Particle Accelerators, 1996, Vol. 55, pp. [365–374] /119–128
Reprints available directly from the publisher
Photocopying permitted by license only

BEAM DYNAMICS CALCULATIONS AND PARTICLE TRACKING USING MASSIVELY PARALLEL PROCESSORS

ROBERT D. RYNE[1] and SALMAN HABIB[2]

[1]*Mail Stop H817, Accelerator Operations and Technology Division, Los Alamos National Laboratory, Los Alamos NM 87545, USA*

[2]*Mail Stop B288, Theoretical Division, Los Alamos National Laboratory, Los Alamos NM 87545, USA*

(Received 31 December 1995; in final form 31 December 1995)

During the past decade massively parallel processors (MPPs) have slowly gained acceptance within the scientific community. At present these machines typically contain a few hundred to one thousand off-the-shelf microprocessors and a total memory of up to 32 GBytes. The potential performance of these machines is illustrated by the fact that a month long job on a high end workstation might require only a few hours on an MPP. The acceptance of MPPs has been slow for a variety of reasons. For example, some algorithms are not easily parallelizable. Also, in the past these machines were difficult to program. But in recent years the development of Fortran-like languages such as CM Fortran and High Performance Fortran have made MPPs much easier to use. In the following we will describe how MPPs can be used for beam dynamics calculations and long term particle tracking.

Keywords: Beam dynamics; tracking; parallel computing.

1 INTRODUCTION

During the 1980s vector supercomputers established themselves as the machines of choice for physicists performing the most numerically intensive calculations; but by the end of the decade many were turning away from the timeshared vector machines of centralized supercomputer centers and obtaining their own dedicated high end workstations. Today such workstations offer users performance approaching a few hundred MFLOPS with memories up to 1 GByte. Since high end workstations are becoming a commodity item, one can rightly question the usefulness of centralized supercomputer centers offering vector performance of at best 2 GFLOPS per processor. In fact, many supercomputer centers have turned to massively parallel processors (MPPs) to provide their users with the highest performance and largest memories available today. For example, since the next generation of MPPs will have up to a few thousand processors like those found in high end workstations, they have a potential performance of a few hundred to one thousand GFLOPS. Also, with order of 100 MByte of RAM on each processor, users will be able to run jobs requiring up to several hundred GBytes of memory.

Though MPPs have potentially very high performance it is often not easily achievable in practice. For example, some algorithms are not easily parallelizable. In these cases significant effort might be required to implement an algorithm on a parallel machine, or one might need to find a new way of thinking about the problem that is amenable to parallel solution. Also, in the past these machines were difficult to program and they required one to learn a new programming language. However, in recent years the development of parallel scientific software libraries and Fortran-like languages have made MPPs much easier to use.

There are two main paradigms for programming MPPs. The most flexible, and the most difficult from a programming standpoint, is the Multiple Instruction Multiple Data (MIMD) paradigm. In this approach, the programmer essentially tells every processor what to do, including when to send data to other processors, when to wait for data from other processors, *etc.* This is accomplished using message passing libraries such as PVM (Parallel Virtual Machine) and MPI (Message Passing Interface). This style of programming has been called the "assembly language of MPPs." The other main paradigm is the Single Instruction Multiple Data (SIMD) approach, in which every processor executes the same instructions but on different data. A similar paradigm is the Single Program Multiple Data (SPMD) approach, where every processor executes the same program, asynchronously, on different data. For those problems that are trivially parallel the SIMD approach is sufficient. This includes the integration of trajectories of particles in specified fields. For problems that are not easily parallelizable MIMD programming provides the best means to achieve good performance, though it could involve the effort of a small or large research project. An example of this is the process of depositing charge on a grid, which has received much attention in the plasma particle simulation community.[1,2]

