Scientific Programming 11 (2003) 159–176
IOS Press


# A performance-prediction model for PIC applications on clusters of Symmetric MultiProcessors: Validation with hierarchical HPF+OpenMP implementation

Sergio Briguglio[a], Beniamino Di Martino[b] and Gregorio Vlad[a]
[a]*Associazione EURATOM-ENEA sulla Fusione, C.R. Frascati, C.P. 65, 00044, Frascati, Rome, Italy*
*E-mail: {briguglio,vlad}@frascati.enea.it*
[b]*Dip. Ingegneria dell'Informazione, Second University of Naples, Italy*
*E-mail: beniamino.dimartino@unina.it*

Accepted July 16, 2002

**Abstract**. A performance-prediction model is presented, which describes different hierarchical workload decomposition strategies for *particle in cell* (PIC) codes on Clusters of Symmetric MultiProcessors. The devised workload decomposition is hierarchically structured: a higher-level decomposition among the computational nodes, and a lower-level one among the processors of each computational node. Several decomposition strategies are evaluated by means of the prediction model, with respect to the memory occupancy, the parallelization efficiency and the required programming effort. Such strategies have been implemented by integrating the high-level languages High Performance Fortran (at the inter-node stage) and OpenMP (at the intra-node one). The details of these implementations are presented, and the experimental values of parallelization efficiency are compared with the predicted results.

## 1. Introduction

Particle-in-cell (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. Two main strategies have been developed for the workload decomposition related to porting PIC codes on distributed memory parallel systems: the *domain decomposition* strategy and the *particle decomposition* one. Standard *domain decomposition* [1,6,7,10, 11] techniques assign different portions of the physical domain and the corresponding portions of the grid to different computational nodes, along 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 is, the maximum spatial resolution) that can be simulated with the number of nodes. This makes the *domain decomposition* approach very attractive, in principle. Two important problems with these techniques are however represented by the communication overhead and the need for dynamic load balancing, both associated to particle migration from one portion of the domain to another one. While the former problem could possibly affect the parallelization efficiency, depending on the effective amount of particle migration per time step, the latter one can be by-passed, at the expense of a deep restructuring of the original serial code and the adoption of a message-passing approach. It is generally accepted, however, that such an approach, based on manual partition of data, insertion of communication library calls, handling of boundary cases, is very complicated, time consuming and error prone, and af-

ISSN 1058-9244/03/$8.00 © 2003 – IOS Press. All rights reserved

fects the portability of the resulting program. In order to avoid these features, it is worth to resort, for distributed architectures, to the *particle decomposition* [5] technique, which is suited to be implemented, with relatively little effort, by the use of high-level programming languages, such as the High Performance Fortran (HPF) [8]. Particle decomposition consists in statically distributing the particle population among the computational nodes, while replicating the data relative to grid quantities. As no particle has to be transferred (reassigned) from one computational node to another, the communication and load-balancing problems associated to particle migration are automatically overcome. The implementation of such a strategy with high-level languages is then, in principle, relatively straightforward. On the opposite side, an overhead on memory occupancy, given by the replication of data related to the domain, and a computation overhead related to the updating of the fields (each node manages only the partial updating associated to its portion of particle population) forbid a good scalability of the maximum domain size with the number of nodes, and limit the efficiency of such a technique to cases in which both memory and computational loads on each node are dominated by the particle-related ones.

When porting a PIC code on a hierarchical distribut ed-shared memory system such as a cluster of SMPs, a two-stage workload decomposition can be envisaged: a distributed-memory level decomposition (among the computational nodes), and a shared-memory one (among the processors of each node). The latter decomposition qualitatively differs from that at the distributed-memory level. Indeed, the alternative between particle and domain decomposition no longer corresponds to the alternative between high-level and low-level languages: even in the framework of a *domain decomposition* approach, particle migration from one processor to another does not require communication, and a high-level parallel programming language such as OpenMP [12] can still be used. Both the *domain decomposition* strategy and the *particle decomposition* one can then be implemented within the framework of a high-level language programming and integrated with the *particle decomposition* strategy devised at the distributed-memory level, looking for an optimal balance of merits and defects.

In this paper we present a performance-prediction model describing the above mentioned different hierarchical workload decomposition strategies, in terms of efficiency and memory occupancy. The prediction-model results are compared with the experimental results from a high-level language based porting (obtained with integration of HPF and OpenMP) of the Hybrid MHD-Gyrokinetic Code (HMGC) [3], which includes all the relevant properties of general PIC codes.

The paper is structured as follows. Section 2 describes the main computational aspects of the chosen application. It introduces the performance-prediction model and its application to the different decomposition strategies devised, both on distributed memory architectures and on distributed-shared memory ones, analytically modeling and predicting their main features in terms of the expected parallelization efficiency and memory requests. The implementation of such strategies, based on integrating the HPF and OpenMP programming environments by means of the EXTRINSIC feature of the HPF language, is presented in Section 3. Section 4 reports the experimental results obtained by running the corresponding parallel versions of HMGC on a IBM SP. Finally, the main results are summarized in Section 5.

## 2. Prediction model of hierarchical workload decompositions for PIC applications

Particle simulation [2] consists in evolving the phase-space coordinates of a set of $N_{\rm part}$ simulation particles (each of them in fact representing, by its weight $w$, a cluster of physical particles) in the electromagnetic fields computed, at each time step, consistently with the particle distribution function (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 the fields 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 condition for an accurate description of the plasma behaviour can be written as $N_{\rm ppc} \equiv N_{\rm part}/N_{\rm cell} \gg 1$, where $N_{\rm cell}$ is the number of grid cells and $N_{\rm ppc}$ 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 $N_{\rm ppc}$, large number of particles. Such a goal requires resorting to parallelization techniques aimed to distribute the computational loads related to the particle population among several processors.

