Workload Decomposition for Particle Simulation Applications on Hierarchical Distributed-Shared Memory Parallel Systems with integration of HPF and OpenMP

Sergio Briguglio
Associazione Euratom-ENEA sulla Fusione, C.R. Frascati, C.P. 65 - I-00044 - Frascati, Rome, Italy.
briguglio@frascati.enea.it

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

Gregorio Vlad
Associazione Euratom-ENEA sulla Fusione, C.R. Frascati, C.P. 65 - I-00044 - Frascati, Rome, Italy.
vlad@frascati.enea.it

ABSTRACT

A crucial issue in programming hierarchical distributed-shared memory systems is the *workload decomposition*. In this paper we address this issue in the framework of porting typical *particle in cell* (PIC) applications on hierarchical distributed-shared memory parallel systems. The workload decomposition we have devised consists in a two-stage procedure: a higher-level decomposition among the computational nodes, and a lower-level one among the processors of each computational node. We have implemented the described work decomposition without large programming effort by using and integrating the high-level languages High Performance Fortran and OpenMP.

1. INTRODUCTION

Hierarchical distributed-shared memory multiprocessor architectures are gaining more and more importance for High Performance Computing.

Bus-based shared memory multiprocessor systems (SMPs) are rapidly spreading out, expecially in the industrial and commercial world, at a wide range of scale. They range from two-four processor configurations typical of desktop systems, to large servers moving to one hundred processors. In addition, advances in Very Large Scale Integration (VLSI) technology are pushing large scale production of multiprocessor chips. Rapidly increasing availability and cost-effectiveness of SMP systems are imposing them as the composing nodes of large scale distributed memory architectures: current examples range from IBM SP to Compaq/Quadrics QM to SGI Origin 2000. At the other end of the scale, *clusters of SMPs*, where moderately sized multiprocessor workstations and PCs are connected with a high-bandwidth interconnection network, are increasingly established and used to provide high performance computing at a low cost.

Hierarchical distributed-shared memory multiprocessor architectures are thus emerging as a flexible architectural model: it combines the two paradigms of shared and distributed address space in one system, thus exploiting at best the properties of hierarchical parallelism present in most applications.

Current parallel programming models are not yet designed to take into account hierarchies of both distributed and shared memory parallelism into one single framework. Programming hierarchical distributed-shared memory systems is currently achieved by means of integration of environments/languages/libraries individually designed for either shared or distributed address space model. They range from explicit message-passing libraries such as MPI (for the distributed memory level) and explicit multithreaded programming (for the shared memory level), at a low abstraction level, to high-level parallel programming environments/languages, such as High Performance Fortran (HPF) [9] (for the distributed memory level), and OpenMP [13] (for the shared memory level).

A crucial issue in programming hierarchical distributed-shared memory systems is the *work decomposition*, i.e. the assignment of tasks composing the parallel application under development among processors. The adoption of an appropriate workload decomposition is crucial for achieving the desired performance results. Primary performance goals of work decomposition are balancing the workload among processes/threads, reducing interprocess communication (for the distributed memory level) or data access contention (for the shared memory level) and reducing the overhead due to managing the work decomposition itself.

Work decomposition task can be accomplished without large programming effort with use and integration of high-level languages such as HPF and OpenMP, expecially when the issue is porting large sequential codes to hierarchical archi-

tectures. While the developer is leveraged, by the adoption of high-level languages, from a large code restructuring effort, particular care must be payed in order to achieve performance goals, because such languages allow for a low level of control over issues such as load balancing, optimization of interprocess communication (for distributed memory) or locality of data access (for shared memory).

In this paper we address the issue of work decomposition on hierarchical distributed-shared memory parallel systems, considering the class of *particle in cell* (PIC) simulations as case study. The PIC simulation consists [2] in evolving the phase-space coordinates of a particle population in certain fields computed (in terms of particle contributions) only at the points of a discrete spatial grid and then interpolated at each particle (continuous) position. The PIC application category represents a suitable case study for the issue of work decomposition on hierarchical distributed-shared memory parallel systems. In fact, two workload decomposition strategies have been devised for this application category: the *domain decomposition* [11, 8] strategy and the *particle decomposition* [6] one. There is a trade-off between the respective merits of each of these methods in terms of little program restructuring effort, high time efficiency and low memory occupancy. More precisely, the *particle decomposition* approach comes out to be preferable with respect to programming effort and amount of interaction among tasks (and, thus, communication overhead on distributed memory architectures) while the *domain decomposition* one yields lower memory requirements. When programming hierarchical distributed-shared memory systems, besides to extend one single strategy to both the distributed and the shared memory decomposition, thus emphasizing the specific features of that strategy, it is possible to integrate the two stategies in a hierarchical way in order to get a suited balance of merits and defects. The class of PIC applications is then a relevant case study for the systems under consideration.

The workload decomposition we have devised for the execution of PIC applications on hierarchical architectures consists in a two-stage procedure: a higher-level decomposition among the computational nodes, and a lower-level one among the processors of each computational node. The inter-node, *particle decomposition* strategy is adopted at the distributed-memory level. Two alternative decomposition strategies, based respectively on *particle decomposition* and *domain decomposition*, are instead considered for the intra-node, shared-memory, level.

With regard to the implementation, we adopt the high-level language approach thus implementing the distributed memory decomposition in HPF, and the shared memory one in OpenMP, and integrating the two programming environments by means of the EXTRINSIC feature of the HPF language.

