LBL-22584
Preprint

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Physics Division

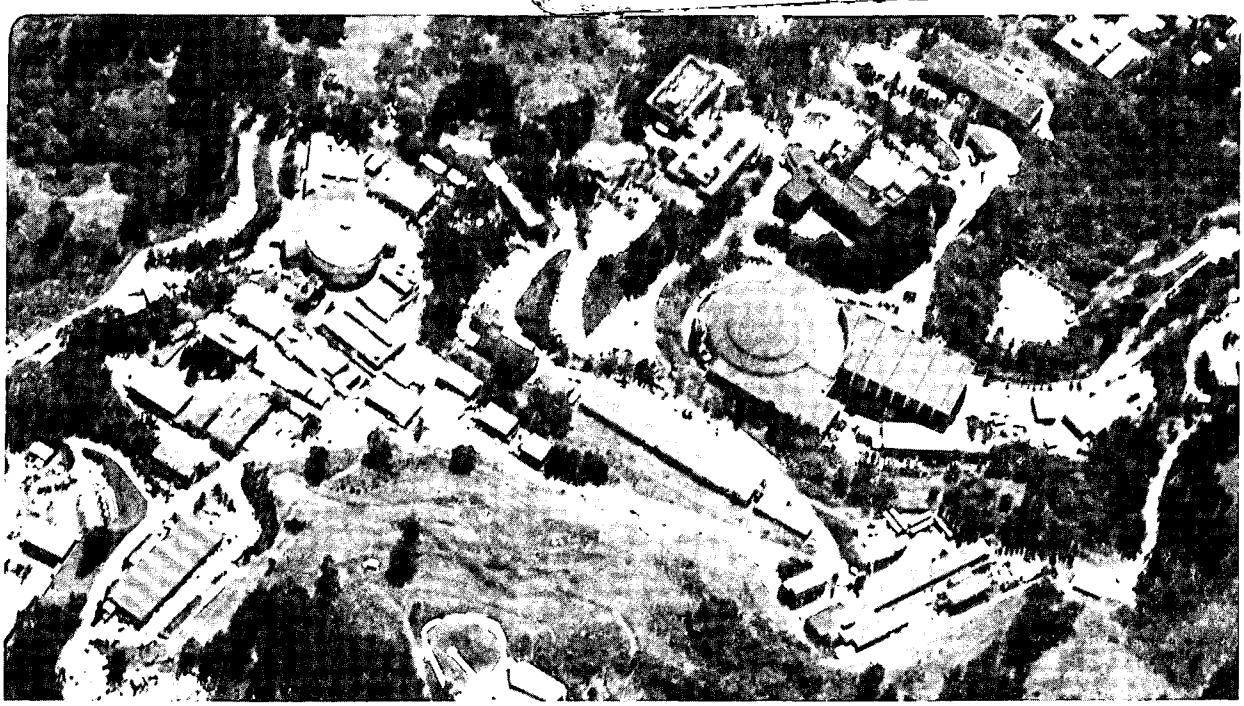
Mathematics Department

To be submitted for publication

DYNAMIC LOAD BALANCING OF A VORTEX CALCULATION RUNNING ON MULTIPROCESSORS

S.B. Baden

December 1986



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

DISCLAIMER

LBL-22584

DYNAMIC LOAD BALANCING OF A VORTEX CALCULATION RUNNING ON MULTIPROCESSORS[1]

Scott B. Baden

Computer Science Division and Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

December 1986

[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.

DYNAMIC LOAD BALANCING OF A VORTEX CALCULATION RUNNING ON MULTIPROCESSORS

Abstract

We discuss a dynamic load balancing strategy intended for various mathematical physics calculations that partitions the work fairly across a multiplicity of processors. Anderson's Method of Local Corrections serves as a model problem; it is a type of vortex method for computational fluid dynamics. Because computational effort follows particles which congregate and disperse irregularly about the domain, this problem is hard to partition in a way that distributes the work evenly among the processors. The load balancing strategy was tested on 32 processors of an Intel Personal Scientific Computer, a message-passing hypercube multiprocessor. The load balancer may be implemented as a small subroutine library that requires no special hardware support. The library should apply to diverse problems, including finite difference methods, and to diverse machines, for instance shared memory architectures, without entailing massive reprogramming.

1. Introduction

Ideally a multiprocesser system would satisfy two conditions: (1) programs that run well on it should not look very different from those that run on a uniprocessor; (2) its performance would be linearly proportional to the number of processors in use without a substantial additional amount of hardware dedicated to overhead functions. However, uniprocessor programs that are adapted simply for multiprocessors can spend the majority of their time sitting idle rather than doing productive work. The major difficulty often lies in splitting up the computational work evenly across the processors, as work tends to concentrate among a few processors, leaving the others relatively unloaded. Attempts at coping with this *load balancing problem* have led to novel computer architectures and programming languages, such as dataflow [2,7], that distribute work automatically among the processors. However, the automatic work distribution schemes are disappointingly inefficient despite extensive hardware support. There are two problems with these schemes: (1) they partition work into parallel tasks with a fine *granularity*; (2) they fail to exploit a key *locality property* inherent in many mathematical physics problems. Task granularity is an important issue in load balancing. Parallel tasks must be small enough that they can be divided evenly among the processors but not so small as to be overwhelmed by the cost of managing them. In data flow implementations, for instance, the tasks tend to be too small. The locality property also has important ramifications. It states that physical effects at two arbitrary points in space and time interact more strongly (and hence convey more information to one another per unit of computation) at short distances than at long ones. This means that there should be a strong incentive to place nearby points on the

same processor. In dataflow implementations, however, no such attempt is made. This is especially problematic in dataflow since small tasks do a lot of communication relative to computation.

We propose a dynamic load balancing strategy that, unlike the automatic schemes, exploits the locality property and that doesn't suffer from the granularity problem when the processors number in the tens. 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 rare for the kinds of problems for which the strategy makes sense. The strategy parcels work into "chunks" that are somewhat larger and much easier to manage than the small pieces of work used in the dataflow schemes.

Our scheme assumes a particular programming discipline:

(1) Each processor computes on its assigned part of the problem at its own rate and out of its own private memory.

(2) Each processor's local memory is augmented by something that looks like memory shared with neighboring processors thereby providing a limited access to information that would otherwise reside outside its address space.

(3) The programmer explicitly invokes software utilities to handle load balancing and to provide for the limited memory sharing.

We have found that a small subroutine library is sufficient for enforcing this discipline. The library requires no special purpose hardware support to operate efficiently and may be installed on existing systems. Though the library applies to the localized part of a computation only, the scheme may still be effective more

generally, so long as the amount of non-localized work to be done is not too great. The library hides considerable detail from the programmer– in particular how processors communicate. The programmer must supply a small amount of code that the utilities call, but the code depends solely on the application and in no way on the architecture. Code written in a higher level language that calls on our utilities should run without major alteration reasonably well on other machines provided the utilities have been implemented there.

The library tends to diminish the importance of what distinguishes one architecture's communication mechanism from another's. Message passing architectures conveniently come to resemble shared memory architectures, and hence become easier to program. On architectures that provide it, shared memory gets used in a restricted way that can reduce memory contention and that also exploits local memory, if provided.

We have implemented our load balancing utilities on the Intel Personal Scientific Computer (iPSC), a hypercube-type multiprocessor. To test the strategy we applied it to an implementation of Anderson's Method of Local Corrections [1], a two dimensional vortex method for incompressible inviscid flow. This particle method typifies various problems that are compute bound, and that appear well-suited to parallel computation, but which are hard to partition. The difficulty arises because these calculations expend effort that varies non-uniformly over the spatial domain of the problem and unpredictably with time.

We obtained speedups of 22 on 32 processors. The overhead of the load balancing utilities was less than 10%. We expect that the utilities will apply to a diversity of other mathematical physics calculations besides the Method of Local

Corrections– not only to particle methods arising in fluid dynamics, astrophysics, and plasma physics, but also to finite difference methods.

Section 2 briefly summarizes the important details of the model problem, gives a simple multiprocessor implementation strategy, and introduces the load balancing problem. Sections 3 and 4 present the two major load balancing utilities. Section 5 discusses iPSC implementations of both the model calculation and the utilities, section 6 evaluates computational results. Section 7 concludes the paper.

2. A Model Calculation and its Parallel Computation

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:

$$\frac{\partial \omega}{\partial t} + \mathbf{u}\cdot\nabla\omega = 0 \tag{2.1a}$$

$$\omega = -\Delta\psi \text{ in } \Omega, \tag{2.1b}$$

where $\mathbf{u}(\mathbf{x}(t),t)$ is the velocity of the fluid at position $\mathbf{x}(t)$ at time t, ω is *vorticity*, defined as the curl of $\mathbf{u}$, ψ is the *stream function*, $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the two dimensional Laplacian operator, and Ω is a square box. The flow satisfies the *no flow* boundary conditions, i.e., the fluid may not penetrate the solid walls of the box, which will be satisfied if:

$$\psi = 0 \text{ on } \partial\Omega. \tag{2.1c}$$

For a thorough discussion of these equations, see Chorin and Marsden's introductory text on fluid mechanics [6].

