Parallel Particle Simulation On The TC2000 And CM2 *

Timothy J. Williams, Y. Matsuda, and E. Boerner

Massively Parallel Computing Initiative
Lawrence Livermore National Laboratory
Livermore, California 94550

Appeared in
The 1991 MPCI Yearly Report: The Attack of the Killer Micros,
UCRL-ID-107022, p. 133.

*This work performed by LLNL under DoE contract No. W–7405–ENG–48.

Parallel Particle Simulation On The TC2000 And CM2 *

Timothy J. Williams, Y. Matsuda, and E. Boerner

Massively Parallel Computing Initiative
Lawrence Livermore National Laboratory
Livermore, California 94550

Abstract

Gyrokinetic particle-in-cell (PIC) simulation is an important tool for studying low-frequency ($\omega \ll \omega_{ci}$) plasma instabilities and the associated transport of particles and heat in tokamaks. In spite of the efficiency gained by not following ion cyclotron motion, more efficiency is needed in performing simulations on a transport timescale: using 1 million particles in a system of $64 \times 128 \times 32$ grid cells, a 3D code took about 100 hours on the Cray2 (using one processor) to simulate the η_i mode for 3 to 4 wave periods. In order to run larger and longer simulations, we pursue efforts to create parallel gyrokinetic PIC codes.

We study parallel PIC algorithms for the BBN TC2000 computer. We use the Parallel Fortran Preprocessor (PFP) and Parallel C Preprocessor (PCP) [1], which are parallel enhancements to Fortran and C based on the efficient split-join programming model [1]. We report progress on using PFP in a conventional 1D PIC code. We report on studies of a parallel sorting algorithm and its incorporation into parallel PIC codes.

We also convert a 3D electrostatic gyrokinetic slab code[1] to run on the Thinking Machines CM2 computer. We report on the code conversion and efficiency gains and contrast the effort with that on the BBN TC2000.

*This work performed by LLNL under DoE contract No. W–7405–ENG–48.

[1]Developed by R. D. Sydora with partial support from W. W. Lee.

1 Goal; Methods

1.1 Goal: Simulation of Fusion Plasma Turbulence Using PIC Codes

Particle-in-cell (PIC) codes simulate plasmas using superparticles moving under self-consistent electromagnetic fields defined on a spatial grid. Simulation of turbulent plasma phenonema and the associated transport of energy and particles requires *lots* of particles and grid cells. For example, 3D gyrokinetic simulation of a physically useful section of a large tokamak may require as many as 8×10^9 particles and 8×10^5 grid cells [11]. Since simulations of $O(10^6)$ ions (Boltzmann electrons) and $O(2 \times 10^5)$ grid cells have taken as long as $O(100)$ hours on the Cray2, it is clear that the Cray2 is not fast enough for the large problems we want to run. The goal of this research is to exploit parallelism to run these big problems within the 10–20 hours of CPU time we consider practical. That is, favorable scaling with increased problem size is what we need.

Much of the PIC method is embarassingly parallel. For example, once particle velocities are computed, all particles can theoretically be pushed simultaneously. There is already some history of success in parallel PIC code building; see, for example, Liewer, *et. al.* [2][3][4]—message passing conventional PIC codes on MIMD machines; and J. Reynders & W. W. Lee [5]—gyrokinetic PIC codes on the CM2 SIMD machine.

1.2 Two Methods Of Attack

We undertake two paths of approach to the goal of simulating physically interesting fusion plasma turbulence problems. Both methods have as their goal the construction of scalable, parallel gyrokinetic PIC algorithms.

1.2.1 Method 1: BBN TC2000

The BBN TC2000 machine has been shown to be able to compete with Cray2 speeds on some physics simulation codes [16]. As stated earlier, however, Cray2 speeds are not enough for the simulations we want to do, so we view the TC2000 as a testbed for preparation of algorithms to run on future machines which are bigger and faster. We use parallel programming paradigms which we assume will be available on future machines and perhaps portable among different types of machines. If the 126-processor TC2000 at LLNL proves to be significantly faster than the Cray2 for our particular code, then we will of course also consider running actual physics simulations there.

