

Lawrence Berkeley National Laboratory

Recent Work

Title

Programming Direct N-Body Solvers on Connection Machines

Permalink

<https://escholarship.org/uc/item/2zm48219>

Authors

Greenberg, A.

Mesirov, J.P.

Sethian, J.A.

Publication Date

1992-08-01



Lawrence Berkeley Laboratory

UNIVERSITY OF CALIFORNIA

Physics Division

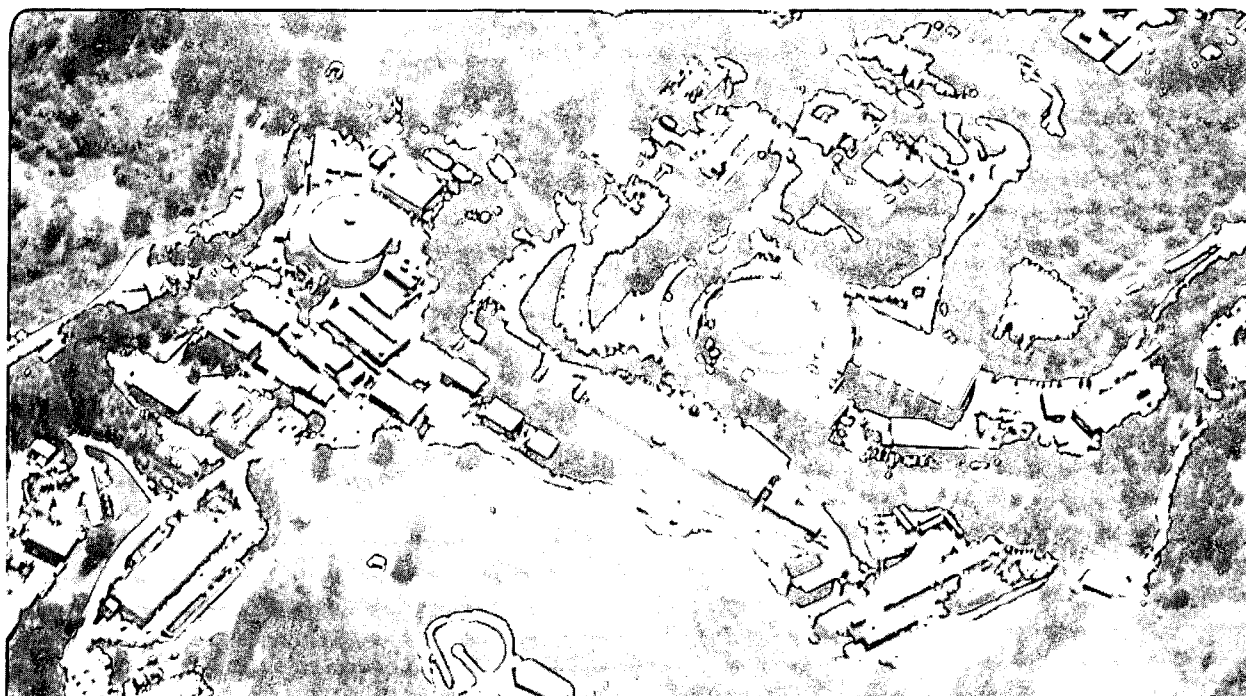
Mathematics Department

To be submitted for publication

Programming Direct N -body Solvers on Connection Machines

A. Greenberg, J.P. Mesirov, and J.A. Sethian

August 1992



1 LOAN COPY 1
1 Circulates 1
1 for 4 weeks 1

Bldg. 50 Library.
Copy 2

LBL-32899

DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. 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 liability or 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 and shall not be used for advertising or product endorsement purposes.

This report has been reproduced directly
from the best available copy.

Available to DOE and DOE Contractors
from the Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road, Springfield, VA 22161

Lawrence Berkeley Laboratory is an equal opportunity employer.

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.

Programming Direct N -body Solvers on Connection Machines

Adam Greenberg
Jill P. Mesirov

Thinking Machines Corporation
Cambridge, Massachusetts 02142

J.A. Sethian *

Department of Mathematics
University of California

and
Lawrence Berkeley Laboratory
Berkeley, California 94720

August 1992

Abstract

We analyze various ways to program a simple direct N -body solver on the Connection Machine CM-2 and CM-5, using both CM Fortran and CMMD Message Passing.

* This author was supported in part by the Applied Mathematics Subprogram of the Office of Energy Research under contract DE-AC03-76SF00098, and the National Science Foundation and DARPA under grant DMS-8919074.

PROGRAMMING DIRECT N -BODY SOLVERS ON CONNECTION MACHINES

I. INTRODUCTION

In this paper, we analyze various ways to program a direct N -body solver on both the CM2 and the CM5. The goal is to produce timing numbers over a fairly wide range of problems, and compare different ways to program the basic algorithm. In CM Fortran on both the CM2 and the CM5, we analyzed the performance of CSHIFTs, broadcasts, and SPREADs, and the effect of masking operations. On the CM5, without vector units, we analyze the performance of a CMMD message passing direct solver. All of the above tests (except CMMD) are also performed on a small test CM5 vector machine. A different implementation, which makes use of a multi-wire broadcast available on the CM2 (see [Brunet, Edelman, and Mesirov]), is not studied here.

In this paper, we took the perspective the average user might take to program a parallel version of the direct N -body algorithm. That is, we assumed that the user was well-acquainted with CMFortran, and tried to consider various ways that such a user might think of programming the algorithm. We then performed timings of these various approaches. We wanted to know the timings that such a user might be reasonably expected to obtain. We did not try to obtain maximum speed using lower level programming or a collection of tricks.

The direct N -body solver is based on a force law used in a vortex calculation, and is typical of the calculations performed in N -body codes. We perform all of the above tests for a two-dimensional N -body problem. The balance between communication and computation changes significantly between two and three dimensions.