2.1. The Calculation

A *vortex blob method* [5] will be used to solve the equations (2.1). 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 "vortices." The particular method we will use is Anderson's Method of Local Corrections [1], henceforth called "the MLC." 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 vortices number in the thousands or more, the calculation spends almost all of its time computing local N-body interactions between nearby vortices. Vortices that are not close to one another interact indirectly through the relatively inexpensive global finite difference computation. We will focus primarily on the local interactions.

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 Fig. 2.1. This technique is discussed in the text on particle-based calculations by Hockney and Eastwood [10]. 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 Fig. 2.1 where $C = 1$. In practice the bins used in the MLC are much smaller than shown in Fig. 2.1 and the vortices interact directly over short distances only (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).

2.2. Dynamic Load Balancing

A simple way to divide up the work in the MLC is to split the bins into a regular pattern of box-like subproblems, as shown in Fig. 2.3a, and to assign each subproblem to a unique processor. 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– the completion time for a subproblem may not be proportional to its area. Fig. 2.3b shows a better way to split up the problem that compensates for the uneven distribution of vortices over the domain. This strategy 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 three.

But the partitioning cannot be left fixed for all time; the vortices move and must be reapportioned as shown in Fig. 2.4. If the work were not redistributed, then some processors would become overloaded while others would only stand and wait. Thus, redistribution advances the latest completion time by shifting work from the more heavily loaded processor(s) to the more lightly loaded one(s), i.e. it balances the workloads. This happens at run time, hence the term "dynamic load

balancing."

2.3. A Local Memory Computational Model and Programming Discipline

Programs that employ the load balancing utilities must use a simple model of parallel execution in which each processor executes its own program out of a private address space and at its own pace. A call to a *partitioner* utility splits the computational domain into regions, as shown in Fig 2.2. Subsequent calls keep loads balanced by adjusting the shape and location of the regions according to how the vortices have moved since last they were partitioned. As a result of such work redistribution, a vortex could find itself owned by a different processor than before. Therefore, redistribution incurs a side effect of having to shuffle particles between private memories. Each processor must decide which particles to shuffle out, where to send them, and what to do with incoming ones. A *mapper* utility handles all the details of how this is accomplished. All the architecture-dependent details are hidden from the programmer. When *mapper* finishes, each processor knows about all the vortices that migrated into its assigned region of space.

What *mapper* does is to provide local paths of communication between the subproblems. Because vortices move very slowly, their redistribution in the MLC is a gradual process. Slender regions of the computational domain, containing small numbers of vortices, shift between processors. Though *mapper* does not provide support for global communication, we have found that a set of utilities discussed by Moler [13] provide nearly all the functionality needed to cope with the global computations in the MLC. These routines apply primarily to finite difference computations: one accumulates arrays stored on different processors into one array stored on one processor, and another broadcasts an array stored on one

processor onto a designated set of processors. These have been used for the global finite difference part of the computation about which we shall have little more to say.

In summary, our strategy for dynamic load balancing entails adopting a local memory model of parallel execution, and periodically invoking two utilities called *partitioner* and *mapper*. The next two sections discuss each utility in turn.

3. Partitioning

The *partitioner* utility splits a problem into a given number of subproblems that all complete at roughly the same time. For the MLC, subproblems correspond to subarrays of the bins; for a finite difference calculation, a subproblem would correspond to a subarray of a finite difference mesh. The particulars of these data structures are not important so long as the computational domains they represent subdivide naturally into rectangles.

The *internal boundaries* that separate the subproblems are distinct from the *physical boundary*, $\partial\Omega$, of the problem. These internal boundaries partition space into rectangular blocks of bins. Internal boundaries may not sever bins but must lie between them as shown in Fig. 2.2. A less restricted scheme that allowed internal boundaries to pass through the bins would impose additional coding overhead, and appears to confer no advantage.

The bins, then, represents a small scale subdivision of space that is used by the numerical method. A larger scale partitioning is imposed on top of this for the purposes of splitting up the problem among multiple processors. This partitioning strategy imposes some overhead costs, i.e. interprocessor communication,

but it does not change any of the arithmetic operations that would be done on a uniprocessor, other than to reorder them. This means that results from successive runs using the same initial conditions will agree to within roundoff, no matter how many processors are used.

3.1. Recursive Bisection

The *recursive bisection algorithm* is a simple but effective way to partition calculations such as the MLC. It has been used by Dippé and Swensen [8] and Dippé and Wold [9] for realistic rendition of computer graphics images and by Berger and Bokhari [3] 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 Fig. 3.1. This simple procedure generalizes trivially to higher dimensional problems.

The algorithm takes two inputs: P and *workGrid*. P is the number of processors and *workGrid* is a mapping used to estimate the completion time for subproblems. The algorithm returns two outputs: the number of subproblems actually rendered – it may not always be able to generate the requested number – and a table describing the subproblems. Each entry in the table has attributes "origin" and "shape."

The *workGrid* is a rectangular array of integers supplied by the user; each integer is proportional to the amount of work required to compute on a small subregion of the computational domain represented by a single bin. To bisect a region of space the algorithm advances along the columns (or the rows) of the *workGrid* until the two sums of all the entries in the left and right hand parts (or

upper and lower parts if advancing along the rows) match as closely as possible. To cut P subproblems the algorithm gets called $\lceil \log_2(P) \rceil$ times. Though we restrict P to be an integer power of two, a simple change to the algorithm allows it to cope with other values of P. If box-like subproblems are desired, the algorithm alternates the direction of the cuts from one invocation to the next. If strip-like subproblems are desired, then cuts lie in one direction only.

3.2. Work Estimation

Work estimation entails producing an array giving the estimated completion times for each bin. The *workGrid* is *set-additive*, meaning that the work in any rectangular region is the sum of the work in two constituent rectangular subregions. It follows, then that a subproblem comprising a collection of bins finishes in time that is proportional to the sum of its corresponding *workGrid* entries. Strictly speaking, our partitionings of the MLC are not set-additive since they introduce extra work in the neighborhoods of the internal boundaries that divide the subproblems. However, the extra work can be ignored since it is negligible compared with what what gets done regardless of how the calculation has been partitioned.

A reasonable work estimate mapping for the MLC ignores the long-range interactions computed in the global part of the calculation. When the vortices are numerous the MLC spends most of its time computing local interactions. The completion time for a bin then is the number of local interactions computed involving its vortices. This value 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:

$$\mathrm{pop}(i,j)\left[\sum_{\substack{|k|,|l| \le C \\ (i+k,j+l)\in\Omega}} \mathrm{pop}(i+k,j+l)\right] \tag{3.1}$$

where "pop(i, j)" is the number of vortices in bin (i, j) and i and j range over all the bins in the computational domain Ω. The result is an integer array.

3.3. Discussion

The recursive bisection algorithm is a useful abstraction for determining a fair partitioning of work across a multiplicity of processors. Since the algorithm carries around no knowledge about either the application or machine architecture, it is ideally suited to the task at hand. Obviously, the effectiveness of the proposed partitioning strategy depends on the user's ability to construct an inexpensive and accurate work-estimate mapping, a process that entails writing some application-dependent code. However, the work estimation procedure for the MLC was neither long-winded nor difficult to write; it parallelizes, runs in negligible time, and is reasonably accurate.

4. Mapping

The *mapper* utility implements the limited form of memory sharing assumed by our dynamic load balancing strategy. It hides the semantics of the communication model supported by the target computer architecture; a message-passing architecture, for instance, would appear no different from a shared-memory architecture, during the local part of the computation. But *mapper's* action is neither completely transparent nor automatic: the programmer must invoke it explicitly; he must use it correctly; and, he must provide it with a small amount of application-dependent code. However, we believe that these requirements will be as acceptable in other applications of our strategy as they were for the MLC. We

present a *mapper* for two dimensional calculations, though the strategy readily generalizes to problems of any dimension.

Mapper allows the interdependent subproblems that *partitioner* assigns to private address spaces to interact as if executed in a single address space. Vortices in the MLC, for instance, that interact across internal boundaries must somehow become visible to processors that don't own them, as shown in Fig. 4.1a. Therefore, each processor must obtain information about vortices just on the other side of the internal boundaries of its subproblem.

Mapping must be done synchronously in order to guarantee correct results. No processor may begin with the next step of the computation until it has received all data to be mapped to it nor until it has finished supplying similar such information needed by others. If mapping were handled asynchronously, then a vortex, for instance, could simultaneously appear to different processors to be at different positions and results would be unpredictable. Explicit calls to *mapper* specify *synchronization points* at which times data get set up in a consistent state across all processors before the computation may continue.

