TITLE: PARELLEL BEAM DYNAMICS CALCULATIONS ON HIGH PERFORMANCE COMPUTERS

AUTHOR(S): Robert Ryne, AOT-1
Salman Habib, T-8


DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, 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 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. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
PARALLEL BEAM DYNAMICS CALCULATIONS
ON HIGH PERFORMANCE COMPUTERS

Robert Ryne* and Salman Habib†

*Accelerator Operations and Technology Division and †Theoretical Division
Los Alamos National Laboratory, Los Alamos, NM 87545

Abstract. Faced with a backlog of nuclear waste and weapons plutonium, as well as an ever-increasing public concern about safety and environmental issues associated with conventional nuclear reactors, many countries are studying new, accelerator-driven technologies that hold the promise of providing safe and effective solutions to these problems. Proposed projects include accelerator transmutation of waste (ATW), accelerator-based conversion of plutonium (ABC), accelerator-driven energy production (ADEP), and accelerator production of tritium (APT). Also, next-generation spallation neutron sources based on similar technology will play a major role in materials science and biological science research. The design of accelerators for these projects will require a major advance in numerical modeling capability. For example, beam dynamics simulations with approximately 100 million particles will be needed to ensure that extremely stringent beam loss requirements (less than a nanoampere per meter) can be met. Compared with typical present-day modeling using 10,000-100,000 particles, this represents an increase of 3-4 orders of magnitude. High performance computing (HPC) platforms make it possible to perform such large scale simulations, which require tens of GBytes of memory. They also make it possible to perform smaller simulations in a matter of hours that would require months to run on a single processor workstation. This paper will describe how HPC platforms can be used to perform the numerically intensive beam dynamics simulations required for development of these new accelerator-driven technologies.

Introduction

Many countries are now involved in efforts aimed at developing high power linacs for transmutation of radioactive waste, disposal of plutonium, production of tritium, and as drivers for next-generation spallation neutron sources. For these projects, high-resolution modeling far beyond that which has ever been performed in the accelerator community will be required to reduce cost and technological risk, and to improve accelerator efficiency, performance, and reliability. Such accelerators will have to operate with extremely low beam loss (0.1-1 nA/m) in order to prevent unacceptably high levels of radioactivity. High resolution simulations using on the order of 100 million particles will be needed to help ensure that this requirement can be met. With the advent of HPC platforms such as massively parallel processors (MPPs) and clusters of shared memory processors (SMPs) such simulations are now possible. Near term HPC platforms will have memories of 100’s of GBytes.
and performance of a few TFLOPs. Compared with high-end workstations (500 MFLOPs) and high-end PCs (100 MFLOPs), a 1 TFLOP HPC platform would outperform these by factors of 2000 and 10000, respectively.

Though HPC platforms have great potential, it is a challenge to use them effectively on a wide variety of problems. Some algorithms can only be parallelized with significant effort, while for some other problems it might be necessary to find a different method of solution, involving a different algorithm, that is more amenable to parallel computation. In general, care must be taken to ensure that all processors do roughly the same amount of work (the load balance issue), and that interprocessor communication does not seriously affect performance. Another challenge is to program parallel computers. Message passing libraries are quite mature and portable; also, languages such as High Performance Fortran (HPF) are maturing and becoming more widely available. But software challenges remain, particularly if platforms with memory hierarchies such as clusters of SMPs emerge as the most widespread HPC platforms, as many have predicted.

Approaches to Parallelization

There are two main approaches to using parallel computers. In the data-parallel approach, the style of programming is similar to that used in sequential computing, i.e. the programmer takes an essentially global view of the system and uses a single thread of control. This style of programming is well-suited to the Single-Instruction, Multiple-Data (SIMD) and Single-Program, Multiple-Data (SPMD) paradigms, in which all processors run the same program, synchronously or asynchronously, with their own data. Data-parallel programming is implemented using languages similar to traditional ones, but with extensions for explicitly parallel operations and directives. An example is High Performance Fortran (HPF). HPF includes Fortran 90 as a subset, but it also contains new commands such as the FORALL statement (which is a parallel DO) and compiler directives such as distribute (for distributing data across processors). For problems that are "embarrassingly parallel," such as modeling the dynamics of a number of noninteracting particles, the SIMD/SPMD approach is the natural choice. But, as will be seen later, purely data-parallel programming is extremely limiting, and to be useful for many problems it must be augmented with utility libraries and scientific software libraries.