Let us estimate such loads, in terms of wall-clock time, $t$, and memory needed to store particle and field arrays, $M$. In many concrete situations, as a matter of fact, periodic spatial domains are considered, and the problem admits solution in terms of normal modes. This feature allows to solve the equations for the fields in Fourier space. In this paper, for the sake of simplicity, we assume that this is the case, although in the general case the resort to finite difference methods could be needed at least for some of the spatial coordinates. We also assume that Fast Fourier Transform (FFT) is used to transform electromagnetic fields from the Fourier space to the real one, as well as the pressure field from the real space to the Fourier one.

We can then approximate time and memory (the pedix $S$ means "serial") as:

$$t_S \approx t_{\rm FFT} N_{\rm cell} \log_2 N_{\rm cell} + t_{\rm int} N_{\rm part}, \quad (1)$$

$$M_S \approx (m_{\rm field} + m_{\rm press}) N_{\rm cell} + m_{\rm part} N_{\rm part}. \quad (2)$$

Here $t_{\rm FFT}$ is a coefficient (with the dimensions of a time) related to the details of the FFT algorithm, while $t_{\rm int}$ corresponds to the wall-clock time consumed, per particle, by the interpolation/assignment operations from the grid to the particle position and viceversa. The quantities $m_{\rm field}$, $m_{\rm press}$ and $m_{\rm part}$ are the amounts of memory needed to store, respectively, the real-space fields and pressure at each grid point and the phase-space coordinates and weights for each particle. Note that, in the view of the qualitative arguments we want to present in this Section, we have neglected, in Eq. (1), the amount of calculations needed to solve the field equations in the Fourier space in comparison with that needed to transform the fields back and forth from Fourier to real space. In the same way, in Eq. (2), we have included the amount of memory needed to store the Fourier harmonics of the fields into that needed to store the real-space fields.

### 2.1. *Decomposition strategies on distributed memory architectures*

Two main techniques have been adopted in parallelizing PIC codes on *distributed memory* architectures. The first approach is based on the so-called *domain decomposition* [10]: different portions of the physical domain and of the corresponding grid are assigned to $n_{\rm node}$ different computational nodes, along with the particles that reside on them. In this way the memory resources required to each node are reduced by a factor equal (in an average sense) to $n_{\rm node}$. An almost linear scaling of the attainable physical-space resolution (more precisely, the maximum number of spatial cells) with the number of nodes is then obtained. This can be seen from the following expression for the memory-per-node requirement:

$$M_D \approx \frac{1}{n_{\rm node}} \left[(m_{\rm field} + m_{\rm press}) N_{\rm cell} + m_{\rm part} N_{\rm part}\right]. \quad (3)$$

From such expression we can obtain the maximum value of $N_{\rm cell}$ ($N^D_{{\rm cell}_{\max}}$) by imposing the condition $M_D \leq M_0$, where $M_0$ is the Random Access Memory (RAM) equipment of each node. This yields

$$N^D_{{\rm cell}_{\max}} = \frac{M_0 n_{\rm node}}{m_{\rm field} + m_{\rm press} + m_{\rm part} N_{\rm ppc}} \approx \frac{M_0 n_{\rm node}}{m_{\rm part} N_{\rm ppc}}, \quad (4)$$

where we have neglected the quantity $m_{\rm field} + m_{\rm press}$ in comparison with $m_{\rm part} N_{\rm ppc}$, because of the general PIC condition $N_{\rm ppc} \gg 1$.

On the opposite side, such a decomposition technique presents two relevant problems:

- inter-node communication is required to update the fields at the boundary between two different portions of the domain, as well as to transfer those particles that migrate from one domain portion to another;
- load imbalance can occur because of particle migration.

Both problems can cause efficiency degradation for the parallel implementation. The effect of the communication overhead can be quantified by the following approximate evaluation of the *domain decomposition* wall-clock time:

$$t_D \approx \frac{1}{n_{\rm node}} \left[ t_{\rm FFT} N_{\rm cell} \log_2 N_{\rm cell} + t_{\rm int} N_{\rm part} + t_{\rm com} \left(\frac{n_{\rm node}}{N_{\rm cell}}\right)^{\frac{1}{3}} N_{\rm part} \right], \quad (5)$$

with $t_{\rm com}$ being the communication time per real variable. In the above expression, we have estimated

the fraction of particles subjected to migration from one portion of the domain to another as the ratio between the number of boundary grid cells, proportional to $(N_{\rm cell}/n_{\rm node})^{\frac{2}{3}}$, and the whole number of cells assigned to each node, proportional to $N_{\rm cell}/n_{\rm node}$; in other words, we have assumed that the time-step distance covered by each particle is of the same order of the cell size, as it is usually required by accuracy and/or stability criteria. Such a fraction is then approximately given by $(N_{\rm cell}/n_{\rm node})^{-\frac{1}{3}}$.

In terms of speed-up values, $s_u$, defined as the ratio between the serial wall-clock time – Eq. (1) – and the parallel one – Eq. (5), in this case –, we have

$$s_u^D \approx \frac{n_{\rm node}\left(1+\frac{t_{\rm FFT}}{t_{\rm int}}\frac{\log_2 N_{\rm cell}}{N_{\rm ppc}}\right)}{1+\frac{t_{\rm FFT}}{t_{\rm int}}\frac{\log_2 N_{\rm cell}}{N_{\rm ppc}}+\frac{t_{\rm com}}{t_{\rm int}}\left(\frac{n_{\rm node}}{N_{\rm cell}}\right)^{\frac{1}{3}}}. \quad (6)$$

From this expression it can be seen that, in the limit $N_{\rm ppc} \gg 1$, the goal of efficient parallelization impose a limit on the number of nodes that can be used; the condition $\eta \equiv s_u/n_{\rm node} > \eta_*$ can indeed be written as

$$n_{\rm node} < \left(\frac{1}{\eta_*}-1\right)^3\left(\frac{t_{\rm int}}{t_{\rm com}}\right)^3 \left[1+\frac{t_{\rm FFT}}{t_{\rm int}}\frac{\log_2 N_{\rm cell}}{N_{\rm ppc}}\right]^3 N_{\rm cell}. \quad (7)$$

The load-balancing problems related to particle migration can make such estimations meaningless, unless a dynamic load balancing is ensured at the expenses of further computation and inter-node communication and of a greater effort in the implementation of the parallel version of the code.