Programming MPPs is becoming easier due to the availability of languages like Connection Machine Fortran (CMF) and High Performance Fortran (HPF) that allow one to do SIMD and SPMD programming in a Fortran-like environment. Though it is the authors' view that CMF, a product of Thinking Machines Corporation, is the most mature language of this type, HPF is an emerging standard and it is expected that CMF will be HPF compliant sometime in 1996. HPF is a language that includes Fortran 90 as a subset, but it also contains some additional commands (such as the FORALL which is a parallel DO) as well as compiler directives to specify how data is to be distributed across processors. In fact, intelligently laying out the data turns out to be a key consideration in this approach. Compared with a Fortran 77 program, an HPF program uses array syntax whenever possible (which is available in Fortran 90); it uses FORALL statements instead of DO loops whenever possible; and it has compiler directives interspersed among array declarations. Also, these programs often contain calls to mathematical libraries, such as the CM Scientific Software Library on the CM-5.

As an example of array layout, consider a program that integrates the trajectories of 4096 particles having coordinates (x, p_x, y, p_y, z, p_z). Suppose the data is stored in an array dimensioned a(6,4096). The compiler directive "distribute a(*,block)" would cause quantities specified by the first dimension to reside on the same processor, 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 1–1024 would reside on processor 1; particles 1025–2048 would reside on processor 2; particles 2049–3072 would reside on processor 3; and particles 3073–4096 would reside on processor 4. This is shown in Table I.

TABLE I Layout of a two dimensional array distributed (*,block)

Processor 1	*Processor 2*	*Processor 3*	*Processor 4*
a(1–6,1)	a(1–6,1025)	a(1–6,2049)	a(1–6,3073)
a(1–6,2)	a(1–6,1026)	a(1–6,2050)	a(1–6,3074)
a(1–6,3)	a(1–6,1027)	a(1–6,2051)	a(1–6,3075)
⋮	⋮	⋮	⋮
a(1–6,1023)	a(1–6,2047)	a(1–6,3071)	a(1–6,4095)
a(1–6,1024)	a(1–6,2048)	a(1–6,3072)	a(1–6,4096)

As a concrete example, consider the propagation of particles in a drift space and the following subroutine:

$$\begin{aligned} \vec{x}^{\,\text{fin}} &= \vec{x}^{\,\text{in}} + \vec{v}^{\,\text{in}} t \\ \vec{v}^{\,\text{fin}} &= \vec{v}^{\,\text{in}} \end{aligned} \tag{1}$$

```
      subroutine drift(a,b,t)
c a(1-6,n) = initial (x,vx,y,vy,z,vz) of nth particle
c b(1-6,n) =    final (x,vx,y,vy,z,vz) of nth particle
      real a,b,t
      dimension a(6,10000000),b(6,10000000)
cmf$  layout a(:serial,:news)
cmf$  layout b(:serial,:news)
!hpf$ distribute a(*,block)
!hpf$ distribute b(*,block)
      b(1,:)=a(1,:)+a(2,:)*t
      b(2,:)=a(2,:)
      b(3,:)=a(3,:)+a(4,:)*t
      b(4,:)=a(4,:)
      b(5,:)=a(5,:)+a(6,:)*t
      b(6,:)=a(6,:)
      return
      end
```

In the above subroutine, the directives beginning with `!hpf` are HPF compiler directives; those beginning with `cmf` are the equivalent directives in CMF. Note the portability of this routine: It should compile without changes on a workstation with Fortran 90 (since the compiler directives would be interpreted as comments), on a multiprocessor workstation with HPF, and on an MPP with CMF or HPF.

2 MODELING BEAM HALO IN ULTRA-LOW LOSS ACCELERATORS

An area where MPPs have already had a significant impact is in modeling beam halo in next-generation, moderate-to-high average power accelerators that must operate with very low beam loss. This effort is motivated by recent activity aimed at utilizing accelerator driven technologies for waste transmutation, plutonium conversion, tritium production, fission energy production, and the production of spallation neutrons for materials science and biological science research. At the high energy end of a 1 GeV linac the allowed beam loss is less than one nanoampere per meter. It is now known that a major source of beam loss is the formation of a very low density halo far from the beam core. Understanding and predicting beam halo and finding ways to minimize it will have a major impact on the above-mentioned technologies.