The paper is structured as follows. Section 2 describes the main physical and computational aspects of the chosen application. The integration of the inter-node decomposition and the intra-node one in the framework of the high-level languages HPF and OpenMP is outlined in Sect. 3. The inter-node, *particle decomposition* strategy, adopted in the distributed-memory context, is presented in Sect. 4. Different, alternative decomposition strategies for the intra-node shared-memory parallelization, based on *particle decomposition* and, respectively, *domain decomposition* approaches, are discussed in Sect. 5. Experimental results obtained with the Hybrid MHD-Gyrokinetic Code (HMGC) [3] PIC code and validating performance models are reported in Sect. 6. Conclusions are drawn in Sect. 7.

2. THE PLASMA PARTICLE SIMULATION APPLICATION

The investigation of turbulent plasma behaviour deals with solving the Vlasov equation (the collisionless version of the Boltzmann equation),

$$\frac{dF}{dt} \equiv \frac{\partial F}{\partial t} + \sum_i \frac{dZ^i}{dt}\frac{\partial F}{\partial Z^i} = 0, \tag{1}$$

for the plasma particle distribution function $F(t, Z)$, with Z indicating the whole set of phase-space coordinates. In the above equation, the phase-space "velocities", dZ^i/dt, have a known dependence on the fluctuating electromagnetic fields, which can be in turns computed in terms of certain moments – e.g., pressure – of the particle distribution function.

A formal, approximate solution of the Vlasov equation can be obtained by representing the distribution function $F(t, Z)$ by its discretized form,

$$\begin{aligned} F(t, Z) &\equiv \int dZ' F(t, Z')\ \delta(Z - Z') \\ &\approx \sum_l w_l(t)\, \delta(Z - Z_l), \end{aligned} \tag{2}$$

where $w_l(t) \equiv \Delta_l(t)\ F(t, Z_l(t))$ is the number of physical particles contained in the volume element Δ_l around the phase-space marker Z_l. It is immediate to show that such an expression of F satisfies the Vlasov equation if each marker evolves in time according to the equations of motion for physical particles and the corresponding number of particles w_l is conserved (constant in time). Such phase-space markers can then be interpreted as the phase-space coordinates of a set of N_{part} macroparticles, each of them representing – by its weight – a cluster of (non mutually interacting) physical particles. Particle simulation [2] then consists in numerically evolving the phase-space coordinates of the macroparticles (simulation particles) in a selfconsistent way, i.e., by computing the electromagnetic fields, at each time step, consistently with the particle distribution (through the calculation of its suited moments).

The most widely used method for particle simulation is represented by the PIC approach. At each time step, a PIC simulation code

- computes the electromagnetic fields only at the points of a discrete spatial grid (*field solver phase*)
- interpolates them at the (continuous) particle positions in order to evolve particle phase-space coordinates (*particle pushing phase*);
- collects particle contribution to pressure at the grid points to close the field equations (*pressure computation phase*).

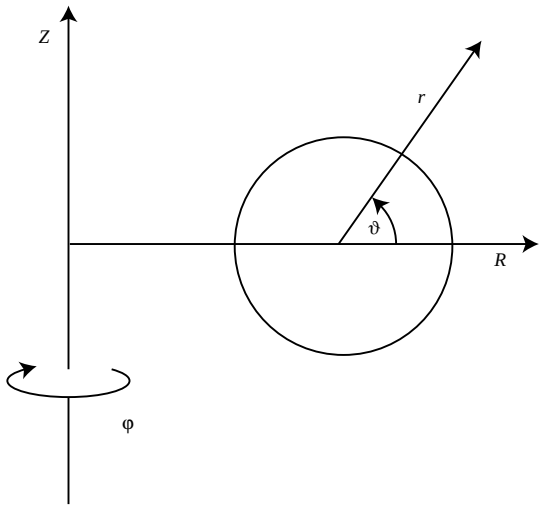

Figure 1: Toroidal coordinate system (r, ϑ, φ) for a tokamak plasma equilibrium.

The interest in investigating systems characterized by a very large number of particles makes the adequate exploitation of parallel architectures a high-priority task. Several contributions exist, in literature, on this issue, mainly concerning parallelization on distributed architectures (see, for example, Refs. [11, 7, 4, 12, 1, 6]). Most of them [11, 7, 4, 12, 1] are based on the *domain decomposition* strategy, while the particle decomposition approach has been adopted in Ref. [6]. Several different aspects have been addressed in these papers: Ref. [7] and, especially, Ref. [4] compare the results obtained by the parallelized code on different architectures; Ref. [12] discusses the benefits of the object-oriented approach to the parallel PIC simulation; Refs. [1] and [6] present the results of the implementation of parallel PIC codes in HPF (with the former one also comparing such results with those obtained in a Message Passing framework).

Here we consider the parallelization on hierarchical distributed-shared memory architectures of a specific PIC code, HMGC [3], consisting of approximatively 16,000 F77 lines distributed over more than 40 procedures. Particles move in a three-dimensional toroidal spatial domain, described in terms of quasi-cylindrical coordinates (see Fig. 1): the minor radius of the torus, r, and the poloidal and toroidal angles, ϑ and φ, respectively. Each particle is characterized by its phase-space coordinates (real space and velocity space ones) and its weight w.

