Lawrence Berkeley National Laboratory

Recent Work

Title

VERY LARGE VORTEX CALCULATIONS IN TWO DIMENSIONS

Permalink

<https://escholarship.org/uc/item/9b92k97j>

Author

Baden, S.B.

Publication Date

1987-11-01

e.2

Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

Physics Division

RECEIVED
LAWRENCE
BERKELEY LABORATORY

FEB 19 1988

LIBRARY AND
DOCUMENTS SECTION

Mathematics Department

Presented at the UCLA Workshop on
Vortex Methods, Los Angeles, CA,
May 20-22, 1987, and to be
published in the Proceedings

Very Large Vortex Calculations in Two Dimensions

S.B. Baden

November 1987

LBL-24334
e.2

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

**VERY LARGE VORTEX CALCULATIONS
IN TWO DIMENSIONS¹**

Scott B. Baden

Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

November 1987

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

Very Large Vortex Calculations in Two Dimensions

Scott B. Baden

Mathematics Department, Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

Abstract

We describe an implementation of Anderson's Method of Local Corrections running on the Cray X-MP multiprocessor with 4 CPUs. We will show how the code was vectorized and parallelized, including dynamic load balancing. We present computational results from very large runs of up to 25702 vortices.

... in the Vortex you are given just one momentary glimpse ... and somewhere in it a tiny little marker, a microscopic dot on a microscopic dot, which says 'you are here.'

—Douglas Adams, The Hitchhiker's Guide to the Universe

1. Introduction

Until recently, large vortex blob calculations involving tens of thousands of vortices were considered intractable, even on the fastest computers. The problem was that an N -body problem had to be solved directly, at a cost that was quadratic in the number of vortices. Today there exist techniques for approximating N -body interactions reasonably well; they compute only a small fraction of the interactions directly and can run in a fraction of the time of the fully direct method. These algorithms work by exploiting the fact that a vortex blob behaves like a point vortex outside the core. Most interactions, involving distant length scales, can then be effectively lumped or averaged since the induced velocity field will be a harmonic function. We use one strategy that takes advantage of that fact, known as Anderson's Method of Local Corrections (MLC) [2]. Another such strategy, the multipole expansion method of Rokhlin and Greengard [11], appears elsewhere in this proceedings.

The MLC differs from the direct method in that it applies computational effort non-uniformly over the vortices. It uses special data structures to concentrate its effort where the vortices are the densest. Such data structures are generally more complicated than the arrays that are traditionally used in the direct method, and require special treatment on a vector-type multiprocessor like the Cray X-MP. First, inner loops must be carefully coded

in order they may vectorize and execute with reasonably long vectors. Second, all processors must be coerced to share the work evenly, and this must be done with a modest overhead.

The purpose of this paper is to consider the problem of implementing the MLC on the Cray X-MP with 4 processors, though the ideas should carry over to other kinds of particle methods and to other types of multiprocessors, too. The next section discusses the MLC algorithm. Section 3 specifies a portable parallel algorithm that can run on diverse multiprocessor architectures. Section 4 presents an implementation of the parallel computation for the particular instance of the Cray X-MP, section 5 presents computational results, and section 6 summarizes the paper.

2. Anderson's Method of Local Corrections

We will use the MLC to solve the *vorticity-stream function formulation* of Euler's equations for incompressible flow confined to a square box Ω with boundary $\partial\Omega$. Let $\mathbf{u}(\mathbf{x}(t), t)$ be the velocity of the fluid at position $\mathbf{x}(t)$ at time t , with x and y components u and v ; $\mathbf{u} = (u, v)$. Let $\omega = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}$ be the vorticity and ψ the stream function such that $\mathbf{u} = \text{curl } \psi$. The vorticity-stream function form of the Euler's equations is:

$$\frac{D\omega}{Dt} = 0 \quad (2.1)$$

$$\omega = -\Delta\psi \text{ in } \Omega \quad (2.2)$$

where $\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla$ is the material derivative and $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the two-dimensional Laplacian operator. Finally, let the flow satisfy free-space boundary conditions:

$$\mathbf{u} = 0 \text{ at } \mathbf{x} = \infty. \quad (2.3)$$

(Other kinds of boundary conditions, such as no-flow boundary conditions, can also be handled.)

To discretize the above equations we initially place a collection of N vortex blobs on a regular two-dimensional mesh and then compute the paths of the vortices over a sequence of timesteps. The following system of ordinary differential equations describes the motion of the vortices:

$$\frac{d}{dt} \mathbf{x}_i(t) = \sum_{j=1}^N \chi_{\sigma}(\mathbf{x}_j(t) - \mathbf{x}_i(t)) \omega_j, \quad i = 1, \dots, N, \quad i \neq j, \quad (2.4)$$

where $\mathbf{x}_j(t)$ is the position of the j th vortex blob at time t , ω_j is its strength, σ is the cutoff

radius, and $\chi_\sigma(\mathbf{r})$ is the influence function. $\chi_\sigma(\mathbf{r})$ gives the velocity induced at the point $\mathbf{r}$ by a unit-strength vortex blob at the origin, and comes in many forms; see Hald [12] for a discussion of different kinds of functions and of their effect on accuracy. Equations (2.4) are an approximation to the Lagrangian form of (2.1-2); for a derivation see Anderson and Greengard [1]. We will not discuss time integration, but focus on the velocity computation on the R.H.S. of (2.4).

