Workload Decomposition Strategies for Hierarchical Distributed-Shared Memory Parallel Systems and their Implementation with Integration of High Level Parallel Languages

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

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

Abstract. In this paper we address the issue of *workload decomposition* in programming 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.
By focussing on porting of a case study PIC application, 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) [11] (for the distributed memory level), and OpenMP [15] (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 architectures. 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 [3] 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. Two workload decomposition strategies have been devised for this application category: the *domain decomposition* [13, 10] strategy and the *particle decomposition* [8] one. There is a trade-off between the respective merits of each of these methods in terms of little program restructuring effort and high time efficiency, on one side, and low memory occupancy, on the other side. More precisely, the *particle decomposition* approach comes out to be preferable with respect to programming effort and time efficiency, 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 strategies 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 on *particle decomposition* and, respectively, on *domain decomposition*, are instead considered for the intra-node, shared-memory, level.

With regard to the implementation, we adopt the high-level language approach. We thus implement the distributed memory decomposition in HPF and the shared memory one in OpenMP, 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 decomposition strategies for the intra-node shared-memory parallelization, based on *particle* and *domain decomposition* approaches, respectively, are discussed in Sect. 5, which also reports experimental results obtained with the PIC Hybrid MHD-Gyrokinetic Code (HMGC) [4], as well as validating performance models. Conclusions on the validity of the proposed strategy are drawn in Sect. 6.

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 Z^i. 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 N-point discretized form,

$$\begin{aligned} F(t, Z) &\equiv \int dZ' F(t, Z')\ \delta(Z - Z') \\ &\approx \sum_{l=1}^{N} w_l(t)\, \delta(Z - Z_l), \end{aligned} \tag{2}$$

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} ($\equiv N$) macroparticles, each of them representing – by its weight w_l – a cluster of (non mutually interacting) physical particles. It can be easily shown that all the relevant parameters (e.g., the Debye length, λ_D) of this macroparticle plasma coincide with the corresponding parameters of the physical plasma, notwithstanding that the macroparticle density is much lower than the physical-particle one. Particle simulation [3] 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 the required moment of the distribution function (e.g., pressure) at the grid points to close the field equations (*pressure computation phase*).

The presence of a discrete grid, with spacing L_c between grid points, leaves the physically relevant dynamics related to the scales larger than L_c unaffected. At the same time, the condition corresponding to long-range particle interactions dominating over the short-range ones results in a much more relaxed requirement than the usual plasma condition, $n_0 \lambda_D^3 \gg 1$, with n_0 being the density of simulation particles. Indeed, it comes out to be satisfied if $n_0 L_c^3 \gg 1$.

The condition $n_0 L_c^3 \gg 1$ can be written as $N_{ppc} \equiv N_{part}/N_{cell} \gg 1$, where N_{cell} is the number of grid cells and N_{ppc} is the average number of particle per cell. As one is typically interested in simulating small-scale turbulence, an important goal in plasma simulation is represented by dealing with large number of cells and, *a fortiori*, for the above condition on N_{ppc}, large number of particles. Such a goal requires to resort to parallelization techniques aimed to distributing the computational loads related to the particle population among several processors.

Several contributions exist, in literature, on this issue, mainly concerning parallelization on distributed architectures (see, for example, Refs. [13, 9, 6, 14, 1, 8]). Most of them [13, 9, 6, 14, 1] are based on the *domain decomposition* strategy, while the *particle decomposition* approach has been adopted in Ref. [8]. Several different aspects have been addressed in these papers: Ref. [9] and, especially,

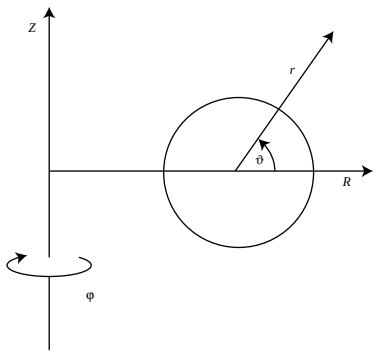

Fig. 1. Toroidal coordinate system (r, ϑ, φ) for a tokamak plasma equilibrium.

