MPI+OPENMP IMPLEMENTATION OF MEMORY-SAVING PARALLEL PIC APPLICATIONS ON HIERARCHICAL DISTRIBUTED-SHARED MEMORY ARCHITECTURES

Sergio Briguglio, Giuliana Fogaccia
and Gregorio Vlad
Associazione EURATOM-ENEA sulla Fusione,
C.R. Frascati,
C.P. 65, 00044, Frascati, Rome, Italy
{briguglio,fogaccia,vlad}@frascati.enea.it

Beniamino Di Martino
Dip. Ingegneria dell'Informazione,
Second University of Naples, Italy
beniamino.dimartino@unina.it

This work has been partly supported by the CNR - Consiglio Nazionale delle Ricerche, Italy (Agenzia 2000 Project *ALCOR* - n. CNRG00A41A).

KEYWORDS

Physics, Multiprocessors, Combined, Hierarchical, Parallel methods.

ABSTRACT

The combination of inter-node and intra-node domain-decomposition strategies for the development of memory-saving parallel Particle-in-cell simulation codes, targeted to hierarchical distributed-shared memory architectures, is discussed in this paper, along with its MPI+OpenMP implementation. Particular emphasis is given to the devised dynamic workload balancing technique.

INTRODUCTION

One of the main issues of thermonuclear fusion research is represented by the investigation of small-scale electromagnetic fluctuations in magnetically confined plasmas and their connection with the high values of the transport coefficients observed in the tokamak experiments (Wesson 1997). The full comprehension of such phenomena requires a self-consistent treatment of the interactions between electromagnetic waves and the plasma, which cannot be satisfactorily described in terms of few moments of the particle distribution function (i.e., by a fluid treatment). A full kinetic approach is instead requested, and significant results cannot be obtained, apart from the case of particular asymptotic regimes, on the basis of analytical studies. Numerical simulation is then needed to get insight in the plasma turbulent behaviour and, in particular, to determine the saturation mechanisms, the level of the saturated fluctuations and their effect on the plasma confinement.

Among the many existing numerical tools, the particle simulation technique (Hockney and Eastwood 1981; Birdsall and Langdon 1985) seems to be the most suited one, as it directly study the time evolution of the particle distribution function and the mutual interaction between such a distribution and the fluctuating electromagnetic fields. Particle simulation indeed consists in performing self-consistent *particle pushing*: the phase-space coordinates of a set of N_{part} particles are evolved in the electromagnetic fields computed, at each time step, in terms of the contribution yielded by the particles themselves (e.g., pressure).

The most important class of particle simulation codes is that of the so-called *particle-in-cell* (PIC) codes, which compute the fluctuating fields and the required particle contribution only at the nodes of a spatial grid, then interpolating the fields at the (continuous) particle position in order to evolve particle coordinates.

Because of the large ratio between the equilibrium scale lengths and the fluctuation ones (typically, several tens or more), high spatial resolution is required in such simulations (up to millions grid cells for three-dimensional PIC codes). Such a requirement, along with the need for ensuring an adequate description of the particle distribution function in the velocity space, makes the usage of large numbers of simulation particles (tens or hundreds millions) necessary and, hence, the full exploitation of parallel computers unavoidable.

Two main workload decomposition strategies have been adopted in parallelizing PIC codes on distributed-memory architectures: the so-called *particle decomposition* (Di Martino et al. 2001b) and *domain decomposition* (Fox et al. 1988; Ferraro et al. 1993). The former strategy statically distributes the particle population among the processes, while assigning the whole domain (and the spatial grid) to each process. The static character of particle assignment yields a perfect load balancing and allows the use of high-level languages like High Performance Fortran (HPF) (High Performance Fortran Forum 1997) in the implementation of this strategy. On the opposite side, an overhead on memory occupancy, given by the replication of data related to the domain, and a computation overhead related to the updating of the fields (each node manages only a partial pressure updating, associated to its portion of particle population) forbid a good scalability of the maximum domain size with the number, n_{node}, of computational nodes. Moreover, though the efficiency of such a technique can be very high, at moderate values of n_{node}, it rapidly degrades when computational loads on each node become dominated by the field-related ones.