Finally, both reassignment of migrating particles to nodes and dynamic load balancing preclude the usage of a high-level programming language such as HPF. A *message passing* library, such as MPI, has to be used instead, involving manual partition of data, insertion of communication library calls, handling of boundary cases. This is generally very complicated, time consuming and error prone, and affects the portability of the resulting program.

The aim of avoiding reassignment and load balancing problems and – more stringently – that of adopting high-level parallel programming languages motivated the development of an alternative approach, based on *particle decomposition* [5]. It corresponds to replicating the whole spatial domain on each node, while distributing the particle population. No particle migrates from one node to another, because no particle meets, in its motion, the unphysical boundaries introduced by domain decomposition. Moreover, load balancing is perfectly ensured during any simulation.

On the opposite side, the memory request associated to the grid fields, which are replicated on each node, gives rise to a bottle-neck on the scalability of physical resolution with nodes: even in the limit $n_{\rm node} \to \infty$, in which each node must treat a vanishingly small number of particles, the maximum resolution that can be reached is determined by the largest size of the (replicated) grid arrays that can be stored on each node.

Similarly, a bottle-neck in the parallelization efficiency is given by the updating of the electromagnetic fields on the grid, as the related computation (FFT included) is no longer distributed among the nodes. Further departures from ideal memory distribution ($\propto 1/n_{\rm node}$) and speed-up ($\sim n_{\rm node}$) are related to the fact that, in order to avoid competitions between nodes updating the same element of the (replicated) pressure array and, more generally, a very frequent inter-node communication of the updated elements, an auxiliary copy of this array must be distributed to each node: because of the associative and distributive properties of the updating laws for the pressure array with respect to the contributions of different particles, the copy can be updated with no regard to what the other nodes are doing; however, after the pressure computation, it will retain only the contribution of the corresponding portion of the particle population. Different-node copies have then to be summed together into the whole-pressure array before updating the electromagnetic fields, and this fact introduces overheads in inter-node communication and memory requirements [5].

Such features of the *particle decomposition* on distributed memory architectures are summarized by the following approximate expressions of memory-per-node requirements, wall-clock time and speed-up:

$$M_P \approx (m_{\rm field}+2m_{\rm press})N_{\rm cell}+m_{\rm part}\frac{N_{\rm part}}{n_{\rm node}}, \quad (8)$$

$$t_P \approx t_{\rm FFT}N_{\rm cell}\log_2 N_{\rm cell}+t_{\rm int}\frac{N_{\rm part}}{n_{\rm node}}+t_{\rm com} N_{\rm cell}\log_2 n_{\rm node}, \quad (9)$$

$$s_u^P \approx \left[n_{\rm node}\left(1+\frac{t_{\rm FFT}}{t_{\rm int}}\frac{\log_2 N_{\rm cell}}{N_{\rm ppc}}\right)\right] \Big/ \left[1+\frac{n_{\rm node}}{N_{\rm ppc}}\left(\frac{t_{\rm FFT}}{t_{\rm int}}\log_2 N_{\rm cell} +\frac{t_{\rm com}}{t_{\rm int}}\log_2 n_{\rm node}\right)\right]. \quad (10)$$

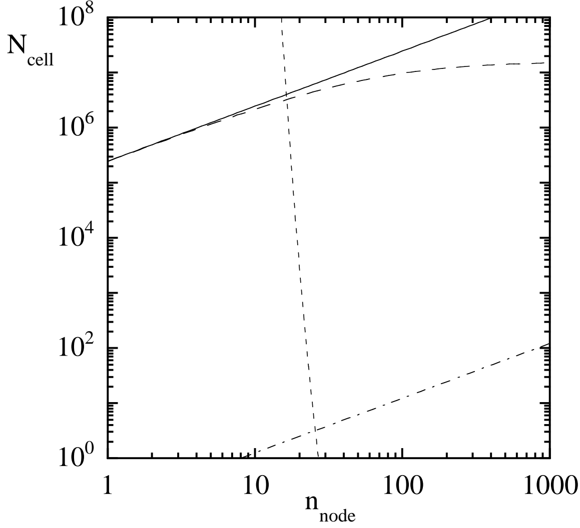


Fig. 1. Regions of allowed resolution and high efficiency ($\eta > 50\%$) in the $(n_{\rm node}, N_{\rm cell})$ space. For the *domain decomposition* strategy, the allowed resolution, corresponding to Eq. (4), is represented by the area below the solid line; for the *particle decomposition* case, it corresponds to Eq. (11) and the area below the dashed line. The high-efficiency regions for the two strategies are those on the left of the dashed-dotted line and, respectively, the dotted line. Both lines correspond to an efficiency value $\eta = 50\%$. Here we have fixed $N_{\rm ppc} = 64$, $M_0 = 1024$ Megabytes, $m_{\rm part} = 64$ bytes, $m_{\rm field} = 48$ bytes, $m_{\rm press} = 8$ bytes, $t_{\rm com}/t_{\rm int} = 0.5$, $t_{\rm FFT}/t_{\rm int} = 0.1$.

In writing Eq. (8), we have taken into account the further amount of memory related to the inclusion of the auxiliary copy of the pressure array. Moreover, we have assumed that the reduction of the different-node contributions to the pressure field is performed by vectorized and collective minimum-cost communications, yielding a logarithmic dependence of the communication term on $n_{\rm node}$.