Ref. [6] compare the results obtained by the parallelized code on different architectures; Ref. [14] discusses the benefits of the object-oriented approach to the parallel PIC simulation; Refs. [1] and [8] 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 [4], developed, in the framework of controlled nuclear fusion research, for the investigation of the effects of energetic particles produced by fusion reactions on the dynamics of Alfvén modes in tokamaks [5]. The code consists 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. The pushing loop can be schematically represented as follows:

```
real*8, dimension (n_part) :: r,...
do l = 1,n_part
 r_l=r(l)
 ...
 r(l) = r_l + g_r(r_l,...)
 ...
enddo
```

with `n_part`$\equiv N_{part}$ being the number of particles and `g_r`, ... being rather complicate nonlinear functions of the particle phase-space coordinates, which give the time-step increment of the particle quantities in terms of the electromagnetic

fields at the neighbouring grid points. The dots, "...", stay for all the other, not reported, phase-space coordinates and related functions.

The pressure 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, `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 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* [11]. 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 procedures. 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* [13, 10] 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 one. 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* [8] 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 (indeed, different from the domain decomposition case, the grid arrays are replicated on each node), and increasing the number of nodes only allows increasing the number, N_{ppc}, of particles per cell, i.e. the velocity-space resolution. Specific communication and computation overheads are introduced, also in this case, be-

cause 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.

The relevance of both memory and efficiency problems, however, is directly related to the grid size. The former ones are negligible as far as the size of the grid arrays is much smaller than that of the portion of particle arrays distributed to each node. Such a condition can be expressed as $N_{cell} \ll (N_{part}/n_{node})$, or $N_{ppc}/n_{node} \gg 1$ [8].

The efficiency problems can be neglected if the amount of grid computation (not distributed) is much smaller than that of the particle one (distributed). For a workload decomposition among single-processor nodes, the condition for efficient parallelization would be the same as the above one, $N_{ppc}/n_{node} \gg 1$ [8]. For a decomposition among multi-processor nodes, the further intra-node workload decomposition must be taken into account. The efficiency condition becomes more stringent, as it will be discussed in Sect. 5.

The implementation of *particle decomposition* parallelization in HPF is, in principle, relatively straightforward and has been discussed in Ref. [8]. 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.

4.1 Particle pushing

The particle-pushing phase is inherently parallel, with no communication required by non-local accesses. Indeed, pushing of each particle consists in updating the particle coordinates in terms of the electromagnetic field interpolated at the particle position, with no dependence on other-particle quantities: both the replicated fields and the particle quantities are locally available. The HPF calling program assumes the form

```
       real*8, dimension (n_part) :: r,...
!HPF$ DISTRIBUTE (CYCLIC) :: r,...

       INTERFACE
       EXTRINSIC(HPF_LOCAL)
      &subroutine extr_push(r,...)
       real*8, dimension(:), intent(inout) :: r,...
!HPF$ DISTRIBUTE (CYCLIC) :: r,...
       end subroutine extr_push
       END INTERFACE

       call extr_push(r,...)
```

with the local procedure given by

```
subroutine extr_push(r,...)
real*8, dimension(:), intent(inout) :: r,...
do l=1, UBOUND(r,dim=1)
  r_l=r(l)
  ...
  r(l) = r_l + g_r(r_l,...)
  ...
enddo
end subroutine extr_push
```

Note that each local procedure only updates the coordinates of the particles local to the node (`l=1,UBOUND(r, dim=1)`).

4.2 Pressure updating

Different from the particle pushing, the updating of particle pressure at the grid points presents 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 restructuring the code in the following way:

- the data structure that store the values of the pressure, is replaced, within the bodies of the distributed loops, by a corresponding data structure augmented by one dimension ($p_{par}(n_r, n_\vartheta, n_\varphi, :)$), with extent equal to the number of available nodes;
- this temporary data structure is 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 each iteration of the loop over the particles, the contribution of the corresponding particle to an element of the pressure array is added to the appropriate element of the distributed page;
- at the end of the iterations, the temporary data structure is reduced along the added and distributed dimension, and the result is assigned to the corresponding original data structure; this is implemented by using the HPF intrinsic reduction function `SUM`.