A second approach to parallelization makes use of message passing libraries. In this approach the programmer takes a more local view of the problem, telling processors when to send and receive data, when to synchronize, etc. This style of programming accommodates the Multiple-Instruction, Multiple-Data (MIMD) paradigm. Though more difficult to use than the data-parallel approach, MIMD-style programming provides the flexibility needed to deal with problems that are not suited to the data-parallel model. Also, it often results in excellent performance.

It is possible to use a combination of these two approaches. For example, one can call non-HPF routines from inside an HPF program using a feature called HPF..EXTRINSIC. This style of parallel programming could become popular in the future since it allows people with traditional programming experience to work
In a familiar environment that also allows them to call message-passing-based software, written by themselves or others, to deal with those parts of a calculation that are not amenable to data-parallel programming.

In the remainder of this paper we begin by presenting two simple examples. The first describes a way to run multiple copies of a single particle beam transport code on a parallel computer, and the second shows a data-parallel implementation of a drift subroutine. After these two examples, we discuss the need to perform internal data processing prior to writing the results of large scale parallel simulations. The remaining sections describe progress in developing Particle-In-Cell and Vlasov-Poisson codes for modeling intense beams on parallel computers.

**Single Particle Tracking to Predict Dynamic Aperture**

Dynamic aperture is an important consideration for large hadron colliders. One way to predict this is to separately track hundreds to thousands of particles for several million turns (assuming the particles do not interact with one another), and see whether they survive or strike the beam pipe during the simulation. This can be done by running a single-particle tracking code on several independent computers, or by using a parallel computer. In the latter case, an obvious approach would be to use message passing. A second, more tedious approach, applicable to codes that track many (non-interacting) particles simultaneously, would be to use data-parallel programming, replacing DO loops over particles in all the subroutines with \texttt{FORALL} statements. Within the confines of HPF a third approach is also possible that is extremely easy to implement and requires no knowledge of message passing. Consider a code called \texttt{MYTRACK} that tracks a single particle. One could write a main program using HPF that calls \texttt{MYTRACK(V)} as a subroutine, where \texttt{V} denotes the 6-vector of coordinates and momenta of the particle to be tracked. In addition to array declarations, compiler directives, etc., the main program would contain the following loop:

\begin{verbatim}
!HPF$ INDEPENDENT
    do N=1,NPARTICLES
        call MYTRACK(V(I))
    end do
\end{verbatim}

The \texttt{INDEPENDENT} directive states that all iterations of the DO loop are independent of one another. The main program would also require an \texttt{INTERFACE} block specifying that \texttt{MYTRACK} is a “pure” subroutine, i.e. one having no side-effects.

**Data-Parallel Implementation of a Drift Subroutine**

Consider a subroutine to propagate the coordinates and momenta, of 100 million particles in a drift space. A crucial consideration, as with any parallel program, is the distribution of the data across processors. In this case the coordinates and momenta of a given particle should be on the same processor so that the above transformation
requires no interprocessor communication. Suppose the data is stored in an array dimensioned \(a(6,100000000)\). The HPF compiler directive \texttt{distribute a(*,block)} would cause quantities specified by the first dimension to reside on the same processor, as desired, while the quantities specified by the second dimension would be stored in “blocks” across processors. For example, if 4 processors were in use, then the six coordinates of particles 1-25000000 would reside on processor 1; particles 25000001-50000000 would reside on processor 2; and so on. See Table 1.

Table 1: Layout of a two dimensional array distributed (*,block)

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
<th>Processor 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a(1-6,1))</td>
<td>(a(1-6,2))</td>
<td>(a(1-6,3))</td>
<td>(a(1-6,4))</td>
</tr>
<tr>
<td>(a(1-6,2))</td>
<td>(a(1-6,25000001))</td>
<td>(a(1-6,50000001))</td>
<td>(a(1-6,75000001))</td>
</tr>
<tr>
<td>(a(1-6,3))</td>
<td>(a(1-6,25000002))</td>
<td>(a(1-6,50000002))</td>
<td>(a(1-6,75000002))</td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>(\vdots)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(a(1-6,24999999))</td>
<td>(a(1-6,49999999))</td>
<td>(a(1-6,74999999))</td>
<td>(a(1-6,99999999))</td>
</tr>
<tr>
<td>(a(1-6,25000000))</td>
<td>(a(1-6,50000000))</td>
<td>(a(1-6,75000000))</td>
<td>(a(1-6,100000000))</td>
</tr>
</tbody>
</table>