The most relevant computational effort is concentrated in the loops over the particle population related, respectively, to the pushing phase and to the pressure computation one. As the parallelization of the pushing loop is trivial in comparison with that of the pressure loop, we limit our discussion to the latter one. Such a loop can be schematized by the following one:

```
real*8, dimension (n_r,n_theta,n_phi):: p
real*8, dimension (n_part) :: r,...,w
p = 0.
do l = 1,n_part
 j_r = f_r(r(l))
 j_theta  = f_theta(theta(l))
 j_phi = f_phi(phi(l))
 p(j_r,j_theta,j_phi) = p(j_r,j_theta,j_phi)
&                       + h(r(l),...,w(l))
enddo
```

Here, `n_part` ($\equiv N_{part}$) is the number of particles, and `f_r`, `f_theta` and `f_phi` are nonlinear functions of the corresponding real-space particle coordinates, determining the indices of the closest of the $n_r \times n_\vartheta \times n_\varphi$ spatial grid points. The dots, " ... ", stay for the not explicitly reported phase-space coordinates. The pressure `p` at that grid point receives a contribution from the particle determined by the function `h`, which takes into account the relative position of the particle and the grid point, the velocity-space coordinate of the particle and its weight. In practice, a more complicate assignment prescription is adopted, which involves a higher number (eight) of neighbouring grid points, in order to get a less noisy description of the pressure field. In the spirit of the present discussion, however, we may neglect such details.

3. INTEGRATION OF HPF AND OPENMP

In this section we describe the technique we use to integrate the inter-node decomposition and the intra-node one, and thus to express the multiple level of parallelism, within the framework of the high-level languages used for their implementation, namely HPF and OpenMP.

Such an integration can be obtained at a negligible programming effort with the help of the HPF extrinsic procedures `HPF_LOCAL`. High Performance Fortran programs may call non-HPF subprograms as *extrinsic procedures* [9]. This allows the programmer to use non-Fortran language facilities, handle problems that are not efficiently addressed by HPF, hand-tune critical kernels, or call optimized libraries. An extrinsic procedure can be defined as explicit SPMD code by specifying the local procedure code that is to execute on each computational node. High Performance Fortran provides a mechanism for defining local procedures in a subset of HPF that excludes only data mapping directives, which are not relevant to local code. If a subprogram definition or interface uses the extrinsic-kind keyword `HPF_LOCAL`, then the HPF compiler will assume that the subprogram is coded as a local procedure. All distributed HPF arrays passed as arguments by the caller to the (global) extrinsic procedure interface are logically divided into pieces; the local procedure executing on a particular computational node sees an array containing just those elements of the global array that are mapped to that node. A call to an extrinsic procedure results in a separate invocation of a local procedure on each node. The execution of an extrinsic procedure consists of the concurrent execution of a local procedure on each executing node. Each local procedure may terminate at any time by executing a `RETURN` statement. However, the extrinsic procedure as a whole terminates only after every local procedure has terminated.

In our case, we will use the extrinsic mechanism to embed the computations that can express multiple levels of parallelism (inter- and intra-node) into calls to extrinsic proce-

dures. Each local procedure executing on a given node will manage only the portion of the arrays assigned to that node. The bodies of the extrinsics can therein be parallelized at the intra-node, shared-memory level, by inserting suited OpenMP directives. The extrinsic procedures are then simply compiled by an OpenMP compiler, while the calling HPF programs is compiled by a HPF compiler; finally, the resulting objects are linked by the HPF linker.

In the next sections we will discuss in detail the decomposition strategies we adopt at the inter-node (HPF) level and the intra-node (OpenMP) one.

4. INTER-NODE DECOMPOSITION STRATEGY

Standard *domain decomposition* [11, 8] techniques assign different portions of the physical domain and the corresponding portions of the grid to different computational nodes, together with the particles that reside on them. The distribution of all the arrays among the computational nodes gives this method an intrinsic scalability of the maximum domain size that can be simulated with the number of nodes. On the opposite side, an important problem with these techniques is given by the need of a dynamic load balancing, associated to particle migration from one portion of the domain to another. Such a load balancing can make the parallel implementation of a serial code complicate, expecially with high-level languages, besides introducing extra computation and communication overheads.

In order to avoid facing the migration of particles from one domain portion to another, which in practice precludes the usage of a high-level programming language like HPF, we adopt the *particle decomposition* [6] approach to the inter-node parallelization of HMGC: particle population is statically distributed among processors, while the data relative to grid quantities are replicated. As no particle has to be transferred from one node to another, load balancing is automatically enforced; moreover, no communication overhead associated to particle migration affects the parallelization efficiency. On the opposite side, the linear scaling of the spatial resolution with nodes, possible, in principle, in the framework of a domain decomposition, is lost: the maximum achievable resolution is limited by the Random Access Memory (RAM) resources of the single node (different from the domain decomposition case, the grid arrays are replicated), and increasing the number of nodes only allows increasing the number, $N_{ppc} \equiv N_{part}/N_{cell}$, of particles per cell, i.e. the velocity-space resolution. Moreover, specific communication and computation overheads are introduced, because partial contributions to particle pressure coming from different portions of the population must be summed together, at each time step, before updating the electromagnetic fields. Both memory and efficiency bottlenecks, however, scale with the grid size and come out to be negligible as far as the number of particles per cell treated by each of the n_{node} computational nodes is large: $N_{ppc}/n_{node} \gg 1$ [6].