In order to see how mapper works, consider a computation involving four processors. Assume that *partitioner* had previously split the computational domain Ω into 4 sub-problems Ω_0, Ω_1, Ω_2, Ω_3, and that each Ω_i had been assigned a unique processor P_i, as shown in Fig. 4.1. Each processor owns its assigned subproblem and knows only about the computational elements assigned to it by *partitioner*, as shown in Fig. 4.2a. But interdependent subproblems interact through logically overlapping regions of space that straddle the internal boundaries as shown in Fig. 4.1. Each processor must somehow find out about computational elements

lying just outside its subproblem, as shown in Fig. 4.2b. For the MLC, the computational elements are vortices; they lie in the subproblem's *dependence region*, a surrounding region of space that logically overlaps other subproblems, as shown in Fig. 4.3. Conversely the interacting processors view this region of space as part of their *influence regions*, as shown in Fig. 4.3. The influence and dependence regions are shells whose thickness is given by an *interaction radius*, which for the MLC equals C, the maximum distance over which vortices may interact directly. What *mapper* does is to physically connect the logically overlapping influence and dependence regions. In the MLC, for instance, each processor will, through its dependence region, know about all the vortices that that had at one time lay only in the influence regions of other processors. The communications paths provided by *mapper* appear to implement a memory sharing mechanism. This mechanism, however, is primitive and provides only a subset of the functionality of a classic shared memory: it shares information only to the extent specified by the *interaction radius* and it buffers memory writes until the next time *mapper* gets called. Since writes do not propagate instantaneously calls to *mapper* must be made at the correct time or results would be non-deterministic.

Having now discussed the externally visible behavior of *mapper* we now move to the internal behavior of the utility. For the purpose of discussion, *mapper* will be specified in architecture-independent terms, assuming a message based model of communication. Let us characterize that model, perhaps simplistically, by two primitive operations *send* and *receive* (see, for example, the *iPSC User's Guide* [11] 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*. 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 strings of bytes.

Mapping divides into two activities called an *influence action* and a *dependence action*, respectively. For the MLC, the influence action collects vortices from the bins in the influence region and copies them into any processor needing them. The dependence action copies incoming information into the bins of the dependence region. Both influence and dependence regions divide into patches. Owing to the reciprocal nature of interactions, patches that make up the regions come in matched pairs: "dependence patches" from the dependence region, and "influence patches" from the influence region, as shown in Fig 4.4. Each patch contains computational elements that interact with just one subproblem. Every processor has an *interactions list* describing its matched sets of patches, see Fig. 4.5. Each element of the list specifies an interaction with one subproblem and provides three pieces of information: an identification of the interacting subproblem; two pointers; and specifiers that tell *mapper* how to access the access the information referenced by the pointers. The pointers refer to two dimensional data structures reserved for storing information in the dependence and influence patches. The specifiers provide index bounds and a memory stride that tell how to access the data structures. For now nothing is said about how interactions lists come to exist.

Mapper uses the interactions list to guide its actions; the *influence* action uses the influence patch information and the *dependence action*, the dependence patch information. Each action calls a different subroutine that the user passes to *mapper* via *mapper's* parameter list. Two subroutines, called *pack* and *unpack* are supplied; they convert between the message buffer representation and the

representation of the application's data structures. The influence and dependence actions execute concurrently and separate parallel processes execute for each interactions list element. *Influence* first calls *pack* to collect computational elements in each influence patch and to copy them into a memory buffer; then it sends the packed data to the interacting subproblem. *Dependence* receives a message, examines the id of the processor that sent the message, looks up the id in the interactions list for the appropriate dependence patch descriptor, and then calls on *unpack* to copy the data in the buffer into the region of memory reserved for the appropriate dependence patch. Each processor exits *mapper* when it finishes executing both the dependence and influence actions. This implements a synchronization mechanism that ensures that data always gets mapped in a consistent state across all the processors. When done mapping, a processor knows about all the computational elements in its dependence region that interact with its own.

So far little has been said about the *pack* and *unpack* routines passed to *mapper*. These routines are necessary because *mapper* knows how to deal with one dimensional data structures only – message buffers. Since the internal behavior of these routines is irrelevant to *mapper* they will not yet be discussed. These routines were not difficult to write for the MLC and were small in comparison to *mapper*.

A *mapper* utility has been introduced that handles communication in a private memory execution model. *Mapper* has two attributes that together contribute to the writing of simpler code, in the local part of the computation: (1) it hides the details of how processors communicate from the programmer; (2) it doesn't need to know about how the application's data structures are represented. These attri-

butes are attractive because application-dependent code and system-dependent code need never become intertwined; were the code transported to a new machine, the parts that must change to accommodate a different communication model would be restricted to code the programmer never sees. *Mapper* does, however, impose some restrictions, in order to streamline its operations. However, we found the restrictions to be reasonable in our implementation of the MLC, and believe that they will also be reasonable for other calculations, too.

5. iPSC Implementation

To evaluate our load balancing library we implemented it along with the MLC on the Intel Personal Scientific Computer (iPSC), a hypercube-type multiprocessor manufactured by Intel Scientific Computers. We describe the iPSC, then implementations of the MLC and of the dynamic load balancing library.

5.1. The iPSC

The iPSC is scientific multiprocessor inspired by the Caltech cosmic cube [16]. An iPSC system may be configured with 32, 64, or 128 processor nodes, which communicate by sending messages over a hypercube interconnection network. The 32 processor model d5 used here is nominally a 1.1 megaflop machine – each processor runs at about 0.033 megaflops – and delivers about 0.8 megaflops on Gaussian elimination [13]. Each node has 512 kilobytes of local memory of which about 300 kilobytes are accessible to the user. Nodes may communicate with a host processor, having 4 megabytes of memory, but may not communicate with the outside world in any other way. Both host and nodes are designed around the Intel 80286 central processor and 80287 arithmetic co-processor, and run a modified version of

the Intel XENIX-286 operating system. The node's operating system provides a 60 Hz timer; timing measurements may be resolved to about 33 milliseconds, but only on the nodes. Host times must be measured from a processor node.

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 288 kilobytes/sec for the maximum-sized message of 16 kilobytes. For short messages no longer than 1 kilobyte, a 5 millisecond *message startup cost* dominates the message transmission time.

5.2. iPSC Implementation of the Method of Local Corrections

All software was written in FORTRAN 77, compiled using the Intel *ftn286* compiler, and run under version 2.0 of the node operating system. Two programs were written, one for the host and the other for the nodes. The host did all the I/O on behalf of the nodes, such as reading in simulation parameters. It ran the partitioner but all numerical computations ran on the nodes only. All nodes executed the same program.

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. Time integrations were accurate to second order and done with a Runge-Kutta time integration scheme (Heun's method) that does two velocity field evaluations per timestep. In addition to computing local interactions, the MLC also does some finite difference computations, that include a global calculation to solve Poisson's equation. However, most of the finite difference computations, that

will not be discussed in detail, are localized. All finite difference calculations were accurate to fourth order and all arithmetic was done with 8 byte double precision numbers. The program used three major data structures that were duplicated on all the processors. These were: three 42×42 finite difference meshes; three 84×84 2-byte integer arrays used for the bins and to do work estimation; and vortex records, each describing a single vortex. A vortex record consists of 154 bytes of information: 2 real-valued position vectors; 2 real-valued velocity vectors; real-valued vortex *strength*, that is similar to an electrostatic charge; 5 complex-valued interpolation coefficients; and a 2 byte pointer used to link vortices into the bins. To economize the iPSC's scarce memory, a short form of the vortex record was also used; it consisted of only 26 bytes of information: 1 position vector, strength and a link. The major data structures accounted for total of 162 kilobytes of storage. The remaining 140 kilobytes of node memory contained mostly code.

5.3. Partitioner

The partitioner implementation was straightforward since it invoked no communication system calls. Use of *partitioner* involves inserting the following call immediately preceding the first velocity field evaluation of a timestep:

partitioner(P,m,n,workEst,ldw,actualP,parts)

where all arguments are integers. Since the vortices don't move very quickly there is little to be gained by partitioning more often than every other velocity field evaluation. P is the number of subproblems requested and *actualP* returns the actual number of subproblems rendered; $actualP \leq P$ with the inequality holding only if *partitioner* is unable to render all P subproblems. *WorkEst* is a 2-D integer

array that holds the work estimate for each bin of vortices; *ldw* is the leading dimension of the array, of which an $m \times n$ subarray gets used by *partitioner*. *Parts* is the table of the *actualP* subproblems; each entry consists of 4 integers giving the origin and shape of a single subproblem in the index space of *workEst*. *Partitioner* has some freedom in making cuts in either of two directions. Normally it tries a cut in a single direction only, but alternates the direction of the cut from one level of recursion to the next. However, it rejects any cut that would leave one region with no work to do and in this case may be unable to alternate the direction of the cut, if it can cut at all. A strategy that chose the better of the two possible cuts, regardless of whether one had to be rejected, has not yet been tried.