Domain decomposition consists instead in assigning different portions of the physical domain and the corresponding portions of the grid to different processes, along with the particles that reside on them. Such a strategy has pros

and cons that are specular to those of particle decomposition. The distribution of all the arrays among the computational nodes gives indeed this method an intrinsic scalability of the maximum domain size (that is, the maximum spatial resolution) that can be simulated with the number of nodes. Particle migration from one portion of the domain to another, however, requires step-by-step particle-to-process reassignment, which can affect the parallelization efficiency, because of the communication and computation overhead. Moreover, it can give rise to serious load unbalances, which require the application of dynamic load-balancing techniques (Cybenko 1989; Ferraro et al. 1993). Both facts make a deep restructuring of the serial code necessary. Moreover, they prevent the use of a high-level data-parallel language and compel one to resort to explicit message-passing libraries, such as MPI.

Such workload decomposition strategies can be extended to the case of shared-memory parallel systems (Di Martino et al. 2001a). In this framework, the use of high-level language, like OpenMP (OpenMP Architecture Review Board 1997), is possible. They can also be combined, when porting a PIC code on a hierarchical distributed-shared memory system (e.g., a cluster of SMPs), in two-level strategies: a distributed-memory level decomposition (among the n_{node} computational nodes), and a shared-memory one (among the n_{proc} processors of each node). In previous papers we have investigated some of these strategies applied to the simulation of thermonuclear plasma confinement. In particular, we have designed and implemented *i*) the particle-decomposition strategy for distributed-memory architectures, with the use of HPF (Di Martino et al. 2001b); *ii*) both particle and domain decomposition strategies for shared-memory architectures, with the use of OpenMP (Di Martino et al. 2001a); *iii*) hierarchically combined particle-particle and particle-domain decomposition strategies for hierarchically structured *distributed-shared memory* architectures, with the integrated use of HPF and OpenMP (Briguglio et al. 2002). The task of optimizing the memory usage requires, however, to enriches this scenario of hierarchically-combined decomposition strategies with the development of the integrated domain-domain combination. Aim of this paper is discussing the MPI+OpenMP implementation of this strategy, with particular emphasis to the dynamic workload balancing technique we have devised.

In the following Sections we describe the inter-node, domain-decomposition strategy, adopted in the distributed-memory context, along with its MPI implementation, and the integration of such inter-node strategy with the intra-node domain-decomposition strategy. Conclusions are drawn in the final Section.

MPI IMPLEMENTATION OF THE INTER-NODE DOMAIN DECOMPOSITION

The typical structure of a PIC code for plasma particle simulation can be represented as follows. At each time step, the code *i*) computes the electromagnetic fields only at the points of a discrete spatial grid (*field solver* phase); *ii*) interpolates the fields at the (continuous) particle positions in order to evolve particle phase-space coordinates (*particle pushing* phase); *iii*) collects particle contribution to the pressure field at the grid points to close the field equations (*pressure computation* phase). We can schematically represent the structure of this time-iteration by the following code excerpt:

```
call field_solver(pressure,field)
call pushing(field,x_part)
call compute_pressure(x_part,pressure)
```

Here, `pressure`, `field` and `x_part` represent pressure, electromagnetic-field and particle-position arrays, respectively. In order to simplify the notation, we will refer, in the pseudo-code excerpts, to a one-dimensional case, while the experimental results reported in the following refer to a three-dimensional (3-D) application.

In implementing a parallel version of the code, according to the distributed-memory domain-decomposition strategy, inter-node communication is required to update the fields at the boundary between two different portions of the domain, as well as to transfer those particles that migrate from one domain portion to another. Such a particle migration possibly determines a severe load unbalancing of the different processes, then requiring a dynamic balancing, at the expenses of further computations and communications.