II. BACKGROUND

II.A. The Basic N -body Algorithm

We imagine that we are given N bodies, each described by a position $\vec{x}$ and a strength C . The goal is to compute the total force exerted on each body, which consists of the superposition of the individual forces exerted by all other bodies. More precisely, Let $F(I, J)$ be the force exerted on body i by body j . Then the following serial algorithm computes the total force $Force(I)$ on each body:

N-Body Algorithm:

```

Do I=1,N
  Force(I) = 0.0
  Do J=1,N
    Force(I) = Force(I) + F(I,J)
  Enddo
Enddo

```

As written, this is an $O(N^2)$ algorithm. A somewhat faster algorithm may be obtained by noting that it is more convenient to compute both $F(I, J)$ and $F(J, I)$ when I and J are in place, which then suggests the following "triangular" N -body algorithm:

Triangular N-Body Algorithm:

```

Do I=1,N
  F_Static = 0.0
  F_Dynamic = 0.0
Enddo

Do I=1,N-1
  F_Static(I) = F_Static(I) + F(I,I)
  Do J=I+1,N
    F_Static(I) = F_Static(I) + F(I,J)
    F_Dynamic(J) = F_Dynamic(J) + F(J,I)
  Enddo
Enddo

F_Static(N) = F_Static(N) + F(N,N)

Do I=1,N
  Force(I) = F_Static(I) + F_Dynamic(I)
Enddo

```

The important thing to notice in the above algorithm is that the J index now runs from $I+1$ to N ,

rather than from 1 to N as it does in the previous algorithm. The reason that this algorithm may run faster is because the calculation of $F(I,J)$ may contain quantities that may be used in the calculation of $F(J,I)$ without recomputing. For example, a typical force law may require the evaluation of a square root to compute the distance between body I and body J , and need to be calculated only once for both $F(I,J)$ and $F(J,I)$.

II.B. The Two-Dimensional Vortex Code

For our purposes, the N -Body solver under study forms a kernel of vortex method for computing viscous, incompressible flow, see [Sethian, Brunet, Greenberg and Mesirov]. The force law for a collection of discrete vortex particles in two dimensions is given by

$$Force(I) = (U(I), V(I)) = \frac{1}{2\pi} \sum_{J=1}^N C(J) \frac{(Y(J)-Y(I)), -(Y(J)-Y(I))}{\max(\sigma, (Y(J)-Y(I))^2 + (X(J)-X(I))^2)}$$

where $(X(I), Y(I))$ is the position of body I , $C(I)$ is its strength, and σ is a small cutoff parameter to insure that the velocities remain bounded, see [Sethian].

III. BASIC IMPLEMENTATION OF N -BODY AND TRIANGLE N -Body ALGORITHM IN CM FORTRAN.

We investigated coding the two basic N -body algorithms using CSHIFTs, SPREADs, and broadcasts.

III.A Using Cshifts

Here, we consider a static and dynamic copy of the vortex elements, containing the position and strengths. The data is always laid out in one-dimensional arrays. For the full N -body algorithm, the dynamic copy is rotated through $N-1$ CSHIFTs, and between shifts the forces of the dynamic copy on the static copy are computed. The algorithm is given by:

The Full N -body Cshift Algorithm

```

subroutine cs_1d( n )
integer n
real, array( 5, n ) :: static
real, array( n ) :: dist
real, array( 3, n ) :: dynamic
integer i
cmf layout static( :serial, :news )
cmf layout dynamic( :serial, :news )
cmf layout dist( :news )
call cmf_random( static(1:3,:), 1.0 )
call cm_timer_clear( 0 )
call cm_timer_start( 0 )
call cm_timer_stop( 0 )
call cm_timer_clear( 0 )
call cm_timer_start( 0 )
dynamic = static(1:3, :)
static(4:5, :) = 0.0
dist = dynamic(3,:) / (max( .001,
.   (dynamic(1,:) - static(1,:))
.   * (dynamic(1,:) - static(1,:))
.   + (dynamic(2,:) - static(2,:))
.   * (dynamic(2,:) - static(2,:)))
static(4,:) = static(4,:) +
.   (dynamic(2,:) - static(2,:)) * dist
static(5,:) = static(5,:) -
.   (dynamic(1,:) - static(1,:)) * dist
do i = 1, n - 1
dynamic(1:3,:) = CSHIFT(dynamic(1:3,:), 2, 1)
dist = dynamic(3,:) / ( max( .001,
.   (dynamic(1,:) - static(1,:))
.   * (dynamic(1,:) - static(1,:))
.   + (dynamic(2,:) - static(2,:))
.   * (dynamic(2,:) - static(2,:)))
static(4,:) = static(4,:) +
.   (dynamic(2,:) - static(2,:)) * dist
static(5,:) = static(5,:) -
.   (dynamic(1,:) - static(1,:)) * dist
end do
call cm_timer_stop( 0 )
print *, 'c-shift: time for ', n, ' bodies = ',
.   cm_timer_read_elapsed(0)
return
end

```

Here, we include the self-interaction term of each body against itself, which may be important in other N -body problems.

An implementation of the N -body Triangle Algorithm using CSHIFTS requires two accumulators. This algorithm is very similar to the one above, but is given below for completeness.

The Triangle N -body Cshift Algorithm

```

subroutine cs_1d_triangular( n )
integer n
real, array( 5, n ) :: static, dynamic
real, array( n ) :: dist
integer, array( n ) :: mask
integer i, nrot
cmf layout static( :serial, :news )
cmf layout dynamic( :serial, :news )
cmf layout dist( :news )
do i = 1, n
static(1,i) = i
static(2,i) = i + 1
static(3,i) = 1.0
end do

