Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Physics Division

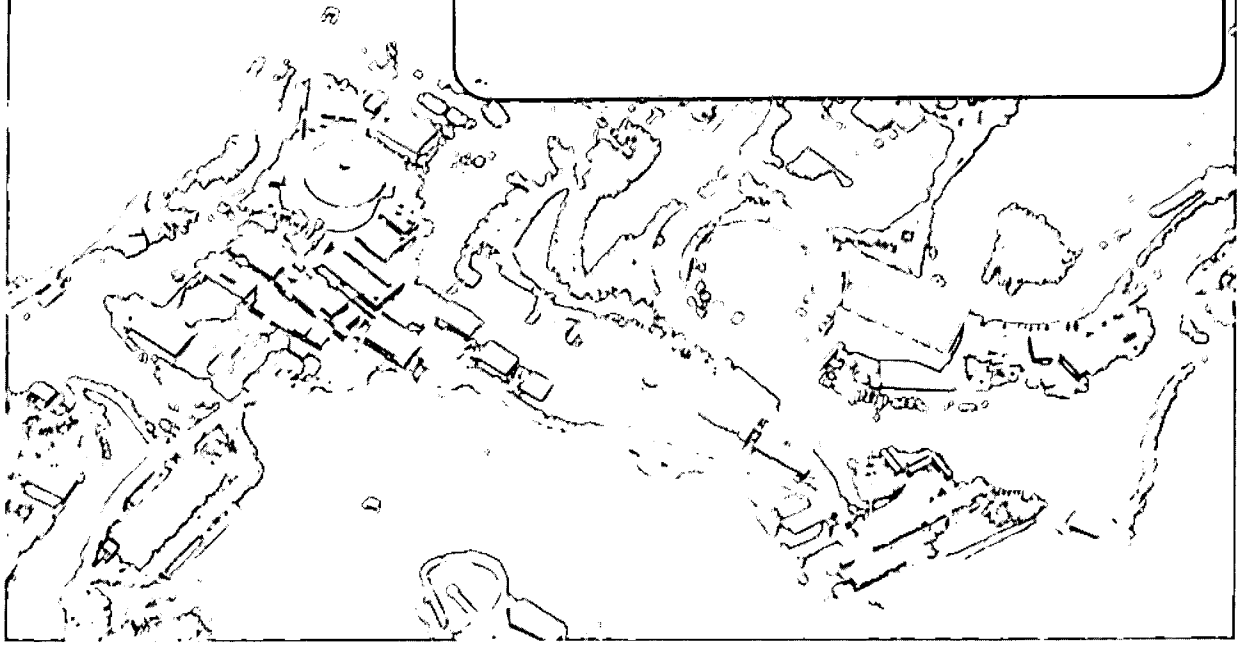
Mathematics Department

Presented at the 3rd SIAM Conference on
Parallel Processing for Scientific Computing,
Los Angeles, CA, December 1–4, 1987

Programming Abstractions for Run-Time Partitioning of Scientific Continuum Calculations Running on Multiprocessors

S.B. Baden

January 1988



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

DISCLAIMER

LBL-24643

PROGRAMMING ABSTRACTIONS FOR RUN-TIME PARTITIONING OF SCIENTIFIC CONTINUUM CALCULATIONS RUNNING ON MULTIPROCESSORS*

Scott B. Baden

Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

January 1988

To appear in: *Proc. Third SIAM Conference on Parallel Processing for Scientific Computing*, Dec. 1-4, 1987, Los Angeles, Calif.

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract DE-AC03-76SF00098; a California Fellowship in Microelectronics; Intel Scientific Computers; and Cray Research Inc.

Programming Abstractions for Run-Time Partitioning of Scientific Continuum Calculations Running on Multiprocessors

*Scott B. Baden**

Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Abstract. I will discuss a set of software abstractions for implementing various math-physics calculations on a team of processors. I tried out the abstractions on Anderson's Method of Local Corrections, a type of vortex method for computational fluid dynamics. I ran experiments on 32 processors of the Intel iPSC – a message-passing hypercube architecture – and on 4 processors of a Cray X-MP – a shared-memory vector architecture – and achieved good parallel speedups of 24 and 3.6, respectively. The abstractions 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.

1. Introduction

A major application for multiprocessors is in obtaining solutions to partial differential equations arising out of various areas of science and engineering. A major outstanding difficulty in using them is how to construct robust software that can run efficiently on diverse systems without having to go through major changes in programming. This is particularly troublesome for calculations that apply computational effort non-uniformly over space according to time-dependent phenomena, and which must therefore be dynamically partitioned. I will discuss a set of abstractions that can hide many of the details entailed in dynamically partitioning and coordinating a computation among a team of processors and that can improve the robustness of software with respect to those activities.