Three additional procedures then characterize the structure of the parallel code: at each time step, *i*) the number of particles managed by a process has to be checked, in order to avoid excessive load unbalancing among the processes (if such an unbalancing is verified, the load-balancing procedure must be invoked); *ii*) particles that moved from one subdomain to another because of particle pushing must be transferred from the original process to the new one; *iii*) the values of the pressure array at the boundaries between two neighbor subdomains must be corrected, because their local computation takes into account only those particles which belong to the subdomain, neglecting the contribution of neighbor subdomain's particles.

Let us report here the schematic representation of the time iteration performed by each process:

```
 call field_solver(pressure,field)
 call load_checking(i_check,n_part,
&  n_part_left_v,n_part_right_v)
 if(i_check.eq.1)then
  call load_balancing(n_part_left_v,
&   n_part_right_v,n_cell_left,
&   n_cell_right,n_part_left,
&   n_part_right)
  n_cell_new=n_cell+n_cell_left+
```

```
&            n_cell_right
  if(n_cell_new.gt.n_cell)then
   allocate(field_aux(n_cell))
   field_aux=field
   deallocate(field)
   allocate(field(n_cell_new))
   field(1:n_cell)=field_aux(1:n_cell)
   deallocate(field_aux)
  endif
  n_cell=max(n_cell,n_cell_new)
  n_cell_old=n_cell
  call send_receive_cells(field,
&   x_part,n_cell_left,n_cell_right,
&   n_part_left,n_part_right)
  if(n_cell_new.lt.n_cell_old)then
   allocate(field_aux(n_cell_old))
   field_aux=field
   deallocate(field)
   allocate(field(n_cell_new))
   field(1:n_cell_new)=
&       field_aux(1:n_cell_new)
   deallocate(field_aux)
  endif
  n_cell=n_cell_new
  n_part=n_part+n_part_left+n_part_right
 endif
 call pushing(field,x_part)
 call particle_transfer(x_part,n_part)
 allocate(pressure(n_cell))
 call compute_pressure(x_part,pressure)
 call correct_pressure(pressure)
```

In order to avoid continuous reallocation of particle arrays (here represented by `x_part`) because of the particle migration from one subdomain to another, we overdimension (e.g., +20%) such arrays with respect to the initial optimal-balance size, N_{part}/n_{node}. Fluctuations of `n_part` around this optimal size are allowed within a certain oscillation band (e.g., $\pm 10\%$). This band is defined in such a way to avoid a too frequent resort to the time-consuming load-balancing procedure and, at the same time, to prevent an excessive load unbalancing. One of the processes (the MPI rank-0 process) collects, in subroutine `load_checking`, the values related to the occupation level of the other processes and checks whether the band boundaries are exceeded on any process. If this is the case, the "virtual" number of particles (`n_part_left_v`, `n_part_right_v`) each process should send to the neighbor processes to recover the optimal-balance level is calculated (negative values means that the process has to receive particles), and `i_check` is set equal to 1. Then, such informations are scattered to the other processes. These communications are easily performed with MPI by means of the collective communication primitives `MPI_Gather`, `MPI_Scatter` and `MPI_Bcast`. Load balancing is then performed as follows. Particles are labelled (subroutine `load_balancing`) by each process according to their belonging to the units (e.g., the `n_cell` spatial-grid cells) of a finer subdivision of the corresponding subdomain: each particle is then characterized by its unit index and an index ranging from 1 to the number of particles belonging to the same unit. The portion of the subdomain (that is, the number of elementary units) the process has to release, along with the hosted particles, to neighbor subdomains in order to best approximate those virtual numbers (if positive) is then identified. Communication between neighbor processes allows each process to get the information related to the portion of subdomain it has to receive (in case of negative "virtual" numbers). Net transfer information is finally put into the variables `n_cell_left`, `n_cell_right`, `n_part_left`, `n_part_right`. Series of `MPI_Sendrecv` are suited to a deadlock-free implementation of the above described communication pattern.