The following HPF subroutine implements the drift transformation:

```hpf
subroutine drift(a,b,t)
c a(1-6,n) = initial (x,px,y,py,z,pz) of nth particle
c b(1-6,n) = final (x,px,y,py,z,pz) of nth particle
real a,b,t
dimension a(6,100000000),b(6,100000000)
!hpf$ distribute a(*,block)
!hpf$ distribute b(*,block)
  forall(i=1:100000000)b(1,i)=a(1,i)+a(2,i)*t
  forall(i=1:100000000)b(2,i)=a(2,i)
  forall(i=1:100000000)b(3,i)=a(3,i)+a(4,i)*t
  forall(i=1:100000000)b(4,i)=a(4,i)
  forall(i=1:100000000)b(5,i)=a(5,i)+a(6,i)*t
  forall(i=1:100000000)b(6,i)=a(6,i)
return
end
```

In the above subroutine, the \texttt{FORALL} statements could be replaced with array syntax. For example, the first executable statement could be written as

\[b(1,:) = a(1,:) + a(2,:)*t\]

If the arrays had a seventh component to indicate whether or not a particle was lost, a masked \texttt{FORALL} statement could be used to transform only the desired particles:

\[\text{forall}(i=1:100000000,a(7,i).eq.0.)b(1,i)=a(1,i)+a(2,i)*t\]
Internal Processing of Large Data Sets

Parallel calculations are of no value unless there is a way to make the resulting data available after the simulation is finished. But considering the size of arrays in large scale simulations, this is not a trivial issue. Consider, for example, a beam dynamics simulation with 100 million particles. A six-dimensional, double precision array would require 4.8 GB of memory, and writing out the data at a few hundred locations in the beamline would require TBytes of disk space. Furthermore, in many situations it is insufficient to simply write out a randomly chosen fraction of the data. Thus, it is often essential to do internal data processing in large scale parallel simulations. The following example describes a situation where the parallel data processing is more difficult to implement than the physics calculation itself.

Consider a simulation whose goal is to illustrate the dynamics of a chaotic system. The system is a one-dimensional double well potential with a sinusoidal drive. To illustrate the dynamics, 100 million particles are initially distributed around a ring of radius \( r = 0.01 \) in \( x-p_x \) space, and the system is allowed to evolve for roughly 12 drive periods. (For this simulation the use of 100 million particles is not excessive, since adjacent particles have trajectories that diverge exponentially due to the presence of chaos.) By the end of the simulation the ring has been repeatedly stretched and folded into the complicated shape shown in Figure 1.

This is another example of an “embarrassingly parallel” problem, since all 100 million particles evolve independently. But the coding to output the data for the figure is complicated (more complicated than the code to integrate the trajectories) and not easily parallelizable. It is obviously wasteful and unnecessary to print all 100 million points, which would require 1.6 GB. Instead, enough points are plotted so that, when connected by straight lines, the curve is smooth. To do this, the tangent vector of the curve is first computed at all 100 million points, and the angle between tangent vectors, \( \theta \), is computed. Next, a cumulative angle array, \( \Theta \), is computed, whose elements are given by \( \Theta_n = \sum_{i=1}^{n} \theta_i \). This is done using a utility routine called SCAN.ADD, a type of parallel prefix operation generally included in parallel software libraries. Next, points are marked for printing based on the following criterion: Having chosen a point in the array to print, the next point to be printed is the next one in the array where \( \Theta \) has changed by some threshold value (1 degree in this example). This is done using a SCAN.COPY routine. Finally, the points cannot simply be printed as follows,

\[
\text{do 100 } i=1,100000000 \\
100 \quad \text{<if the ith point is marked for printing, print it>} \\
\]

since the DO loop is a scalar operation that would run far more slowly than anything else in the code. Instead, the points to be printed are moved to the start of a temporary array by ordering them using a mask of zeroes and ones as keys to indicate which points are marked or not marked for printing, respectively. The data is printed as shown,

\[
\text{do 100 } i=1,\text{ntot} \\
100 \quad \text{<print the ith point of the temporary array>} \\
\]
where ntot is the total number of points to be printed. Figure 1 contains approximately 62000 points, corresponding to a data reduction factor of 1600.