5.4. Mapper Implementation

On the iPSC mapping involves two major steps: (1) *pack* vortices lying in influence patch bins into message-buffers and send to the appropriate processor; (2) receive incoming vortices and *unpack* them into dependence patch bins. The steps are interleaved to roughly balance incoming and outgoing data traffic; this helps avoid transient deadlocks that won't lock up the code permanently but which could slow it down. Since iPSC allows the nodes to be multiprogrammed, the two steps could have been executed as concurrent processes, but this wasn't tried.

Mapper packs and unpacks vortices *en masse* to amortize the startup cost of sending messages over the several vortices contained in a message buffer. In general, the optimal buffer-size is a function of the startup cost, which is an architecture-dependent parameter. The message buffers used on the iPSC were 6720 bytes long.

In addition to processing external influence of vortices, *mapper* also found use in the *sort* routine used to sort vortices into bins. During *sort* it becomes necessary to migrate vortices between processors that find themselves under the aegis of a different processor than before as the result of their own motion, repartitioning, or both, see Fig. 5.1.

Use of *mapper* involves calling the following subroutine from the user-code:

mapper(interactionsList,pack,unpack,inBuffer,outBuffer,bufLen)

where interactionsList has been previously computed by an interactions list generator, *pack* and *pack* are external subroutines, and *inBuffer* and *outBuffer* are message buffers *bufLen* bytes long. *Pack* takes *outBuffer*, *bufLen* and an *interactionsList* element as arguments and puts the information it collects into *outBuffer*. *Unpack* takes *inBuffer*, *bufLen*, and an *interactionsList* element as arguments and doesn't write into any of the parameters.

Pack copies vortices from the bins selected by the influence patch description passed to it into a contiguous array. *Pack* expects the patch description to be an ordered pair of the form (*origin*,*shape*) where "origin" gives the row and column address of the patch's origin in the bin index space and "shape" the number of rows and columns comprising the patch. Thus, the ordered pairs ((2,1),(3,1)), ((2,2),(1,1)), and ((2,1),(1,3)) define the range of bin-indices covered by the influence patches shown in Fig. 4.4, going clockwise around the subproblem. Each non-empty bin in a patch points to a list of vortices lying in the region of space covered by the bin, see Fig. 5.2. Each element of the list contains two fields: a *data* field, containing such information as the vortex's position, and a *link* field, a pointer to

the next vortex in the list. A distinguished pointer, the *null pointer*, terminates the list. *Pack* traverses the linked list pointed to by each bin in the patch and copies data found in the list into an array, see Fig. 5.2. *Unpack* reverses the packing process. It copies information from a message buffer into freshly allocated vortex list cells, threading each new list cell onto the head of the appropriate bin's linked list. The correct bin address for a vortex is determined by examining the position vector that is a part of the vortex record's data field.

An *interactions list generator* produces the interactions list that *mapper* uses to guide the actions of *pack* and *unpack*. The load balancing library supplies some default generators, though the user is free to write his own. A generator is parameterized by the *interaction radius* and by the table of subproblems returned by *partitioner*. It works by doing simple geometric operations on the table of subproblems. Influence patches, for instance, lie inside the subproblem, at the intersection of the subproblem with the subproblems of other processors extended by the interaction radius, as shown in Fig. 5.3. For the MLC, interaction radius equals the correction radius C. For finite difference calculations, the interaction radius would be the radius of the finite difference stencil.

6. Evaluation

6.1. Experimentation and Evaluation Methodology

All results were obtained from runs involving the "two entraining patch problem" of Fig 2.4: 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

used in the runs were made to vary linearly with P, the number of processors; this is consistent with the philosophy that ever larger problems may be handled as computational resources increase. Each processor was initially assigned about 100 vortices independent of the size of the problem, though as a result of the motion of the particles 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 only; 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. Problems should have at least several hundred vortices and preferably several thousand.

A calculation involving 3180 vortices ran for about 5.4 *hours* on an iPSC with 32 processors. All runs lasted 64 timesteps and Δt, the timestep, was 0.05. A 36×36 finite difference grid was used in the finite difference computations with spacing $h = 1/30$. The correction radius C was $2h$. The bin spacing was 0.5h and vortices, therefore, were sorted into a 72×72 array of bins.

Each processor recorded its own timing information and at the end of the calculation sent its measurements to the host, which then wrote the data to a file. The file contained the times spent in the various phases that made up each velocity field evaluation of each time step of the calculation. A separate program reduced the raw data to determine the times spent communicating, doing localized computation, solving Poisson's equation, doing task partitioning and so on. The data reduction program also computed parallel efficiency, a familiar performance metric for evaluating multiprocessor implementations. Kuck [12, p. 33] defines η_p as the efficiency with p processors:

$$\eta_P = \frac{T_1/P}{T_P} \tag{6.1}$$

where T_P is the time to complete on P processors. T_1 is the time taken on a uniprocessor; in this special case $P = 1$, various overheads, such as communication, that would be incurred on a multiprocessor, are non-existent. By definition $\eta_1 = 1$. Because the size of the problem scales with the number of processors, T_1 cannot be measured directly. But since *partitioner* conserves the total amount of work that would be done in a uniprocessor computation, T_1 can be reasonably approximated by summing up the completion times for all P processors.

η_P simultaneously measures three factors that degrade efficiency: (1) communication overhead; (2) non-parallelizable computation that must run on only one processor; (3) load imbalances. Another important measure is $\tilde{\eta}_P$, the *maximum theoretical efficiency*. It measures the effect of the second factor only. This measure is important because it tells us the best that our load balancing utilities could do were communication instantaneous and loads perfectly balanced. It therefore provides a figure of merit for evaluating our strategy. Unlike η_P, $\tilde{\eta}_P$ considers task partitioning to be work that gets done on a uniprocessor; $\widetilde{T}_1$ includes the time spent partitioning. Thus

$$\tilde{\eta}_P = \frac{\widetilde{T}_1/P}{\widetilde{T}_P} \tag{6.2}$$

where

$$\widetilde{T}_1 = T_1 + T_{partition}\ , \tag{6.3}$$

$T_{partition}$ is the time spent partitioning, and $\widetilde{T}_P$ is defined as:

$$\widetilde{T}_P = T_1/P + T_{partition}, \tag{6.4}$$

i.e. T_1 ideally divides into P equal parts. Thus $\widetilde{T}_P$ places a lower bound on the

time taken to run the calculation using our load balancer.

6.2. Computational Results

Consider the top and bottom curves of Fig. 6.1 that plot η_P and $\tilde{\eta}_P$ vs. P. The 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, this non-parallelizable work grows slowly with the number of processors. The observed efficiency, in contrast, decreases more sharply than the maximum theoretical efficiency, as the number of processors increase. It ranges from 89% for 4 processors to 69% for 32 processors. This means that the calculations would run at worst only about 40% ($1/\eta_P$) slower than they would under ideal conditions. Load imbalances are the major difficulty here; communication and other overhead are comparatively benign. This can be seen by looking the idealized efficiency successively degraded by the various sources of overhead, plotted in Fig. 6.1. The two new curves divide the gap between the idealized and observed efficiency curves 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 imbalances. It can be seen that the cost of load imbalances increases much more rapidly with the number of processors than do the costs of the other factors. This happens because the total communication overhead increases gently with the number of processors and the solver overhead decreases gently, while load imbalances increase sharply.

Exclusive of the Poisson solver, communication overhead ranged from 3.1% on 4 processors to 5.5% on 32. Communication serves two purposes: (1) to manipulate

the finite difference grids used by the solver and (2) to do mapping. The second activity was much less expensive than the first; *mapper* incurred communication overhead that ranged from 0.2% to 1.9% 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. *Partitioner* overhead was never more than 1.6%; the recursive bisection algorithm is fast, doing only integer arithmetic and running in time that is proportional to the logarithm of the number of processors. So the total time spent in the load balancing utilities is never greater than 3.5%.

6.2.1. Discussion

Processor idleness has been shown to be the major performance bottleneck for the Method of Local Corrections running on the iPSC. The load balancing utilities incur negligible overhead. The high message latency time of the iPSC appears to have very little impact on the running time of the calculation; on 32 processors communication accounts for no more than about 10% of total running time of the computation. This is because communication phases are brief and occur relatively infrequently between the much longer computation phases.

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 10% of what was observed. The 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 Fig. 6.2, 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.

Using a larger number of bins to keep track of the particles appears to be a far simpler way of improving the workload balance than changing the geometry of the partitions. Making the bins more numerous decreases the granularity of the work represented by a bin. In the case of the C.MMP multiprocessor [14] reducing task granularity was shown to improve efficiency, as long as the extra time spent processing the increased number of pieces of work was reasonable. So far, an optimum tradeoff between efficiency and the number of bins has not yet been determined, although simulations have shown that for best results the number of the bins should be increased when the number of processors increases. A scheme that took advantage of the sparsity of the bins would substantially increase the number of non-empty bins that could be accommodated with a given amount of memory. So far such a scheme hasn't yet been tried; the storage overhead was not a major issue on the model *d5* iPSC with 32 processors. But it would be an issue for the model *d7* with 128 processors, as well as for any machine with several hundred or a few thousand processors.