The implementation of *particle decomposition* parallelization in HPF is, in principle, relatively straightforward and has been discussed in Ref. [6]. In particular, HPF directives for data distribution can be applied to all the data structures (e.g., `r(n_part)`) related to the particle quantities. By embedding the particle loops related to the particle-pushing and the pressure-updating phases into calls to extrinsic procedures, the distribution of the loop iterations among the nodes according to the *owner computes* rule applied to the distributed data is automatically enforced. Different from the particle-pushing phase, which is inherently parallel, the updating of the particle pressure at the grid points presents, however, two strictly linked problems: (i) such a quantity is replicated, and thus must be kept consistent among the nodes; (ii) each element of the pressure array `p` takes contribution from particles that reside on different nodes. The strategy adopted to solve this problem relies on the associative and distributive properties of the updating laws for the pressure array with respect to the contributions given by every single particle: the computation for each update is split among the nodes into partial computations, involving the contribution of the local particles only; then the partial results are reduced into global results, which are broadcasted to all the nodes.

The scheme to handle with this "inhibitor of parallelism" within the loops over the particles, can be implemented in HPF by introducing a temporary data structure, `p_par(n_r, n_theta, n_phi,:)`, augmented by one dimension, with extent equal to the number of available nodes, and distributed, along the added dimension, over the nodes. Each of the distributed "pages" will store the partial computations of the pressure, which include the contributions of the particles that are local to each node. At the end of the pressure-loop iterations, the temporary data structure is reduced along the added dimension, and the result is assigned to the corresponding original data structure. This is implemented by using the HPF intrinsic reduction function `SUM`. The restructured calling HPF program then looks like the following:

```
       real*8, dimension (n_r,n_theta,n_phi):: p
       real*8, dimension
      &            (n_r,n_theta,n_phi,
      &             number_of_processors()):: p_par
       real*8, dimension (n_part) :: r,...,w
!HPF$ DISTRIBUTE (CYCLIC) :: r,...,w
!HPF$ ALIGN WITH w(:) :: p_par(*,*,*,:)

       INTERFACE
       EXTRINSIC(HPF_LOCAL)
      &subroutine extr_pressure(r,...,w,p_par)
       real*8, dimension(:), intent(in) :: r,...,w
       real*8, dimension(:,:,:,:),
      &                intent(out) :: p_par
!HPF$ DISTRIBUTE (CYCLIC) :: r,...,w
!HPF$ ALIGN WITH w(:) :: p_par(*,*,*,:)
       end subroutine extr_pressure
       END INTERFACE

       call extr_pressure(r,...,w,p_par)
       p(:,:,:) = SUM(p_par(:,:,:,:),dim=4)
```

and the local procedure becomes:

```
       subroutine extr_pressure(r,...,w,p_par)
       real*8, dimension(:), intent(in) :: r,...,w
       real*8, dimension(:,:,:,:),
      &                intent(out) :: p_par
       p_par = 0.
       do l=1, UBOUND(w,dim=1)
```

```
      j_r = f_r(r(l))
      j_theta  = f_theta(theta(l))
      j_phi = f_phi(phi(l))
      p_par(j_r,j_theta,j_phi,1)=
     &           p_par(j_r,j_theta,j_phi,1)
     &           + h(r(l),...,w(l))
      enddo
      end subroutine extr_pressure
```

Note that each local procedure executes only the set of loop iterations that access the particles local to the node (`l=1,UBOUND(w, dim=1)`), and updates the page of `p_par` assigned to it. At the end of the execution of the local extrinsic procedure, all the partial updates of the components of `p_par` are collected in the global-HPF-index-space `p_par`, which is then reduced to `p`.

5. INTRA-NODE DECOMPOSITION STRATEGY

Once completed the distributed memory work decomposition, in the framework of a *particle decomposition* approach, the issue of the intra-node shared-memory decomposition must be addressed.

The most natural parallelization strategy for shared memory architectures consists in distributing the loop iterations over the particles among the different processors, without respect to the portion of the domain in which each particle resides. For this reason, such a technique can be referred to as a *particle decomposition* one. OpenMP allows for a straightforward implementation of this strategy by means of the `parallel do` directive. All the variables that are set and then used within the `do` loop are explicitly defined as `private`, with the others being `shared` by default. The immediate parallelization of the pressure loop is however inhibited, once again, by the updating of the array `p_par`. Such a computation is indeed an example of *irregular array-reduction operation* (cfr., e.g., [10]), where the elements to be reduced are the elements of the particle arrays `r`, ... , `w`, and the results of the reduction are the pressure values (the elements of the array `p_par`). The operation is a reduction because the updating function `h` has associative and distributive properties with respect to the contributions given by every single particle (i.e. with respect to the quantities `r(l)`, ... `w(l)`), but it is not regular because the indices of the updated element (`j_r`, `j_theta`, `j_phi`) are not induction variables of the loop, but functions of it (`j_r = f_r(r(l))`, `j_theta = f_theta(theta(l))`, `j_phi = f_phi(phi(l))`), having the property that for two given values of the induction variable l (l_i, l_j, with $l_i \neq l_j$) the corresponding computed values of the updating indices can be equal: (`j_r,j_theta,j_phi`)$_i$ =(`j_r,j_theta,j_phi`)$_j$. If particles that concur to updating the same element of the array `p_par` are assigned to different processors, a *race condition* can occur, if the processors try to update the array element "simultaneously". In such a case, the correctness of the parallel computation would be affected, because some of the contributions of the concurrent particles would be retained, with the others being lost.