This example illustrates the fact that, in the data-parallel environment, it is crucial to have a library of utility routines (in this case SCANS and ORDER), to perform those operations that, by their nature, are not data-parallel. It also illustrates the fact that, though it will never be possible to take an arbitrary scalar code and have it converted automatically to an efficient parallel code, it would be extremely useful to have this capability for small segments of code.

![Figure 1: The final configuration of an initial circle of points of radius 0.01 after it has evolved in a driven, double well system. The plot contains approximately 62000 points out of 100 million that were initially distributed around the circle.](image)

**Particle-In-Cell Simulations**

MPPs have had a significant impact in modeling beam halo in moderate-to-high average power linacs for the accelerator-driven technologies mentioned previously. Using resources provided by the Los Alamos Advanced Computing Laboratory, parallel beam dynamics codes have been written using CM Fortran (a precursor to HPF) that run on a 1024-node Thinking Machines CM5 computer. The codes use standard Particle-In-Cell (PIC) techniques, and the main difficulty in parallelization is related to the space charge routines, as described later.
Parallel Version of PARMILA

In addition to writing new codes, a parallel version of the beam dynamics code PARMILA (originally written in Fortran 77) has also been developed. To do this, a number of steps were required. First, all DO loops over large arrays, such as the particle array, were replaced with FORALL loops. This required rewriting large sections of the serial code between DO/ENDO statements, replacing scalar temporaries with parallel temporary arrays. Complications such as testing for lost particles inside of FORALL statements could be easily dealt with by using masked FORALLs. A more complicated situation arose when tests inside loops affected the program flow. Consider, for example, the serial code used to generate a 4D waterbag distribution:

```
  do 100 i=1,nptcls <loop over particles>
    <generate 4 random numbers x1,x2,x3,x4>
    if(x1**2+x2**2+x3**2+x4**2.gt.1)goto 50
    <generate coordinates and momenta for this particle>
  100 continue
```

This had to be replaced with code of the following form:

```
  100 <generate four LARGE arrays x1,x2,x3,x4>
    <mask off if x1**2+x2**2+x3**2+x4**2.gt.1>
    <pack good data into final array>
    <if final array is not complete, goto 100>
    <generate coordinates and momenta for all particles>
```

This exemplifies another situation where a utility routine (namely PACK) is essential to perform an operation that cannot be performed within the confines of data-parallel programming.

Besides rewriting large sections of code associated with DO loops, some other simple tasks were required to port PARMILA. As mentioned above, it is necessary to insert compiler directives in subroutines to specify the layout of parallel arrays. Another task was related to subroutine calls and data reshaping. In data-parallel languages, parallel arrays cannot be reshaped through subroutine calls as they can in Fortran 90. Thus, a 2D array coord(6,ntot) cannot be used as in `call mysub(coord(1))` and treated like a 1D array in subroutine mysub. Also, a 1D array x(ntot) cannot be used as in `call mysub(x(ntot/2))` and treated as a 1D array of half the original length in the subroutine. These situations are straightforward to deal with, but it can be tedious to find all such occurrences and they can easily go unnoticed until the program fails to execute properly.

The major difficulty in developing a parallel version of PARMILA, as with any parallel PIC code, is the space charge calculation. This is discussed in the next.

Space Charge Calculation

In the PIC approach, the self-fields are computed in three steps: Charge deposition on a grid, field solution on the grid, and field interpolation at the particle positions.
These steps are not easily parallelizable. Consider, for example, charge deposition on a two-dimensional grid using area weighting. A serial routine would look like the following:

```
do 100 n=1,np
   i=(x(n)-xmin)/hx
   j=(y(n)-ymin)/hy
   ab=xmin-x+i*hx
   cd=ymin-y+j*hy
   rho(i,j)=rho(i,j) + ab*cd
   rho(i+1,j)=rho(i+1,j)+cd*(hx-ab)
   rho(i,j+1)=rho(i,j+1)+ab*(hy-cd)
   100 rho(i+1,j+1)=rho(i+1,j+1)+(hx-ab)*(hy-cd)
```