```

```

call cm_timer_clear( 0 )
call cm_timer_start( 0 )
call cm_timer_stop( 0 )
call cm_timer_clear( 0 )
call cm_timer_start( 0 )
dynamic = static
static(4:5,:) = 0.0
dynamic(4:5,:) = 0.0
nrot = (n - 1) / 2
do i = 1, nrot
  dynamic = CSHIFT(dynamic, 2, 1)
  dist = 1.0 / ( max( .001,
    ( dynamic(1,:) - static(1,:) )
    * ( dynamic(1,:) - static(1,:) )
    + ( dynamic(2,:) - static(2,:) )
    * ( dynamic(2,:) - static(2,:) ) ) ) )
  static(4,:) = static(4,:) +
    ( dynamic(2,:) - static(2,:) ) * dist * dynamic(3,:)
  static(5,:) = static(5,:) -
    ( dynamic(1,:) - static(1,:) ) * dist * dynamic(3,:)
  dynamic(4,:) = dynamic(4,:) +
    ( static(2,:) - dynamic(2,:) ) * dist * static(3,:)
  dynamic(5,:) = dynamic(5,:) -
    ( static(1,:) - dynamic(1,:) ) * dist * static(3,:)
end do
if (mod(n,2) .eq. 0) then
  nrot = nrot + 1
  dynamic = CSHIFT(dynamic, 2, 1)
  dist = dynamic(3,:) / (max( .001,
    ( dynamic(1,:) - static(1,:) )
    * ( dynamic(1,:) - static(1,:) )
    + ( dynamic(2,:) - static(2,:) )
    * ( dynamic(2,:) - static(2,:) ) ) ) )
  static(4,:) = static(4,:) +
    ( dynamic(2,:) - static(2,:) ) * dist
  static(5,:) = static(5,:) -
    ( dynamic(1,:) - static(1,:) ) * dist
end if
static(4:5,:) = static(4:5,:) + CSHIFT(dynamic(4:5,:), 2, -nrot)
call cm_timer_stop( 0 )
print *, 'c-shift: time for ', n, ' bodies = ',
  cm_timer_read_elapsed(0)
return
end

```

III.B Using Spreads

Here, we consider a static and dynamic copy of the vortex elements, containing the position and strengths. For the full N -body algorithm, the dynamic copy is created by spreading each element from the fixed copy to form the entire dynamic copy. Between each SPREAD, we compute the influence of the dynamic copy on the static copy. After N such SPREADs, the complete interaction has been summed. The algorithm is given by:

The Full N -body Spread Algorithm

```

subroutine cs_1d( n )
integer n
real, array( 5, n ) :: static
real, array( n ) :: dist
real, array( 3, n ) :: dynamic
integer i
cmf layout static( :serial, :news )
cmf layout dynamic( :serial, :news )
cmf layout dist( :news )

```

```

static = 0.0
do i = 1, n
  static(1,i) = i
  static(2,i) = i + 1
  static(3,i) = 1.0
end do
call cm_timer_clear( 0 )
call cm_timer_start( 0 )
call cm_timer_stop( 0 )
call cm_timer_clear( 0 )
call cm_timer_start( 0 )
do i = 1, n
  dynamic(1,:) = spread(static(1,i),1,n)
  dynamic(2,:) = spread(static(2,i),1,n)
  dynamic(3,:) = spread(static(3,i),1,n)
  dist = dynamic(3,:) / ( max( .001,
    (dynamic(1,:) - static(1,:))
    * (dynamic(1,:) - static(1,:))
    + (dynamic(2,:) - static(2,:))
    * (dynamic(2,:) - static(2,:))) )
  static(4,:) = static(4,:) +
    (dynamic(2,:) - static(2,:)) * dist
  static(5,:) = static(5,:) -
    (dynamic(1,:) - static(1,:)) * dist
end do
call cm_timer_stop( 0 )
print *, 'broadcast: time for ', n, ' bodies = ',
  cm_timer_read_elapsed(0)
return
end

```

III.C Using Broadcasts

Here, we consider a static and dynamic copy of the vortex elements, containing the position and strengths. For the full N -body algorithm, the dynamic copy is created by broadcasting each element from the fixed copy to form the entire dynamic copy. Between each broadcast, we compute the influence of the dynamic copy on the static copy. After N such broadcasts, the complete interaction has been summed. The algorithm is given by:

The Full N -body Broadcast Algorithm

```

subroutine cs_1d( n )
integer n
real, array( 5, n ) :: static
real, array( n ) :: dist
real, array( 3, n ) :: dynamic
integer i
cmf layout static( :serial, :news )
cmf layout dynamic( :serial, :news )
cmf layout dist( :news )
static = 0.0
do i = 1, n
  static(1,i) = i
  static(2,i) = i + 1
  static(3,i) = 1.0
end do
call cm_timer_clear( 0 )
call cm_timer_start( 0 )
call cm_timer_stop( 0 )
call cm_timer_clear( 0 )
call cm_timer_start( 0 )

```

```

do i = 1, n
  dynamic(1,:) = static(1, i)
  dynamic(2,:) = static(2, i)
  dynamic(3,:) = static(3, i)
  dist = dynamic(3,:) / ( max( .001,
    ( dynamic(1,:) - static(1,:) )
    * ( dynamic(1,:) - static(1,:) )
    + ( dynamic(2,:) - static(2,:) )
    * ( dynamic(2,:) - static(2,:) ) ) )
  static(4,:) = static(4,:) +
    ( dynamic(2,:) - static(2,:) ) * dist
  static(5,:) = static(5,:) -
    ( dynamic(1,:) - static(1,:) ) * dist
end do
call cm_timer_stop( 0 )
print *, 'broadcast time for ', n, ' bodies = ',
  cm_timer_read_elapsed(0)
return
end

```

A different broadcast may be obtained by using the CMF array transfer utility function, known as a bit-blit, to send the entire CM array to the front end of the CM-2 (or control processor of the CM-5) at once, and then broadcast one body at a time as in the above. The following code gives the array transfer/broadcast:

The Bit-blit Full N -body Broadcast Algorithm

```

program bit-blit-broadcast
  integer n, iterations
  real fe_static( 5, 100000 )
  print *, 'bit-blit broadcast 1d'
  print *, 'Enter number of bodies'
  read *, n
  call bit-bc_1d( n, fe_static )
  stop
end

subroutine bit-bc_1d( n, fe_static )
  integer n
  real, array( 5, n ) :: static
  real, array( 5, 100000 ) :: fe_static
  real, array( n ) :: dist
  real, array( 3, n ) :: dynamic
  integer, array( 2 ) :: end
  integer i
cmf layout static( :serial, :news )
cmf layout dynamic( :serial, :news )
cmf layout dist( :news )
  static = 0.0
  do i = 1, n
    static(1,i) = i
    static(2,i) = i + 1
    static(3,i) = 1.0
  end do
  end(1) = 5
  end(2) = n
  call cm_timer_clear( 0 )
  call cm_timer_start( 0 )
  call cm_timer_stop( 0 )
  call cm_timer_clear( 0 )
  call cm_timer_start( 0 )
  call cmf_from_cm( fe_static, static, end )
  do i = 1, n

```

```

dynamic(1,:) = fe_static(1, i)
dynamic(2,:) = fe_static(2, i)
dynamic(3,:) = fe_static(3, i)
dist = dynamic(3,:) / ( max( .001,
.   (dynamic(1,:) - static(1,:))
.   * (dynamic(1,:) - static(1,:))
.   + (dynamic(2,:) - static(2,:))
.   * (dynamic(2,:) - static(2,:)))
static(4,:) = static(4,:) +
.   (dynamic(2,:) - static(2,:)) * dist
static(5,:) = static(5,:) -
.   (dynamic(1,:) - static(1,:)) * dist
end do
call cm_timer_stop(0)
print *, 'broadcast: time for ', n, ' bodies = ',
.   cm_timer_read_elapsed(0)
return
end

```

IV. "FILLS" TO AVOID MASKING IN CM FORTRAN.

IV. A. Fills and Cshifts

If the number of vortices is not exactly matched with the layout of the array on the CM, the compiler inserts garbage masking, which flags the processors that do not contain actual bodies. While this is transparent to the user, it unfortunately means that operations such as CSHIFT become significantly slowed while the garbage mask is checked. A simple fix is to determine the number of processors (here, processors means virtual processors) that the machine will allocate for a given number of bodies, and initially load the "extra" processors with bodies of zero strength. Adding these extra bodies of zero strength does not affect the forces calculated on the "live" bodies. We call this an "*N*-body algorithm with fill". First, we note that this approach does not require any extra memory, since those processors are automatically allocated during compile time. Second, by filling the "extra" processors with bodies of zero strength, the garbage masking is turned off, and hence the speed of such operations as CSHIFT is increased.

Determining the number of bodies to add to reach the size of array allocated during compilation depends on the particular machine. Let P be the number of processing nodes on the machine (by a processing node, we mean a floating point unit on a CM-2). A CM-2 will allocate an array with an across-processor axis of length a multiple of $4P$, while a CM-5 with vector units will allo-

cate an array with an across-processor axis of length a multiple of $8P$, and a CM-5 without vector units will allocate an array with an across-processor axis of length a multiple of P . The idea of an " N -body problem with fill" is to make sure that the number of bodies is always equal to an integral multiple of this across-processor axis length. For example, given 300 bodies, and a CM-2 with 256 processing nodes, garbage masking can be avoided by augmenting the 300 live bodies with 724 bodies of strength zero to bring the total number of bodies up to 1024. In this case, the compiler will allocate an across-processor axis of length 1024, which in practice means that the across-processor axis will have a serial component of dimension 4. It is unfortunate that CM Fortran does not currently allow the user to access this serial axis.

In order to add these extra "zero" strength bodies, we must be careful to provide an extra copy of the initial live bodies at the end of the static array to maintain the full interaction of the dynamic copy with the static copy. This idea is best illustrated through an example. Suppose we have five live vortices, labeled "A" through "E". The static copy must contain these live bodies plus an extra copy at the end, making a total of 10 bodies. Imagine for a moment a CM-2 with four floating point nodes. Given these 10 bodies, the compiler will allocate an across-processor axis of length 16, which once again in practice means that the across-processor axis will have a serial component of dimension 4. Thus, we must fill the static array as follows:

Initially

```
Dynamic:  A B C D E _ _ _ _ _ A B C D E
Static:   A B C D E _ _ _ _ _ _ _ _ _
```

After one CSHIFT

```
Dynamic:  A B C D E _ _ _ _ _ A B C D E
Static:   B C D E _ _ _ _ _ _ _ _ A
```

where the blanks denote processors without live bodies. We note that the set up of the extra bodies at the end of static array requires a cross-geometry move, and gathering the accumulated velocities

which are distributed among the two copies in the static array requires an additional two CSHIFTs and add. The program to accomplish this for a CM-2 is given below. Here we include the driver which calculates the size of the fill array, as well as the N -body kernel subroutine.

The N -body Cshift Algorithm with Fill

```

      program CSHIFT_1d_fill
      implicit none
      integer m, n, iterations, proc
#include <cm/CMF_defs.h>
#ifdef CM5
#include <cm/timer-fort.h>
#else
#include <cm/paris-configuration-fort.h>
#endif
      print *, 'CSHIFT 1d'
      print *, 'Enter number of bodies'
      read *, n
      proc = CMF_number_of_processors()
      m = 0
#ifdef CM5
      else
#endif
      if (2 * n .le. m) go to 2
      go to 1
      stop
      end

      subroutine cs_1d_fill_top( m, n )
      implicit none
      integer m, n
#include <cm/CMF_defs.h>
#ifdef CM5
#include <cm/timer-fort.h>
#else
#include <cm/paris-configuration-fort.h>
#endif
      real, array( 5, n ) :: static
      integer i
      cmf layout static( :serial, :news )
      do i = 1, n
         static(1,i) = i
         static(2,i) = i + 1
         static(3,i) = 1.0
      end do
      call cm_timer_clear( 0 )
      call cm_timer_start( 0 )
      call cm_timer_stop( 0 )
      call cs_1d_fill( m, n, static )
      return
      end

      subroutine cs_1d_fill( m, n, static )
      implicit none
      integer m, n
      real, array( 5, n ) :: static
      integer i
      real, array( 5, m ) :: big_static
      real, array( 3, m ) :: dynamic
      real, array( m ) :: dist
#include <cm/CMF_defs.h>
#ifdef CM5
#include <cm/timer-fort.h>
#else
#include <cm/paris-configuration-fort.h>
#endif
      cmf layout static( :serial, :news )
      cmf layout big_static( :serial, :news )

