Hierarchical MPI+OpenMP implementation of parallel PIC applications on clusters of Symmetric MultiProcessors

Sergio Briguglio[1], Beniamino Di Martino[2],
Giuliana Fogaccia[1] and Gregorio Vlad[1]

[1] Associazione EURATOM-ENEA sulla Fusione, C.R. Frascati,
C.P. 65, 00044, Frascati, Rome, Italy
{briguglio,fogaccia,vlad}@frascati.enea.it
[2] Dip. Ingegneria dell'Informazione,
Second University of Naples, Italy
beniamino.dimartino@unina.it*

Abstract. The hierarchical combination of decomposition strategies for the development of 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.

1 Introduction

Particle-in-cell (PIC) simulation consists [1] in evolving the coordinates of a set of N_{part} particles in certain fluctuating fields computed (in terms of particle contributions) only at the points of a discrete spatial grid and then interpolated at each particle (continuous) position. Two main strategies have been developed for the workload decomposition related to porting PIC codes on parallel systems: the *particle decomposition* strategy [5] and the *domain decomposition* one [7, 6]. Domain decomposition consists 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. Particle decomposition, instead, statically distributes the particle population among the processes, while assigning the whole domain (and the grid) to each process. As a general fact, the particle decomposition is very efficient and yields a perfect load balancing, at the expenses of memory overheads. Conversely, the domain decomposition does not require a memory waste, while presenting particle migration between different portions of the domain, which causes communication overheads and the need for dynamic load balancing [3, 6].

Such workload decomposition strategies can be applied both for distributed-memory parallel systems [6, 5] and shared-memory ones [4]. They can also be

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

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 two-level strategies applied to a specific application domain, namely the simulation of thermonuclear plasma confinement. In particular, we have designed and implemented the hierarchically combined particle-particle and particle-domain decomposition strategies, with the integrated use of HPF and OpenMP [2].

The task of a good scalability of the domain size with n_{node}, requires, however, to avoid the replication of the grid data proper of the particle decomposition at the distributed-memory level. The scenario of hierarchically-combined decomposition strategies has then to be completed by developing the domain-particle combination and, specially, the domain-domain one. A high-level data-parallel language like HPF is not suited, in these cases, to face problems like the inter-process particle migration and the related dynamic workload unbalance. We have then to resort to explicit message-passing libraries, such as MPI. Aim of this paper is discussing the MPI+OpenMP implementation of the integrated domain-particle and domain-domain decomposition strategies, with particular emphasis to the dynamic workload balancing technique we have devised and its MPI-based implementation.

In Sect. 2 we describe the inter-node, domain-decomposition strategy, adopted in the distributed-memory context, along with its MPI implementation, while the integration of such inter-node strategy with both intra-node particle and domain decomposition strategies is discussed in Sect. 3.

2 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, different portions of the physical domain and of the corresponding grid are assigned to the n_{node} different nodes, along with the particles that reside on them. This approach yields benefits and problems that are complementary to those yielded by the particle-decomposition one [5]: on the one hand, the memory resources required to each node are approximately reduced by the number of nodes; an almost linear scaling of the attainable physical-space resolution (i.e., the maximum size of the spatial grid) with the number of nodes is then obtained. On the other hand, 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