The equations of (2.4) imply that we must solve an N -body problem, an $O(N^2)$ calculation that is prohibitively expensive when the vortices number ten thousand or more. Computing the mutually induced velocities for a distribution of 12848 vortices, for example, consumes 56 seconds of CPU time on a single processor of a Cray X-MP, and a full length run comprising 10000 velocity evaluations would consume over 150 hours. By comparison, the MLC can complete the run in just under 21 hours. Unlike the direct N -body method, the MLC distinguishes between two kinds of vortex interactions: (1) long-range interactions approximated by solving a discrete Poisson equation on a finite-difference grid (“global” computation); (2) N -body interactions computed accurately for vortices close enough to one another (“local” computation). When they number in the thousands or more, the calculation spends most of its time computing local N -body interactions between nearby vortices. Vortices that are not so close to one another interact indirectly through the relatively inexpensive global finite difference computation. The method is fast because it can effectively approximate most interaction inexpensively—typically greater than 95% of the summands of (2.4). Unlike the direct method, the MLC is not grid-free. It computes the velocity on a set of vortices in two steps:

- (1) Solve Poisson’s equation for an approximate velocity field on a finite difference grid and interpolate to the centers of the vortices.
- (2) Locally correct the velocity of each vortex by cancelling the portion of the approximate velocity due to nearby vortices and substituting in its place local direct interactions.

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

Step 1 calculates $\tilde{u}$, an approximation to u on a grid, in three sub steps:

- (1.a) Set up a right hand side for the discrete Poisson equation.
- (1.b) Solve.

(1.c) Interpolate.

Assume a single vortex with unit vorticity centered at the origin; this simple case generalizes to arbitrary collections of vortices by superposition and linearity. Setting up the right hand side entails first computing a “spread function” g_D at a discrete set of grid points surrounding the vortex. g_D approximates the discrete Laplacian of the vortex’s velocity field by truncating to zero at sufficient distances from the vortex:

$$g_D(ih, jh) = \begin{cases} \Delta^h u(ih, jh) & 0 < |ih| \text{ and } |jh| \leq D \\ 0 & \text{otherwise.} \end{cases} \quad (2.5)$$

Here h is the grid spacing, D is called the “spreading distance” and is chosen by the user, Δ^h is the discrete Laplacian, and u is the x component of the *point* vortex velocity function:

$$\mathbf{u}(x, y) = (-y, x)/2\pi r^2 \quad (2.6)$$

For the boundary conditions we use the exact velocities induced by the vortex at the discrete positions of the boundary of the grid, computing according to (2.6). The first step finishes by inverting the right hand side with a Poisson solver, and then interpolating the resultant far-field velocity field onto the center of the vortex. We used the Lagrange interpolation formula for complex analytic functions used by Anderson [2].

For the ensuing discussion on the local corrections now consider an arbitrary distribution of vortices. At the end of the first step each vortex will have accumulated *point* vortex influences from all the others. This far-field approximation, however, is invalid wherever the vortices are close enough that they behave as blobs rather than as point sources of vorticity. These interactions will have to be recomputed using the direct method, this time using a vortex blob velocity formula. We cannot use the finite difference strategy to compute these interactions since only point vortices induce a harmonic velocity field. The local interactions in the MLC are computed in much the same way as direct interactions involving charged particles. The MLC requires that a “correction distance” C be chosen by the method’s user to distinguish nearby vortices (closer than C) from distant ones. These nearby vortices are the ones that participate in the local interactions. 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 (Figure 1). Convenience dictates setting the correction distance C to a multiple of the bin width. Let C now stand for that multiple. C typically takes on values like 1 or 2. All the vortices influencing bin (i, j) can now be found in bins whose indices differ from i and j by integers no bigger than C . This surrounding region of space is referred to as the “correction neighborhood.”

Figure 1. Vortices, shown as x's, get sorted into bins, demarcated here by hyphenated lines. In practice the bins used in the MLC are much smaller than shown in the figure, so the vortices interact directly only over short distances. Each shaded region designates the correction neighborhood for a bin at the region's center, with $C = 1$. The region contains all the vortices that influence those in the bin via local interactions, and includes the bin itself.

The local corrections phase finishes by cancelling out the far-field influences from the vortices in the correction neighborhood. This entails computing the exact point-vortex velocities (using (2.6)) induced by the vortices in the correction neighborhood at the positions of the interpolation stencil surrounding the bin, interpolating, and then subtracting from what had already been accumulated on the vortices in step 1. In sum, the local corrections divide into two parts:

- (2.a) Locally cancel the far-field approximation for all the vortices in the correction neighborhood.
- (2.b) Compute local interactions with those same vortices.