1.2.2 Method 2: Thinking Machines CM2

Reynders and Lee [5] have demonstrated a 2D gyrokinetic PIC code on the CM2 which is 10 times as fast as their Cray2 code and probably at least 6 times as fast as our slightly more optimized Cray2 code. This speed is enough to do some new physics simulations now which wouldn't be practical on the Cray2. Our plan for the CM2 is to use a 3D slab gyrokinetic PIC code to do some physics simulations as soon as possible. The porting of an existing Cray2 code to the CM2 is nearly completed.

1.3 Machine horsepower comparison

The NERSC Cray2 computers have been our gyrokinetic PIC simulation machines for some time now, and we have invested considerable effort into vectorization and development of new algorithms to optimize our codes there [12][13][14]. The Cray2 is the base machine we compare others with. To give an idea about the relative capabilities of the three machines we are using, here are some descriptive numbers:

- Reference point—Cray2: 4 processors
  - 1 GByte memory
  - ∼2 GFLOPS peak
    [based on ∼0.5 GFLOPS per processor]
- BBN TC2000: 63 (126) processors
  - 1 GByte memory
  - ∼1.26 (2.52) GFLOPS peak with 32-bit arithmetic
    [based on ∼20 MFLOPS per node]
  - ∼0.63 (1.26) GFLOPS peak with 64-bit arithmetic
    [based on ∼10 MFLOPS per node]
  - MIMD, shared memory (switch-access time penalty of $\sim 3\times$)
- Thinking Machines CM2: 64k processors
  - 2-8 GByte memory
  - ∼32 GFLOPS peak with 32- or 64-bit arithmetic
    [based on 2048 floating-point processors at 16 MFLOPS each]
  - SIMD, distributed memory

These numbers further emaphasize that the TC2000 is a research testbed for future machines, while the CM2 is fairly big and fast in its current form.

2 Research On BBN TC2000

Our current project on the TC2000 is algorithm research using the 1D PIC code ES1 [15]. The lessons learned from this work are to be applied to a port of the 3D gyrokinetic code from the Cray2. Our basic approach is to use the PCP/PFP extensions to C/Fortran to modify the raw Fortran codes to run in parallel on the TC2000.

The Parallel C Preprocessor and Parallel Fortran Preprocessor (PCP and PFP) [1][6] are implementations of the *split-join* parallel programming paradigm which are descended from SPMD model [7] and The Force [8]. In this paradigm, a fixed-CPU-count *team* of processors enters the code; all members execute the code unless instructed otherwise (via PCP/PFP embedded statements). The team has a *master* processor, which can be specifically accessed for scalar code blocks or other special purposes. The team can be *split* to do independent code blocks or to execute the same code on independent data. Finally, and importantly, *all these operations are nestable.* This paradigm is conceptually much the same as fork-join paradigms. In practice it is distinct from fork-join models because it is much more efficient, and because existing fork-join implementations are generally not nestable.

To use PCP/PFP, one inserts special kewords and control statements into his plain C/Fortran code. These are preprocessed and the output is fed into the TC2000 C/Fortran compiler. The number and variety of these extra coding constructs is limited, making the system relatively easy to program with. To run the codes, one specifies the fixed number of processors in the team which enters the code before the execution begins.

2.1 Easy Problem: Parallel Particle Push In ES1

This example from ES1.PFP shows a couple of PFP modifications. The `doall` loop replaces an ordinary Fortran loop. The `barrier` statement forces each processor in the team to wait there they all arrive at that point.

```
doall i=1,nparticles
  x(i) = x(i) + vx(i)
  if (x(i) .lt. 0.0) x(i) = x(i) + xn
  if (x(i) .ge. xn)  x(i) = x(i) - xn
enddoall
barrier
```

This loop shows one of the embarassingly parallel parts of a PIC code. The `x` array stores the positions of all the particles; the `vx` array stores their x-velocities. Having computed all the elements of `vx`, there is no reason why every iteration of this loop over the elements of `x` and `vx` cannot be done simultaneously. This, the "push" part of the ES1 PIC algorithm, is easy to parallelize. The difficult parts are (1) the accumulation of scalar charge density onto the spatial grid on which the fields are defined, (2) the computation of the forces at the particle positions, and (3) the FFT algorithm used to solve the field equations. An experimental method for parallelizing (1) is detailed in the remaining sections on TC2000 research.