The abstractions apply to the important class of calculations that spend most of their time in *spatially localized* computation in which two data points communicate far more information with respect to the computation done on them when they are close together than when they are far apart. Consider, for example, the particle-particle particle-mesh solution to the N-body problem. Such a calculation arises in problems in computational fluid dynamics, plasma physics, and particle physics; it entails following a set of particles that move under mutual interaction, congregating and dispersing unpredictably with time (see Figure 2). The particles move under the influence of a logarithmic potential, which computation divides into two parts: a local part that does roughly 90% of the computational work when the problem is large, and a relatively inexpensive global part whose data dependencies are not localized. The cost of computing the local part of the potential is a position- and time-dependent function of the local density of particles.

*This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract DE-AC03-76SF00098; a California Fellowship in Microelectronics; Intel Scientific Computers; and Cray Research Inc.

A simple way of applying a team of processors to spatially localized particle methods is to partition the domain into rectangular regions and assign the computation and data associated with each region to a processor. There are many ways to partition the domain, two of which are shown in Figure 1. A uniform partitioning, in which all partitions have the same area, is the most straightforward. Such a partitioning, however, utilizes only a small fraction of the total power of the processing team because work is distributed non-uniformly in space, as shown in Figure 1a. A better way is to partition adaptively into somewhat irregularly-sized regions that all complete in roughly the same time, as shown in Figure 1b. Such a strategy compensates for the uneven distribution of work, and can substantially accelerate the computation. Of course, the partitioning must be periodically recomputed, as shown in Figure 2, or otherwise the workloads would gradually drift out of balance as the particles redistribute themselves. Some processors would become overloaded, while others would sit idle waiting; the cost of the computation would steadily increase with time, as shown in Figure 3.

The decision to change the work assignments dynamically, rather than to assign work statically, can substantially complicate the user's software. The trouble is that the best way to handle the communication and the bookeeping that come as a side effect of shuffling work among the processors can depend on various overhead costs – such as memory latency or message startup time – that generally vary from system to system. Thus, the code required to effectively parallelize a calculation on a shared-memory multiprocessor differs substantially from that required to run on a message-passing architecture, and code can vary even among members of one family of architecture. To facilitate in the construction of robust software, I propose that the user program a generic multiprocessor whose partitioning and coordinating operations have the same semantics regardless of where implemented. I will discuss a particular generic multiprocessor called "genMP." GenMP can help desensitize substantial portions of the user's software from a change in various system parameters such as communication or memory latency, numbers of processors, processor interconnection structure, and the semantics of system library calls that handle various aspects of concurrency. GenMP isn't universal, however, and applies to localized computation only; the user will have to parallelize any non-localized computation himself – though separately from the local part – according to the particular architecture in use. The question of how best to parallelize non-localized computation is beyond the scope of this paper, and I consider only spatially localized computation.

2. The GenMP Abstractions

GenMP can be implemented by a layer of software on most any traditional multiprocessor system. It provides a set of run-time utilities that the user will invoke from his code. GenMP assumes a particular style of localized computation, a lattice model computation, in which the calculation maps onto a regular lattice of boxes, the work lattice, subdividing the domain. The computation updates the state of each box as a function of the previous state of only those bins within a given distance C, the local interaction distance. (In contrast, the data dependencies for the updates done in the global part of the computation are not constrained to be localized.) The cost of updating a bin generally depends on the state of the surrounding bins and can be reasonably estimated with an inexpensive auxiliary computation.

The lattice model of computation assumed by genMP is a reasonable one for a variety of computations in science and engineering, and for this reason I believe that it will prove useful for a diversity of applications such as:

- Finite difference calculations that use a fixed rectangular mesh trivially fit the model, as do methods such as Adaptive Mesh Refinement (AMR) [3] that employ dynamic grids.
- Localizable Particle methods such as Particle-Particle Particle-Mesh (PPPM) [11], and Rokhlin and Greengard's fast multipole method [10] are naturally organized around a lattice, and they spend the majority of their time computing direct interactions between nearby particles or doing other localized computation.
- Finite element methods may also be mapped onto a lattice [14], and divide naturally into localized and non-localized computation.
- Ray tracing for computer graphics may also be organized around a lattice, and has a localized communication structure (see Swensen and Dippé [9]).