The equivalent parallel routine is shown below:

```
i=(x-xmin)/hx   ! i,j,x,y,ab,cd = arrays
j=(y-ymin)/hy   ! hx,hy,xmin,ymin = scalars
ab=xmin-x+i*hx
cd=ymin-y+j*hy
forall(n=1:np)rho(i(n),j(n))=rho(i(n),j(n))+ab(n)*cd(n)
forall(n=1:np)rho(i(n)+1,j(n))=rho(i(n)+1,j(n))+cd(n)*(hx-ab(n))
forall(n=1:np)rho(i(n),j(n)+1)=rho(i(n),j(n)+1)+ab(n)*(hy-cd(n))
forall(n=1:np)rho(i(n)+1,j(n)+1)=rho(i(n)+1,j(n)+1)+(hx-ab(n))*(hy-cd(n))
```

Unfortunately the above parallel routine has poor performance. First, the `FORALL` statements cause significant interprocessor communication. Second, if the density array `rho` is uniformly spread across processors, then the routine will not be load balanced. For example, if one deposited a Gaussian charge distribution on the grid, then processors associated with the tail of the distribution would finish accumulating charge sooner than processors associated with the core.

The performance of this algorithm can be improved using several approaches. One can use optimized utility routines that send data to processors based on index arrays to perform binary operations on the data (e.g. add, overwrite, min, max). This is easy to implement but the performance improvement is modest. Another approach is to first reorder the particles so that all those corresponding to the same grid point are contiguous, then use parallel prefix routines to efficiently perform the charge deposition. This approach is more difficult to implement than the one mentioned previously, but the performance improvement is significant. It has been used by Ferrell and Bertschinger in an N-body code for astrophysical simulations [1]. A third method for improving the performance uses routines written with message passing libraries. In this approach the programmer explicitly writes the code that includes logic to determine how to partition the data so that the load is balanced. Though this approach is the most difficult to implement, it yields the best performance improvement. This approach was developed within the plasma simulation community, and has been used as part of the Numerical Tokamak Project, a DOE-funded High Performance Computing and Communications project [2][3].
Besides charge deposition and field interpolation, the space charge calculation requires the solution of the field equations on the grid. The parallel beam dynamics codes developed at the Advanced Computing Laboratory use an FFT-based approach to convolve the charge density on the grid with a Green function defined on the grid. Using standard techniques it is possible to treat a bunch of charge in the presence of open boundary conditions [4]. We have also implemented a procedure that uses open boundary conditions transversely and periodic boundary conditions longitudinally.

Performance

We have used our parallel beam dynamics codes to perform linac simulations with 1-30 million particles. For example, the parallel version of PARMILA has been used to model a superconducting linac for the Accelerator Production of Tritium project. The code contains a new 3D space charge algorithm based on the method of Ferrell and Bertschinger as described previously. A 10 million particle simulation on a $128^3$ grid, involving 900 space charge calculations, required 4.4 hours and 6.1 GBytes on the 256 node partition of the CM5. A sample output of the horizontal phase space for an initially mismatched beam shown in Figure 2. Symbols associated with the particles show the density in phase space, normalized to a maximum value of one.

![Figure 2: Results from a parallel version of PARMILA run on the CM5 at the Los Alamos Advanced Computing Laboratory. The simulation used 10 million particles.](image-url)
The success of the parallel approach depends on scalability, i.e., the ability to run larger problems in the same amount of time using more processors, or the ability to run problems of a fixed size in less time using more processors. The parallel version of PARMILA has excellent scalability as shown in Table 2:

Table 2: Scaling Results (3.75M particles, 64x64x64 grid)

<table>
<thead>
<tr>
<th>Procs</th>
<th>CPU (min)</th>
<th>MEM (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>15.5</td>
<td>1.8</td>
</tr>
<tr>
<td>256</td>
<td>8.1</td>
<td>2.0</td>
</tr>
<tr>
<td>512</td>
<td>4.3</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Direct Vlasov-Poisson Simulations

Direct Vlasov-Poisson solvers provide an alternative to PIC methods. In a direct solver, the beam distribution function is defined on a grid in phase space. Since no particles are needed there is no sampling noise in determining the distribution function at low densities. One difficulty with this approach is the enormous amount of memory required: a 3D simulation (with a 6D phase space) that uses 64 mesh points per dimension has a total of 68 billion mesh points.