2.2 Hard Problem: Parallel Charge Accumulation In ES1

The accumulation of scalar charge density in ES1 proceeds as follows: The spatial domain of the simulation is mapped onto a 1D grid. The charge density is stored in an array `rho` having one element for each cell in the grid. To do the charge accumulation, each particle's position is examined and the particle contributes a weighted amount to each of its nearest neighboring grid points in the `rho` array:

```
do i=1,nparticles
  j = x(i)
  drho = qdx*(x(i) - j)
  rho(j)   = rho(j) - drho+qdx
  rho(j+1) = rho(j+1) + drho
enddo
```

There is clearly a data dependency in this charge accumulation loop, since different loop iterates may try to grab and update the same element of `rho`. Simply changing `do` to `doall` would not give correct answers.

However, because the nearest-grid-point interpolation scheme affects only two `rho` elements at a time, a parallel update of every other cell would be legal if there were some way to select particles whose positions were located in every other cell. If the `x` array were sorted, particles could be accumulated from every other cell in a deterministic fasion. Particle sorting has been used for vectorizing PIC codes [9]. On the TC2000, it makes sense to use a *parallel* sort. A description of a parallel sort we have implemented on the TC2000 follows.

2.2.1 Parallel Quicksort Using Pcp

The Quicksort sorting algorithm [10] is elegant and efficient for many kinds of data. The basic idea is to select a pivot element from the array to be sorted and then divide the array into two segments, one of which contains only elements less than the pivot value and the other of which contains only elements greater than the pivot value. Then this procedure is repeated on each of the two segments, and so on until the array is sorted. Figure 1 illustrates the procedure.

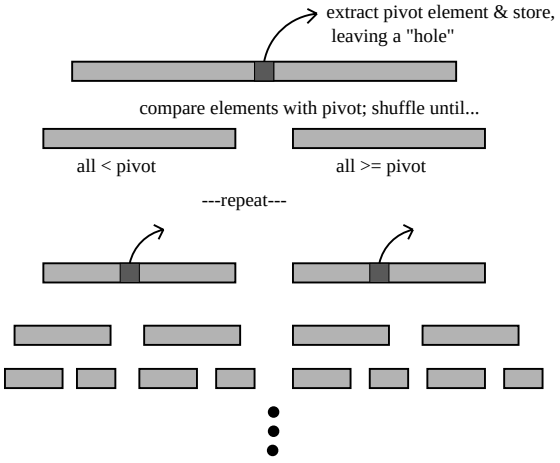

Figure 1: Flow of Quicksort algorithm.

The potential parallelism in this algorithm is apparent: all of the disconnected segments at any stage can be sorted simultaneously. We have written a PCP routine to do a parallel Quicksort. The basic program flow is as follows:

1. each array-division is done by scalar processing (by the team master processor)

2. after division, *split* the team and give half to each subteam

3. do this recursively to fully sort the array:

The following code segment is the heart of the algorithm:

```
/* The array to be sorted is a global array. It is
   not explicitly referenced in this routine */

void sort(LeftElement, RightElement)
     int LeftElement, RightElement;
{
  int MidElement;

  /* Divide the current array section about a pivot: */
  MidElement = DivideOrSort(LeftElement,RightElement);
  barrier;
  MidElement = TeamMidElement;

  /* If sort is not finished yet: */
  if (MidElement != -1)
    {
      split
        {
          /* Give one part to one subteam: */
          if (MidElement - LeftElement > 1)
             sort(LeftElement,MidElement-1);
        }
      and
        {
          /* Give other part to the other subteam: */
          if (RightElement - MidElement > 1)
             sort(MidElement+1,RightElement);
        }
    }
}
```

A timing study of this algorithm is shown in Figure 2. The data to be sorted was generated by putting random fluctuations on ordered data; this was designed to mimic particle-position data arrays which are nearly sorted and uniformly distributed into grid-cell domains, as is the case in the physics simulations of interest to us. The CPU time for sorting a fixed-size array scales like $1/\sqrt{N_{\rm processors}}$.