The only need for communication is related to this reduction and the subsequent broadcast, and thus it is embedded in the execution of the intrinsic function. If

the underlying HPF compiler supports the implementation of highly optimized versions of the HPF intrinsic procedures for distributed parameters, these communications are performed as vectorized and collective minimum-cost communications. 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 r(:) :: 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 r(:) :: 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(r,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
```

Analogously to the *particle pushing* case, each local procedure executes only the set of loop iterations that access the particles local to the node and updates only 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 Strategies

Once completed the distributed memory work decomposition, in the framework of a *particle decomposition* approach, the issue of the intra-node decomposition must be addressed. Here, we assume that each node is represented by a shared-memory multi-processor machine, with a single thread running on each processor. It will execute both pushing and pressure loops embedded in local extrinsic procedures, like those described in Sect. 4.

5.1 Particle pushing

It is easy to see that the particle pushing loop is suited for trivial work distribution among different processors. The natural parallelization strategy for shared memory architectures consists in distributing the work needed to update particle coordinates among different threads (and, then, processors). OpenMP allows for a straightforward implementation of this strategy: the `parallel do` directive can be used to distribute the loop iterations over the particles. All the variables that are set and then used within the `do` loop are explicitly defined as `private`, with the other ones being `shared` by default. The loop contained in the extrinsic procedure outlined in Sect. 4.1 then becomes:

```
!$OMP parallel do private(l,r_l,...)
        do l = 1,UBOUND(r,dim=1)
         r_l=r(l)
...
         r(l) = r_l + g_r(r_l,...)
         ...
        enddo
!$OMP end parallel do
```

We have tested this strategy (and the others presented in the following), by running the corresponding HPF+OpenMP version of HMGC 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. A spatial grid with $n_r \times n_\vartheta \times n_\varphi = 32 \times 16 \times 8$ has been considered ($N_{cell} = 4096$). The average number of particles per cell has been varied from $N_{ppc} = 4$ to $N_{ppc} = 256$, which corresponds to N_{part} ranging approximately from 16 k to 1 M.

Figure 2 shows the scaling of the speed-up (s_u) of the *particle pushing* procedure with respect to the number of processors per node, n_{proc}, at two values of the number, n_{node}, of (8-processors) nodes. Figure 3 reports the values of s_u,

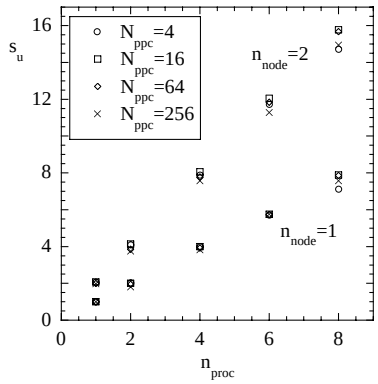

Fig. 2. Speed-up of the *particle pushing* procedure versus the number of processors per node, for two values of the number, n_{node}, of (8-processors) nodes. Here, and in the following Figures $N_{cell} = 4096$. Four different values of the average number of particles per cell, from $N_{ppc} = 4$ to $N_{ppc} = 256$, have been considered (corresponding to N_{part} ranging approximately from 16 k to 1 M).

for the same procedure, versus the number of (2-processors) nodes, at two values of the number of processors per node, n_{proc}. The speed-up has been defined as the ratio between the wall-clock time yielded by the serial execution of the HPF+OpenMP version of the code and the one 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. Note that speed-up values not far from their ideal limit are obtained.

5.2 Pressure updating

The immediate intra-node parallelization of the pressure loop is inhibited, as in the inter-node case, by the updating of the array `p_par`. Such a computation is indeed an example of *irregular array-reduction operation* (cf., e.g., [12]), where the elements to be reduced are the particle coordinates (the elements of the arrays `r`, `theta`, `phi`), 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,

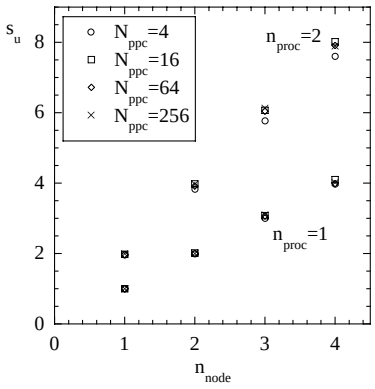

Fig. 3. Speed-up of the *particle pushing* procedure versus the number of (2-processors) nodes, for two values of the number of processors per node, n_{proc}. The other parameters are chosen as in the previous Figure.

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.

In the following, we discuss some possible intra-node parallelization strategies for the pressure updating loop, which present close analogies to the *particle decomposition* strategy or the *domain decomposition* one, discussed at the beginning of Sect. 4 for the distributed memory, inter-node, decomposition.

5.2.1 Particle Decomposition Strategy As stated above, the most natural parallelization strategy for shared memory architectures consists in distributing the particle loop iterations among different processors, without respect to the portion of the domain in which each particles resides. For this reason, such a technique can be referred to as a *particle decomposition* one. It can be implemented very easily in OpenMP, by using the `parallel do` directive, and it is fully satisfactory for the particle-pushing loop. With regard to the pressure loop, however, attention must be payed to protect the *critical sections* of the pressure loop from race conditions, that is to ensure *mutual exclusion* among threads accessing shared data. The most obvious solution to this problem (and the least expensive, in terms of code restructuring effort) consists, in OpenMP, in enclosing the updating of `p_par` by the OpenMP `critical` and `end critical` directives. The relevant portion of the *pressure updating* 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(r,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_par` represents a bottleneck that heavily affects the performances (almost no speed-up) [7]. Such a bottleneck can be eliminated, at the expenses of memory occupation, by means of an alternative strategy, analogous to that envisaged within the framework of the inter-node decomposition, which 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 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. The easiest way to implement such a strategy consists, once again, in introducing an auxiliary array, `p_aux`, defined as a `private` variable with the same dimensions and extent as `p`. Each processor works on a separate copy of the 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 corresponding 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(r,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_lock)
        p_par(:,:,:,1) = p_par(:,:,:,1) + p_aux(:,:,:)
!$OMP end critical (p_lock)
!$OMP end parallel
```