From Eqs (8) and (10), we see that the maximum resolution ($N^P_{{\rm cell}_{\rm max}}$) and the condition for efficient parallelization ($s^P_u/n_{\rm node} > \eta_*$) are given, respectively, by

$$N^P_{{\rm cell}_{\rm max}} = \frac{M_0 n_{\rm node}}{(m_{\rm field} + 2m_{\rm press})n_{\rm node} + m_{\rm part}N_{\rm ppc}} \qquad (11)$$

and

$$\log_2 N_{\rm cell} < \frac{t_{\rm int}N_{\rm ppc}\left(\frac{1}{\eta_*} - 1\right) - t_{\rm com}n_{\rm node}\log_2 n_{\rm node}}{t_{\rm FFT}\left(n_{\rm node} - \frac{1}{\eta_*}\right)}, \qquad (12)$$

or

$$n_{\rm node} < \frac{N_{\rm ppc}\left[\left(\frac{1}{\eta_*} - 1\right) + \frac{1}{\eta_*}\frac{t_{\rm FFT}}{t_{\rm int}}\frac{\log_2 N_{\rm cell}}{N_{\rm ppc}}\right]}{\left(\frac{t_{\rm FFT}}{t_{\rm rint}}\log_2 N_{\rm cell} + \frac{t_{\rm com}}{t_{\rm int}}\log_2 n_{\rm node}\right)}. \qquad (13)$$

The latter condition, given in an implicit form, means that the number of particles per cell per node is so high that the particle computation dominates over the grid one [5]. If such a condition is satisfied, the former one, for the maximum resolution, essentially reduces to that obtained for the *domain decomposition* strategy, Eq. (4). Figure 1 shows the regions in the space $(n_{\rm node}, N_{\rm cell})$ characterized by allowed resolution and high efficiency ($\eta > \eta_*$, with $\eta_* = 50\%$). Both the *domain decomposition* case and the *particle decomposition* one are considered, for given values of the single-node memory resources, $M_0$, and of the other, simulation-related, parameters that appear in Eqs (4), (6), (11) and (12). In particular, as an example, we have fixed $N_{\rm ppc} = 64$, $M_0 = 1024$ Megabytes, $m_{\rm part} = 64$ bytes, $m_{\rm field} = 48$ bytes, $m_{\rm press} = 8$ bytes. Moreover, on the basis of empirical estimates related to the specific platform considered in Section 4, we assume $t_{\rm com}/t_{\rm int} = 0.5$, $t_{\rm FFT}/t_{\rm int} = 0.1$ (note that the same reference to such platform will be done, in the following, when estimating the other time ratios). We see that, for a limited number of nodes (much lower than the number of particles per cell), there is no reason to adopt the much more complicate (and less efficient) *domain decomposition* technique. This is no longer true for higher number of nodes: particle decomposition comes out to be both inefficient and limited in resolution ($N_{\rm cell}$) by memory constraints, while domain decomposition allows efficient high-resolution simulations. Note, however, that within the *domain decomposition* framework, due to the strong dependence of the efficiency on $t_{\rm com}/t_{\rm int}$ (see Eq. (7)) a slight increase of such a parameter can imply a significant reduction of the high-efficiency region or even make the high-efficiency condition incompatible with the memory constraint.

### 2.2. *Decomposition strategies on distributed-shared memory architectures*

The increasing relevance that hierarchical distributed-shared memory architectures are assuming in High Performance Computing requires a deeper consideration of the respective merits of the two decomposition techniques examined in the previous Section, also in the view of adopting mixed decomposition schemes.

Let us assume that the target architecture is composed by $n_{\rm node}$ computational nodes, each of them being a shared memory multiprocessor system, with $n_{\rm proc}$ processors. We still indicate the node memory resource as $M_0$.

The workload decomposition we consider here 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 already observed that, at the inter-node, distributed memory, level, only the *particle decomposition* strategy can be implemented within the framework of a high-level language such as HPF, while the *domain decomposition* strategy compels to resort to explicit message-passing libraries, such as MPI. At the intra-node, shared memory, level, both techniques can instead be developed in a high-level parallel programming environment, like OpenMP.

### 2.2.1. Intra-node particle decomposition

The most natural intra-node parallelization strategy for shared memory architectures consists in distributing the particle loop iterations (both for the particle-pushing loop and for the pressure-updating one) among 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. It is fully satisfactory for the particle-pushing loop; with regard to the pressure loop, however, caution must be payed to protect the pressure updating from race conditions, that is to ensure *mutual exclusion* among threads accessing shared data. Such conditions can be avoided at the expenses of memory occupation: 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 technique consists, similarly to the inter-node *particle decomposition* case, in introducing an auxiliary array with the same dimensions and extent as the pressure-array copy assigned to the node and making each processor working on a separate copy of the auxiliary array. There is no conflict, in this way, between processors updating the same element of the array. At the end of the loop, however, each copy 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 node array in order to obtain the whole-node contribution.

We can conjugate this intra-node technique with the *particle decomposition* inter-node strategy discussed in the previous Section. We indicate the resulting workload decomposition, which distributes particles both to nodes and processors without any reference to the portion of domain they reside in, as the *particle-particle* strategy. The memory-per-node requirement and the wall-clock time can then be approximated by the following expressions:

$$M_{PP} \approx (m_{\rm field} + 2m_{\rm press})N_{\rm cell} + m_{\rm press}N_{\rm cell}n_{\rm proc} + m_{\rm part}\frac{N_{\rm part}}{n_{\rm node}}, \quad (14)$$

$$t_{PP} \approx t_{\rm FFT}N_{\rm cell}\log_2 N_{\rm cell} + t_{\rm int}\frac{N_{\rm part}}{n_{\rm proc}n_{\rm node}} + t_{\rm red}n_{\rm proc}N_{\rm cell} + t_{\rm com}N_{\rm cell}\log_2 n_{\rm node}, \quad (15)$$

$$s_u^{PP} \approx \left[n_{\rm node}n_{\rm proc}\left(1+\frac{t_{\rm FFT}}{t_{\rm int}}\frac{\log_2 N_{\rm cell}}{N_{\rm ppc}}\right)\right] \Big/ \left[1+\frac{n_{\rm node}n_{\rm proc}}{N_{\rm ppc}}\left(\frac{t_{FFT}}{t_{\rm int}}\log_2 N_{\rm cell} + \frac{t_{\rm com}}{t_{\rm int}}\log_2 n_{\rm node}\right) + \frac{t_{\rm red}}{t_{\rm int}}\frac{n_{\rm node}n_{\rm proc}^2}{N_{\rm ppc}})\right]. \quad (16)$$