7. Conclusions

A versatile low-overhead load balancing strategy has been presented. It has been implemented on Intel's iPSC, a message passing multiprocessor, and parallel speedups of 22 have been achieved on 32 processors running a particle-type calculation. The strategy uses a small set of utilities that may be implemented in the

form of a subroutine library. The library makes sense for calculations meeting the following three criteria such that:

(1) Local interactions predominate.
(2) The computational domain partitions naturally into a fine lattice of box-like subproblems.
(3) The time to complete each subproblem can be computed in a simple way.

Thus it should apply not only to particle methods but also to finite difference methods such as adaptive mesh refinement for treating hyperbolic partial differential equations. Our load balancing utilities generalize to computational domains of arbitrary dimensionality. They apply to the localized part of the computation only; system-dependent calls that handle interprocessor communication should not appear in that part of the application code. The user must provide some application-dependent code but the code should not be difficult to write. Such code would be similar to what is provided, for instance, in library packages that solve elliptic partial differential equations or that do zero-finding.

Because the load balancing library supports a local memory model, it may arrange data to exploit local memory on shared-memory architectures. Its use can therefore avoid costly memory access conflicts by causing a high percentage of accesses to go to local memory. The use of the local memory model comes at the cost of redundantly storing some information but the storage overhead will be be reasonable for localized computations. This redundant storage scheme can also help reduce memory contention on shared-memory architectures that do not provide local memory. Memory accesses that would go to the same location in shared memory instead go different copies of the same piece of information.

The proposed load balancing library may fail if workloads cannot be estimated accurately, or if the estimate is expensive to compute. The loads may then be poorly balanced, or the estimation phase may be too expensive to justify balancing the workloads. Clearly the success of such a strategy is problem-dependent. Because of the need to store information redundantly, the storage overhead of our load balancer would be unacceptable for some calculations with strong long-range coupling. However, 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 a logarithmic potential governs the motion of the vortices; distant interactions can be computed by means of fast Poisson solver and only a small fraction of all interactions need to be computed directly.

Because our load balancer can schedule tasks statically it can avoid the high overhead associated with dynamic scheduling. Dynamic scheduling can become necessary on multiprogrammed systems, which aren't considered here. It also becomes necessary if the tasks don't have known completion times or if the tasks are indivisible (See Saltz's dissertation [15] for a discussion of how to schedule indivisible tasks on multiprogrammed multiprocessors). Under our strategy tasks have known completion times and are assumed to be divisible. This allows the balancer to split them up with reasonable fairness and then to fix the assignments of work to processors. Since the balance of work drifts slowly over time work can be reshuffled at a convenient stopping place in the program, such as between timesteps, rather than while the computation runs. The *pack* and *unpack* routines

that the programmer passes to *mapper* handle all the details of task subdivision. Static scheduling is generally preferable to dynamic scheduling because processors can access a static, local data structure to get new work assignments or to obtain information about interdependent subproblems. This is extremely attractive on private memory architectures that have high message startup costs but can also reduce memory contention on shared memory architectures by reducing memory accesses to shared information. Finally, use of a static scheduler on private memory architectures avoids the costly reshuffling of tasks among the processors that would occur were dynamic scheduling employed.

Our strategy also makes no attempt to assign the more heavily communicating tasks in a way that best utilizes the communication links with the greatest bandwidth. The need to place heavily communicating tasks preferentially would arise in calculations that do considerable amounts of communication relative to computation, which are beyond the scope of this paper. See Bokhari's paper [4] on the *processor mapping problem* for the details.

Although our load balancing strategy isn't automatic, we believe that it achieves a better balance than the automatic schemes, such as dataflow, between efficiency and ease-of-use:

(1) It may be implemented as a subroutine library using existing programming languages and existing multiprocessor architectures, including those with vector-mode capabilities.

(2) It doesn't impose on the programmer. It requires only a few routines that depend solely on the application.

(3) It incurs modest overhead.

The library can go a long way to making multiprocessors an attractive means of solving some very difficult scientific problems.

8. Acknowledgments

I gratefully acknowledge the persistent encouragement of my advisor, W. Kahan, and of Phil Colella, Chris Anderson, and Cleve Moler. Chris provided an initial version of the local corrections code and Cleve a parallel Poisson solver. I also wish to extend my thanks to Professors Paul Concus and Alexandre Chorin for their support of this work. Professor Alberto Sangiovanni-Vincentelli generously provided access to the iPSC at Berkeley. Many thanks go to Mark Dippé for suggesting a subdivisive strategy at an opportune moment.

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; 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.

9. References

1. C. R. Anderson, "A Method of Local Corrections for Computing the Velocity Field Due to a Distribution of Vortex Blobs, *in press*, 1985.
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 PDES Across Multiprocessors, Submitted for publication to IEEE Trans. Comput., 1985.
4. S. Bokhari, "On the Mapping Problem," *IEEE Trans. Comput. C-30*,3 (1981).
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. 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.
8. M. E. Dippé and J. A. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," in *SIGGRAPH '84 Conference Proceedings*, ACM, Minneapolis, July 1984, pp. 149-158.
9. 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.
10. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*, McGraw-Hill, 1981.

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

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

13. 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.

14. 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.

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

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

FIGURES

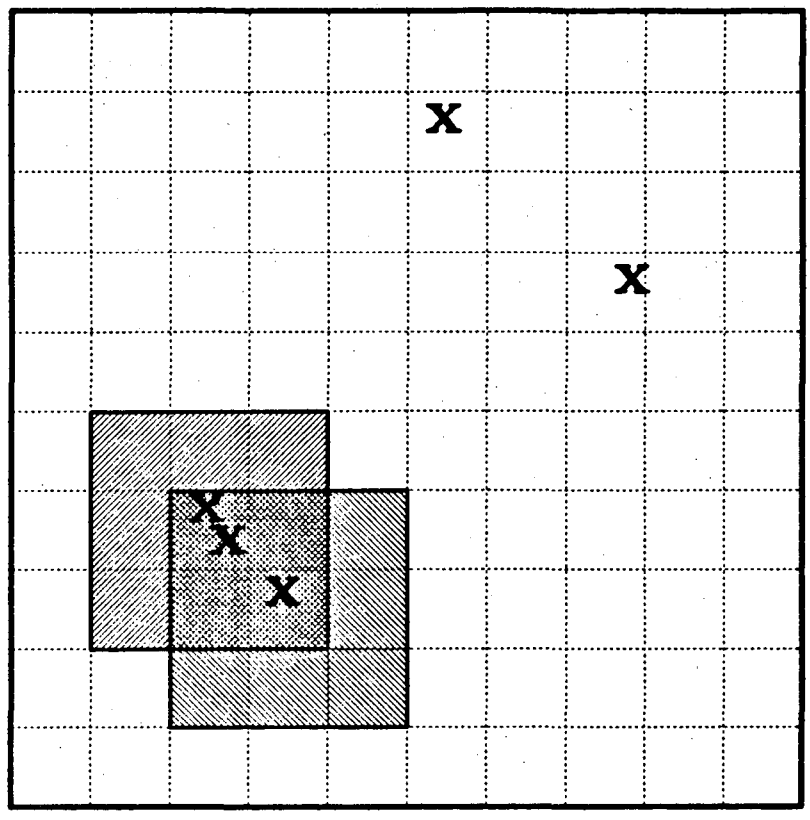


Fig. 2.1.
Vortices, shown as x's, get sorted into bins, demarcated here by hyphenated lines. Associated with each bin is a pointer to a list of votices that lie within the bin's region of space. In practice, the bins are much smaller and more numerous than shown here. Each shaded region designates the 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.

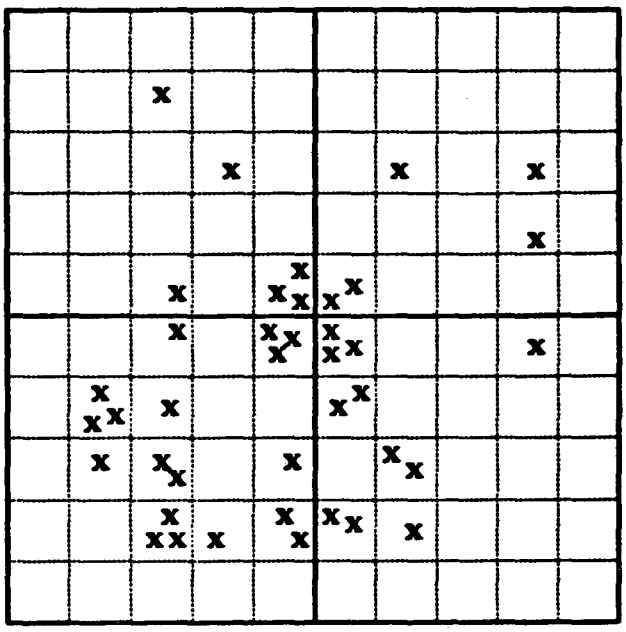


Fig. 2.2.
Partitioning the computational domain among four processors. Each processor computes on its assigned region of space only.