The MLC concentrates most of its work in the local interactions phase, and its overall running time can be reasonably estimated by summing the number of local interactions computed against all the vortices. A *work density mapping* gives the number of local interactions computed against the vortices in each bin. It 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 in the correction neighborhood:

$$workMap(i, j) = \text{card}(i, j) \left[\sum_{|k|, |l| \leq C} \text{card}(i+k, j+l) \right] \quad (2.7)$$

where “ $\text{card}(i, j)$ ” is the number of vortices in bin i, j .

Assuming that the influence function, the cutoff radius σ , and the vorticity mesh spacing h , are fixed, two parameters affect the spread and accuracy of the MLC— the spreading distance D and the correction distance C . D affects the accuracy of the far-field interactions and can be no smaller than σ . It cannot hurt to make D slightly too large, although the cost to set up the right hand side for Poisson’s equation increases roughly as D^2 . C can be no smaller than σ because the finite difference approximation to u isn’t accurate inside the core of the vortex ($r < \sigma$) and must be corrected there. The answers get more accurate as C increases, but for a price: the cost to do the local corrections varies roughly as the square of C , depending on the local density of vortices. We obtained good results by setting $C = D = 2h$, where $h = 1/M$ is the finite difference mesh spacing used by the Poisson solver.

3. A Parallel Implementation

We present a parallel implementation of the MLC. We suppress architecture-specific details and focus on how to organize the computation for a generic multiprocessor. In the next section we will flesh out the details of the implementation on a particular instance of

multiprocessor architecture, the Cray X-MP. Our generic multiprocessor can be implemented by a layer of software on most any traditional multiprocessor system; see Baden [3] for the details. It provides a set of subroutines that the MLC will invoke from time to time and that we will refer to as “run-time utilities.” The semantics of these utilities are independent of the underlying architecture.

The MLC uses three major data structures. First, there are two finite difference meshes with dimensions $(M+1) \times (M+1)$ and spacing $h = 1/M$. There is also a bin data structure with dimensions $M \times M$, called the workLattice. Each bin in the workLattice contains a collection of vortices and can be thought of as a pointer to a linked list of vortex records. Each vortex record contains fields for the position and strength of a single vortex, and some additional information used in time integration. The record also contains a pointer field to string together all the vortices in a single bin into a linked list.

To parallelize the MLC, we subdivide the workLattice 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 workLattice. 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.

The boss plays a minor role and primarily sits idle. Its major purpose is to initiate and terminate the parallel computation. Whether the boss task executes on its own processor or shares a processor with a worker isn’t said. The boss furnishes each worker with an initial assignment of work, consisting of a description of the assigned region of the workLattice and the initial configuration of bins (and hence vortices) belonging there. Each worker maintains a private copy of its assigned part of the workLattice and of the finite difference meshes. 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 2). The task uses this external interaction region to maintain copies of vortices belonging to other tasks that directly interact with its own; hence, the region is C bins thick. Because our generic multiprocessor has no shared memory a task may directly modify only the vortices in its private memory and it has no information about any vortices that lie beyond the external interaction region. If a task modifies a copy of a vortex in the interaction region, then the owner of the “original” won’t know that the change was made.

Figure 2. Task i is assigned L^i , a subregion of the workLattice, and an external interaction region D^i . D^i is a surrounding shell of bins C bins thick, where $C = 2$. Since D^i and L^j do not intersect, subproblems L^i and L^j are independent during the local interactions computation, i.e. they do not interact. But D^i and L^k do intersect, and so subproblems L^i and L^k are dependent during local interactions.

Similarly, when a task modifies an original any tasks that possess copies will be unaware of the changes. To ensure correctness tasks must periodically satisfy a synchronization constraint such that all vortex copies will be consistent with the originals. Tasks synchronize by calling a run-time utility.

The computation begins with all tasks cooperating to set up a right hand side for the Poisson solver. Each task computes only over its assigned bins (exclusive of the external interaction region) and calls a run-time utility to determine the bounds of its assigned region of the workLattice. When done, it passes the local right hand side it has computed to the Poisson solver. The Poisson solver combines all the local contributions into a single global right hand side and then inverts the global right hand side. Each task will generally encounter the solver at a different time, according to the amount of work assigned to it. All tasks leave the solver together, however, and upon return each will have a copy of the result in its private memory. How the solver combines the local right hand sides and how it distributes its result isn't said; we assume that the solver comes in an application library and that its internal structure is hidden from the programmer. Once a task has obtained a copy of the inverted right hand side, it is free to interpolate far-field velocities onto its vortices and then do the local corrections. This finishes the velocity evaluation.

The next velocity evaluation may commence after the time integration procedure has been called. The evaluation begins by sorting the vortices into their correct bins, since they moved since the last evaluation. (Assume for now that no vortex has moved into a bin owned by another task.) At this point the private copies of vortices in the external interaction region are inconsistent with the originals since these were changed by the owner during time integration and sorting. To ensure correctness these copies must be refreshed. Each task must synchronize at a "local barrier" by communicating with all tasks that overlap its external interaction region. It does this by calling a run-time utility called `lBar`. This utility shuffles vortices between processors and hides most of the details. When a task encounters a call to `lBar` it will not return until it has finished synchronizing with all interacting tasks (the "local barrier"), at which time it is free to continue with the velocity evaluation. Thus, `lBar` specifies "Local BARrier synchronization points," and hence its name. A separate call to `lBar` may also be used to migrate vortices that changed owners as a side-effect of the sorting procedure.