As each process could be requested, in principle, to host (almost) the whole domain, overdimensioning the grid arrays (`pressure` and `field`) would cause losing of the desired memory scalability (there would be, indeed, no distribution of the memory-storage loads related to such arrays). We then have to resort to dynamical allocation of the grid arrays, possibly using auxiliary back-up arrays (`field_aux`), when their size is modified.

Portions of the array `field` have now to be exchanged between neighbor processes, along with the elements of the array `x_part` related to the particles residing in the corresponding cells. This is done in subroutine `send_receive_cells` by means of `MPI_Send` and `MPI_Recv` calls. The elements of the grid array to be sent are copied in suited buffers, and the remaining elements are shifted, if needed, in order to be able to receive the new elements and to fill possibly occurring holes. After sending and/or receiving the buffers to/from the neighbor processes, the array `field` comes out to be densely filled in the range `1:n_cell_new`. Analogously, the elements of `x_part` corresponding to particles to be transferred are identified on the basis of the labelling procedure performed in subroutine `load_balancing` and copied into auxiliary buffers; the residual array is then compacted in order to avoid the presence of "holes" in the particle-index space. Buffers sent by the neighbor processes can then be stored in the higher-index part of the array `x_part` (remember that such an array is overdimensioned).

After rearranging the subdomain, subroutine `pushing` is executed, producing the new particle coordinates, `x_part`. Particles whose new position falls outside the original subdomain have to be transferred to a different process. This is done by subroutine `particle_transfer`. First, particles to be transferred are identified, and the corresponding elements of `x_part` are copied into an auxiliary buffer, ordered by the destination process; the remaining elements of `x_part` are compacted in order to fill holes. Each process sends to the other processes the corresponding chunks of the auxiliary buffer, and receives the new-particle coordinates in the higher-index portion of the array `x_part`. This is a typical all-to-all communication; the fact that the chunk size is different for each destination process makes the `MPI_Alltoallv` call the tool of choice.

Finally, after reallocating the array `pressure`, subroutine `compute_pressure` is called. Pressure values at the boundary of the subdomain are then corrected by exchanging the locally-computed value with the neighbor process (subroutine `correct_pressure`), by means of `MPI_Send` and `MPI_Recv` calls. The true value is obtained by adding the two partial values. The array `pressure` can now be yielded to the subroutine `field_solver` for the next time iteration.

Note that, for the sake of simplicity, we referred, in the above description, to one-dimensional field arrays. In the real case we have to represent field informations by means of multi-dimensional arrays. This requires us to use MPI derived datatypes as arguments of MPI calls in order to communicate blocks of pages of such arrays.

INTEGRATION OF INTER-NODE AND INTRA-NODE DOMAIN DECOMPOSITION STRATEGIES

The implementation of the domain decomposition strategy for a PIC code at the shared-memory level in a high-level parallel programming environment like OpenMP has been discussed in Refs. (Di Martino et al. 2001a). It consists in further decomposing the node subdomain and assigning a pair of the resulting portions (we will refer to them as to "intervals", looking at the subdivision along one of the dimensions of the subdomain) along with the particles residing therein to each thread. This requires labelling particles according to the interval subdivision. The loop over particles in subroutine `compute_pressure` can be restructured as follows. A pair of parallel loops are executed: one over the odd intervals, the other over the even ones. A loop over the interval particles is nested inside each of the interval loops. Race conditions between threads are then removed from the pressure computation, because particles treated by different threads, will update different elements of `pressure` as they belong to different, not adjacent, intervals. Race conditions can still occur, however, in the labelling phase, in which each particle is assigned, within a parallel loop over particles, to its interval and labelled by the incremented value of a counter: different threads can try to update the counter of a certain interval at the same time. The negative impact of such race conditions on the parallelization efficiency can be contained by avoiding to execute a complete labelling procedure for all the particles at each time step, while updating such indexing "by intervals" only in correspondence to particles that have changed interval in the last time step (Di Martino et al. 2001a).