2.2.2 Use Of Parallel Quicksort For Charge Accumulation In ES1

To use the parallel Quicksort routine to parallelize the charge accumulation in ES1, we have installed extra coding to:

1. maintain an integer array of particle labels which is the only array in the main code whose elements are actually shuffled in the sort
2. call the (PCP) parallel sort routine from the ES1.PFP
3. accumulate in parallel the charge density contributions from all the particles in even-numbered grid-cells, then do the same for the particles in odd-numbered grid cells

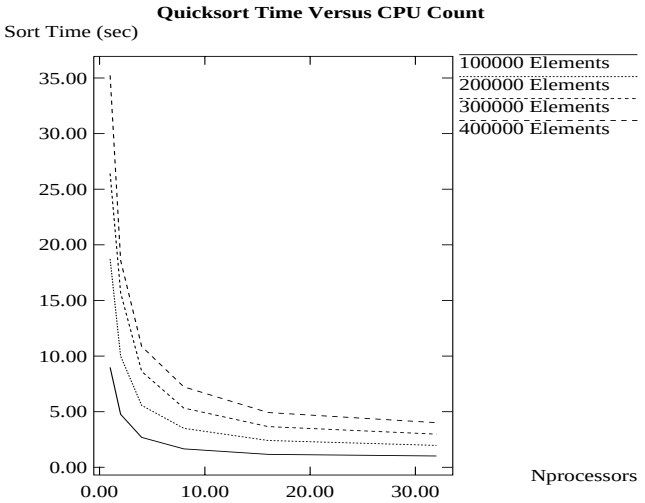

Figure 2: TC2000 Timings for the parallel Quicksort algorithm for various array sizes.

It is important to note that the particle positions need not be *fully* sorted, since we only need to know which grid cell each particle belongs to. Sorting the particles within the cells is unnecessary. The algorithm we constructed takes this into account by first determining the cell containing each particle and storing these numbers in an integer array, which is then sorted. We have generalized this procedure to work for 2D and 3D simulations as well.

Figure 3 shows TC2000 timings for the charge accumulation part of ES1.PFP using two different methods. One method is simply to put locks around the charge density array accesses in the accumulation loop:

```
shared lock rholock
...
doall i=1,nparticles
  j = x(i)
  drho = qdx*(x(i) - j)
  lock(rholock)
    rho(j)   = rho(j) - drho+qdx
    rho(j+1) = rho(j+1) + drho
  lock(rholock)
enddoall
```

This prohibits simultaneous updates of more than one element of `rho`, but allows parallel computation of values of the increment `drho` for many particles. This method is actually the best method for small simulations and for small numbers of processors, but Figure 3 shows that it becomes less efficient as the number of processors increases (presumably because of butterfly switch contention in the TC2000, though this hasn't been proved yet). The second method for charge accumulation is the particle-sorting scheme described ear-

lier. Figure 3 shows that although this method starts out with lots of overhead computation relative to the other at low CPU numbers, it improves dramatically with increased CPU numbers, and eventually overtakes the lock method for problems as large as the one pictured.

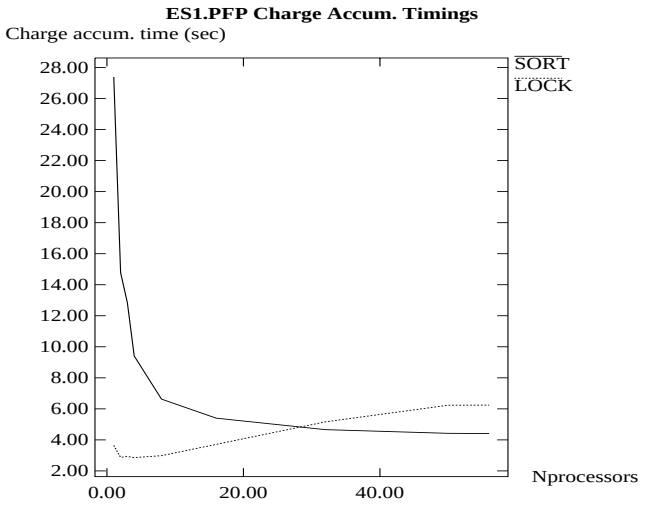

Figure 3: TC2000 timings for the charge accumulation part of ES1.PFP using the simple lock method and using the parallel Quicksort algorithm to sort the particles into grid cells then accumulate charge density contributions in parallel. This simulation had 128K particles and 2K grid cells