We then find, for the maximum value of $N_{\rm cell}$ allowed by the memory constraint, Eq. (14), and the condition for efficient parallelization ($\eta > \eta_*$, with $\eta \equiv s_u/n_{\rm node}n_{\rm proc}$) are given, respectively, by

$$N_{{\rm cell}_{\max}}^{PP} = \frac{M_0 n_{\rm node}}{[m_{\rm field} + m_{\rm press}(2+n_{\rm proc})]n_{\rm node} + m_{\rm part}N_{\rm ppc}} \quad (17)$$

and

$$\log_2 N_{\rm cell} < \left[t_{\rm int}N_{\rm ppc}\left(\frac{1}{\eta_*}-1\right) - n_{\rm proc}n_{\rm node}(t_{\rm red}n_{\rm proc} + t_{\rm com}\log_2 n_{\rm node})\right] \Big/ \left[t_{\rm FFT}\left(n_{\rm proc}n_{\rm node} - \frac{1}{\eta_*}\right)\right], \quad (18)$$

or, in implicit form,

$$n_{\rm node} < \left[N_{\rm ppc}\left(\frac{1}{\eta_*}-1\right) + \frac{1}{\eta_*}\frac{t_{\rm FFT}}{t_{\rm int}}\frac{\log_2 N_{\rm cell}}{N_{\rm ppc}}\right] \Big/ \quad (19)$$

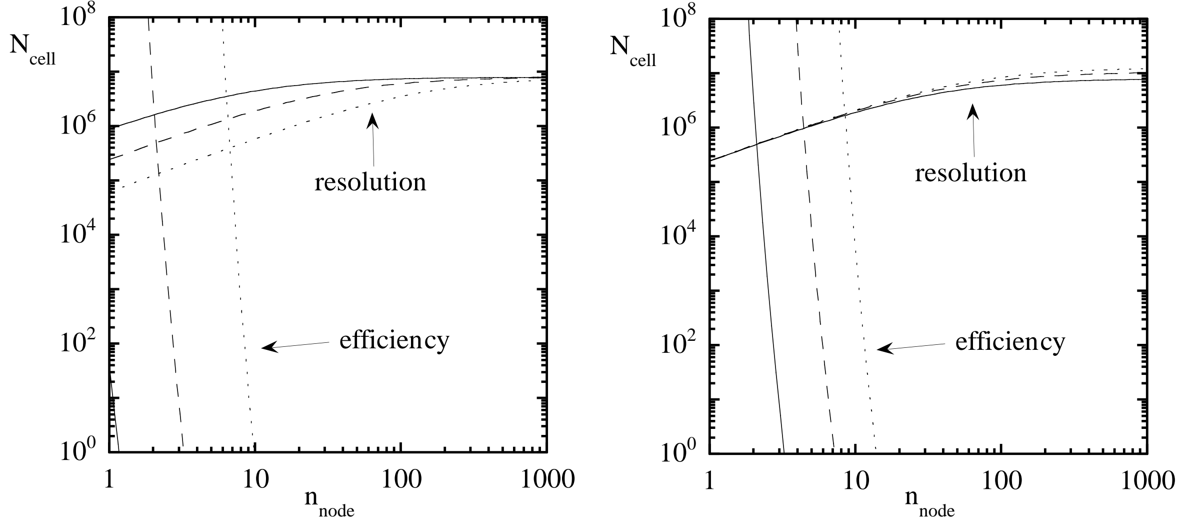


Fig. 2. Regions of allowed resolution and high efficiency ($\eta > 50\%$) for the *particle-particle* decomposition. The allowed resolution is represented by the area below the resolution curves, corresponding to Eq. (17). The high-efficiency region is that on the left of the efficiency curves, corresponding to Eq. (18). Here we have fixed $t_{\rm red}/t_{\rm int} = 0.2$, and the other parameters as in Fig. 1. The left frame considers the case with $n_{\rm proc} = 8$ and three different values of the number of particles per cell: $N_{\rm ppc} = 16$ (solid lines), $N_{\rm ppc} = 64$ (dashed lines) and $N_{\rm ppc} = 256$ (dotted lines). The right frame refers to the case with $N_{\rm ppc} = 64$ and three different values of the number of processors: $n_{\rm proc} = 2$ (dotted lines), $n_{\rm proc} = 4$ (dashed lines) and $n_{\rm proc} = 8$ (solid lines).

$$\left[n_{\rm proc}\left(\frac{t_{\rm FFT}}{t_{\rm int}}\log_2 N_{\rm cell} + \frac{t_{\rm com}}{t_{\rm int}}\log_2 n_{\rm node}\right) + \frac{t_{\rm red}}{t_{\rm int}} n_{\rm proc}^2\right].$$

Note that, as in the case of purely-distributed-memory *particle decomposition* approach, in the region of parameters in which the efficiency condition, Eq. (19), is satisfied, Eq. (17) essentially reduces to Eq. (4). Such region, however, is reduced, in comparison with that case, at least by a factor $n_{\rm proc}$.