In this approach one attempts a direct solution of the Vlasov-Poisson equations,

\[ \frac{\partial f}{\partial t} + (\vec{\mathbf{p}} \cdot \nabla f) - (\nabla V \cdot \mathbf{\partial p}) f = 0, \]

where \( f(\zeta, t) \) is a distribution function on phase space \( (\zeta = (x, p)) \). Here, the potential \( V \) is a sum of an externally applied potential and a space charge potential which is obtained self-consistently from Poisson’s equation. A second-order accurate stepping algorithm for this system is given by

\[ f(\zeta, t) = M(t)f(\zeta, t = 0). \]

where the mapping \( M \) is given by

\[ M(t) = e^{-(\frac{1}{2}(\varphi + \varphi V) + t(\nabla V \cdot \mathbf{\partial p}) + e^{-(\frac{1}{2}(\varphi + \varphi V))}. \]

This is analogous to the leap-frog algorithm used to advance particles in PIC codes, since the operators involving drifts and space charge “kicks” are done separately. In a split-operator spectral code, the mappings generated by these operators are evaluated using Fourier transforms. By transforming variables appropriately, the exponential operators turn into multiplications in Fourier space. It is worth noting

---

1Increasing the number of processors while keeping the problem size fixed cannot cause the execution time to decrease indefinitely since, if the problem size is too small, the processors will do too little calculation, and the execution time will be dominated by communication.
that, using a method due to Yoshida, one can construct fourth order (and higher order) stepping algorithms [5].

Figure 3 shows the results of a 2D beam halo simulation using a direct Vlasov-Poisson solver. The figure shows the beam density, $\rho(x, y)$, of an initially mismatched Gaussian beam after 22 periods in a quadrupole channel. The simulation, which was performed on the CM5 at the Advanced Computing Laboratory, used a 128 mesh points in each phase space dimension, for a total of 268 million mesh points.

Conclusions

HPC platforms make it possible to perform the numerically intensive calculations needed to model future accelerators for accelerator-driven transmutation technologies and applications. As discussed in this paper, simulations of intense beams with up to 30 million particles are already being performed, and with the arrival of the next-generation of HPC platforms, simulations with 100 million particles are expected in the 1997-1998 time frame. Though the development of parallel beam dynamics codes would be a daunting task if done from scratch, the accelerator community is fortunate to be able to build on progress already made by researchers in the computational plasma physics and cosmology communities.

Though we have emphasized modeling intense charged particle beams, HPC platforms will have many other uses in the accelerator community. As mentioned in this paper, parallel tracking calculations to predict dynamic aperture in large
circular machines can be done with ease, either through message passing or within the data-parallel framework of HPF. Also, HPC platforms will have a major impact in modeling complicated 3D electromagnetic structures with high accuracy. Parallel, 2D finite element codes have already been developed for this purpose, and development of 3D codes is underway [6].

In summary, HPC platforms will be needed to solve a variety of important simulation problems within the accelerator community. By the end of the decade we can expect to see many more accelerator physicists doing production computing on these platforms, thanks to maturing programming languages such as HPF, the availability of parallel scientific software libraries, and the availability of cycles at locations such as the Los Alamos Advanced Computing Laboratory and the National Energy Research Scientific Computing Center at Lawrence Berkeley National Laboratory.

Acknowledgements

The authors thank consultants of the National Energy Research Scientific Computing Center, including Youngbae Kim, for suggesting the use of the INDEPENDENT directive to run multiple copies of single particle tracking codes. This research was supported by the U.S. Department of Energy, Office of Energy Research, through the Division of Mathematical, Information, and Computational Sciences, the Division of High Energy and Nuclear Physics, and by the Office of Defense Programs, Accelerator Production of Tritium program. This research was performed in part using the resources located at the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545.

References

Title of Article (in caps) (Spell out all symbols)
PARALLEL BEAM DYNAMICS CALCULATIONS ON HIGH PERFORMANCE COMPUTERS

Intended for:

Deadline Date

<table>
<thead>
<tr>
<th>TECHNICAL CONTENT</th>
<th>GROUP</th>
<th>PHONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew Jason</td>
<td>AOT-1</td>
<td>7-2842</td>
</tr>
</tbody>
</table>

CLASSIFICATION DETERMINATION

<table>
<thead>
<tr>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADTO</td>
</tr>
</tbody>
</table>

Signature Derivative Classifier

EDITING REQUIREMENTS

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edited</th>
<th>Logged In</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group Leader Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew Jason</td>
<td>12/2/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Publications Department Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12/3/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12/3/96</td>
</tr>
</tbody>
</table>

Please send all publications with this form whether the papers are to be edited or not to the Publications Department, MS H811, all papers that require editing should be double spaced.