In the interest of brevity I will consider the abstract problem of how to parallelize a lattice model computation using genMP. The interested reader should consult Baden [2] for the details regarding a specific application. To parallelize a lattice model computation, we subdivide the work lattice into subregions, assign each such subproblem to a unique processor, and let each processor compute on its assigned subproblem in parallel with the others. Ignoring roundoff, results will be independent of the number of processors used. The computation begins with a distinguished task called the "boss." The boss reads in the input data and spawns P additional "worker tasks," where P is chosen by the user. These worker tasks participate in the numerical part of the computation. All execute the same program but each on a different set of data – a single subregion of the work lattice. Each worker executes out of a private address space and communicates with the others through a mechanism to be discussed. There is no shared memory.

Each worker maintains a private copy of its assigned part of the work lattice. It also maintains a copy of a surrounding collection of bins, called an "external interaction region," which augments the task's assigned sublattice (see Figure 4). The task uses this external interaction region to maintain copies of data belonging to other tasks that directly interact with its own; hence, the region is C bins thick, where C is local interaction distance previously discussed. As a consequence of using this distributed storage strategy, no task may access any bins beyond its external interaction region. Furthermore, a task may only indirectly access bins, owned by other tasks, that overlap its external interaction region. For example, when a task modifies a copy of a bin in the external interaction region, then the owner of the "original" won't know that the change was made. Similarly, when a task modifies an original any tasks that possess copies will be unaware of the changes. These changes must eventually be propagated, however, to ensure correctness; at certain points in the calculation each task must suspend computation and communicate with the other tasks in such a way that all bin-copies be consistent with the originals. To this end, all tasks periodically invoke a run-time utility called `lBar`. When a task encounters a call to `lBar` it communicates with all tasks overlapping its external interaction region and returns when it has finished communicating with all of them. This set of interacting tasks acts as a local barrier synchronization mechanism. Each task will generally encounter and leave the local barrier at a different time, according to the amount of work assigned to it. We refer to the barrier as being a local one because generally it involves only a local subset of tasks, rather than the entire set of tasks as in traditional (global) barrier synchronization; the name `lBar` stands for "Local BARrier synchronization." `LBar` is passed it two subroutines as arguments. These perform gather and scatter operations on the local user data structures. For details see [2].

To ensure they share the work evenly, the workers must periodically invoke a run-time utility called `Partitioner`. `Partitioner` readjusts each worker's assignment of bins according to a time-varying "work density mapping," supplied by the user. This mapping comes in the form of an array; each entry estimates the cost of updating one bin of the work lattice. All tasks leave `Partitioner` together and upon return each will be assigned a unique rectangular region of the work lattice. A task determines the set of indices for the bins assigned to it with the aid of querying functions provided as run-time utilities. As a result of calling `Partitioner`, some bins may change owners, and must therefore be transmitted to the correct task. A call to the `lBar` utility can handle the necessary exchanges of data. I chose to implement `Partitioner` with a recursive bisection algorithm similar to that used by Berger and Bokhari [4]. The user, however, is unaware of how `Partitioner` works, and any strategy that was fast and that rendered partitionings with a low surface area to volume ratio would suffice.

3. Computational Results

I evaluate genMP on the Intel Personal Scientific Computer (iPSC), manufactured by Intel Scientific Computers, and on the Cray X-MP, manufactured by Cray Research Inc.. I will show that the performance of either of these systems running genMP can scale reasonably well with the number of processors in use. A detailed description of the iPSC and the Cray X-MP is beyond the scope of this paper; see Baden for a summary of the relevant details, or the manufacturer's manuals [8, 12]. (The pamphlet by S. Chen et al. [5] on the Cray X-MP is a more accessible document than the manufacturer's Hardware Reference Manual.) Table 1 summarizes the relevant characteristics of the two machines.

The application I used as a test problem was a vortex dynamics calculation chosen from fluid dynamics known. This calculation solves the *vorticity-stream function formulation* of Euler's equations for incompressible flow in two dimensions in an infinite domain:

$$\frac{D\omega}{Dt} = 0 \tag{3.1}$$

$$\omega = -\Delta\psi \tag{3.2}$$

$$\mathbf{u} = 0 \text{ at } \mathbf{x} = \infty, \tag{3.3}$$

where $\mathbf{u}(\mathbf{x}(t),t)$ is the velocity of the fluid at position $\mathbf{x}(t)$ at time t; ω is the vorticity, defined as the curl of $\mathbf{u}$; ψ is the stream function; $\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u}\cdot\nabla$ is the material derivative and $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the two-dimensional Laplacian operator. (For an explanation of these equations consult Chorin and Marsden's introductory text on fluid mechanics [7]. Also see Chorin's original paper on the vortex blob method [6], or Leonard's survey of vortex methods [13].) The above equations were solved for an initial vorticity distribution that was constant inside two disks centered about the origin, and zero elsewhere. These disks are referred to as Finite Area Vortices. To discretize the above equations we place a collection of N marker particles, called vortices, on a regular mesh of points, and then compute the path of the vortices over a sequence of timesteps. The following system of ordinary differential equations describes the motion of the vortices:

$$\frac{d}{dt}\mathbf{x}_i(t) = \mathbf{u}_i(t), \; i = 1, \cdots, N \tag{3.4}$$

where $\mathbf{x}_i(t)$ is the position of the ith vortex at time t, $\mathbf{u}_i(t)$ the velocity, and ω_i is its strength, which is like a charge. A PPPM-type algorithm, Anderson's Method of Local Corrections [1], was used to compute the mutually-induced velocities on the RHS of (3.4). When the vortices number in the thousands this method typically spends less than 5% in a Poisson solver – global computation – and most of the remaining time in localized computation. The positions of the vortices were evolved by discretizing (3.4) in time with a second order Runge-Kutta time integration scheme. All software was written in FORTRAN 77. On the iPSC, the code was compiled with FTN286 and run under release 2.1 of the node operating system. On the Cray, the code was compiled with CFT (version 1.14), and run under COS (version 1.16). All arithmetic was done using 8-byte floating point numbers (double precision on the iPSC, single precision on the Cray). Experiments were run with various number of vortices N and processors P. Owing to the differences in processor speed, numbers of processors, and memory capacity, the values of N on the Cray were different from those used on the iPSC.

I use parallel efficiency as the figure-of-merit. Define η_P as the parallel efficiency on P processors:

$$\eta_P = \frac{T_1/P}{T_P}, \tag{3.5}$$

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. By definition $\eta_1 = 1$. Table 2 gives the efficiency and speedup measurements obtained from the iPSC runs, and Table 3, the measurements obtained from the Cray. Efficiency was quite good on both machines. On the iPSC, η_P ranged from 90% with 4 processors to 74% with 32. The efficiency on the Cray was about 90% on 4 processors. Thus, if efficiency were somehow increased to 100%, that would speed up the iPSC computations by at most 35% ($(1-\eta_P^{-1})\times 100\%$) and by 12% on the Cray.

Overall, genMP's overhead seems reasonable; it never exceeded 2.4% on the iPSC, and I estimate that it never exceeded 5% on the Cray (overheads on the Cray could not be measured directly). The computations vectorized on the Cray as well as they did in the uniprocessor version of the code, and ran at roughly 250 megaflops/sec. on 4 processors. Surprisingly, the iPSC's high message startup time – roughly 5 msec. – appeared to have very little impact on the running time of the calculation. GenMP incurred a low communication overhead during local barrier synchronization, for example, because it can transmit data in bulk rather than an element at a time, and this was facilitated by restricting the partitions to have simple shapes.

4. Summary

I have outlined a simple approach to parallelizing numerical software for multiprocessors that insulates the programmer from many of the machine-dependent and low-level details. Application-dependent code and system-dependent code need not become heavily intertwined; when code is 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. I tried out my ideas on a realistic application, and obtained good parallel speedups on architectures that represent two extremes in multiprocessor design philosophy.

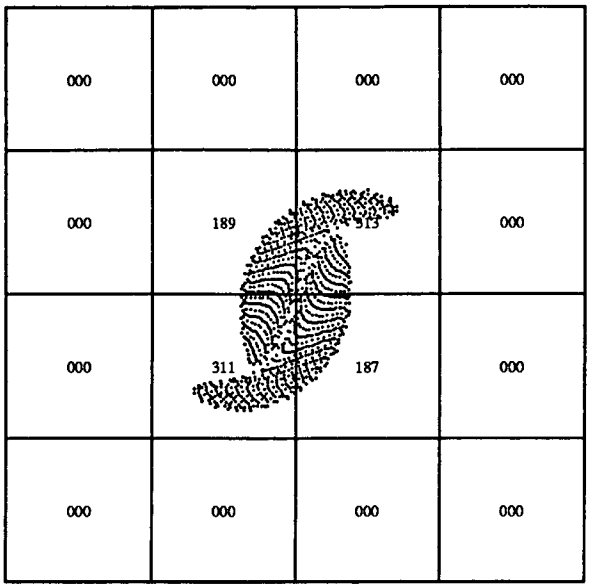
My approach is to have the user program a generic multiprocessor, called "genMP," with abstractions for hiding the details of task decomposition and coordination activities from the user. GenMP employs domain partitioning to subdivide workly fairly among a team of processors, and local barrier synchronization to ensure correctness. 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 estimates for the computation in each bin, and must supply routines for converting these data to and from byte streams. GenMP will assign bins to tasks in order to even the workload and will allow each task to access and communicate the necessary boundary data. GenMP is intended for a diversity of calculations, previously identified, that fit a simple model of spatial locality. It is neither universal nor complete, however, and leaves some programming details up to the discretion of the user.