Using the resources of the Advanced Computing Laboratory (ACL) at Los Alamos National Laboratory (LANL), we have developed tools that enable one to model the dynamics of intense charged particle beams with very high speed and accuracy. Previously, most linacs were designed using particle simulation codes run with 10000 particles. Though this was adequate in the past, in the new ultra-low loss regime it will be necessary to use at least 10 million particles to predict the beam halo with confidence. Using the CM-5 at the ACL, we have developed 1D, 2D, and 3D particle-in-cell beam dynamics codes. These codes use symplectic, split-operator integration algorithms to advance the particles.[3] This process is trivially parallel except for the space charge calculation that is required. Though it is possible to implement charge deposition and field interpolation in Fortran 90 (in the form of statements involving indirect addressing of arrays), this is not very efficient. Instead, we have implemented a procedure as described by Ferrell and Bertschinger in the context of cosmology simulations.[4] Besides using particle simulation codes, it is also possible to use direct Vlasov/Poisson solvers to determine the evolution of an intense charged particle beam. This was difficult in the past due to the large memory requirement of such programs. Since the distribution function is represented by a grid on phase space, not real space, the grid is 2N-dimensional for an N-dimensional problem. For example, a 2D problem with a grid length of 256 requires 268 million grid points. This is not unreasonable for an MPP having a few GBytes of memory. A direct Vlasov/Poisson code looks very similar in structure to a particle simulation code except that instead of using split-operator techniques to advance particles they are used to advance the distribution function. For example, a Vlasov/Poisson code solves the equation

$$\frac{\partial f}{\partial t} + (\vec{p} \cdot \partial_{\vec{x}}) f - (\nabla V \cdot \partial_{\vec{p}}) f = 0, \tag{2}$$

where $f(\zeta, t)$ is a distribution function on phase space ($\zeta = (\vec{x}, \vec{p})$). 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 the Vlasov/Poisson equation is given by

$$f(\zeta, t) = \mathcal{M}(t) f(\zeta, t = 0). \tag{3}$$

where the mapping $\mathcal{M}$ is given by

$$\mathcal{M}(t) = e^{-\frac{t}{2}(\vec{p}\cdot\partial_{\vec{x}})} e^{t(\nabla V\cdot\partial_{\vec{p}})} e^{-\frac{t}{2}(\vec{p}\cdot\partial_{\vec{x}})}. \tag{4}$$

It is important to note that, given an algorithm of order $2n$, it is possible to construct an algorithm of order $2n + 2$ using a technique originally due to Yoshida.[5,6] Thus it is a simple matter to construct high order particle simulation programs and Vlasov/Poisson codes. Using these particle simulation techniques and direct methods we have also developed codes to model large scale structure formation in the early universe, as well as direct solvers to model quantum systems represented by a Schrödinger wave function, a density matrix and a Wigner distribution function.

3 LONG TERM TRACKING

Compared with the multi-million particle simulations of intense beams described in the previous section, there are significant differences in using MPPs to perform long term tracking in circular machines. First, codes written in CMF or HPF usually perform well only when they use large amounts of memory, but tracking codes usually utilize only a few to a few hundred particles and hence use little memory. Second, matrix codes and Lie algebraic beam transport codes usually involve irregular data movement, and this can cause excessive interprocessor communication and poor performance. On the other hand high order codes would alleviate the memory issue since they utilize somewhat long arrays. In a Lie algebraic code the length of arrays as a function of order is shown in Table II.