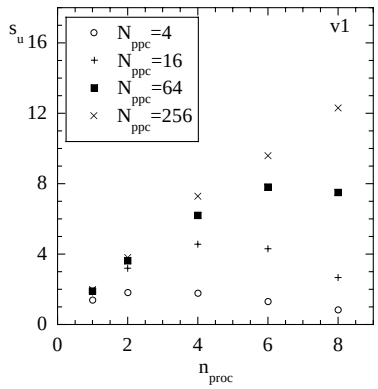

Fig. 4. Speed-up of the *pressure updating* procedure versus the number of processors per node for the *particle decomposition* version (*v1*), at fixed number of (8-processors) nodes, $n_{node} = 2$, and different values of the average number of particles per cell, N_{ppc}.

Note that this strategy (hereafter, version *v1*), based on the introduction of an auxiliary array, makes the execution of the `UBOUND(r,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 $N_{cell} \times n_{proc}$. Such product is much smaller than N_{part}/n_{node}, as long as the $N_{ppc} \gg n_{proc} \times n_{node}$; under this condition, a good parallelization efficiency can be obtained. 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.

Figure 4 shows the scaling of the speed-up (s_u) of the *pressure updating* procedure (version *v1*) with respect to the number of processors per node, n_{proc}, at fixed number of (8-processor) nodes, $n_{node} = 2$. Figure 5 reports the values of s_u for the same procedure versus the number of (2-processor) nodes, at $n_{proc} = 2$. Speed-up values refer only to the execution of the section related to the updating of the (whole) pressure array `p`.

In practice, the "serial" and "parallel" times, t_s and t_{v1} respectively, can be evaluated as

$$t_s \approx t_{loop}|_{\text{serial}} \ ,$$
$$t_{v1} \approx t_{parallel} + t_{sum} \ ,$$

where 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`

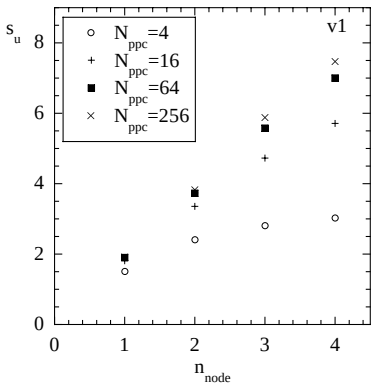

Fig. 5. Speed-up of the *pressure updating* procedure (*particle decomposition* version *v1*) versus the number of 2-processor nodes, at fixed number of processors per node, $n_{proc} = 2$ and different values of N_{ppc}.

(by the HPF intrinsic reduction function `SUM`). We can approximate the above expressions as follows

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

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 intra-node and inter-node array reductions, respectively. Here we have taken into account the logarithmic character of the `SUM` reduction and we have neglect the time required to create and terminate threads and distribute the work among them. From such approximations, we expect a speed-up approximately given by

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

From Fig. 4, 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. 5, 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 "double" *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 criterion

for this approach not to be too memory demanding, $n_{proc}n_{node}/N_{ppc} \lesssim 1$, is *a fortiori* satisfied.