Because vortices will migrate among tasks over time, some tasks may eventually become overloaded with work, while others become lightly loaded. As a result, the time required to do a velocity evaluation will gradually increase with time, unless the work-assignments are readjusted (see Figure 3). To this end the workers must periodically invoke a run-time utility called `Partitioner`, say every couple of velocity evaluations.

Figure 3. The cost of doing a velocity evaluation will steadily increase (top curve) unless workloads are periodically rebalanced (bottom curve). We balanced workloads every time step, or every other velocity evaluation. These measurements were obtained from the Intel iPSC with 32 processors running with 3180 vortices. The top run was aborted after 36 velocity evaluations; some of the iPSC's processors had accumulated sufficient additional vortices to exceed the capacity of their local memory.

A worker obtains its new work assignments by computing a “work density mapping,” in the form of an array, and then passing that array to the partitioner. This work density mapping estimates the cost of evaluating the velocities in each bin of the worker’s assigned region of the `workLattice`. It is computed according to equation (2.7) and tallies the number of local interactions computed against each bin. This simple estimate gives a reasonable guess of the exact cost of the overall computation since the local interactions account for a large fraction of the MLC’s running time. The partitioner combines the local mappings using a mechanism hidden from the programmer, and then uses this global mapping to decide how to divide up the work. All tasks leave the partitioner together and upon return each will be assigned a unique rectangular region of the `workLattice`, as shown in Figure 4. These regions change with time as shown in Figure 5. 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 the `Partitioner`, some vortices may change owners, and must therefore be transmitted to the correct task. As was the case with sorting, a call to the `lBar` utility can handle the necessary exchanges of data. One detail that has been left out about `lBar` is that the user must pass it two subroutines as arguments. These implement gather and scatter operations that are familiar to users of vector architectures, and will not be discussed. The interested reader is referred to [3] for the details. The complete parallel MLC code appears in Figure 6.

4. Cray X-MP Implementation

4.1. The Cray X-MP Multiprocessing System

The Cray X-MP is a shared-memory multiprocessor with vector arithmetic capabilities. We used the largest model, known as the X-MP/416. The X-MP/416 has four processors and 16 million words of main memory (1 word = 64 bits) and is capable of computing at a sustained rate of more than 400 million floating point operations per second. Each processor executes its own instruction stream and communicates with the others through either the shared memory or through a special set of shared registers. The definitive source of information about the X-MP is the hardware reference manual published by Cray Research, Inc. [10], although the pamphlet by S. Chen et al. [6] is a more accessible document.

The key to using the X-MP effectively is to keep the processors busy doing useful work most of the time. We must ensure that most arithmetic operations execute in the fast *vector* mode and that the vectors be reasonably long (40 elements or more). We must also divide the work up fairly among the processors, ensuring that the cost of coordinating the

$T = 12.500$

$T = 12.500$

Figure 4. To parallelize the MLC we subdivide the domain into regions and have each processor compute on just one region. We show two such partitionings on 16 processors. To compensate for the uneven distribution of vortices over the domain we use an adaptive partitioning (a) that generates somewhat irregularly sized subproblems that all complete in roughly the same time. If we partitioned the domain uniformly (b), only 4 of 16 processors would be given much work to do, and the computation would run four times more slowly than in (a). The labels give each processor's share of the workload normalized to 1000 units of total work. If loads were perfectly balanced, then each subproblem would get 062 units of work. The calculation began with 2 finite area vortices consisting of 795 vortices, shown as dots, placed evenly on a mesh confined to a disk. Some vortices have been omitted for the purpose of clarification.

Figure 5. The distribution of vortices changes with time, so the work must be periodically repartitioned. If this were not done, then some processors would become overloaded while others would only stand and wait. Owing to a time step constraint, vortices jump by only small amounts between time steps; repartitioning need therefore be done every few, say 2 or 4, timesteps. This series of snapshots was taken from the same calculation used to produce Figure 4. Some vortices have been omitted for the purpose of clarification.

```

sort the vortices
local synchronization barrier #1; handle vortex migration

if it's time, repartition to rebalance the work loads
local synchronization barrier #2; handle load shifting

local synchronization barrier #3; obtain copies of locally interacting vortices

compute velocities over this task's work assignment
    set down right hand side
    call the Poisson solver [global synchronization barrier]
    interpolate and do local corrections

```

Figure 6. The parallel MLC code contains three local barrier synchronization points.

effort doesn't overwhelm the benefits. In this section we will show how we did this for the MLC, though many of the ideas carry over to other kinds of computations, in particular, to the Multipole Expansion Method discussed elsewhere in this proceedings. We will discuss parallelization and vectorization separately, since the two may be handled independently of one another in the case of the MLC.