The least expensive way, in terms of code restructuring effort, to protect the *critical sections* of the pressure loop from such race conditions (i.e., to ensure *mutual exclusion* among threads accessing shared data) consists in enclosing the updating of `p_par` by the OpenMP `critical` and `end critical` directives. The relevant portion of the extrinsic procedure described in Sect. 4 then becomes:

```
        p_par = 0.
!$OMP parallel do private(l,j_r,j_theta,j_phi)
        do l = 1,UBOUND(w,dim=1)
         j_r = f_r(r(l))
         j_theta = f_theta(theta(l))
         j_phi = f_phi(phi(l))
!$OMP critical
         p_par(j_r,j_theta,j_phi,1)=
       &           p_par(j_r,j_theta,j_phi,1)
       &           + h(r(l),...,w(l))
!$OMP end critical
        enddo
!$OMP end parallel do
```

Unfortunately, the intra-node serialization induced by the protected critical section on the shared access to the array `p` represents a bottleneck that heavily affects the performances (almost no speed-up) [5]. Such a bottleneck can be eliminated, at the expenses of memory occupation, by means of an alternative strategy, which resembles the one adopted within the distributed-memory parallelization: the computation for each update is split among the threads into partial computations, each of them involving only the contribution of the particles managed by the responsible thread; then the partial results are reduced into global ones. Such a strategy can be implemented (version *v1*; see next Section) by introducing an auxiliary array, `p_aux`, defined as a `private` variable with the same dimensions as `p`. Each processor works on a separate copy of the array (analogously to what obtained, in the inter-node decomposition, by the introduction of the augmented array) and there is no conflict between processors updating the same element of the array. At the end of the loop, however, each copy of `p_aux` contains only the partial pressure due to the particles managed by the owner processor. Each processor must then add its contribution, outside the loop, to the global, shared array `p_par` in order to obtain the whole-node contribution; the `critical` directive can be used to perform such a sum. The code section then reads as follows:

```
        p_par = 0.
!$OMP parallel private(l,j_r,j_theta,j_phi,p_aux)
        p_aux = 0.
!$OMP do
        do l=1,UBOUND(w,dim=1)
         j_r      = f_r(r(l))
         j_theta  = f_theta(theta(l))
         j_phi    = f_phi(phi(l))
         p_aux(j_r,j_theta,j_phi) =
       &           p_aux(j_r,j_theta,j_phi)
       &           + h(r(l),...,w(l))
        enddo
!$OMP end do
!$OMP critical
        p_par(:,:,:,1) = p_par(:,:,:,1) + p_aux(:,:,:)
!$OMP end critical
!$OMP end parallel
```

Note that this alternative strategy makes the execution of the `UBOUND(w, dim=1)` ($\approx N_{part}/n_{node}$) iterations of the

loop perfectly parallel. The serial portion of the computation is limited to the reduction of the different copies of `p_aux` into `p_par`. Then, its size scales with $N_{cell} \times n_{proc}$, with N_{cell} and n_{proc} being the number of grid points and processors (equal to the number of threads), respectively. Such product is much smaller than N_{part}/n_{node}, as long as the $N_{ppc} \gg n_{proc} \times n_{node}$. The price payed to obtain such an improvement is represented by the increased memory requirement: $N_{cell} \times n_{proc}$ more *real*8* elements must be stored on each node. In order to evaluate the effective relevance of such further requirement, this number has to be compared with the number of elements of the shared particle arrays stored on the same node. Under the above condition, $N_{ppc} \gg n_{proc} \times n_{node}$, the whole memory requirement is not significantly affected.

In those situations in which the memory overhead associated to the introduction of the auxiliary array `p_aux` comes out to be anyway intolerable, a completely different, *domain decomposition*, strategy can be adopted for the work distribution among the different processors of each computational node. This strategy, which requires a heavier restructuring of the code and, possibly, the addressing of load-balancing problems, consists in reordering the particle population according to the portion of domain in which each particle resides, and assigning a different portion to each processor. Such a reordering gives rise, once again, to the risk of race conditions (the particles belonging to a certain domain portion have to be counted within a particle loop, and the updating of the counter is a *critical* operation). Once assigned to the processors (threads), however, no further race condition occurs in updating the pressure array element, as loop iterations that could, in principle, concur to the updating of the same element are executed by the same thread.

A possible implementation of this strategy (version *v2a*) consists in decomposing the domain along one of its dimensions (e.g., along the radial coordinate) and is based on the following items:

- A particle loop is executed in order to identify the elementary portion of the domain in which each particle falls. The number of particles that belong to each portion is updated inside a critical section. Each particle is labelled, inside the same critical section, by an index that spans the population belonging to the corresponding elementary domain portion.
- The different elementary portions of the domain are assigned to each processor. Load balancing is enforced by adding elementary portions to a given-processor load until the number of particles assigned to the processor approximately equals the average number of particles per processor, $(N_{part}/n_{node})/n_{proc}$. Particles are then reordered according to the processor they belong to.
- The pressure loop is executed in the form of a parallel loop over processors in which a loop over the particle belonging to the processor is nested. Race conditions are automatically avoided.