5.2.2 Domain Decomposition Strategy In the previous Paragraph, we have discussed the implementation of what we can indicate as a *particle decomposition* strategy. Indeed, the intra-node work distribution consists in assigning the particle loop iterations to different processors, without respect to the portion of the domain in which each particles resides. We have seen that such a strategy is characterized by a perfect load balancing among the different processors and a very limited code restructuring effort. On the opposite side, the need of avoiding race conditions introduces a trade-off between parallelization efficiency and memory requirements.

In order to overcome such a trade-off, at the price of a heavier restructuring of the code and, possibly, the need of addressing load-balancing problems, a completely different strategy can be adopted for the work distribution among the different processors of each computational node: namely, the *domain decomposition* strategy[1]. This strategy 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, 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 processor.

A possible implementation of this strategy (version *v2a*, whose schematic representation is reported in the Appendix) 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.

[1] Note that the inter-node decomposition remains a particle decomposition.

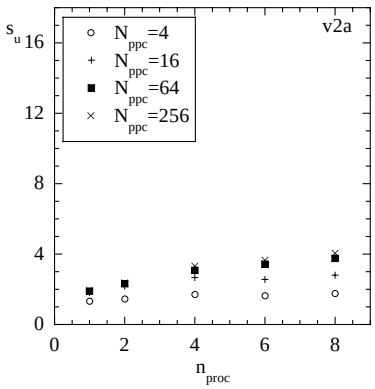

Fig. 6. Speed-up of the *pressure updating* procedure versus the number of processors per node, at fixed number of (8-processors) nodes, $n_{node} = 2$, for the *domain decomposition* version, *v2a*.

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 per node.

Figure 6 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 *domain decomposition* version, *v2a*, of the *pressure updating* procedure. Figure 7 reports the values of s_u for the same procedure versus the number of (2-processor) nodes, at $n_{proc} = 2$. The wall-clock time can be evaluated, in this case, as follows:

$$t_{v2a} \approx t_{pre-loop} + t_{assign} + t_{reorder} + t_{loop} + t_{sum} \ .$$

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 in the pressure updating loop (both over particles), respectively.

We note that, at least for the specific application here considered, this *domain decomposition* strategy appears to be an interesting compromise between the two extremes obtained in the framework of the *particle decomposition* approach (namely, the low-efficiency and the large-memory versions). We also observe that the bottleneck, with regard to the efficiency performances, is still represented by the critical section, although this bottleneck is not so penalizing as in the low-efficiency *critical section* version, discussed at the beginning of Subsect. 5.2.1.

A significant improvement of the efficiency can be obtained, for specific (but rather common) applications characterized by a contained particle migration per time step from one portion of the domain to another one, by limiting the

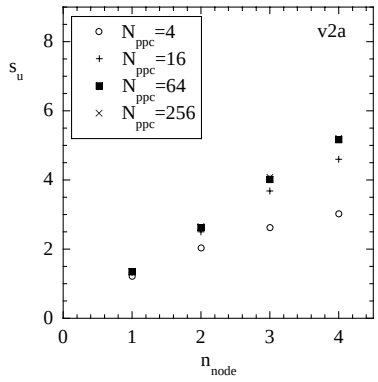

Fig. 7. Speed-up of the *pressure updating* procedure (*domain decomposition* version *v2a*) versus the number of 2-processor nodes, at fixed number of processors per node, $n_{proc} = 2$ and different values of N_{ppc}.

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 slow-varying coordinate. This is moderately true for the specific application we have tested, as it can be seen from Figs. 8 and 9 (corresponding to Figs. 6 and 7, respectively), which show the results from a modified *domain decomposition* version, *v2b*, implementing such a selective reordering. A comparison between the different versions examined, in this Section, for the *pressure updating* procedure is shown in Fig. 10 (speed-up versus n_{proc} for the case $N_{ppc} = 256$).

6 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, *particle 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.

Some different implementations of them, based on the use and integration of the high level languages HPF and OpenMP, have been discussed with respect to program restructuring effort, time efficiency and memory occupancy, with particular regard to the not-trivial *pressure updating* procedure. We observe a trade-off between the merits of each method with respect to such requirements. More precisely, the *particle decomposition* approach yields, at the expense of a little programming effort, high speed-up values, while requiring a supplementary memory resource level that scales with the number of processors and the size of

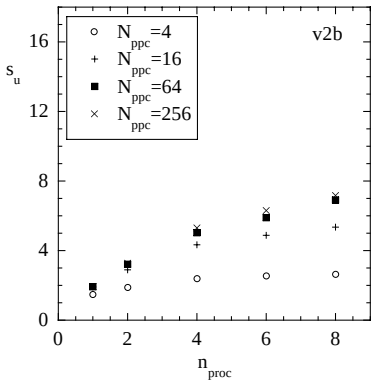

Fig. 8. Speed-up versus the number of processors per node, at fixed number of (8-processors) nodes, $n_{node} = 2$, for the *selective reordering* version, *v2b*, of the *pressure updating* procedure.