In integrating the inter-node domain-decomposition strategy with the intra-node one, the most delicate issue is represented by the need of containing the effect of the labelling-procedure race conditions, and, hence, identifying particles whose interval indexing cannot be maintained. Two factors make such a task more complicate in comparison with the simple shared-memory case: the subdomain rearranging (due to load balancing) and the particle migration from one subdomain to another. The former factor may even make the previous-step interval subdivision meaningless; we then choose to reset the interval assignment of particles after each invocation of the load-balancing procedure:

```
...
call send_receive_cells(...)
n_cell=n_cell_new
n_part=n_part+n_part_left+n_part_right
call assign_to_interval(x_part)
...
```

The latter factor enriches the family of particles that change interval: beside those leaving their interval for a different interval of the same subdomain, particles leaving the subdomain or coming from a different subdomain have to be taken into account. In the framework of such domain-domain decomposition strategy, subroutine `particle_transfer` will then include, in addition to the check on inter-subdomain particle migration, the check on inter-interval migration. Particles that left the subdomain will affect the internal ordering of the original interval only; particles who came into the subdomain will be assigned to the proper interval, then affecting only the internal ordering of the new interval; particles that changed interval without leaving the subdomain will continue to affect the ordering of both the original and the new interval.

The analysis aimed to identify, in subroutine `particle_transfer`, inter-subdomain or inter-interval migrating particles can still be performed by a parallel loop over intervals (with a nested loop over interval particles). Race conditions can occur when updating the counters related to particles leaving the subdomain and those related to particles reaching a new interval without changing subdomain. Race conditions can also be presented, of course, when parallelizing the interval assignment of the particles imported from the others subdomains.

Table 1: Elapsed Times (in Seconds) for the Different Procedures of a 3-D Skeleton-Code Implementation of the Decomposition Strategy at Different Pairs n_{node}/n_{proc}, along with the Overall Speed-Up Values

	1/1	*1/2*	*2/1*	*2/2*	*3/1*	*3/2*	*4/1*	*4/2*
Load checking $(\times 10^{-4})$	0.13	0.13	4.38	4.87	7.20	7.95	11.2	11.0
Pushing	8.52	4.34	4.26	2.14	2.77	1.39	2.10	1.06
Particle transfer	1.27	0.68	0.65	0.36	0.44	0.24	0.33	0.19
Pressure	7.82	3.98	3.85	2.00	2.51	1.34	1.92	1.05
Speed-up	**1.00**	**1.88**	**1.99**	**3.54**	**3.01**	**5.84**	**4.04**	**7.56**

The domain-domain hierarchical decomposition strategy has been implemented for a 3-D skeleton-code. The parallel version of the code has been successfully tested on an IBM SP parallel system, equipped with four 2-processors SMP Power3 nodes, with clock frequency of 200 MHz and 1 GB RAM. Two sources of departures from ideal

speed-up performances can be identified: the computation and communication overheads associated to the sporadic load-balancing procedure (invoked if `i_check.eq.1`) and those related to the every-time-step procedures. Concerning the former source, the execution frequency of that potentially heavy procedure depends positively on the strength of particle dynamics and negatively on the size of the particle-array oscillation band: a larger strength implies a larger migration; a larger size implies a less frequent *load balancing* invocation (but also a larger memory waste). There is apparently a trade-off between parallelization efficiency and memory saving. Note however that the memory waste does not affect the scalability of the problem size with the number of processors, as it is distributed among them. Then we can fix the oscillation-band size in such a way to maintain the load-balancing invocation frequency and its effect on the overall speed-up values at very low levels.

Concerning the latter source of non-ideality, the average elapsed time (in seconds) for the different every-time-step procedures are reported in Table 1 for different pairs n_{node}/n_{proc}, along with the overall speed-up values. Note that the "Pressure" procedure includes both `compute_pressure` and `correct_pressure` subroutines. A case with a spatial grid of $128 \times 32 \times 16$ cells and $N_{part} = 1048576$ particles has been considered.