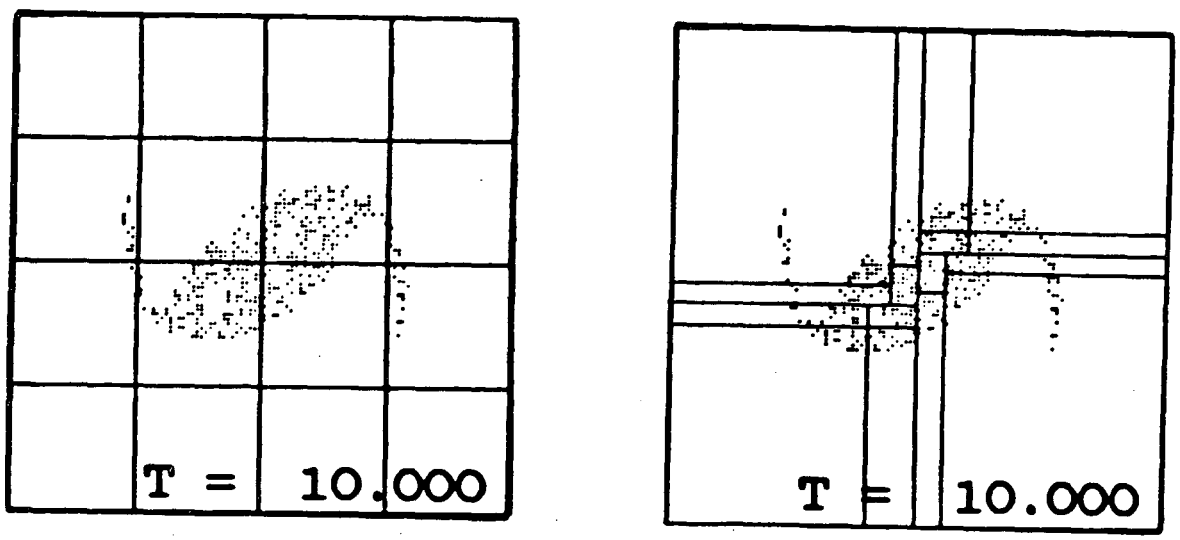


Fig. 2.3.
Partitions with (a) equal areas and (b) amounts of equal work Each dot represents a vortex and also a unit of work The calculation began with 2 patches of vorticity of radius 0.120 with centers separated by 0.25. Each patch had 732 vortices, placed evenly on lattice points spaced 7.5×10^{-3} units apart.

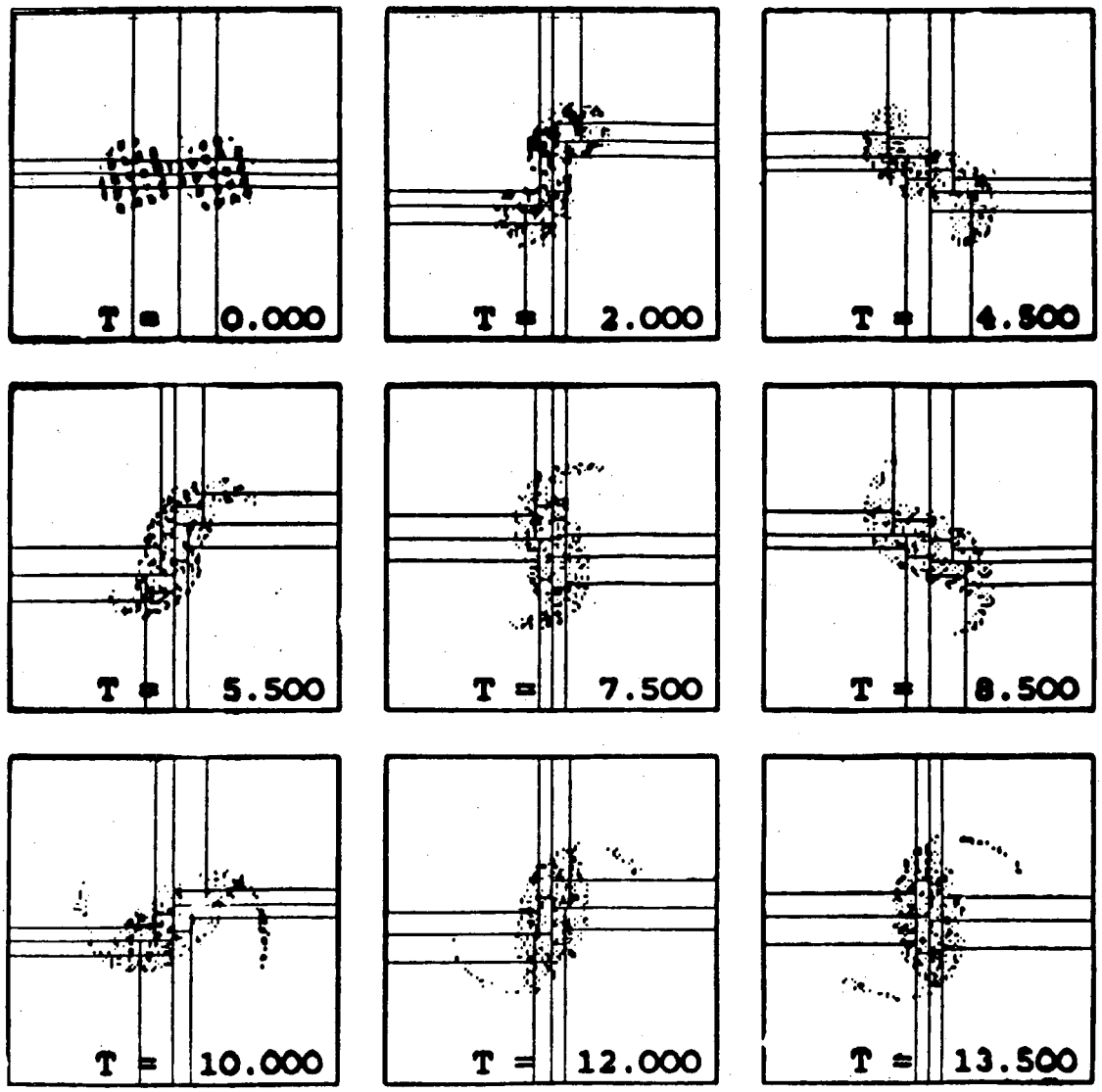


Fig. 2.4.
The distribution of 1464 vortices changes with time, so the work must be periodically repartitioned. This series of snapshots was taken from the same calculation used to produce Fig. 2.3. The calculation consumed 11 minutes of CPU time on just one processor of a Cray X-MP/22.

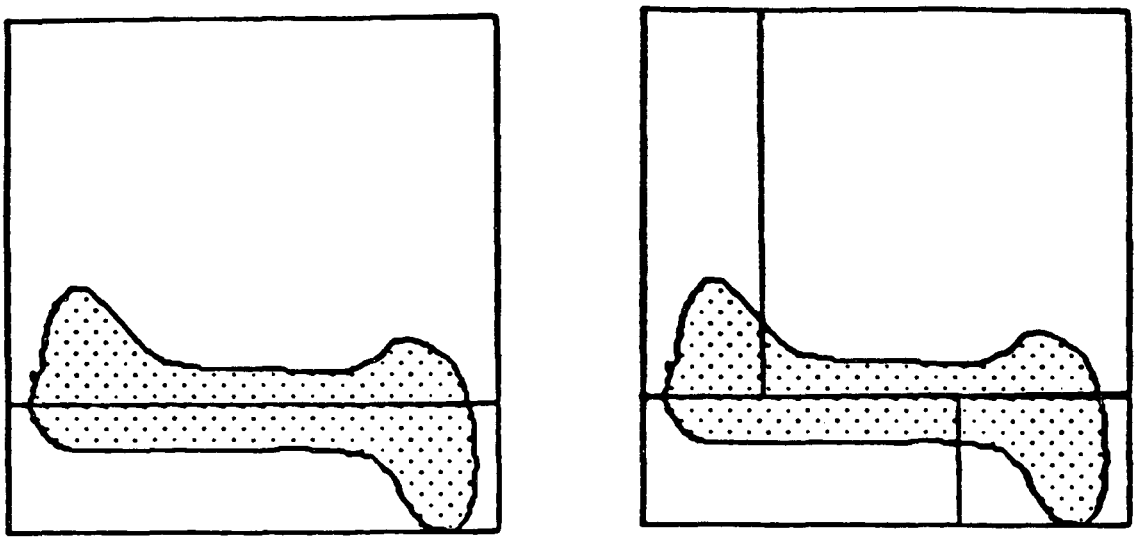


Fig. 3.1.
Recursive bisection.