In order to explore its generality, I have begun to apply genMP to other kinds of applications; a boundary layer calculation that solves the incompressible Navier-Stokes equations in two dimensions (in collaboration with E. G. Puckett), an adaptive grid method for hyperbolic partial differential equations (in collaboration with M. J. Berger and P. Colella), and a three-dimensional vortex calculation (in collaboration with T. Buttke and P. Colella).

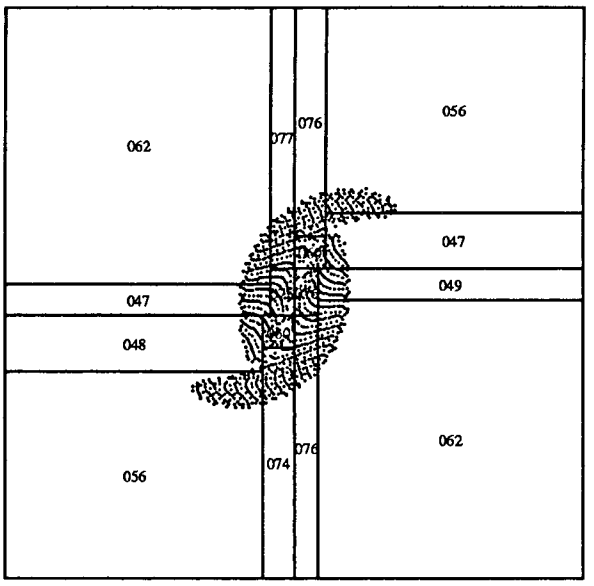
The research described here was part of my Ph. D. dissertation research [2] done in the Computer Science Division at the University of California at Berkeley. I gratefully acknowledge the encouragement and moral support of my thesis advisor, W. Kahan; Phillip Colella also helped to supervise the work. Many thanks go to Erling Wold for reading the final draft of this paper.

5. 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, "Run-Time Partitioning of Scientific Continuum Calculations Running On Multiprocessors," LBL-23625, Mathematics Department, University of California, Lawrence Berkeley Laboratory, June 1987. (Ph. D. Dissertation in the Computer Science Division at the University of California, Berkeley, Tech. Report # 87/366)).
3. M. J. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations," *J. Comput. Phys. 53*,3 (March 1984), pp. 484-512.
4. M. J. Berger and S. Bokhari, "A Partitioning Strategy for Non-Uniform Problems on Multiprocessors," *IEEE Trans. Comput. C-36*,5 (May 1987).
5. 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..
6. A. J. Chorin, "Numerical Study of Slightly Viscous Flow," *J. Fluid Mech. 57*(1973), pp. 785-796.
7. A. J. Chorin and J. E. Marsden, *A Mathematical Introduction to Fluid Mechanics*, Springer-Verlag, New York, 1979.
8. *Cray X-MP Hardware Reference Manual*, Cray Research, Inc., 1986. Order number HR-0097.
9. 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.
10. L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations," YALEU/DCS/RR-459, Yale Univ., Dept. of Computer Science, April 1986.
11. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*, McGraw-Hill, 1981.
12. *iPSC User's Guide*, Intel Corporation, Beaverton, Oregon, October 1985. Order Number: 175455-003.
13. A. Leonard, "Vortex Methods for Flow Simulation," *J. Comput. Phys. 37*(1980), pp. 289-335.
14. B. Nour-Omid, A. Raefsky and G. Lyzenga, "Solving Finite Element Equations on Concurrent Computers," *Proc. ASME Symp. on Parallel Computations and Their Impact on Mechanics*, December 13-18, 1987.