All our software was written in Cray X-MP FORTRAN [8], a superset of FORTRAN 77 which Cray Research, Inc. has extended to improve the vectorizability of code and to deal with certain aspects of concurrency. Several new Cray FORTRAN library routines are provided in support of these activities and are described both in the FORTRAN manual and in the Cray X-MP Multitasking User's Guide [9]. Cray made two major changes to FORTRAN in support of parallel execution. First, code is now re-entrant and so local variables no longer persist across subroutine calls. Second, a new kind of common block was introduced called `task common`. Whereas ordinary common is storage shared among all tasks, a `task common` block is private to a single task. Task common is dynamically allocated and initially it is undefined. We used it to implement the local memory of our generic multiprocessor.

4.2. Parallelization

In a parallel computation certain parts must run on just one processor to ensure correctness, e.g. program initialization. Other parts may be executed concurrently to reduce the running time of the program. Not all parallelizable parts need be parallelized, however, but only those parts that consume a “noticeable” fraction of the code’s running time. Just what this noticeable fraction is depends on the number of processors in use, and can be determined from Amdahl’s law. This law tells us the best parallel speedup S we can expect on P processors, given that only a fraction f of the code parallelizes is:

$$S(P, f) = ((1-f) + (f/P))^{-1} \quad (4.1)$$

Ideally our speedup equals P ($f = 1.0$). Thus, our first task is to determine which parts of the code, if parallelized, could substantially reduce the running time of the calculation. To this end we used the Cray *flowtrace* facility to profile the MLC code. We found, for example, that the MLC spent only 0.6% of its time in the Poisson solver with a calculation using 12848 vortices and a 64×64 solver mesh. Thus, Amdahl’s law tells us that if we don’t parallelize the Poisson solver, then our speedup will only drop to 3.92 (out of a possible 4.0 if we did parallelize). We therefore chose not to parallelize the solver since doing so would only speed up the overall computation by at most 2% ($P/S - 1$). After all tasks have contributed their local right hand sides to the solver, a conditional statement selects a distinguished task to do the solver’s computation. The others wait at a synchronization barrier until the solver returns. This barrier is a global one since all tasks must wait, and is implemented by a run-time utility.

The next part of parallelizing the MLC is to implement the boss task. The boss calls various Cray X-MP multitasking library routines to spawn the worker tasks. The calling sequences for these routines are described in the X-MP Multitasking User’s Guide and will not be discussed.

The final part of the implementation is to set up the remaining calls to the generic multiprocessor’s run-time utilities to handle partitioning and local barrier synchronization. We have already discussed this in §3 and will not go into detail here. The major point to be made is that the programmer doesn’t know how the utilities work though he will pass them some subroutines as arguments. These routines accounted for roughly 700 out of 7000 lines of code in our implementation.

4.3. Vectorization

The desire to have `do` loops execute in vector mode had a major impact on the design of the numerical portion of the MLC code. Owing to the high cost of following pointers on the Cray, collections of vortices represented as linked lists should first be

gathered into contiguous vectors before they can be used within an innermost loop. Later, the result of the computation will be scattered from the adjacent locations of the array into the non-adjacent locations of the linked list. The effectiveness of gather and scatter operations is contingent on their being done infrequently relative to numerical operations. We implemented gather and scatter operations in software; the code is straightforward and will not be shown.

Operation counts can be a misleading timing metric on the Cray since operation times are sensitive to how well the Cray's pipelined functional units are kept filled. Even if they execute more arithmetic operations, calculations that have been modified to utilize the pipeline more effectively often run faster than the original computation that executed fewer arithmetic operations. Decisions, for example, are often very expensive within tight inner loops, since they can disrupt the flow of data through the arithmetic pipeline. Some inner loops will have to be rewritten to avoid disturbing the pipeline; this entails evaluating both branches of the loop and then using a Cray intrinsic vector merge function to select the desired result. The strategy effectively converts a conditional statement that doesn't vectorize into a conditional expression that does. In making the conversion however, we have changed the semantics of the loop. We may have introduced spurious illegal operations, such as division by zero, whose results will never be used but which could abort the program. Such erroneous operations must be somehow be avoided in software or rendered harmless. We used a `max` function, for example, to protect against any spurious divisions by zero.

4.4. Storage Utilization

The MLC program consumed a total of 3.6 megawords of memory when running on 4 processors. The boss executed as a fifth task in addition to the 4 worker tasks. The shared code consumed only 43 kilobytes of storage, divided roughly equally among the MLC code and the system libraries. Each of 5 tasks consumed 100 kilobytes of stack space and all tasks shared 400 kilobytes of heap storage. The remainder of the storage was consumed primarily by the major data structures which were duplicated for each task in `task common`—the bins, finite difference grids, gather/scatter buffers (each 4096 elements long), and vortex-records. The first three structures consumed roughly 138 kilobytes per task, the vortex-records 400 kilobytes per task. There were also 70 kilobytes of global common grids shared by all tasks.

4.5. Summary

We took 4 steps to parallelize the MLC code:

- (1) Determine which parts of the code, if parallelized, could substantially reduce the running time of the calculation, and which parts can run serially. Set up serial sections with a conditional statment and a call to a global barrier synchronization utility.
- (2) Set up the boss task.
- (3) Implement data partitioning. Modify loop bounds and generate the work density mapping array. Insert a call to the `Partitioner` utility.
- (4) Handle local barrier synchronization. Write the gather and scatter routines. Insert calls to the `lBar` utility.