Figure 2 (left) shows the regions of allowed resolution and high efficiency ($\eta > \eta_*$, with $\eta_* = 50\%$) for the *particle-particle* strategy for three values of the number of particles per cell, $N_{\rm ppc}$, with $n_{\rm proc} = 8$, $t_{\rm red}/t_{\rm int} = 0.2$ and the other parameters as in Fig. 1. The qualitative arguments exposed with reference to Fig. 1 maintain their validity, although the reduction of the efficiency region for this composed method is apparent. Note, however, the different notion of "high efficiency" in the two cases: speed-up $\sim n_{\rm node}$ for the purely-distributed-memory decomposition; speed-up $\sim n_{\rm node} n_{\rm proc}$ in the present one. Note also that higher values of $N_{\rm ppc}$ penalize the maximum achievable resolution, while improving the efficiency. Viceversa, for low values of $N_{ppc}$, parallelization is expected to be very inefficient even for very low values of $n_{\rm node}$. Figure 2 (right) shows the same regions for $N_{\rm ppc} = 64$ and three values of $n_{\rm proc}$: $n_{\rm proc} = 2$, $n_{\rm proc} = 4$ and $n_{\rm proc} = 8$. The maximum resolution only slightly depends on $n_{\rm proc}$ because of the quite high value of $N_{\rm ppc}$ (cf. Eq. (17)); on the opposite, because of the reduction term in Eq. (18), the efficiency decreases with increasing $n_{\rm proc}$.

### 2.2.2. Intra-node domain decomposition

In Section 2.2.1 we have examined the main features, in terms of memory requirements and efficiency, of the decomposition strategies based on a particle decomposition at the intra-node level. The main defect of this approach is represented by the enhancement of the memory load on each computational node. This further load, in fact, comes out to be negligible (in comparison with the distributed particle-array one) if the efficiency condition is satisfied. In the general case, however, one could be more interested in reaching the highest spatial resolution than in getting efficient parallelization (this is especially true for scientific, not routine, computing). In such a general situation, the terms proportional to $n_{\rm proc}$ in Eq. (17) (associated to the storage of the pressure-array copies) put the strongest constraint on the scalability of the problem size with $n_{\rm node}$: an upper limit on the allowed number of $N_{\rm cell}$ is obtained, approximately given by $M_0/[m_{\rm field} + m_{\rm press}(2 + n_{\rm proc})]$, with no dependence on $n_{\rm node}$.

These considerations justify the introduction of a different intra-node decomposition strategy, based on the *domain decomposition* concept. It consists in decomposing the domain and assigning different portions of the domain and the residing particles to each processor. 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 in the pressure updating, although they can still occur in the sorting phase, in which each particle is assigned to a domain portion and labelled by an intra-portion index. The negative impact of such race conditions on the parallelization efficiency can be contained by limiting the sorting phase to those particles that have changed domain portion in the last time step.

Load balancing can be enforced quite easily, by adopting a finer subdivision of the domain, and adding elementary portions to the load assigned to a given processor until the number of particles assigned to the processor approximately equals the average number of particles per processor, $(N_{\rm part}/n_{\rm node})/n_{\rm proc}$. Differently from the distributed memory context, such a load balancing does not require any communication between processors. Moreover, the increment of memory requirements is very contained (essentially limited to the integer labels of the sorted particles), and does not increase, as it does in the *particle decomposition* intra-node approach, with the number of processors per node.

Let us combine such a strategy with the *particle decomposition* technique envisaged for the inter-node level. Memory, time and speed-up will be given, in this *particle-domain* approach, by

$$M_{PD} \approx (m_{\rm field} + 2m_{\rm press})N_{\rm cell} + (m_{\rm part} + \delta m_{\rm part})\frac{N_{\rm part}}{n_{\rm node}}, \tag{20}$$

$$t_{PD} \approx t_{\rm FFT}N_{\rm cell}\log_2 N_{\rm cell} + t_{\rm int}\frac{N_{\rm part}}{n_{\rm proc}n_{\rm node}} + t_{\rm com}N_{\rm cell}\log_2 n_{\rm node} + t_{\rm sort}\frac{N_{\rm part}}{n_{\rm node}}\left(\frac{n_{\rm node}n_{\rm proc}}{N_{\rm cell}}\right)^{\frac{1}{3}}, \tag{21}$$

$$s_u^{PD} \approx n_{\rm node}n_{\rm proc}\left(1+\frac{t_{\rm FFT}}{t_{int}}\frac{\log_2 N_{\rm cell}}{N_{\rm ppc}}\right)\Big/ \left\{1+\frac{n_{\rm proc}}{t_{\rm int}}\left[\frac{n_{\rm node}}{N_{\rm ppc}}(t_{\rm FFT}\log_2 N_{\rm cell} + t_{\rm com}\log_2 n_{\rm node}) + t_{\rm sort}\left(\frac{n_{\rm node}n_{\rm proc}}{N_{\rm cell}}\right)^{\frac{1}{3}}\right]\right\}. \tag{22}$$

Here $\delta m_{\rm part}$ refers to the integer-label particle arrays, and $t_{\rm sort}$ is related to the computation needed to order particles according to the subdomain they belong to (the factor $(n_{\rm node}n_{\rm proc}/N_{\rm cell})^{\frac{1}{3}}$ has been introduced assuming that the particle sorting is limited to the fraction of particles that changed domain portion in the last time step). Note that such a computation is proportional to $N_{\rm part}/n_{\rm node}$, with no benefit coming from the intra-node parallelization. This is due to the protection of *critical* sections from race conditions, which serializes the computation.