Note that the load balancing is implemented within a loop over processors. It then causes negligible computation overheads. Moreover, different from the distributed memory context, it does not require any communication between processors. Note also that the increment of memory requirements is very contained (essentially limited to the integer labels of the reordered particles), and does not scale with the number of processors.

6. PERFORMANCE EVALUATION AND EXPERIMENTAL RESULTS

The tests we report in this Section have been carried out on a IBM SP parallel system, equipped with, among the others, two 8-processors SMP PowerPC nodes, with clock frequency of 200 MHz and 2 GB RAM, and four 2-processors SMP Power3 nodes, with clock frequency of 200 MHz and 1 GB RAM. The HPF code has been compiled by the IBM *xlhpf* compiler (an optimized native compiler for IBM SP systems), while the extrinsic OpenMP subroutines have been compiled by the IBM *xlf* (ver.6.01) compiler (an optimized native compiler for Fortran95 with OpenMP extensions for IBM SMP systems) under the `-qsmp=omp` option. The resulting objects are then linked by the HPF linker.

In the following we report performance results and validating models for the different HPF+OpenMP implementations, corresponding to coupling the inter-node *particle* workload decomposition with the two intra-node *particle* and *domain* decompositions described above.

The first version investigated, *v1*, implements the coupling of the inter-node particle decomposition described in Sect.4 and the intra-node particle decomposition described in Sect.5. A spatial grid with $n_r \times n_\vartheta \times n_\varphi = 32 \times 16 \times 8$ ($N_{cell} = 4096$) has been considered in this test, as well as in the following ones. Figure 2 shows the scaling of the speed-up (s_u) with respect to the number of processors, n_{proc}, at fixed number of (8-processors) nodes, $n_{node} = 2$. Four different values of the average number of particles per cell have been considered: from $N_{ppc} = 4$ to $N_{ppc} = 256$ (corresponding to the total number of particles ranging from $N_{part} = 16384$ to $N_{part} = 1048576$). Figure 3 reports the values of s_u versus the number of (2-processors) nodes, for $n_{proc} = 2$. Speed-up values refer only to the execution of the section related to the updating of the (whole) pressure array `p`. The speed-up has been defined as the ratio between the wall-clock time, t_s, yielded by the serial execution of the HPF+OpenMP version of the code and the one, t_{v1}, obtained by the parallel execution. By "serial execution" we mean the execution obtained, on the specific node used in the parallel executions, after performing the HPF and OpenMP compilations with the `-qnohpf` option and, respectively, without the `-qsmp=omp` option. The results for the case $n_{proc} = 1$ ($n_{node} = 1$) have been also reported in Fig. 2 (Fig. 3), corresponding to the execution on a single processor (node) of the parallel version. The same values of N_{cell} and N_{ppc} (and, hence, N_{part}) as in Fig. 2 have been considered.

In practice, the times t_s and t_{v1} can be evaluated as

$$\begin{aligned} t_s &\equiv (t_{reset} + t_{loop})_{\text{serial}} \ , \\ t_{v1} &\equiv t_{reset} + t_{parallel} + t_{sum} \ , \end{aligned}$$

where t_{reset} is the time needed to reset the values of the grid

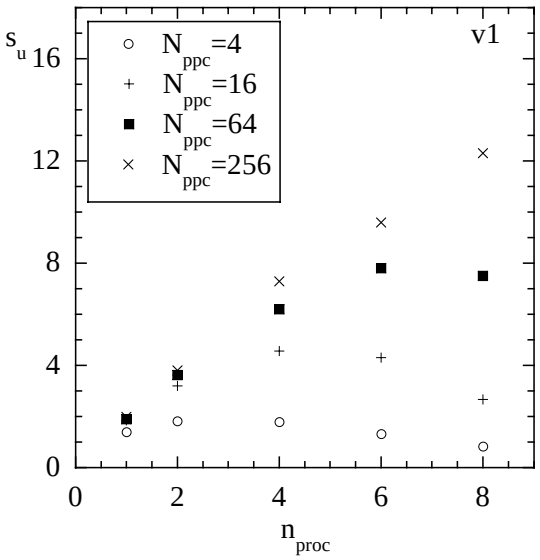

Figure 2: Speed-up of the pressure-updating phase versus the number of processors, at fixed number of (8-processors) nodes, $n_{node} = 2$, for the *particle decomposition* version, *v1*. Here, and in the following Figures $N_{cell} = 4096$. Four different values of the average number of particles per cell, from $N_{ppc} = 4$ to $N_{ppc} = 256$, have been considered (corresponding to N_{part} ranging from 16384 to 1048576).