T = 12.500 Eff = 0.200



T = 12.500 Eff = 0.809

Figure 1. Partitioning of a particle calculation for vortex dynamics on 16 processors. A simple way to divide up the work is (a) to partition the domain 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 particles distribute themselves unevenly so that the completion time for a subproblem may not be proportional to its area. A better way (b) compensates for the uneven distribution of particles 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. At the depicted time each processor's share of the workload is shown in the subdomain assigned to it, normalized to 1000 units of total work. A perfectly balanced workload would correspond to 062 units of work for each subproblem.

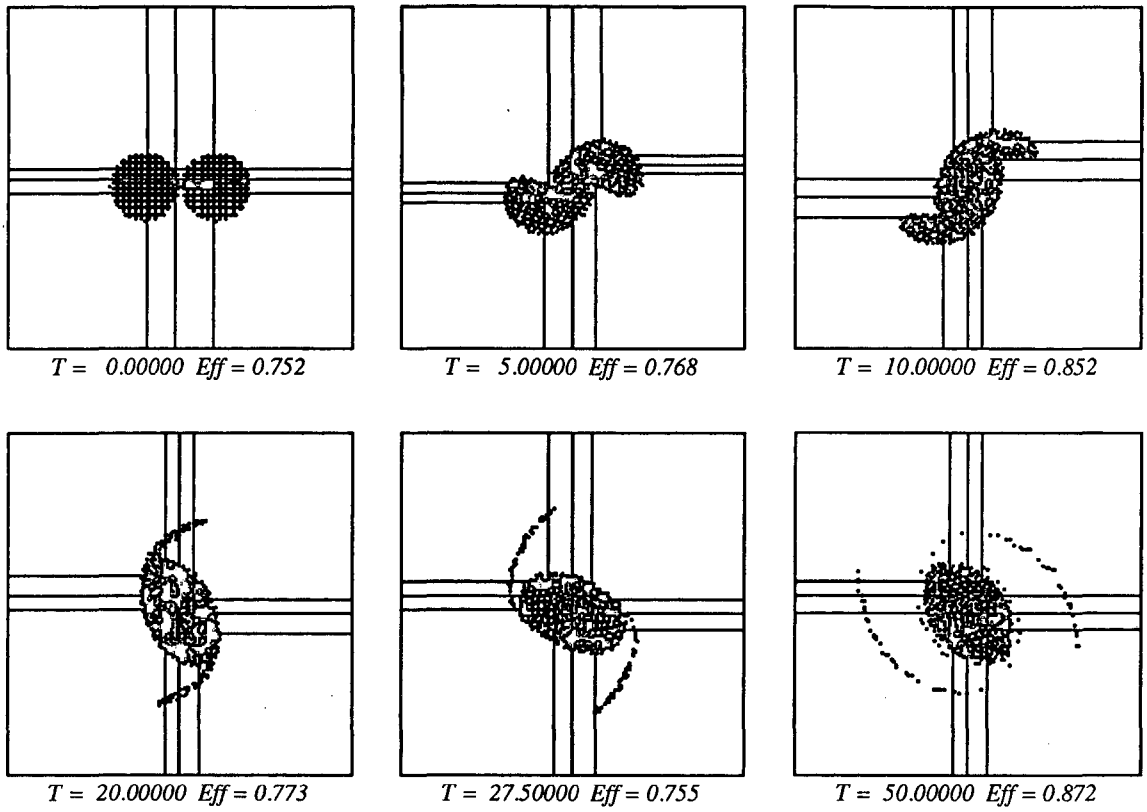


Figure 2. The distribution of particles changes with time, so the work must be periodically repartitioned. This series of snapshots was taken from the same calculation used to produce Figure 1.