```

```

cmf layout dynamic(:serial, :news)
cmf layout dist(:news)
  call cm_timer_clear( 0 )
  call cm_timer_start( 0 )
  big_static = 0.0
  big_static(1:3, 1:n) = static(1:3, :)
  dynamic = big_static(1:3, :)
  big_static = big_static + cshift( big_static, 2, n)
  do i = 1, n - 1
    dynamic = CSHIFT(dynamic, 2, 1)
    dist = dynamic(3,:) / ( max( .001,
      ( dynamic(1,:) - big_static(1,:) )
      * ( dynamic(1,:) - big_static(1,:) )
      + ( dynamic(2,:) - big_static(2,:) )
      * ( dynamic(2,:) - big_static(2,:) ) ) )
    big_static(4,:) = big_static(4,:) +
      (dynamic(2,:) - big_static(2,:)) * dist
    big_static(5,:) = big_static(5,:) -
      (dynamic(1,:) - big_static(1,:)) * dist
  end do
  big_static(4:5,:) = big_static(4:5,:) +
    cshift( big_static(4:5,:), 2, -n)
  static(4:5, :) = big_static(4:5, 1:n)
  call cm_timer_stop( 0 )
  print *, 'c-shift: time for ', n, ' bodies (fill = ', m, ') = ',
    cm_timer_read_elapsed(0)
  return
end

```

The Triangular N -body Cshift Algorithm with fill is given below. The driver program is the same as the above.

The Triangle N -body Cshift Algorithm with Fill

```

      subroutine cs_1d_fill_top( m, n )
      integer m, n
      real, array( 5, n ) :: static
      integer i
cmf layout static(:serial, :news)
      call cmf_random( static(1:3,:), 1.0 )
      do i = 1, n
        static(1,i) = i
        static(2,i) = i+1
        static(3,i) = 1.0
      end do
      call cm_timer_clear( 0 )
      call cm_timer_start( 0 )
      call cm_timer_stop( 0 )
      call cs_1d_fill( m, n, static )
      return
      end

      subroutine cs_1d_fill( m, n, static )
      implicit none
      integer m, n
      real, array( 5, n ) :: static
      integer i, ntot
      real, array( 5, m ) :: big_static, dynamic
      real, array( 2, m ) :: cmfbugtemp
      real, array( m ) :: dist
#include <cm/CMF_defs.h>
#ifdef CM5
#include <cm/timer-fort.h>
#else
#include <cm/paris-configuration-fort.h>
#endif
cmf layout static(:serial, :news)
cmf layout big_static(:serial, :news)
cmf layout dynamic(:serial, :news)
cmf layout cmfbugtemp(:serial, :news)
cmf layout dist(:news)

```

```

call cm_timer_clear( 0 )
call cm_timer_start( 0 )
big_static = 0.0
big_static(1:3, 1:n) = static(1:3, :)
dynamic = big_static
big_static = CSHIFT( big_static, 2, n )
big_static = big_static + dynamic
nrot = (n - 1) / 2
do i = 1, nrot
  dynamic = CSHIFT(dynamic, 2, 1)
  dist = 1.0 / ( max( .001,
    (dynamic(1,:) - big_static(1,:))
    * (dynamic(1,:) - big_static(1,:))
    + (dynamic(2,:) - big_static(2,:))
    * (dynamic(2,:) - big_static(2,:)) ))
  big_static(4,:) = big_static(4,:) +
    (dynamic(2,:) - big_static(2,:)) * dist * dynamic(3,:)
  big_static(5,:) = big_static(5,:) -
    (dynamic(1,:) - big_static(1,:)) * dist * dynamic(3,:)
  dynamic(4,:) = dynamic(4,:) +
    (big_static(2,:) - dynamic(2,:)) * dist * big_static(3,:)
  dynamic(5,:) = dynamic(5,:) -
    (big_static(1,:) - dynamic(1,:)) * dist * big_static(3,:)
end do
if (mod(n,2) .eq. 0) then
  nrot = nrot + 1
  dynamic = CSHIFT(dynamic, 2, 1)
  dist = dynamic(3,:) / ( max( .001,
    (dynamic(1,:) - big_static(1,:))
    * (dynamic(1,:) - big_static(1,:))
    + (dynamic(2,:) - big_static(2,:))
    * (dynamic(2,:) - big_static(2,:)) ))
  big_static(4,:) = big_static(4,:) +
    (dynamic(2,:) - big_static(2,:)) * dist
  big_static(5,:) = big_static(5,:) -
    (dynamic(1,:) - big_static(1,:)) * dist
end if
cmfbugtemp = CSHIFT(big_static(4:5,:), 2, -n)
big_static(4:5,:) = big_static(4:5,:) +
  CSHIFT(dynamic(4:5,:), 2, -nrot) + cmfbugtemp
static(4:5, :) = big_static(4:5, 1:n)
call cm_timer_stop( 0 )
print *, 'c-shift: time for ', n, ' bodies (fill = ', m, ') = ',
  cm_timer_read_elapsed(0)
return
end

```

IV. B. Fills and Broadcasts/Spreads

In the same manner as above, fills can be used with both broadcasts and SPREADs to avoid garbage masking. There is no need to include the additional copy at the end of the static array, and thus we need only add zero-strength bodies to fill the static array up to the size allocated by the compiler. We do not include that code, since it looks almost identical to the previous codes.