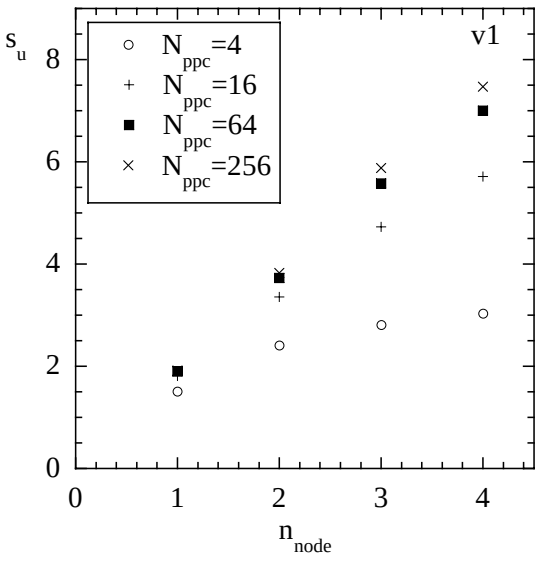

Figure 3: Speed-up versus the number of (2-processors) nodes, at fixed number of processors, $n_{proc} = 2$, for the *particle decomposition* version, *v1*. The other parameters are chosen as in the previous Figure.

arrays elements to zero; t_{loop} is the time required for the execution of the particle loop; $t_{parallel}$ is the time required for the execution of the whole `parallel` section (including the reduction of `p_aux` to `p_par`); t_{sum} is the time needed to reduce `p_par` to `p` (by the HPF intrinsic reduction function `SUM`). We can approximate the above expressions as follows

$$\begin{aligned} t_s &\approx t_{loop} \approx \alpha_{loop} N_{ppc} N_{cell}, \\ t_{v1} &\approx \alpha_{loop} \frac{N_{ppc} N_{cell}}{n_{proc} n_{node}} + \\ &\quad (\alpha_{red} n_{proc} + \alpha_{sum} \log n_{node}) N_{cell}, \end{aligned}$$

with α_{loop}, α_{red} and α_{sum} being suited coefficients, corresponding to the detailed operations and communications needed to perform the single loop iteration and the array reductions, respectively. Here we have taken into account the logarithmic character of the `SUM` reduction; moreover, we have neglected the time required to reset the arrays to zero, create and terminate threads and distribute the work among them. From such approximations, we expect a speed-up approximately given by

$$s_u \approx \frac{n_{proc}}{1 + \frac{n_{proc} n_{node}}{\alpha_{loop} N_{ppc}} (\alpha_{red} n_{proc} + \alpha_{sum} \log n_{node})} \quad . \qquad (3)$$

From Fig. 2, we observe indeed, in agreement with Eq. (3), that the speed-up values depart from the linear scaling with n_{proc} only for n_{proc} greater than a certain value, which is higher, the higher the average number of particles per cell, N_{ppc}, is. A significant departure from the linear scaling with n_{node} can be observed, in Fig. 3, only for the lowest values of N_{ppc}, because of the small number of nodes involved.

Assuming that $\alpha_{red} n_{proc} \gg \alpha_{sum} \log n_{node}$, we can conclude that such a "fully *particle decomposition*" approach (both for inter-node and intra-node decomposition) is efficient as far as $n_{proc}^2 n_{node} / N_{ppc}$ is lower than a certain threshold; for the specific code considered in this paper, such a threshold comes out to be approximately equal to 1. Under the same condition, the criterium ($n_{proc} n_{node} / N_{ppc} \lesssim 1$) for this approach not to be too memory demanding is *a fortiori* satisfied.

The second version considered, *v2a*, implements the coupling of the inter-node particle decomposition and the intra-node domain decomposition. Figure 4 shows the scaling of the speed-up with respect to n_{proc}, at fixed number ($n_{node} = 2$) of 8-processors nodes, obtained by this version. The wall-clock time can be evaluated, in this case, as follows:

$$\begin{aligned} t_{v2a} &\equiv t_{reset} + t_{pre-loop} + t_{assign} + \\ &\quad t_{reorder} + t_{loop} + t_{sum}. \end{aligned}$$

Here $t_{pre-loop}$ refers to the particle loop needed to identify the domain portion in which each particle falls, t_{assign} is the time required by the balanced assignment loop (over processors), $t_{reorder}$ and t_{loop} are the times spent in the reordering loop and, respectively, in the pressure updating loop (both over particles, in fact).

The speed-up values obtained with the version *v2a* do not seem to be very satisfactory. However, a significant improvement of the efficiency can be obtained, for specific (but rather common) applications characterized by a contained

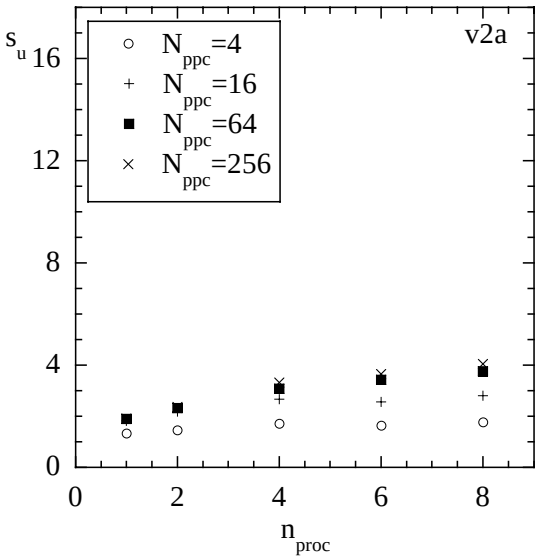

Figure 4: Speed-up versus the number of processors, at fixed number of (8-processors) nodes, $n_{node} = 2$, for the *domain decomposition* version, *v2a*.

