically reduced onto microfiche; a copy may be obtained by writing to the following address:

Scott B. Baden
Lawrence Berkeley Laboratory
50A-2129
University of California
Berkeley, CA 94720
USA

There are a total of 7 listings. The name of each listing appears in block letters at the top of each page of microfiche and is visible to the unaided eye. The listings are as follows:

NAME	EXPLANATION	# FILMS
UNIBLOB	Uniprocessor Cray X-MP MLC program	2
XMP-LOCAL	Local memory multitasked version of above, including all utilities	2
XMP-SHARE	Shared-memory model version of above	2
PLOTTING	Cray plotting program	1
BLOBIPSC	iPSC version of MLC program	3
IPSCLIB	iPSC utility libraries	1
REPORT	iPSC timing report program	1