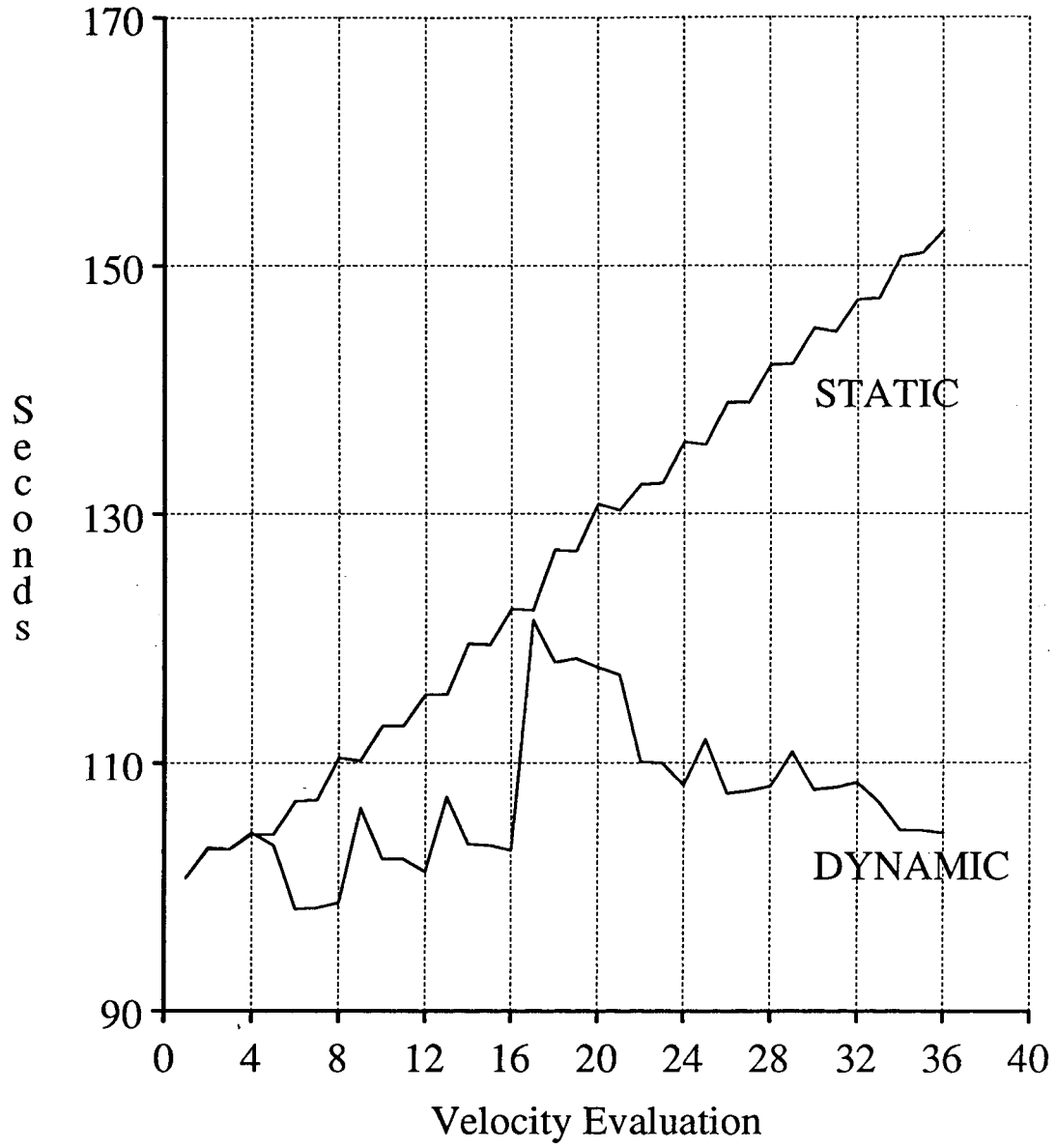


Figure 3. A comparison of static and dynamic load balancing on 32 processors of the Intel iPSC. If the workloads are partitioned only at the beginning of the calculation (static load balancing), the loads will drift gradually out of balance, and the time required to perform a velocity evaluation will steadily increase with time. In contrast, a dynamic load balancing strategy periodically rebalances the workloads and is able to maintain an almost steady running time. Here loads were rebalanced every fourth velocity evaluation.