V. CMMD CODE

The various CM Fortran techniques presented in previous sections share a common feature: they move the bodies as independent entities. This is a consequence of the limited control of data layout provided by the current (Version 2.0) versions of the compiler. Instead, one would like to

configure the vortex array as a two-dimensional array (in actuality, three-dimensional, since the x , y , c , u , v elements are included), where one axis is purely physical (across processors) and the other purely serial. In this case, both the CSHIFT and SPREAD algorithms could be modified to use block transfers which could then make use of vectorization capabilities. In this view, the bodies are distributed uniformly so that either (N/P) or $((N-1)/P)$ bodies reside on each physical processor. That is, the array subgrid which resides on each processor is two-dimensional. Computation then consists of an N -body solver among the bodies on a node interleaved with a communication step that moves all of the bodies (the entire serial axis) from one processor to the next. If the nodes are equipped with vector units, we may expect that the compiler will efficiently optimize (vectorize) the serial axis computations.

For a machine with P nodes, a CM Fortran program for the block move variant would look like:

```

      subroutine CSHIFT-block-triang( n, p, vortices )
      implicit none
      integer n, p
      real, array( 5, n/p, p ) :: vortices
      integer i, j, k
      real, array( 5, n/p, p ) :: dynamic
      real, array( n/p, p ) :: dist
      vortices( 4:5, :, : ) = 0.0
      dynamic = vortices
c     compute interactions of bodies residing on a processor
      do j = 1, n / p - 1
        do k = j + 1, n / p
          (
            ((dynamic( 1, j, : ) - vortices( 1, i, : )) *
            ((dynamic( 1, j, : ) - vortices( 1, i, : ))) +
            ((dynamic( 2, j, : ) - vortices( 2, i, : )) *
            ((dynamic( 2, j, : ) - vortices( 2, i, : )))
          )
          dynamic( 3, j, : ) * dist
          dynamic( 3, j, : ) * dist
        end do
c     block move and compute interactions between resident and visiting
c     vortices
      do i = 1, p / 2 - 1
        dynamic = CSHIFT( dynamic, 3, 1 )
        do k = j + 1, n / p
          (
            ((dynamic( 1, j, : ) - vortices( 1, i, : )) *
            ((dynamic( 1, j, : ) - vortices( 1, i, : ))) +
            ((dynamic( 2, j, : ) - vortices( 2, i, : )) *
            ((dynamic( 2, j, : ) - vortices( 2, i, : )))
          )
          dynamic( 3, j, : ) * dist
          dynamic( 3, j, : ) * dist
          vortices( 3, j, : ) * dist
          vortices( 3, j, : ) * dist
        end do
      end do
    end do
  
```

```

c    add the static and dynamic velocity accumulations.
      dynamic = CSHIFT( dynamic, 3, -p / 2 + 1 )
      vortices( 4:5, :, : ) = vortices( 4:5, :a, : ) + dynamic( 4:5, :, : )
      return
    end

```

The SPREAD Variant would look similar:

```

      subroutine spread-block-triang( n, p, vortices )
      implicit none
      integer n, p
      real, array( 5, n/p, p ) :: vortices
      integer i, j, k
      real, array( 3, n/p, p ) :: dynamic
      real, array( n/p, p ) :: dist
      vortices( 4:5, :, : ) = 0.0
      dynamic = vortices
c    compute interactions of bodies residing on a processor
      do j = 1, n / p - 1
        do k = j + 1, n / p
          (
            (((dynamic( 1, j, :) - vortices( 1, i, : )) *
              ((dynamic( 1, j, :) - vortices( 1, i, : )))) +
              (((dynamic( 2, j, :) - vortices( 2, i, : )) *
              ((dynamic( 2, j, :) - vortices( 2, i, : ))))
            )
            dynamic( 3, j, :) * dist
            dynamic( 3, j, :) * dist
          )
        end do
c    block move and compute interactions between resident and visiting
c    vortices
      do i = 1, p
        dynamic = spread( dynamic(:, :, i), 3, p )
        do k = j + 1, n / p
          (
            (((dynamic( 1, j, :) - vortices( 1, i, : )) *
              ((dynamic( 1, j, :) - vortices( 1, i, : )))) +
              (((dynamic( 2, j, :) - vortices( 2, i, : )) *
              ((dynamic( 2, j, :) - vortices( 2, i, : ))))
            )
            dynamic( 3, j, :) * dist
            dynamic( 3, j, :) * dist
          )
        end do
      end do
      return
    end

```

Note that the SPREAD version should be less efficient since it cannot take advantage of the triangular structure of the computation across SPREADs. That is, only one side of the force interaction is computed per communication step, so p steps must be performed. The CSHIFT version requires only $(p / 2) + 1$ steps.

It seems reasonable to expect that the CSHIFT version will be the optimal CM Fortran implementation once version 2.1 is available. This compiler will allow the explicit data layout described above and will also provide a degree of vectorization for computations on serial axes.

As an experiment, we coded a version of the block CSHIFT algorithm using CMMD, the message passing library available on the CM-5. Here, the CSHIFTs have been replaced by the explicit CMMD_send_and_receive calls. This code is given below.