Note that the utilities are provided by our generic multiprocessor's run-time library.

We took two steps to vectorize the code:

- (1) Gather vortices stored as linked lists into long vectors prior to their use within inner loops, and scatter the results to memory afterwards.
- (2) Execute both branches of a conditional statement within tight inner loops and then select the desired result with an intrinsic merge function.

5. Computational Results

5.1. Accuracy

We used the Cray's default single precision arithmetic (64 bits). We used a second order Runge-Kutta time integration scheme (Heun's method), that does two velocity evaluations per time step, and Chorin's [7] second order cutoff function. All finite difference calculations were accurate to fourth order.

We ran two test problems to determine the accuracy of the MLC. The first test problem runs with a single vortex with strength π and cutoff $1/30$ at the origin. The vortex should remain stationary since there are no self-induced velocities. With the MLC, however, there is some motion, and so we measure the error as the L_2 norm of the position of the vortex at the end of the run (the runs were 1000 timesteps long). Since this run computes no local interactions it measures the accuracy of the finite difference approximation and cancellation procedures. We determined that a good value for the spreading distance D is twice the finite difference mesh spacing. The results are summarized in Table 1.

We ran with another test problem with a known exact solution to determine the interaction of D , C , and Δt with h_v , the initial spacing of the vortices. We set the cutoff $\sigma = h_v^q$, where we chose $q = 0.75$ for all our runs. (See Beale and Majda [4, 5] for a discussion of the relationship between σ and h_v and accuracy.) We set $C = D = 2h$, where h is the spacing of the finite difference mesh, and varied h . The initial vorticity distribution for the test problem is radially symmetric and vanishes outside a circle of radius 0.25 centered about the origin. The vortices are distributed on a uniform mesh of points, with the strength of a vortex at $\mathbf{x} = (x, y)$ given by $4\pi(1 - 4(x^2 + y^2))^7$. The vortices rotate about the origin with an angular velocity that increases with decreasing distance from the origin. See Perlman [14] for the details. The runs were stopped when the fastest moving vortices had rotated one revolution. The L_2 norm of the error was reported. This error is defined as:

$$h_v \left[\sum_{j=1}^N (\text{computed}(x_j) - \text{exact}(x_j))^2 \right]^{1/2} \quad (5.1)$$

The results of the study are presented in Table 2, and tell us three things:

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

D/h	L_2 Error
1	6.23324e-01
2	4.41949e-03
4	1.36450e-07
8	8.40737e-11

Table 1. The accuracy of the finite difference approximation improves as D , the support of Δu , increases. D is measured in units of the mesh spacing h , with h fixed at $1/30$. The error is measured as the L_2 norm of the position of the vortex at the end of 1000 timesteps. The timestep $\Delta t = 0.05$.

- (2) The correction radius C scales roughly with h_v (more precisely with the cutoff σ). Accuracy is insensitive to changes in C , so long as $C > \sigma$. This can be inferred by moving down a column and keeping the number of vortices fixed.
- (3) Accuracy improves significantly as the vortices are initially spaced more closely ($h_v \rightarrow 0$), so long as an appropriate timestep has been chosen.

The second result is significant since it tells us that the cost of the local interactions need not necessarily grow as quickly as N^2 if C and hence h are decreased with h_v . This is indeed what has been observed in practice, as shown in Table 3.

h_v (N)	h	Δt				
		0.1	0.05	0.025	0.0125	0.00625
0.014 (1005)	1/30	9.090×10^{-3}	4.566×10^{-3}	3.497×10^{-3}	3.243×10^{-3}	3.175×10^{-3}
	1/60	9.082×10^{-3}	4.556×10^{-3}	3.488×10^{-3}	3.233×10^{-3}	3.165×10^{-3}
0.007 (4020)	1/30	7.659×10^{-3}	3.648×10^{-3}	1.476×10^{-3}	1.216×10^{-3}	—
	1/60	7.661×10^{-3}	3.650×10^{-3}	1.478×10^{-3}	1.218×10^{-3}	—
	direct	7.658×10^{-3}	3.646×10^{-3}	1.474×10^{-3}	1.214×10^{-3}	—
0.0035 (16043)	1/30	7.172×10^{-3}	1.971×10^{-3}	7.479×10^{-4}	4.828×10^{-4}	4.219×10^{-4}
	1/60	7.173×10^{-3}	1.974×10^{-3}	7.500×10^{-4}	4.850×10^{-4}	4.241×10^{-4}
	1/120	7.174×10^{-3}	1.974×10^{-3}	7.506×10^{-4}	4.856×10^{-4}	4.247×10^{-4}

Table 2. Results from a three parameter study show the effect of varying the initial spacing of vortices h_v , the correction and spreading distances C and D , and the timestep Δt on the accuracy of the computed solution. The cutoff σ was set to $h_v^{0.75}$. In all cases C and D were set to $2h$, where h was the finite difference mesh spacing. For comparison, a result for the direct method is presented for the single case of 4020 vortices; accuracy is not significantly better than it is with the MLC. N gives the number of vortices.