TABLE II Array length as a function of order in a Lie algebraic code

Order	3	4	5	6	7	8	9	10	11	12
Monomials	83	209	461	923	1715	3002	5004	8007	12375	18563

We thus come to the following conclusions: If one wants to perform long term tracking with many (i.e. greater than 10000) particles, then one can achieve good performance by parallelizing over particles. If one is using only a few particles, then one can still achieve good performance by parallelizing over the Lie polynomials. But if one is interested in tracking a small number of particles to low order, than one is not likely achieve good performance on an MPP using data parallel methods in CMF or HPF. It might be possible to parallelize over particles using other methods as is described later.

As mentioned previously, Lie algebraic codes store polynomials as large arrays where each element corresponds to a monomial. When performing a ray trace one of the things that must be done is the numerical evaluation of the monomials. That is, for a given (x, p_x, y, p_y, t, p_t) one must compute $(x^{n_1}, p_x^{n_2}, y^{n_3}, p_y^{n_4}, t^{n_5}, p_t^{n_6})$, where the sum of the n_j is less than or equal to the order of the highest order polynomial. To see how this would be accomplished, consider the following CM Fortran code which could appear in a beam dynamics program using 4th order polynomials:

```
      dimension g(209),nvbl(4,209),vec(0:6)
cmf$  layout g(:news)
cmf$  layout nvbl(:serial,:news)
      forall(i=1:209)
     #g(i)=vec(nvbl(1,i))*vec(nvbl(2,i))
     #    *vec(nvbl(3,i))*vec(nvbl(4,i))
```

In the above, each 4-tuple of the array nvbl denotes a monomial. For example, in the usual indexing scheme nvbl(1-4,31) is equal to (1,1,4,0), which corresponds to the monomial $x^2 p_y$. This code would execute properly on a CM-5, but it would not be efficient due to the communication that it generates. Namely, for a given index i the quantities vec(nvbl(1-4,i)) would not be on the processor of the target element g(i). (On the CM-5 scalars and small arrays are normally stored on the "front end" and not on the processing elements.) The solution to this, on the CM-5, is to make an array svec(0:6,209) that is 209 copies of vec by using the SPREAD command: `svec=spread(vec,ncopies=209,dim=2)`. Though this takes time, it is acceptable so long as it is only done once and the routine is called many times. Thus, a more efficient version of the code would use the following FORALL statement:

```
 forall(i=1:209)
#g(i)=svec(nvbl(1,i),i)*svec(nvbl(2,i),i)
#    *svec(nvbl(3,i),i)*svec(nvbl(4,i),i)
```

This is an example of code that parallelizes over the order of the problem (i.e. over the Lie polynomials), since the FORALL statement runs from one to the length of the polynomials. Alternately, a code that tracked many particles could parallelize over the particles. In that case no extra work (i.e. no explicit parallelization using SPREAD commands) would be required.

Table III shows CM-5 timing results for computing the monomials of 256 particles 1000 times. The code is written to be parallel over the monomials and two cases are shown, 6th order and 12th order, corresponding to array lengths of 923 and 18563, respectively. The program scales well (i.e. the execution time is roughly inversely proportional to the number of processors) only for the 12th order case. Table IV shows CM-5 timing results for computing monomials up to 6th order 1000 times. The code is written to be parallel

TABLE III 256 particles, 6th and 12th order, parallelized over monomials

PNs	*6th order*	*12th order*
32	10 sec, 74 MB	302 sec, 600 MB
64	11 sec, 147 MB	159 sec, 691 MB
128	10 sec, 294 MB	81 sec, 806 MB

TABLE IV 6th order, 1024 and 16384 particles, parallelized over particles

PNs	*1024 particles*	*16384 particles*
32	54 sec, 90 MB	408 sec, 578 MB
64	58 sec, 180 MB	221 sec, 628 MB
128	58 sec, 359 MB	126 sec, 743 MB

over particles, and two cases are shown, 1024 particles and 16384 particles. The program scales well only for the 16384 particle case. Poor scaling using data parallel techniques is usually associated with the processors not having enough work to do, and this is normally accompanied by low memory usage.