2.3 Remaining Problem: Data Localization And Caching

So far we have paid no attention at all to the possibly very important issues of data locality and caching on the TC2000; all of the particle arrays and grid arrays are simply stored in plain shared interleaved memory and directly accessed through whatever switch path is necessary. If the operations in the parallel sort-based charge accumulation, for example, could be performed with cached on-board memory references, an overall speed improvement by as much as a factor of ten might be achieved. Indeed, unless this type of further improvement in this part of ES1.PFP is successful, we haven't gained much for all the contortions necessary to implement the parallel charge accumulation. (Compare absolute times for high-CPU-count runs in Figure 3.) The parallel accumulation (and corresponding force computation) algorithms result in organized data access patterns which we hope can be more easily mapped onto the most efficient local cached memory accesses on the TC2000. This is the next stage of our research on this machine, which is now underway in tandem with our beginning to port the 3D gyrokinetic code using the ES1 port as a template.

3 Research On CM2

Our current project on the Thinking Machines CM2 is porting a 3D gyrokinetic code of Sydora *et.al.* from Cray2; this code is an electrostatic slab code with adiabatic electrons. Our basic approach is to use CM Fortran, which is based largely on Fortran 90 syntax.

To use the CM2 efficiently, one must insure that data arrays are stored and processed on the Connection Machine rather than the front end machine. To do this, one generally must remove all references to specific array elements and replace loops over elements with Fortran 90 syntax. As an example, consider part of the coding to enforce periodic particle boundary conditions on the gyrokinetic simulation particles:

```
do i=1,nparticles
  if (x(i) .lt. ancxl) x(i) = x(i) + boundx
  if (x(i) .gt. ancxr) x(i) = x(i) - boundx
enddo
```

Here `x` is an array of size `nparticles` which stores x-positions of the particles; `ancxl`, `ancxr`, and `boundx` are constants related to x-boundaries of the simulation box. In CM Fortran, the explicit loop over particles is eliminated and the if-tests are done via the `where` syntax:

```
where (x .lt. ancxl) x = x + boundx
where (x .gt. ancxr) x = x - boundx
```

Here the scalars `ancxl`, `ancxr`, and `boundx` are broadcast to all nodes on the CM2 and the if-tests and assignments of all elements of the array `x` occur simultaneously.

As was the case with ES1 on the TC2000, the particle-push part of the algorithm is embarassingly parallel and easy to make run in parallel on the CM2. The charge accumulation likewise is more difficult in theory, but in practice is handled easily using high-level gather/scatter library functions available on the CM2. The availability of library functions to perform this commonly-needed task makes porting PIC codes to the CM2 much easier than porting to the TC2000.

There are also efficient library routines available to perform 3D complex-to-complex parallel FFT's on the CM2. Here again, the availability of this library function greatly helps in porting PIC codes (and other codes requiring FFT's) to the CM2. (No library FFT routines of any kind are available on the TC2000).

The charge accumulation, force calculation, particle push, and field-equation solve (using FFT's) for

the 3D gyrokinetic code now run in parallel on the CM2. The FFT's were the most recent modification (previously, they were done by scalar processing on the CM2's front end). Timings for the code with parallel FFT's are not yet completed; Figure 4 timings are for scalar (front-end) FFT's. Notice that the CM2 utilization (ratio of CM2 time to elapsed execution time) was about 30 to 50% at that stage of conversion.