The CMMD Code

```

program n_body
implicit none
integer alive, n, size
real static(5, 1), dynamic1(5, 1), dynamic2(5, 1)
integer byte_size, bytes_per_float, floats_per_body
integer src, dest, tag, floats_per_velocity
real extra
parameter (tag = 10)
parameter (bytes_per_float = 4)
character*(10) bodies
pointer (s, static), (d1, dynamic1), (d2, dynamic2)
external malloc
external getarg
external iargc
integer malloc
integer iargc
integer i, j, ps, result, swad, swas, t, v_size
#include <cm/cmmd_fort.h>
floats_per_body = 5
c result = ieee_handler('set', 'inexact', handler_in )
c result = ieee_handler('set', 'overflow', handler_ov )
c result = ieee_handler('set', 'underflow', handler_un )
ps = cmmd_partition_size()
if (iargc() .lt. 2) then
  print *, "Usage: n-body size procs"
  stop
end if
call getarg(1, bodies )
read( bodies, * ) size
call getarg(2, bodies )
read( bodies, * ) i
if (i .ne. 0) ps = i
print *, 'bodies = ', size, ' processors in computation = ', ps
print *, 'partition size is ', cmmd_partition_size()
n = size / ps
extra = float( size ) / float( ps ) - n
if (extra .ne. 0.0) then
  i = extra * cmmd_partition_size()
  alive = n
  if (cmmd_self_address() .lt. i) then
    alive = n + 1
  end if
  n = n + 1
end if
print *, 'bodies/processor = ', n, extra, i
byte_size = n * bytes_per_float * floats_per_body
c allocate arrays
s = malloc( byte_size )
d1 = malloc( byte_size )
d2 = malloc( byte_size )
c make standard error and output units independent across pns
result = cmmd_set_io_mode( 0, cmmd_independent )
result = cmmd_set_io_mode( 6, cmmd_independent )
c only do the calculation in the desired processors
if (cmmd_self_address() .lt. ps) then
c initialize velocities
do i = 1, n
  static(4, i) = 0.0
  static(5, i) = 0.0
end do
c initialize vortices
call initialize(static, dynamic1, n, alive )
result = cmmd_node_timer_clear( 0 )
result = cmmd_node_timer_clear( 1 )
result = cmmd_node_timer_start( 1 )
c now compute static interaction
call vortex_kernel_s( n, static, dynamic1, n )
c now cycle through the nodes
dest = cmmd_self_address() - 1
if (dest .eq. -1) then
  dest = ps - 1
endif
src = cmmd_self_address() + 1
if (src .eq. ps) then
  src = 0
endif
swad = mod( cmmd_self_address() + ps/2, ps )

```

```

      swas = mod( ps + cmmd_self_address() - ps/2, ps )
      t = 1
      do i = 1, ps/2 - 1
        t = 1 - t
        if (t .eq. 1) then
          result = cmmd_send_and_receive( src, tag, dynamic1,
            byte_size, dest, tag, dynamic2, byte_size )
          call vortex_kernel_sd( n, static, dynamic1, n )
        else
          result = cmmd_send_and_receive( src, tag, dynamic2,
            byte_size, dest, tag, dynamic1, byte_size )
          call vortex_kernel_sd( n, static, dynamic2, n )
        end if
      end do
c    Now do last pair. If ps is even, this is a one way calculation.
c    Otherwise to do the full bidirectional calculation.
      j = mod(ps, 2)
      t = 1 - t
      if (j .ne. 0) then
c    do the full case
        if (t .eq. 1) then
          result = cmmd_send_and_receive( src, tag, dynamic1,
            byte_size, dest, tag, dynamic2, byte_size )
          call vortex_kernel_sd( n, static, dynamic1, n )
        else
          result = cmmd_send_and_receive( src, tag, dynamic2,
            byte_size, dest, tag, dynamic1, byte_size )
          call vortex_kernel_sd( n, static, dynamic2, n )
        end if
      else
c    do the half case
        if (t .eq. 1) then
          result = cmmd_send_and_receive( src, tag, dynamic1,
            byte_size, dest, tag, dynamic2, byte_size )
          call vortex_kernel_s( n, static, dynamic1, n )
        else
          result = cmmd_send_and_receive( src, tag, dynamic2,
            byte_size, dest, tag, dynamic1, byte_size )
          call vortex_kernel_s( n, static, dynamic2, n )
        end if
      end if
c    now do send with add of velocity.
      if (t .eq. 0) then
        result = cmmd_send_and_receive( swas, tag, dynamic1,
          byte_size, swad, tag, dynamic2, byte_size )
        do i = 1, n
          static(4, i) = static(4, i) + dynamic1(4, i)
          static(5, i) = static(5, i) + dynamic1(5, i)
        end do
      else
        result = cmmd_send_and_receive( swas, tag, dynamic2,
          byte_size, swad, tag, dynamic1, byte_size )
        do i = 1, n
          static(4, i) = static(4, i) + dynamic2(4, i)
          static(5, i) = static(5, i) + dynamic2(5, i)
        end do
      end if
      result = cmmd_node_timer_stop( 1 )
    end if
    call cmmd_sync_with_nodes()
    if (cmmd_self_address() .eq. 0) then
      print *, 'total time = ', cmmd_node_timer_elapsed( 1 )
    end if
    stop
  end

  subroutine initialize( static, dynamic, bodies, alive )
    implicit none
    integer i, bodies, alive
    real dynamic( 5, bodies )
    real static( 5, bodies )
    integer offset
    external rand
    real rand
#include <cm/cmmd_fort.h>
    offset = cmmd_self_address() * bodies
    do i = 1, bodies
      static(1, i) = i + offset
      static(2, i) = i + 1 + offset

```

```

      if (i .le. alive) then
        static(3, i) = 1.0
      else
        static(3, i) = 0.0
      end if
      dynamic(1, i) = static(1, i)
      dynamic(2, i) = static(2, i)
      dynamic(3, i) = static(3, i)
      dynamic(4, i) = static(4, i)
      dynamic(5, i) = static(5, i)
    end do
    return
  end

```

```

subroutine vortex_kernel_sd( bodies, static, dynamic, alive )
implicit none
integer bodies, alive, index1, index2
real inverse_d
real dynamic(5, bodies)
real static(5, bodies)
do index1 = 1, alive
  do index2 = 1, alive
    inverse_d = 1.0 / max( .001,
      . ((dynamic(1, index2) - static(1, index1)) *
      . (dynamic(1, index2) - static(1, index1))) +
      . ((dynamic(2, index2) - static(2, index1)) *
      . (dynamic(2, index2) - static(2, index1)))
      static(4, index1) = static(4, index1) +
      . dynamic(3, index2) *
      . (dynamic(1, index2) - static(1, index1)) * inverse_d
      static(5, index1) = static(5, index1) -
      . dynamic(3, index2) *
      . (dynamic(2, index2) - static(2, index1)) * inverse_d
      dynamic(4, index2) = dynamic(4, index2) -
      . static(3, index1) *
      . (dynamic(1, index2) - static(1, index1)) * inverse_d
      dynamic(5, index2) = dynamic(5, index2) +
      . static(3, index1) *
      . (dynamic(2, index2) - static(2, index1)) * inverse_d
    end do
  end do
  return
end