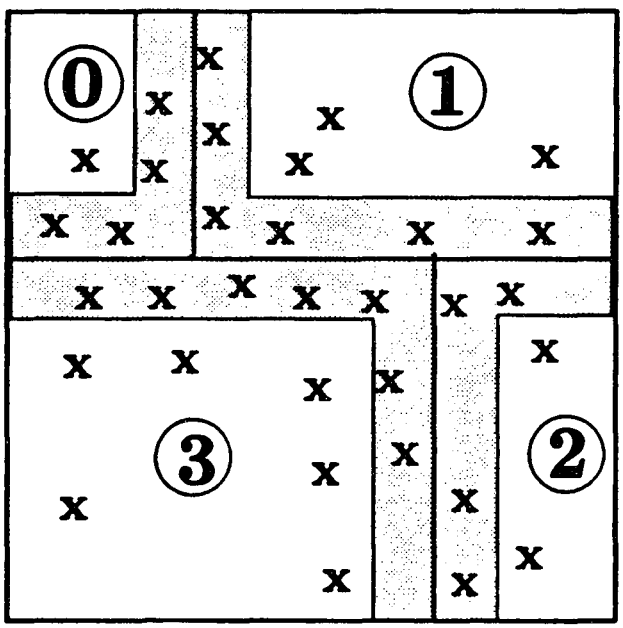


Fig. 4.1
Some of the interactions computed on vortices in the shaded regions involve vortices assigned to different processors.

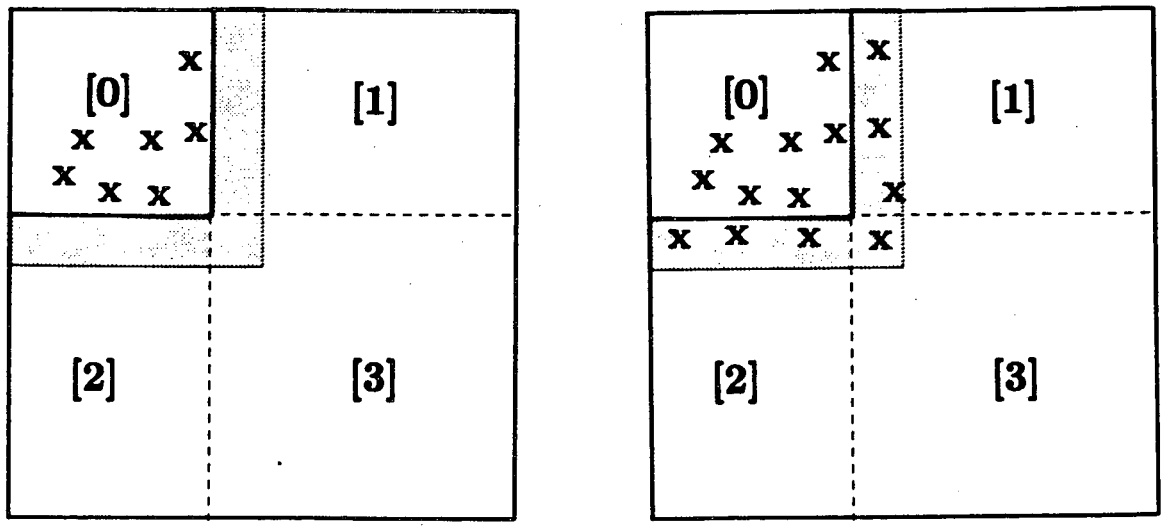


Fig. 4.2
Only processor 0's private bins are shown. (a) Initially vortices lie only in bins owned by the processor. However, some interactions involve vortices owned by other processors so (b) *mapper* fills in the processor's dependence region with those vortices.

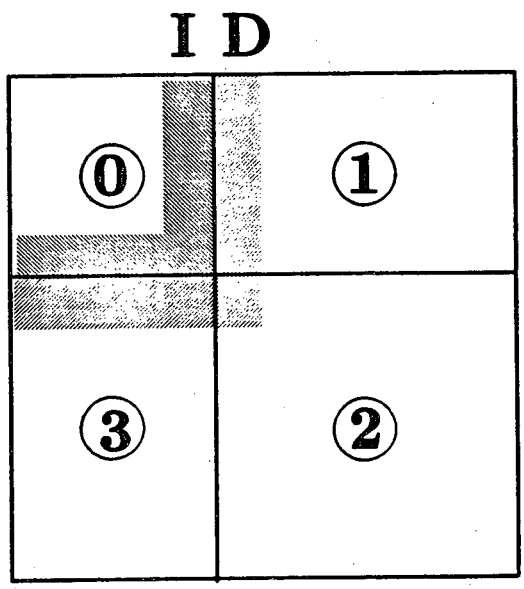


Fig. 4.3
Processor 0's (I) influence region and (D) dependence region. The other processors view these regions as parts of their dependence and influence regions, respectively.

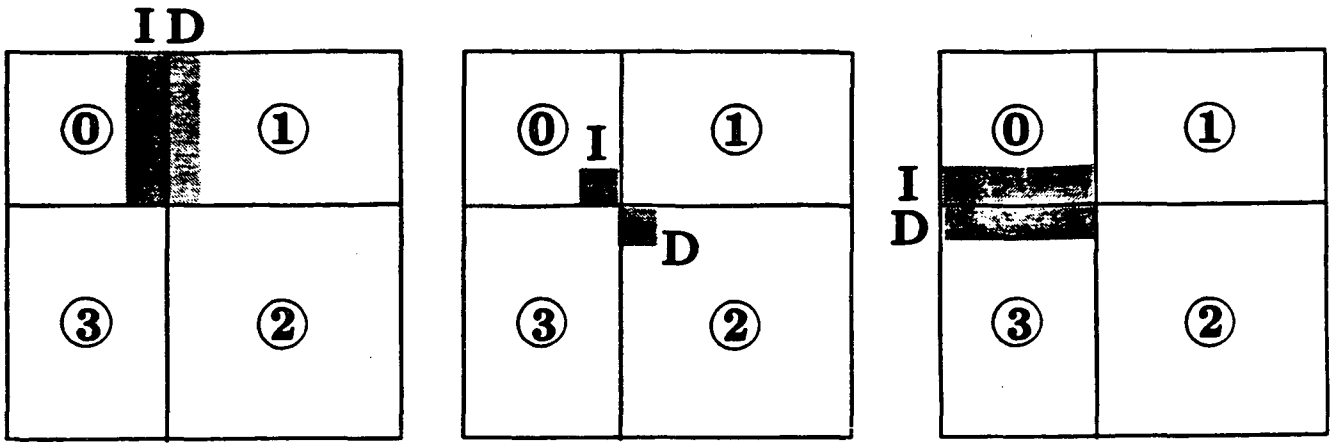


Fig. 4.4
Procesors 0's (I) influence patches and (D) dependence patches.

Influence Region		Dependence Region		Interacting Processor #
Origin	Shape	Origin	Shape	
(1,2)	3 × 1	(1,3)	3 × 1	1
(2,2)	1 × 1	(3,3)	1 × 1	3
(2,1)	1 × 3	(3,1)	1 × 3	2

Fig. 4.5
Processor 0's interactions list describes the interactions of Fig. 4.4, giving both the shape and location of the interacting regions of space and the external processor involved in each interaction.

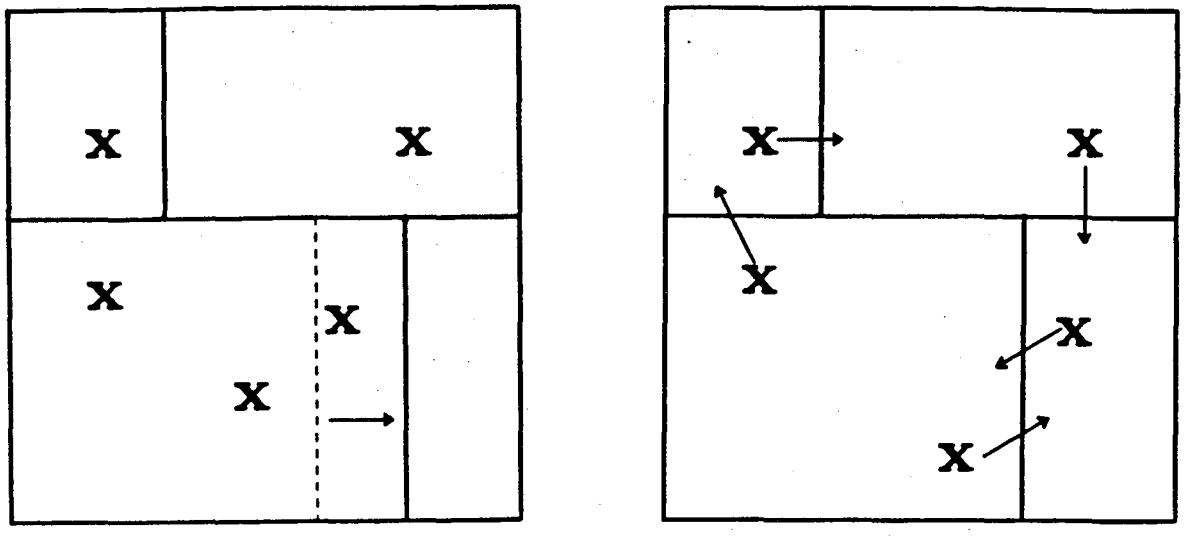


Fig. 5.1
Vortices must be migrated when (a) boundaries slide past vortices, (b) vortices slide past boundaries, or both.