5.2. Merger Question of Two Finite Area Vortices

In the next test problem we repeated the experiments of Zabusky [15] involving the question of merger of two Finite Area Vortices (FAVs). Contrary to what was observed by Zabusky and his coworkers, we observe no merger, in agreement with known analytic results. (For a discussion see Majda [13].) The initial vorticity distribution was set out on a uniform mesh confined to two circular disks centered about the origin. The disks had a radius of 0.1125, with centers separated by 0.3. This corresponded to a run done in Zabusky [15] in which the ratio of the *diameter* of the patches to their separation was set at 0.75. The vortices all had the same strength, equal to one-half the square of their initial spacing. Figure 7 shows the result of a run with 12788 total vortices at an advanced stage of the computation.

We next evaluate the performance of the MLC. (We used the same test problem, though we ran with slightly different parameters; the separation of the FAVs was 0.25 and their radii was 0.12.) We measure floating point operation counts and uniprocessor and multiple processor running times. We used Version 1.14 of the CFT compiler, dated October 8, 1985, and Version 2.2 of the loader (segldr), dated December 1, 1986, and ran under version 1.16BF1 of COS, the Cray Operating System. Since COS is a batch operating system, and therefore non-interactive, jobs were submitted to the Cray mainframe from an interactive front-end processor. Our results for the multiprocessor runs were obtained during special dedicated blocks of time during which we had the entire machine to ourselves and interference from operating system activities was minimal.

In the first experiment we ran on just one processor. We show that the Method of Local Corrections is substantially faster than the fully direct method. With 25,702 vortices, for example, the MLC is 16 times faster than the direct method. It can do a velocity evaluation in just under 15 seconds (see Table 3). To avoid a quadratic growth in the MLC's running time we must scale the correction distance with the initial spacing of vortices. There is, however, a practical lower limit on the size of C , above that implied by the 3-parameter study. The trouble is that the finite mesh spacing h , and hence the bin size, must be decreased along with C , and doing so decreases vector lengths. In addition, the cost of setting up the boundary conditions increases, since the number of boundary points where we must evaluate exact velocities increases. This is documented in Table 4. For the run with 12848 vortices for instance, decreasing C from $1/30$ to $1/60$ didn't speed up the calculation but actually slowed it down slightly. The reason why was that although the total number of floating point operations was reduced roughly by a factor of two, so was the rate at which those operations got done; the average vector length decreased from 160 to 40 elements.

$$T = 28.46480 [x 1.30]$$

Figure 7. We observed no merger of two finite area vortices after 28.4648 units of time. The vortices were spaced on 2.48625×10^{-3} unit centers, the timestep was 8.84×10^{-3} , and the cutoff $\sigma = h_v^{0.75} = 1.1134193 \times 10^{-2}$. Color photographs available from the author can be used to observe the result more closely than is possible in this plot.

We next consider how well the MLC can utilize the X-MP's processors when executed concurrently. We define η_P as the parallel efficiency on P processors:

$$\eta_P = \frac{T_1/P}{T_P}, \quad (5.1)$$

and use this as our figure of merit. Here T_P is the time to complete on P processors and 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 $\eta_1 = 1$. To evaluate the efficiency, we ran with two different problem sizes – 12848 and 25702 vortices. To conserve scarce dedicated computer time, we started the runs at an advanced stage of the simulation, using a snapshot file that had been generated on another X-MP system. The snapshots were taken at 10.0 units of simulated time; this is an interesting point in the simulation where the finite area vortices have begun to entrain. Table 5 gives the timings and the parallel speedup and efficiency for the various runs. Efficiency was never less than 89%; under ideal conditions of 100% efficiency the programs would run only 12% faster. We were able to attribute most of the efficiency loss to the load imbalance and to the fact that we chose to run the Poisson solver on just one processor. Had we parallelized the solver, then most of the efficiency loss would have been due to load imbalance.

6. Summary

We have implemented a vortex code that is capable of handling tens of thousands of vortices. It can do a velocity computation on 25702 vortices, for example, in about 4 seconds on a Cray X-MP with 4 processors. The computation is fast because it can approximate N -body interactions without having to resort to a fully direct computation. Furthermore, it vectorizes on the Cray – each processor computes at a rate of about 65 megaflops/sec – and it parallelizes well – it utilizes the 4 processors with 90% efficiency and can compute at an aggregate rate of 200 to 250 megaflops/sec. The algorithm appears to strike a reasonable balance between speed and accuracy.

The major issue in parallelizing the Method of Local Corrections is to avoid excessive load imbalance. We were, however, able to mitigate load imbalance at a reasonable cost with the aid of some run-time utilities. To a large extent, these utilities insulate task partitioning and synchronization activities from a change of computer architecture and their use therefore improved the portability of our software. These utilities do not, however, apply to the Poisson solver, though the solver may be parallelized separately from the rest of the calculation.

N	C	CPU time per timestep (min)		MLC Speedup
		MLC	Direct	
12848	1/30	0.12	0.95	7.9
25702	1/60	0.24	3.8	16
51376	1/60	0.65	15	23
102822	1/120	1.6	61	38

Table 3. The MLC is substantially faster than the direct method. If we scale the correction distance C with the initial spacing of the vortices h_v , then the cost of the MLC grows more slowly than the square of the number of vortices. The italicized times for the direct method, when $N > 12848$, were extrapolated from the running times for smaller N . Times are reported for a single velocity evaluation and were measured on a single processor of a Cray X-MP/416.