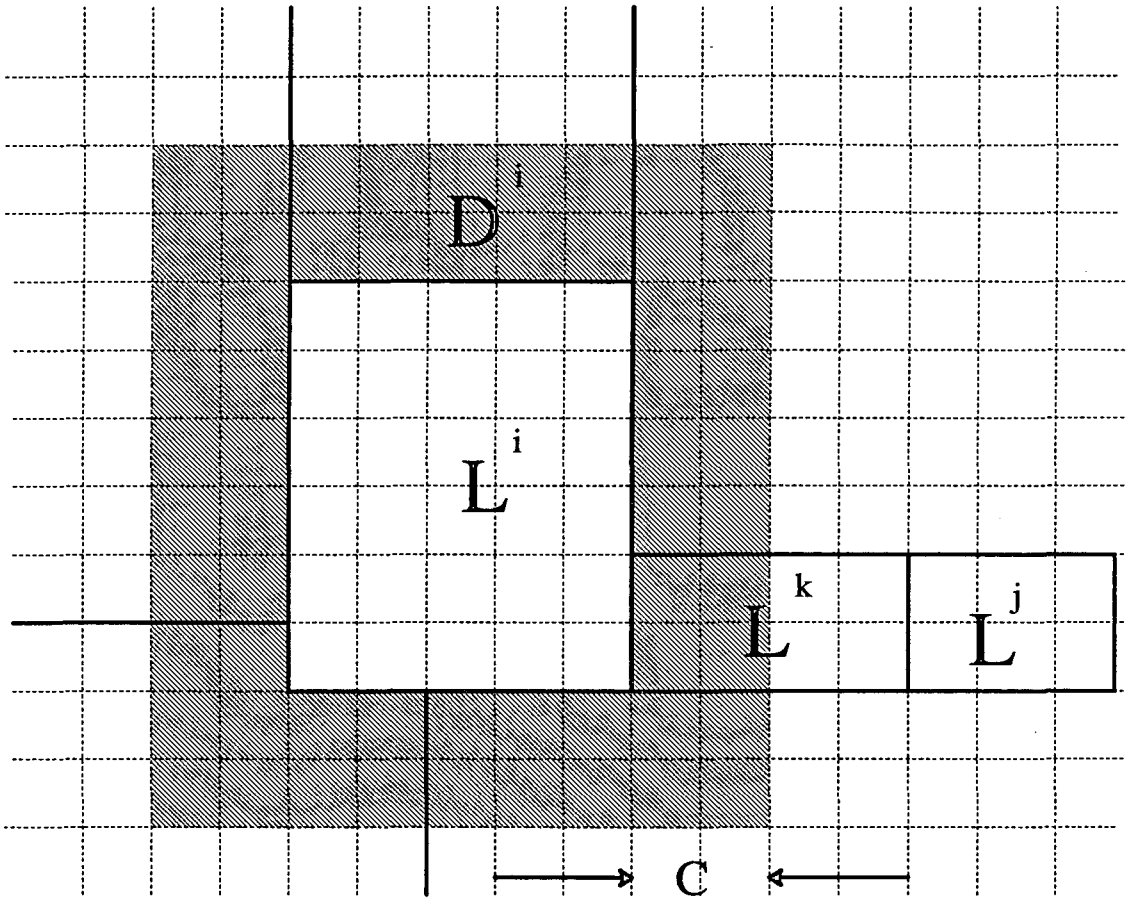


Figure 4. Task i is assigned L^i, a subregion of the work lattice, and an external interaction region D^i. D^i is a surrounding shell of bins and is C bins thick, where we have chosen $C = 2$. Since D^i and L^j do not intersect, subproblems L^i and L^j are locally independent. But D^i and L^k *do* intersect, and so subproblems L^i and L^k are locally dependent.

Tables and Figures

	Cray X-MP	Intel iPSC
Communication Model	Shared Memory	Message-Passing
# Processors used [Max]	4 [4]	32 [128]
Megaflops/sec/cpu	100	0.035
Max Memory (megabytes)	128 total	0.5/cpu

Table 1. Design parameters for the Intel iPSC and the Cray X-MP. We used the iPSC model d5, with 32 processors, and the largest-model Cray, the X-MP/416, with 4 processors 16 megawords of main memory. The megaflop execution rates are typical *sustainable* rates for just one processor.

N	P	η_P (Efficiency)	S_P (Speedup)	%Lbar	%Part
386	4	90	3.6	0.3	1.2
796	8	85	6.8	0.8	1.2
1586	16	79	13	1.4	1.0
3180	32	74	24	1.6	0.8

Table 2. iPSC results, where the number of vortices N varies linearly with the number of processors P. The parallel efficiency η_P (reported as a percentage) and parallel speedup S_P decrease with P. By definition the $\eta_P = S_P/P$. Overhead costs are reported as %Lbar, the fraction of the total time spent in local barrier synchronization, and %Part, the fraction spent partitioning, including the cost of producing the work density mapping. All runs lasted 64 timesteps, two velocity evaluations were done per timestep, and loads were rebalanced every other timestep. Since the larger problems couldn't fit into the memory of a single processor, T_1 could not be measured directly, and the efficiency and speedup figures are pseudo-measurements.

N	P	S_P	η_P	η_P^{max}
12848	1	1.00	1.000	1.000
12848	2	1.95	0.973	0.994
12848	4	3.63	0.908	0.982
25702	1	1.00	1.000	1.000
25702	4	3.57	0.892	0.957

Table 3. Parallel efficiency and speedup for the X-MP runs. η_P^{max} is the maximum theoretical efficiency that could be achieved under ideal conditions, given we chose not to parallelize the global computation done by a Poisson solver. The runs with 12848 vortices ran for 400 timesteps, the larger runs for 240. Loads were balanced every timestep.

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