- 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);
- particles that moved from one subdomain to another because of particle pushing must be transferred from the original process to the new one;
- 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, before giving some detail on the implementation of such procedures:

```
 call field_solver(pressure,field)
 call check_loads(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 transfer_particles(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 band of oscillation (e.g., ±10%). This band is defined in such a way to prevent, under normal conditions, index overflows and, at the same time, to avoid excessive load unbalancing. One of the processes (the MPI rank-0 process) collects, in subroutine `check_loads`, 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. 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 or 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 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 transfer_particles. 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.

3 Integration of the inter-node domain decomposition with intra-node particle and domain decomposition strategies

The implementation of particle and domain decomposition strategies for a PIC code at the shared-memory level in a high-level parallel programming environment like OpenMP has been discussed in Refs. [4, 2]. We refer the reader to those papers for the details of such implementation. Let us just recall the main differences between the two intra-node approaches, keeping in mind the inter-node domain-decomposition context. In order to avoid race conditions between different threads in updating the array `pressure`, the particle-decomposition strategy introduces a private auxiliary array with same rank and size of `pressure`, which can be privately updated by each thread; the updating of `pressure` is then obtained by a reduction of the different copies of the auxiliary array. The domain-decomposition strategy consists, instead, 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 `pressure` can be restructured as follows. A pair of parallel loops are executed: one over to 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 [4].

The integration of the inter-node domain-decomposition strategy with the intra-node particle-decomposition one does not present any relevant problem. The only fact that should be noted is that, though the identification of particles to be transferred from one subdomain to the others can be performed, in subroutine `transfer_particles`, in a parallel fashion, race conditions can occur in updating the counters related to such migrating particles and their destination subdomains. The updating has then to be protected within `critical` sections.

The integration with the intra-node domain-decomposition strategy is more complicate. The need of containing the effect of the labelling-procedure race conditions requires indeed identifying particles whose interval indexing cannot be maintained. Two factors make such a task more delicate 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

		1/1	*1/2*	*2/1*	*2/2*	*3/1*	*3/2*	*4/1*	*4/2*
Load balancing $\times 10^{-4}$	*d-p*	0.14	0.14	4.32	4.25	7.28	7.48	9.59	9.60
	d-d	0.13	0.13	4.38	4.87	7.20	7.95	11.2	11.0
Pushing	*d-p*	8.58	4.39	4.28	2.14	2.78	1.40	2.11	1.06
	d-d	8.52	4.34	4.26	2.14	2.77	1.39	2.10	1.06
Particle transfer	*d-p*	0.85	0.43	0.42	0.21	0.29	0.15	0.22	0.11
	d-d	1.27	0.68	0.65	0.36	0.44	0.24	0.33	0.19
Pressure	*d-p*	9.82	4.89	4.82	2.42	3.16	1.58	2.40	1.20
	d-d	7.82	3.98	3.85	2.00	2.51	1.34	1.92	1.05

Table 1. Elapsed times (in seconds) for the different procedures of 3-D skeleton-code implementations of the domain-particle (*d-p*) and the domain-domain (*d-d*) decomposition strategies at different pairs n_{node}/n_{proc}.

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 `transfer_particles` 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 `transfer_particles`, 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 (as in the above domain-particle case) 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.

Preliminary results obtained for a 3-D skeleton-code implementations of the domain-particle (*d-p*) and the domain-domain (*d-d*) hierarchical decomposition

	1/1	*1/2*	*2/1*	*2/2*	*3/1*	*3/2*	*4/1*	*4/2*
d-p	0.92	1.82	1.83	3.39	2.78	4.98	3.71	7.32
d-d	1.00	1.88	1.99	3.54	3.01	5.84	4.04	7.56

Table 2. Speed-up values for the 3-D skeleton-code implementations of the domain-particle and the domain-domain decomposition strategies at different pairs n_{node}/n_{proc}.

strategies are shown in Table 1. The elapsed time (in seconds) for the different procedures are reported for different pairs n_{node}/n_{proc}. 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. The overall speed-up values, defined as the ratio between the serial-execution elapsed times and the parallel execution ones, are reported in Table 2. These results have been obtained by running the code on an IBM SP parallel system, equipped with four 2-processors SMP Power3 nodes, with clock frequency of 200 MHz and 1 GB RAM.

We note that, for the considered case, the elapsed times decrease with the total number of processors for *pushing*, *particle transfer* and *pressure* procedures, while it increases for the *load balancing* procedure. This result can strongly depend, as far as the *particle transfer* procedure is concerned, on the rate of particle migration (which, in turn, depends on the specific dynamics considered).

Finally, we note that the domain-domain decomposition strategy comes out to be, for this case, more efficient than the domain-particle one. This is due to the need of reducing, in the framework of the latter decomposition strategy, the private copies of the array `pressure`.

References

1. Birdsall, C.K., Langdon, A.B.: Plasma Physics via Computer Simulation. (McGraw-Hill, New York, 1985).
2. Briguglio, S., Di Martino, B., Vlad, G.: 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**, n. 11, (2002) 933–956.
3. Cybenko, G.: Dynamic Load Balancing for Distributed Memory Multiprocessors. J. Parallel and Distributed Comput., **7**, (1989) 279–391.
4. Di Martino, B., Briguglio, S., Vlad, G., Fogaccia, G.: Workload Decomposition Strategies for Shared Memory Parallel Systems with OpenMP. Scientific Programming, IOS Press, **9**, n. 2-3, (2001) 109–122.
5. Di Martino, B., Briguglio, S., Vlad, G., Sguazzero, P.: Parallel PIC Plasma Simulation through Particle Decomposition Techniques. Parallel Computing **27**, n. 3, (2001) 295–314.
6. Ferraro, R.D., Liewer, P., Decyk, V.K.: Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code, J. Comput. Phys. **109**, (1993) 329–341.
7. Fox, G.C., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D.: Solving Problems on Concurrent Processors (Prentice Hall, Englewood Cliffs, New Jersey, 1988).