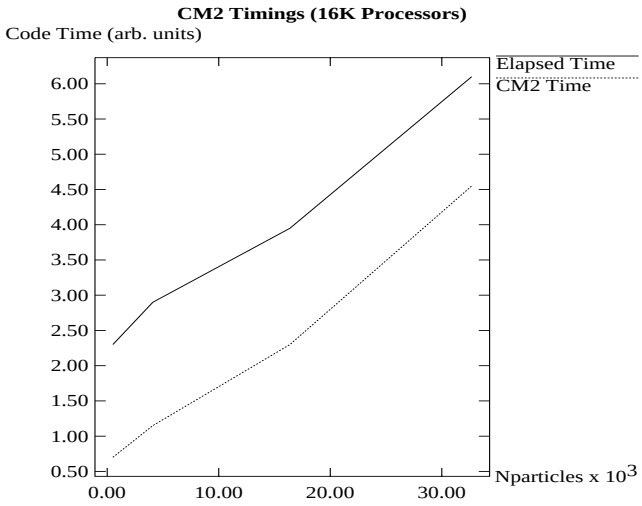

Figure 4: CM2 timings for the 3D slab gyrokinetic code with an $8 \times 8 \times 8$ field grid using 16K-processors. This is *before* the implementation of the parallel FFT routines.

4 Summary

Having done some studies of sticky PIC parallelization issues using ES1.PFP on the TC2000, we are ready to refine the memory managment of that code to better capitalize on the gains made by parallelization. The results of the studies, meanwhile, will be applied to porting the 3D code to the TC2000. Also, we will investigate parallel FFT methods to finish parallelizing the field-solve part of ES1.PFP (which is only a few percent of the total execution time and a reasonable choice for the part to leave for last). We are now working on reconstructing our Cray2 diagnostics and graphics capabilities in the CM2 port of the 3D code in preparation for physics simulations.

References

[1] Eugene D. Brooks III, UCRL-99673, LLNL (1988).

[2] P. C. Liewer and V. K. Decyk, J. Comput. Phys. **85**, 302 (1989).

[3] R.D. Ferraro, P. C. Liewer, V. K. Decyk, and J. M. Dawson, in UCRL-JC-104774 (Proceedings of Workshop On Massively Parallel Computing In Magnetic Fusion Energy, April 1990).

[4] R. W. Huff and J. M. Dawson, in UCRL-JC-104774 (Proceedings of Workshop On Massively Parallel Computing In Magnetic Fusion Energy, April 1990).

[5] J. V. W. Reynders and W. W. Lee, Proc. 13th Conf. Num. Sim. Plasmas, Santa Fe, "Comparison of a 2-D Finite-β Gyrokinetic Simulation in Slab Geometry on Serial and Parallel Computers,"(Sept. 1989).

[6] PFP authors: E. D. Brooks III and K. Warren, LLNL.

[7] *e.g.,* see A. H. Karp, "Programming for Parallelism," Computer **20**, 43 (1987).

[8] H. F. Jordan *et.al.*,"The Force: A Highly Porable Parallel Processing Language," Proc. 1989 International Conference on Parallel Processing, IEEE **II**, 112 (1989).

[9] E. J. Horowitz, J. Comput. Phys. **61**, 519 (1985).

[10] W. H. Press *et.al.*, *Numerical Recipes in C*, (Cambridge, New York, 1990).

[11] R. H. Cohen, B. I. Cohen, and P. F. Dubois, UCRL-ID-105650.

[12] T. J. Williams, Y. Matsuda, and E. Boerner, Bull. Am. Phys. Soc., **35**, 2000 (1990).

[13] T. J. Williams, B. I. Cohen, and R. D. Sydora, Proc. 13th Conf. Num. Sim. Plasmas, Santa Fe, "Improved Gyrokinetic Particle Simulation Techniques," (Sept. 1989).

[14] B. I. Cohen and T. J. Williams, UCRL-JC-103117, accepted for publication by J. Comput. Physics (1990).

[15] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation* (McGraw-Hill, New York, 1985).

[16] M. G. McCoy, G. D. Kerbel, R. W. Harvey, Bull. Am. Phys. Soc., **35**, 2000 (1990).