```

```

subroutine vortex_kernel_s( bodies, static, dynamic, alive)
implicit none
integer bodies, alive, index1, index2
real inverse_d
real dynamic(5, bodies)
real static(5, bodies)
do index1 = 1, alive - 1
  do index2 = index1 + 1, alive
    inverse_d = 1.0 / max( .001,
      . ((dynamic(1, index2) - static(1, index1)) *
      . (dynamic(1, index2) - static(1, index1))) +
      . ((dynamic(2, index2) - static(2, index1)) *
      . (dynamic(2, index2) - static(2, index1)))
      static(4, index1) = static(4, index1) +
      . dynamic(3, index2) *
      . (dynamic(1, index2) - static(1, index1)) * inverse_d
      static(5, index1) = static(5, index1) -
      . dynamic(3, index2) *
      . (dynamic(2, index2) - static(2, index1)) * inverse_d
      dynamic(4, index2) = dynamic(4, index2) -
      . static(3, index1) *
      . (dynamic(1, index2) - static(1, index1)) * inverse_d
      dynamic(5, index2) = dynamic(5, index2) +
      . static(3, index1) *
      . (dynamic(2, index2) - static(2, index1)) * inverse_d
    enddo
  enddo
  return
end

```

VI. RESULTS

In Table 1, we give the results of timing runs to evaluate the various algorithms described above. We perform the CSHIFT, triangle-CSHIFT, SPREAD, and broadcast/broadcast-bit-blit algorithms, both with and without fill, for a 256 floating point node CM2, a 256 Sparc node CM-5, and a 128 pn (512 vector unit) CM-5. The times for the smaller problems are the average of ten runs. The fastest time in the table for 100,000 bodies is for the broadcast/bitblit on the 128 pn (512 VU) CM-5.

VII. OTHER ISSUES AND IMPLEMENTATIONS

It is important to note a seemingly minor change in coding style can make a major impact in timings. For example, in the implementation of the simple SPREAD algorithm, we note that it is unnecessary to spread the stored velocities from the static copy to the dynamic copy. Thus the static copy is of serial dimension 5, whereas the dynamic copy has serial dimension 3. In the implementation given in the text, the x and y positions and strength are each spread separately, that is,

```
dynamic(1,:) = spread(static(1,i),1,n)
dynamic(2,:) = spread(static(2,i),1,n)
dynamic(3,:) = spread(static(3,i),1,n)
```

However, if one were to replace this with the single SPREAD

```
dynamic = spread(static(1:3,i),1,n)
```

the resulting code is fifteen times slower. As another example, in the bit-blits, it makes considerable difference whether or not one bit-blits all or part of the array. For example, given the static array of serial length 5 containing x, y, c, u and v , one only need the x, y , and c components bit-blitted to the front end and then shipped to the CM to build the dynamic array. However, bit-blitting only 3 of the 5 elements causes garbage masking, which once again considerably slows the code. Other programming pitfalls abound.

Finally, the above techniques for programming direct N -body solvers is by no means exhaustive. We have attempted only to program some relatively straightforward implementations, and tried

Table 1: Comparative 2-D Direct N-body Kernel Timings (in seconds)

Alg. \ # Bodies	CM-2 256 Weiteks			CM-2 Fill 256 Weiteks			CM-5 Sparc 256 Nodes			CM-5 Sparc Fill 256 Nodes			CM-5 VU 128 nodes (512 VU)			CM-5 VU Fill 128 nodes (512 VU)		
	1 e3	1e4	1e5	1e3	1e4	1e5	1e3	1e4	1e5	1e3	1e4	1e5	1e3	1e4	1e5	1e3	1e4	1e5
CSHIFT	.996	60.72	2873	.169	7.13	610.0	.336	6.14	744.6	.304	6.46	505.2	.616	7.67	170.2	.970	8.90	227.4
CSHIFT TRI	.710	48.77	2300	.134	8.79	637.7	.245	4.58	637.3	.205	4.79	409.4	.317	4.48	113.3	.349	5.19	159.5
SPREAD	.125	3.96	325.5	.125	4.02	325.6	.236	4.28	246.3	.238	3.88	252.0	.292	5.82	91.01	.364	4.65	116.1
BROADCAST	.125	3.93	327.6	.125	4.03	325.5	.245	3.93	242.0	.238	4.32	252.5	.451	5.98	119.7	.282	5.76	110.7
BC - BITBLT	.065	3.39	307.0										.088	2.11	66.53			
CMMD							.040	1.14	105.7									

to examine how the coding affects the efficiency. Some other techniques might include:

- (1) Use global communication to replace CSHIFTS/broadcasts/SPREADs (most probably considerably slower).
- (2) Place the bodies in an across-physical axis, transpose, and then SPREAD to get all N bodies on each node. Then loop through serial axis in each node, performing the N -body interaction with the body located in that node.
- (3) Place the bodies in an across-physical axis, transpose, and then SPREAD to get all N bodies on each node. Then compute the interaction matrix which gives the distance between body I and J, and then use a matrix-vector multiply to compute the interactions.
- (4) Bit-blit all the bodies to the front end at the beginning, and then broadcast a section of the bodies to each node. This may be especially useful if/when the Fortran compiler performs vectorization along the serial axis.

Acknowledgements: We wish to acknowledge the contributions of Woody Lichtenstein.

References

- 1) Brunet, J-Ph., Edelman, A., and Mesirov, J.P., *An optimal hypercube direct N-body solver on the Connection Machine*, Proceedings of Supercomputing '90, IEEE Computer Society Press (1990), 748--752
- 2) Sethian, J.A., *A Brief Overview of Vortex Methods*, in Vortex Methods and Vortex Motion, Eds. K. Gustafson and J.A. Sethian, SIAM Publications, Philadelphia, 1991.
- 3) Sethian, J.A., J.P. Brunet, A. Greenberg, J. Mesirov, *Two-Dimensional, Viscous, Incompressible Flow on a Massively Parallel Processor*, J. Comp, Phys. **101**, 1, pp. 185-206 (1992).

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