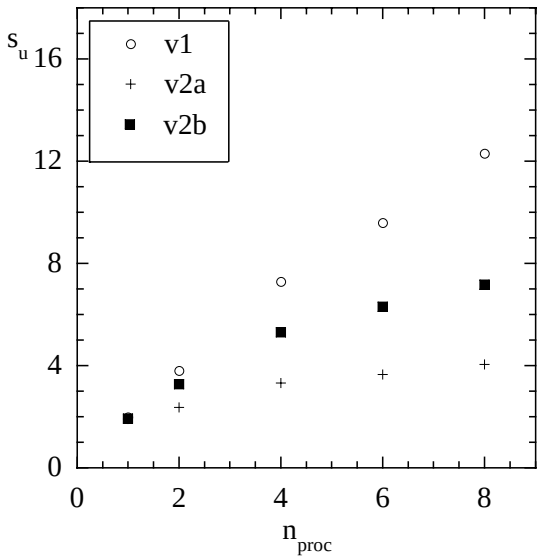

Figure 5: Comparison between the speed-up obtained, at different number of processors and fixed number of 8-processors nodes, $n_{node} = 2$, by the *domain decomposition* version, *v2a*, and a companion *selective reordering* version, *v2b*. The results of the *particle decomposition* implementation, *v1*, are also shown for reference. The case $N_{ppc} = 256$ is considered ($N_{part} = 1048576$).

particle migration per time step from one portion of the domain to another one, by limiting the reordering phase (and then the *critical* computation) to those particles that have changed domain portion in the last step. Their number can be indeed very low if it is possible to decompose the domain along a slowly-varying coordinate. Figure 5 shows a comparison between the results obtained by the version *v2a* and a companion version (*v2b*) which implements such a selective reordering. The results of the *particle decomposition* implementation, *v1*, are also shown for reference. The case $N_{ppc} = 256$ is considered, for example ($N_{part} = 1048576$).

We can conclude that, at least for the specific application here considered, this mixed "*particle-domain decomposition*" strategy represents an interesting compromise between the two competing targets – namely, high speed-up and low memory requirements.

7. CONCLUDING REMARKS

We have described a two-stages workload decomposition strategy for hierarchical distributed-shared memory systems, with application to a case study PIC simulation.

An inter-node, *particles decomposition* strategy is adopted at the higher, distributed-memory, level, while different intra-node alternative decomposition strategies, based on *particle decomposition* as well as *domain decomposition*, have been adopted for the lower, shared-memory, level.

Few different implementations of them, based on the use and integration of the high level languages HPF and OpenMP, have been discussed with regard to program restructuring effort, time efficiency and memory occupancy. We observe a trade-off between the merits of each method with respect to such requirements. More precisely, the *particle decomposition* approach yields, with a little programming effort, high speed-up values, due to the moderate amount of interaction among tasks (and, thus, communication overhead on distributed memory architectures), while requiring a supplementary memory resource level that scales with the number of processors and the size of the grid. This prevents any favourable scaling of the maximum spatial resolution level that can be reached in the simulation with the number of processors. On the opposite, the *domain decomposition* approach preserves such a good (essentially linear) scaling, but it requires a relevant programming effort and produces more moderate speed-up values.

We note that, while in a purely distributed memory system, the targets of a high-level language parallelization and low communication overhead would probably force adopting the *particle decomposition* strategy, with the consequence of a strong penalization in terms of memory occupancy requirements, in the present *hierarchical distributed-shared memory system* case some degree of freedom is left, which allows one to balance the competing targets of high time efficiency and low memory requirements.

8. REFERENCES

[1] E. Akarsu, K. Dincer, T. Haupt and G.C. Fox, Particle-in-Cell Simulation Codes in High Performance Fortran, in: Proc. SuperComputing '96 (IEEE, 1996).

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

[3] Briguglio, S., Vlad, G., Zonca, F., Kar, C.: Hybrid Magnetohydrodynamic-Gyrokinetic Simulation of

Toroidal Alfvén Modes. Phys. Plasmas **2** (1995) 3711–3723.

[4] V.K. Decyk, Skeleton PIC codes for parallel computers, Computer Physics Communications 87 (1995) 87-94.

[5] Di Martino, B., Briguglio, S., Vlad, G., Fogaccia, G.: Workload Decomposition Strategies for Shared Memory Parallel Systems with OpenMP. Submitted for publication to Scientific Programming, 2000.

[6] Di Martino, B., Briguglio, S., Vlad, G., Sguazzero, P.: Parallel Plasma Simulation in High Performance Fortran. In *High Performance Computing and Networking. Proceedings*, LNCS Vol. 1401, Springer, Berlin, 1998, p. 203–212.

[7] R.D. Ferraro, P. Liewer and V.K. Decyk, Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code, J. Comput. Phys. 109 (1993) 329-341.

[8] 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).

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

[10] Labarta, J., Ayguadè, E., Oliver, J., Henty, D.: New OpenMP Directives for Irregular Data Access Loops, Proc. of 2^{nd} *European Workshop on OpenMP - EWOMP'2000*, 14–15 September 2000, Edinburgh (UK).

[11] Liewer, P.C., Decyk, V.K.: A General Concurrent Algorithm for Plasma Particle-in-Cell Codes. J. Computational Phys. **85** (1989) 302–322.

[12] C.D. Norton, B.K. Szymanski and V.K. Decyk, Object Oriented Parallel Computation for Plasma Simulation, Communications of ACM 38(10) (1995) 88-100.

[13] OpenMP Architecture Review Board: OpenMP Fortran Application Program Interface, ver. 1.0, October 1997.