The maximum value of $N_{cell}$ compatible with the memory constraint and the high-efficiency condition ($\eta > \eta_*$) are then given, respectively, by

$$N_{\rm cell_{max}}^{PD} = [M_0 n_{\rm node}]/ [(m_{\rm field} + 2m_{\rm press})n_{\rm node} + (m_{\rm part} + \delta m_{\rm part})N_{\rm ppc}] \tag{23}$$

and

$$\frac{\left(n_{\rm proc}n_{\rm node} - \frac{1}{\eta_*}\right)}{N_{\rm ppc}}\frac{t_{\rm FFT}}{t_{\rm int}}\log_2 N_{\rm cell} + \frac{t_{\rm com}}{t_{\rm int}}\frac{n_{\rm proc}n_{\rm node}}{N_{\rm ppc}}\log_2 n_{\rm node} + \frac{t_{\rm sort}}{t_{\rm int}}\left(\frac{n_{\rm node}^{\frac{1}{3}}n_{\rm proc}^{\frac{4}{3}}}{N_{\rm cell}^{\frac{1}{3}}}\right) < \frac{1}{\eta_*} - 1. \tag{24}$$

Note that, differently from the *particle-particle* case, the efficiency condition not only puts an upper limit, for given values of $N_{\rm cell}$ and $n_{\rm proc}$, on the number of nodes, but also defines, for a given $n_{\rm node}$, a lower limit on $N_{\rm cell}$. This is related to the last term on the left hand side of Eq. (24), and it is due to the fact that particle migration from one domain portion to another and the corresponding particle-sorting (serial) computation increases with the surface-to-volume ratio of the domain portions, which is proportional to $(n_{\rm node}n_{\rm proc}/N_{\rm cell})^{\frac{1}{3}}$. The order of magnitude of such a lower limit can be evaluated by considering the regime in which the sorting term is the dominant one:

$$N_{\rm cell} \gtrsim \left(\frac{\eta_*}{1-\eta_*}\right)^3\left(\frac{t_{\rm sort}}{t_{\rm int}}\right)^3 n_{\rm node}n_{\rm proc}^4. \tag{25}$$

Figure 3 (left) shows the regions of allowed resolution and high efficiency (with $\eta_* = 50\%$) for this "particle-domain" decomposition, at three differ-

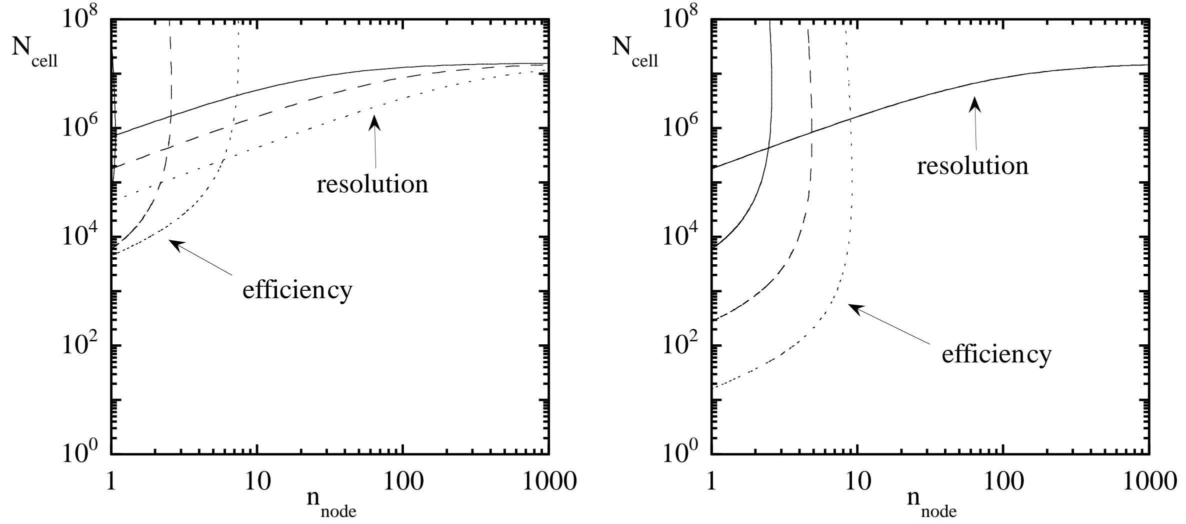


Fig. 3. Curves of allowed resolution and high efficiency ($\eta > 50\%$) for the *particle-domain* decomposition, corresponding to Eqs (23) and (24), respectively. We have fixed $t_{\rm sort}/t_{\rm int} = 1$, $\delta m_{\rm part} = 24$ bytes, and the other parameters as in Fig. 2. The left frame refers to the case with $n_{\rm proc} = 8$ and three different values of the number of particles per cell: $N_{\rm ppc} = 16$ (solid lines), $N_{\rm ppc} = 64$ (dashed lines) and $N_{\rm ppc} = 256$ (dotted lines). The right frame refers to the case with $N_{\rm ppc} = 64$ and three different values of the number of processors: $n_{\rm proc} = 2$ (dotted lines), $n_{\rm proc} = 4$ (dashed lines) and $n_{\rm proc} = 8$ (solid lines).

ent values of $N_{\rm ppc}$. The allowed-resolution regions and the high-efficiency ones correspond to Eq. (23) and (24), respectively. Here we have fixed $t_{\rm sort}/t_{\rm int} = 1$, $\delta m_{\rm part} = 24$ bytes, and the other parameters as in Fig. 1. With such parameters, low values on $N_{\rm ppc}$ ($N_{\rm ppc} = 16$ in this case) do not allow to get efficient results even for very low number of nodes. Note also that, because of the reduced extent of the high-efficiency region, once given a certain resolution level, adopting a number of nodes higher than the required minimum (the value corresponding to the maximum-resolution curves in Fig. 3 is almost useless, with respect to the speed-up values that can be obtained.

Figure 3 (right) shows the resolution and efficiency boundaries for $N_{\rm ppc} = 64$ and three values of $n_{\rm proc}$. A single curve is shown for the maximum resolution, as there is no dependence on $n_{\rm proc}$ in Eq. (23). The dependence of the high-efficiency region extent on $n_{\rm proc}$ is due both to the replicated character of the grid computation (insensitive to $n_{\rm proc}$ and, therefore, inefficient) and to the fact that particle-sorting computation increases with $n_{\rm proc}$, as explained with regard to Eq. (24).