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 choice of a high-level language parallelization 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.

As a final remark, we would like to outline that, once restructured the code in order to implement a *domain decomposition* strategy, one could aim to apply the same strategy to the inter-node parallelization. The introduction of auxiliary arrays like `p_par` could then be avoided, and a better scaling of the maximum size of the domain with the number of nodes could be obtained. However, several facts must be kept into account: first, a time-varying assignment of particles to nodes makes HPF not very suited to the implementation of work distribution. Second, even resorting to a lower-level message passing paradigm for the inter-node parallelization, the migration of particles from one node to another requires communication, different from the case of intra-node parallelization. The amount of communication can be maintained at a reasonably low level, if only a low percentage of the particle population treated by a node migrates at each time step. As the same condition must hold within the intra-node decomposition context (in order to get the improvement associated to the selective reordering of version *v2b*), it can be easily seen that either the application is characterized by two (not just one) slowly-varying coordinates (which is not the case, for

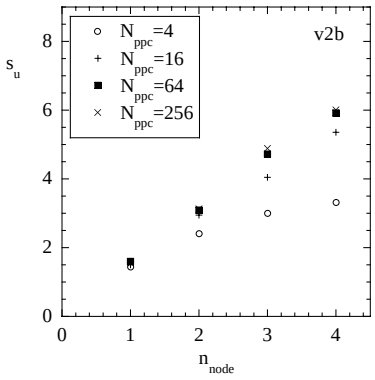

Fig. 9. Speed-up versus the number of 2-processor nodes, at fixed number of processors per node, $n_{proc} = 2$ and different values of N_{ppc}, for the *selective reordering* version, *v2b*, of the *pressure updating* procedure.

example for tokamak plasma simulations) or a "double" *domain decomposition*, in order to be efficient, has to satisfy the condition, e.g., $n_{node} \times n_{proc} \ll n_r$ – that is, the whole number of processors much lesser than the number of cells along the slowly-varying coordinate. Then, although a *domain decomposition* approach would allow for a fair scaling, with the whole number of processors, of the maximum system size that can be simulated, the parallelization efficiency of this approach would heavily decrease above a certain number of processors.

Appendix: Schematic Representation of the Version *v2a*

The intra-node *domain decomposition* strategy can be implemented, e.g., according to the scheme proposed in Subsect. 5.2.2. The domain is decomposed along one of its dimensions. The elementary portion each particle belongs to (`j_r_part`) is identified by a loop over particles. The number of particles belonging to each elementary portion (`n_part_r`) is updated inside a critical section of the loop (in order to avoid race conditions). The relative order of each particle within the population residing in the same portion (`i_r`) is defined inside the critical section too. A global portion, composed by several consecutive elementary portions, is assigned to each processor. The size of each global portion, delimited by `j_r_upper`, is chosen in such a way to ensure an approximate load balancing: each processor will manage a number of particles approximately equal to $(N_{part}/n_{node})/n_{proc}$. Particles are then reordered according to the processor they belong to. Finally, the pressure loop, rewritten as a loop over processors in which a loop over the particle belonging to the processor is nested, can be distributed among processors without concerns for the occurrence of race conditions.

This scheme can be represented as follows:

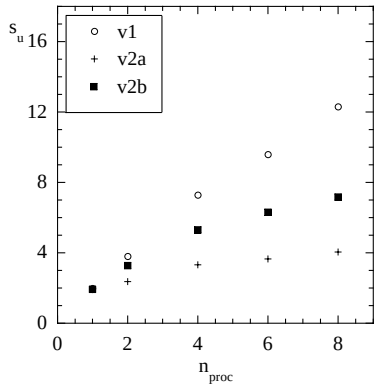

Fig. 10. Comparison between the speed-up obtained, at different number of processors per node and fixed number of 8-processors nodes, $n_{node} = 2$, by the *domain decomposition* version, *v2a*, and the companion *selective reordering* version, *v2b*, of the *pressure updating* procedure. The results of the *particle decomposition* implementation, *v1*, are also shown for reference. The case $N_{ppc} = 256$ is considered.

```
        integer, allocatable :: j_r_upper(:)
        integer j_r_part(n_part),i_r(n_part),l_index(n_part)
        integer n_part_r(n_r),n_part_lower(n_r)
        n_procs=omp_get_max_threads()
        allocate(j_r_upper(0:n_procs))
        p_par = 0.
        n_part_r = 0
c
c   assignment of each particle to elementary portions
c   of the domain
c
!$OMP parallel do private(l,j_r)
        do l = 1,UBOUND(r,dim=1)
         j_r = f_r(r(l))
         j_r_part(l)=j_r
!$OMP critical (n_part_r_lock)
         n_part_r(j_r)=n_part_r(j_r)+1
         i_r(l)=n_part_r(j_r)
!$OMP end critical (n_part_r_lock)
        enddo
!$OMP end parallel do
c
c   definition of the global portions of the domain
c
        n_part_average=float(UBOUND(r,dim=1))/float(n_procs)
        n_part_portion=0
        do j_r=1,n_r
```

```
         n_part_lower(j_r)=n_part_portion
         n_part_portion=n_part_portion+n_part_r(j_r)
         i_proc=n_part_portion/n_part_average+1
         if(i_proc.gt.n_procs)i_proc=n_procs
         j_r_upper(i_proc)=j_r
        enddo
        j_r_upper(0)=0
c
c   reordering of the particles
c
!$OMP parallel do private(l,j_r,l0)
        do l = 1,UBOUND(r,dim=1)
         j_r=j_r_part(l)
         l0=n_part_lower(j_r)+i_r(l)
         l_index(l0)=l
        enddo
!$OMP end parallel do
c
c   pressure loop
c
!$OMP parallel do private(i_proc,j_r,i,l0,l,j_theta,j_phi)
        do i_proc=1,n_procs
         do j_r=j_r_upper(i_proc-1)+1,j_r_upper(i_proc)
          do i=1,n_part_r(j_r)
           l0=n_part_lower(j_r)+i
           l=l_index(l0)
           j_theta = f_theta(theta(l))
           j_phi = f_phi(phi(l))
           p_par(j_r,j_theta,j_phi,1) =
       &                          p_par(j_r,j_theta,j_phi,1)
       &                          + h(r(l),...,w(l))
         enddo
!$OMP end parallel do
```

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). (http://www.supercomp.org/sc96/proceedings/SC96PROC/AKARSU/INDEX.HTM)
2. Benkner, S., Sanjari, K., Sipkova, V., Velkov, B.: Parallelizing Irregular Applications with Vienna HPF+ Compiler VFC. In *High Performance Computing and Networking. Proceedings*, LNCS Vol. 1401, Springer, Berlin, 1998, p. 816–827.
3. Birdsall, C.K., Langdon, A.B.: Plasma Physics via Computer Simulation (McGraw-Hill, New York, 1985).
4. Briguglio, S., Vlad, G., Zonca, F., Kar, C.: Hybrid Magnetohydrodynamic-Gyrokinetic Simulation of Toroidal Alfvén Modes. Phys. Plasmas **2** (1995) 3711–3723.
5. Chen, L.: Theory of Magnetohydrodynamic Instabilities Excited by Energetic Particles in Tokamaks. Phys. Plasmas **1** (1994) 1519–1522.

6. V.K. Decyk, Skeleton PIC codes for parallel computers, Computer Physics Communications 87 (1995) 87-94.
7. 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.
8. 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.
9. 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.
10. 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).
11. High Performance Fortran Forum: High Performance Fortran Language Specification, Version 2.0, Rice University, 1997.
12. 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).
13. Liewer, P.C., Decyk, V.K.: A General Concurrent Algorithm for Plasma Particle-in-Cell Codes. J. Computational Phys. **85** (1989) 302–322.
14. 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.
15. OpenMP Architecture Review Board: OpenMP Fortran Application Program Interface, ver. 1.0, October 1997.
16. Saltz, J., Ponnusamy, R., Sharma, S., Moon, B., Hwang, Y.-S., Uysal, M., Das, R.: A Manual for the CHAOS Runtime Library, Technical Report UMIACS TR CS-TR-3437, Univ. of Maryland, March 1995.