N	C	Local Interactions		Finite Differences		Overall	
		Time (sec)	Mflops per sec	Time (sec)	Mflops per sec	Time (sec)	$\times 10^9$ flops
12848	1/15	67.4	89	3.02	84	72.0	6.30
12848	1/30	28.2	69	5.37	68	36.1	2.33
12848	1/60	19.1	34	13.0	47	38.5	1.35
25702	1/60	95.1	78	8.85	79	108	8.19
25702	1/120	45.8	50	17.9	63	71.6	3.50
25702	1/240	36.9	23	49.8	42	112	3.50

Table 4. The optimal values used for the correction radius C have the property that increasing or decreasing them increases the overall running time of the computation. The optimal values, shown in bold-face, are larger than the minimum allowable values, immediately below, owing to a decrease in vector length and in an increase in the number of operations required to compute boundary conditions. We report on the running time and the execution rate for the two dominant parts of the computation: local interactions and finite difference computations (here finite difference computation doesn't include the Poisson solver). We also report on the entire computation. Though these measurements were taken from runs that lasted just two timesteps, we believe they are representative of much longer runs since operation counts do not appear to fluctuate much.

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

Table 5. Timings, parallel efficiency and speedup for the X-MP runs. S_P is the speedup; by definition $\eta_P = S_P/P$. All runs began at 10.0 units of simulated time. For $N = 12848$, the vortices were spaced at 2.6516×10^{-3} unit intervals; the finite difference mesh spacing h was $1/60$, and the runs lasted 400 timesteps. For $N = 25702$ the vortices were spaced at 1.8750×10^{-3} unit intervals; $h = 1/120$, and the runs lasted 240 timesteps. The cutoff was $h_v^{0.75}$ for all runs, the timestep was 0.0125, and $C = D = 2h$. Loads were rebalanced every timestep, i.e. every other velocity evaluation.

In the near future we believe that our code could handle as many as 10^5 vortices. A multiprocessor with 8 or 16 processors, each somewhat faster than the Cray X-MP's, would suffice and cannot be far off.

7. Acknowledgements

I gratefully acknowledge the advice of my thesis advisor, W. Kahan, and of Phil Colella. I give thanks to Elbridge Gerry Puckett for reading the final draft of this paper. 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, and in part by a California Fellowship in Microelectronics. Additional support was provided by Cray Research, Inc., Intel Scientific Computers, the National Science Foundation, and the Defense Advance Research Projects Agency.

8. References

1. C. Anderson and C. Greengard, "On Vortex Methods," *SIAM J. Numer. Anal.* 22,3 (June 1985), pp. 413-440.
2. C. R. Anderson, "A Method of Local Corrections for Computing the Velocity Field Due to a Distribution of Vortex Blobs," *J. Comput. Phys.* 62(1986), pp. 111-123.

3. S. B. Baden, "Run-Time Partitioning of Scientific Continuum Calculations Running On Multiprocessors," LBL-23625, Mathematics Department, University of California, Lawrence Berkeley Laboratory, Berkeley, California, June 1987. Ph. D. Dissertation in the Computer Science Division, University of California, also published there as a Tech. Report 87/366.
4. J. T. Beale and A. Majda, "The Design and Numerical Analysis of Vortex Methods," PAM-48, Center for Pure and Applied Mathematics, University of California, Berkeley, 1981.
5. J. T. Beale and A. Majda, "Vortex Methods. II: Higher Order Accuracy in 2 and 3 Dimensions," *Math. Comput.* 39,159 (July 1982), pp. 29-52.
6. 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..
7. A. J. Chorin, "Numerical Study of Slightly Viscous Flow," *J. Fluid Mech.* 57(1973), pp. 785-796.
8. *Cray-1 FORTRAN (CFT) Reference Manual (2240009)*, Cray Research, Inc., Mendota Heights, MN, 1978.
9. *CRAY X-MP Multitasking Programmer's Manual*, Cray Research, Inc., March 1986. Order number SN-0222.
10. *Cray X-MP Hardware Reference Manual*, Cray Research, Inc., 1986. Order number HR-0097.
11. L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations," YALEU/DCS/RR-459, Yale Univ., Dept. of Computer Science, April 1986.
12. O. Hald, "Convergence of Vortex Methods, II," *SIAM J. Numer. Anal* 16(1979), pp. 726-755.
13. A. Majda, "Vorticity and the Mathematical Theory of Incompressible Fluid Flow," *Comm. of Pure and Applied Math.* 39(1986), pp. S187-S220. Special issue.
14. M. B. Perlman, "On the Accuracy of Vortex Methods," PAM-192, Center for Pure and Applied Mathematics, University of California, Berkeley, December 1983. Ph. D. Dissertation.
15. N. J. Zabusky, "Contour Dynamics for the Euler Equations in Two Dimensions," *J. Comput. Phys.* 30(1979), pp. 96.

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