We note that, for the considered case, the elapsed times decrease with the total number of processors for *pushing* and *pressure* procedures. This is obvious for the former, which is an intrinsically parallel procedure. The latter contains instead the correction phase, which introduces both computation and communication overheads. Such overheads are however overwhelmed, for the considered case, by the standard loop over particles, which, once rearranged in the described nested-loop way, is intrinsically parallel too.

A similar situation occurs for the *particle transfer* procedure, which includes a loop over (all) particles, aimed to identifying migrating particles, and the reassignment phase, reserved to such particles. The former takes advantage from increasing number of nodes and/or processors. The latter becomes relatively more important as these numbers are increased, because of the larger surface-to-volume ratio of each domain portion and the larger corresponding fraction of migrating particles. In the considered case, the benefits are apparently larger than the costs. Such conclusion could be however deeply modified by a stronger particle dynamics and/or for larger number of nodes and processors (i.e., for larger migration rates).

For the *load checking* procedure the elapsed times increase with the number of nodes (the weak dependence on the number of processors has to be considered within the noise level). This is consistent with the n_{node} dependence of both communication and computation loads associated to the corresponding subroutine.

CONCLUSIONS

In this paper we have faced the task of developing a hierarchically-combined domain-domain decomposition strategy for hierarchically structured distributed-shared memory architectures. Such a task is motivated by the need of minimizing the memory requirement, which constitutes one of the major obstacles in performing realistic particle-in-cell plasma simulations. We have discussed an implementation of this strategy based on the combination of MPI, at the inter-node (distributed memory) level, and OpenMP, at the intra-node (shared memory) one. By applying the decomposition strategy to a 3-D skeleton code, we have shown that a good scalability of the maximum simulable domain size with the number of computational nodes can be obtained (as it does not require replicated arrays), along with a satisfactory parallelization efficiency.

A comparison between the four different MPI+OpenMP combined strategies that are obtained by composing particle and domain decompositions at the inter-node and intra-node levels, will be given, for the same skeleton code, in a future paper. The application of these strategies to a real, large scale, 3-D PIC code is in progress.

REFERENCES

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

Briguglio, S.; B. Di Martino; and G. Vlad. 2002. "Workload Decomposition Strategies for Hierarchical Distributed-Shared Memory Parallel Systems and their Implementation with Integration of High Level Parallel Languages". *Concurrency and Computation: Practice and Experience*, Wiley, Vol. 14, No.11, 933-956.

Cybenko, G. 1989. "Dynamic Load Balancing for Distributed Memory Multiprocessors". *J. Parallel and Distributed Comput.*, 7, 279-391.

Di Martino, B.; S. Briguglio; G. Vlad; and G. Fogaccia. 2001a. "Workload Decomposition Strategies for Shared Memory Parallel Systems with OpenMP". *Scientific Programming, IOS Press*, 9, No. 2-3, 109-122.

Di Martino, B.; S. Briguglio; G. Vlad; and P. Sguazzero. 2001b. "Parallel PIC Plasma Simulation through Particle Decomposition Techniques". *Parallel Computing*, 27, No. 3, 295-314.

Ferraro, R.D.; P. Liewer; and V.K. Decyk. 1993. "Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code". *J. Comput. Phys.*, 109, 329-341.

Fox, G.C.; M. Johnson; G. Lyzenga; S. Otto; J. Salmon; and D. Walker. 1988. *Solving Problems on Concurrent Processors*. Prentice-Hall, Englewood Cliffs, New Jersey.

High Performance Fortran Forum: High Performance Fortran Language Specification, Version 2.0. 1997. Rice University.

Hockney R.W. and J.W. Eastwood. 1981. *Computer Simulation Using Particles*. McGraw-Hill, New York.

OpenMP Architecture Review Board: OpenMP Fortran Application Program Interface, Version 1.0. October 1997.

Wesson J. 1997. *Tokamaks*. Clarendon Press, Oxford.