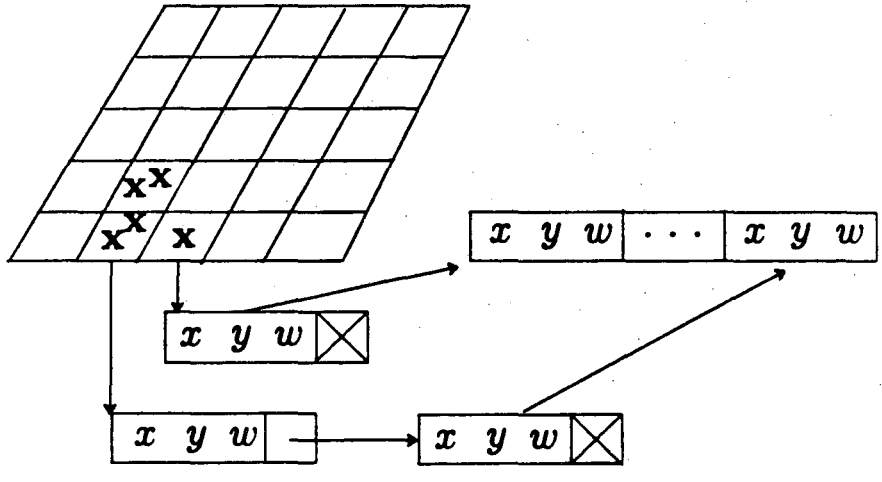


Fig. 5.2
Vortices in the unpacked form, linked into bins, and in the packed form, stored contiguously in an array.

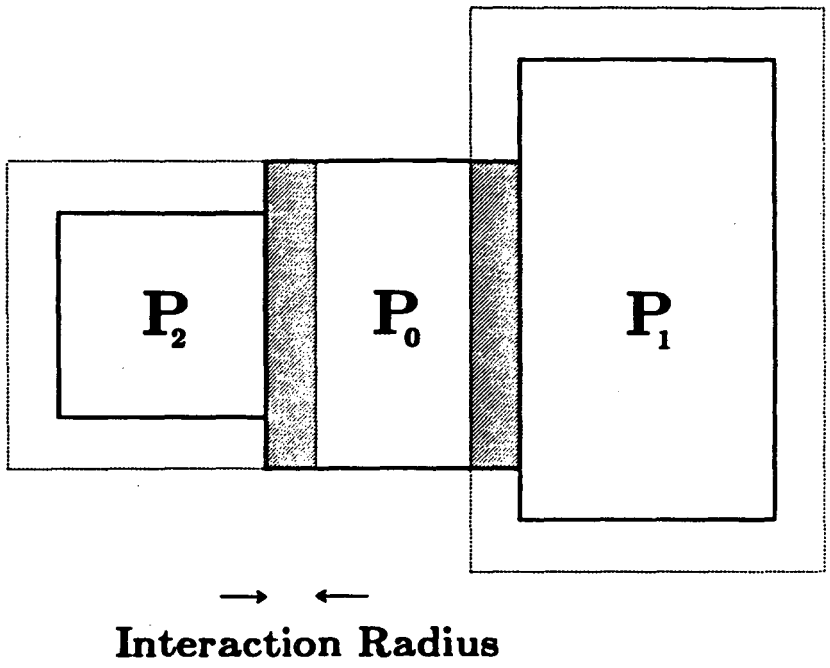


Fig. 5.3
Determining influence patches for partition 0. Each patch lies in the intersection of the partition with all interacting partitions extended by the interaction radius.

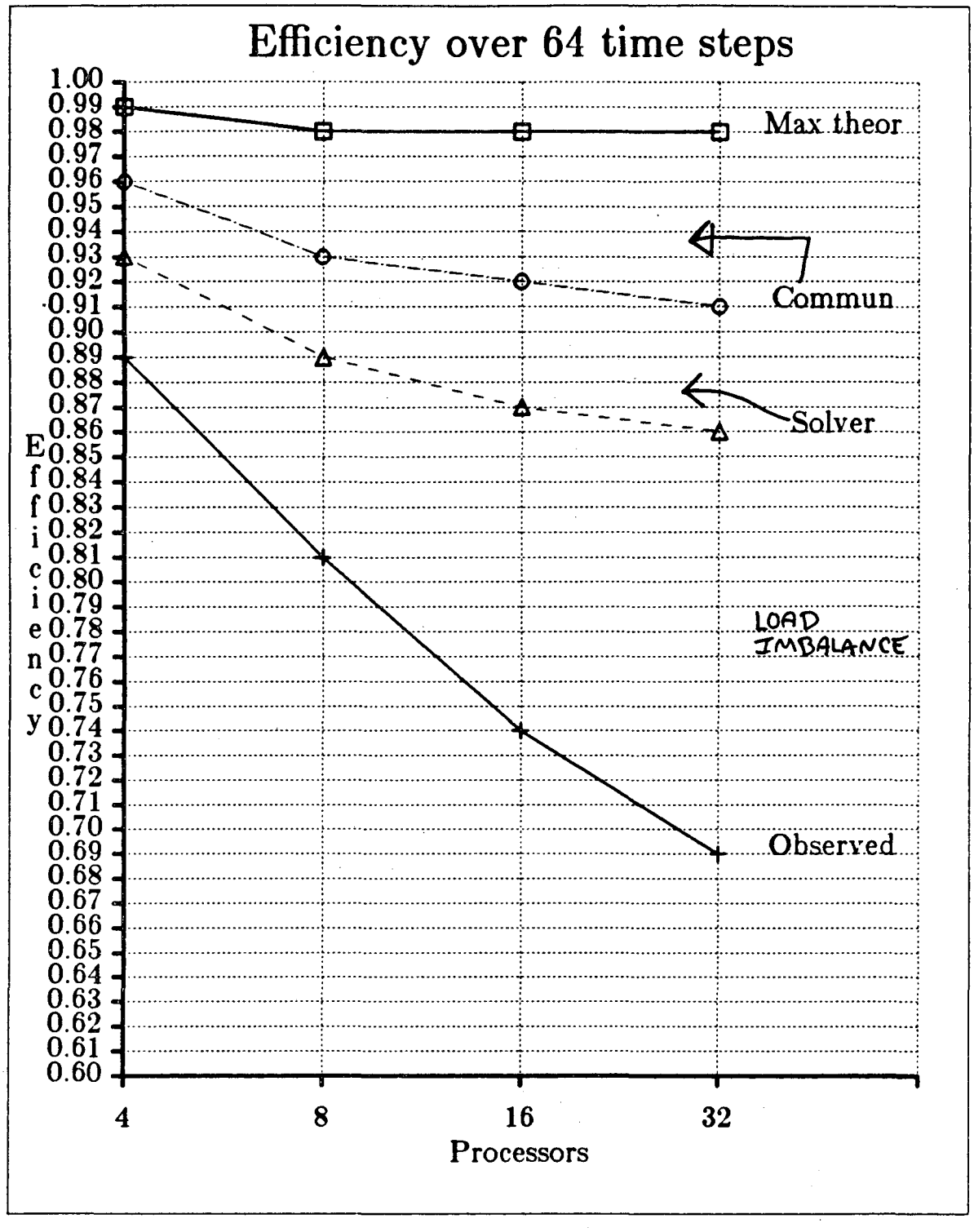


Fig. 6.1

Efficiency vs. the number of Processors. The top curve gives the maximum theoretical efficiency that would be attained under idealized conditions. The bottom curve gives the observed efficiency. The two curves in the middle divide the efficiency losses represented by the gap between the upper and lower curves into 3 regions that correspond to losses due to communication, the Poisson Solver, and and to load imbalances, respectively.

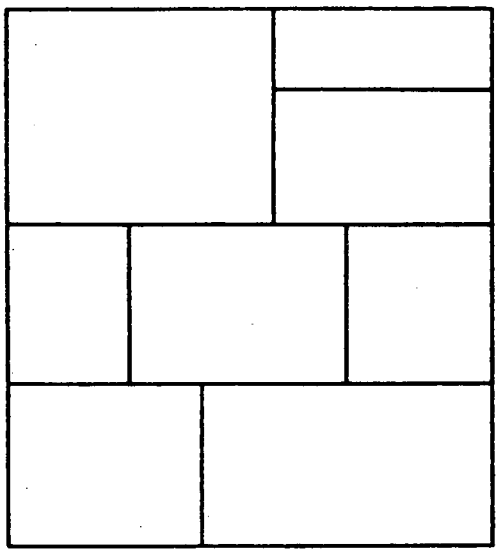


Fig. 6.2
The recursive bisection strategy cannot render this partitioning

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720