Lastly we will consider the implementation of Lie algebraic ray tracing. This discussion assumes some familiarity with Lie methods.[7] Consider, for example, a 4th order, nonsymplectic ray trace:

$$e^{:f_2:}\, e^{:f_3:}\, e^{:f_4:}\zeta_j \approx e^{:f_2:}\left(1+ :f_3: + \frac{:f_3:^2}{2!}\right)\left(1+ :f_4:\right)\zeta_j \tag{5}$$

From a symbolic viewpoint, $e^{(:f_4:)}$ acts first on ζ_j, followed by $e^{(:f_3:)}$ and $e^{(:f_2:)}$. But from a numerical viewpoint the situation is just the opposite, *i.e.* the left-most Lie transformation acts first. In the first approach, one would perform a ray trace by doing algebraic manipulations until the final step of the calculation. First one would apply $(1+ :f_4:)$ to one of the ζ_j to obtain a polynomial; next one would apply $(1+ :f_3: + :f_3:^2 /2!)$ to obtain a new polynomial; then one would transform the polynomial by the matrix representation of $e^{:f_2:}$; and finally one would numerically evaluate the polynomial to obtain the final value of ζ_j. This could of course be done in parallel for all six ζ_j and all particles. In the second approach, one would first multiply the six-vector $\vec{\zeta}$ by the matrix representation of $e^{:f_2:}$; next one would apply the operator $(1+ :f_3: + :f_3:^2 /2!)$ to these six numbers to obtain six new numbers; and finally one would apply $(1+ :f_4:)$ to obtain the final conditions of the ray trace. This could of course be done in parallel for all particles. We are still in the process of studying the efficiency of these two methods, but it is likely that the first method will be best when the number of particles is large since all the algebraic manipulations, including the time-consuming transformation of the polynomial by the matrix, will be amortized over the particles. The second method is likely to be best when the number of particles is small and the order is high, since one would avoid having to transform a large polynomial by a matrix.

4 CONCLUSION

MPPs have the potential to outperform high end workstations by two to three orders of magnitude, but achieving this is not always a simple matter. MPPs have been used successfully to model beam dynamics and halo formation in intense charged particle beams both in particle simulation codes and in direct Vlasov/Poisson solvers. With time MPPs are becoming easier to use; HPF is becoming a widespread standard and HPF compilers are maturing. We have found that, with regard to long term tracking, it will be easy to achieve good performance with MPPs if the number of particles is large or the order is high. But it is our experience that data parallel techniques have a certain amount of overhead associated with them, and if one were to track 128 particles on 128 nodes the relative performance would be poor compared with tracking a single particle on a single-CPU workstation. On the other hand, it is possible to temporarily break out of CMF or HPF in order to execute code on the nodes. Using this SPMD approach it should be possible to exploit the coarse-grained parallelism of the particles to achieve good performance even when the number of particles is equal to the number of processors, if desired.

Acknowledgements

We thank Fillipo Neri for helpful conversations regarding Lie algebraic tracking. This research was supported by the U.S. Department of Energy, Office of Energy Research, through the Division of High Energy Physics and the Division of Mathematical, Information, and Computational Sciences. 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

[1] J. Wang, P. Liewer and V. Decyk, *Computer Physics Communications*, **87** (1995), 35–53.
[2] V.K. Decyk, *Computer Physics Communications*, **87** (1995), 87–94.
[3] E. Forest and R. Ruth, *Physica D*, **43**, (1990) 105.
[4] R. Ferrell and E. Bertschinger, *Int. J. Mod. Phys. C*, **5**, (1994) 933–956.
[5] H. Yoshida, *Phys. Lett. A*, **150**, (1990) 262.
[6] E. Forest *et al.*, *Phys. Lett. A*, **158**, (1991) 99.
[7] A. Dragt, In *Physics of High Energy Particle Accelerators*, AIP Conf. Proc. **87**, R.A. Carrigan *et al.*, ed. (1982).