## 3. Implementation of the decomposition strategies

In this Section we describe the implementation, on hierarchical distributed-shared memory architectures, of the different two-stage parallelization strategies discussed in Section 2: the *particle-particle* decomposition strategy and the *particle-domain* decomposition one. As stated above, such strategies, differently from strategies involving an inter-node domain decomposition, can be implemented in the framework of high-level languages – namely, HPF at the inter-node stage and OpenMP at the intra-node one. Within this framework, the inter-node decomposition and the intra-node one can be integrated with negligible programming efforts with the help of the HPF extrinsic procedures `HPF_LOCAL`. High Performance Fortran programs may call non-HPF subprograms as *extrinsic procedures* [8]. 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.

We apply such techniques to a specific PIC code, HMGC [3], developed, in the framework of controlled nuclear fusion research. 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: the minor radius of the torus, $r$, and the poloidal and toroidal angles, $\vartheta$ and $\varphi$, 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 the pressure computation one. We will concentrate, in the following, on the pressure loop, whose distribution represents the bottle-neck in the parallelization of a PIC code (the particle-pushing loop is indeed inherently parallel, with no communication required by non-local accesses).

The pressure loop can be schematized by Fig. 4.

```
 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
```

Fig. 4.

Here, `n_part`$\equiv N_{\rm part}$ 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. Moreover, `r,...` stays for the radial and the other, not 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 `w`. 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.

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.

### *3.1. Inter-node decomposition strategy*

The *particle decomposition* approach consists (see Section 2.1) in statically distributing the particle-population data among different nodes, while replicating the data relative to grid quantities. Its implementation in HPF is, in principle, relatively straightforward and has been discussed in Ref. [5]. 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 outomatically enforced.

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 $(p_{\rm par}(n_r, n_\vartheta, n_\varphi, :))$, 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 Fig. 5.

Note that each local procedure executes only the set of loop iterations that access the particles local to the node (`l=1,UBOUND(r, dim=1)`) 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`.

### 3.2. 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 multiprocessor machine, with a single thread running on each processor. It will execute particle loops embedded in local extrinsic procedures, like that described in Section 3.1.

The natural parallelization strategy for shared memory architectures consists in distributing the work needed to update particle coordinates among different threads (and, then, processors), with no respect to the portion of the domain in which each particles resides. For this reason this workload decomposition can be referred to as a *particle decomposition*. 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 immediate intra-node parallelization of the pressure loop is however 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., [9]), 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, 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 how this race condition can be avoided by applying, to the parallelization of the pressure updating loop, one of the two different intra-node decomposition strategies presented in Sections 2.2.1 and 2.2.2 respectively.

```
      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
```

Fig. 5.

*3.2.1. Particle-particle decomposition strategy*

When applying the particle decomposition technique to the pressure loop, the simple use of the `parallel do` OpenMP directive is no longer sufficient. Caution must indeed 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 trivial solution to this problem (and the least expensive, in terms of code restructuring effort) would consist, 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 Section 3.1 would assume the form, as seen in Fig. 6.

Unfortunately, the intra-node serialization induced by the protected critical section on the shared access to the array `p_par` represents a bottle-neck that heavily affects the performances (almost no speed-up) [4]. Such a bottle-neck can be eliminated, at the expenses of memory occupation, by means of the strategy described in Section 2.2.1. It 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 and consists in splitting the pressure updating among processors: each processor only computes the partial contribution of its own particles; then the partial results are reduced into global ones. The easiest way to implement such a strategy consists 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

```
      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
```

Fig. 6.

```
      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. 7.

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 can be seen in Fig. 7.

Note that this strategy (version *v1* of the parallel HMGC), based on the introduction of an auxiliary array, makes the execution of the `UBOUND(r,dim=1)` ($\approx N_{\mathrm{part}}/n_{\mathrm{node}}$) 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`.

### 3.2.2. *Particle-domain decomposition strategy*

In order to overcome the trade-off between parallelization efficiency and memory requirements, at the price of a heavier restructuring of the code and, possibly, the need of addressing load-balancing problems, the *domain decomposition* strategy (see Section 2.2.2) can be adopted for the work distribution among the different processors of each computational node. 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 (each schematized by the corresponding code excerpt):

- 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 (see Fig. 8).
- The different elementary portions of the domain are assigned to each processor. Load balancing is enforced by adding elementary portions to a

```
      integer j_r_part(n_part),i_r(n_part)
      integer n_part_r(n_r)
      ...
      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
      ...
```

Fig. 8.

given-processor load until the number of particles assigned to the processor approximately equals the average number of particles per processor, $(N_{\rm part}/n_{\rm node})/n_{\rm proc}$. Particles are then sorted according to the processor they belong to (Fig. 9).

– 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 as can be seen in Fig. 10.

Note that the load balancing is implemented within a loop over processors. It then causes negligible computation overheads. Moreover, differently 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 sorted particles), and does not scale with the number of processors per node.

## 4. Experimental results

We have tested the composed strategies discussed in Section 3.2, by running the corresponding HPF+OpenMP versions of HMGC on a IBM SP parallel system, equipped with, among the others, two 8-processor SMP PowerPC nodes, with clock frequency of 200 MHz and 2 GB RAM, and four 2-processor 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 ($N_{\rm cell} = 4096$). The average number of particles per cell has been varied from $N_{\rm ppc} = 4$ to $N_{\rm ppc} = 256$, which corresponds to $N_{\rm part}$ ranging approximately from 16 $k$ to 1 $M$.

Figure 4 (left) shows the scaling of the speed-up of the *pressure updating* procedure for the *particle-particle decomposition* strategy (version *v1*) with respect to the number of processors per node, $n_{\rm proc}$, at fixed number of (8-processor) nodes, $n_{\rm node} = 2$. Figure 4 (right) reports the values of $s_u$ for the same procedure versus the number of (2-processor) nodes, at $n_{\rm proc} = 2$. 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. Speed-up values refer only to the execution of the section related to the updating of the (whole) pressure array `p` (note that the expressions obtained in Section 2.2 maintain the same form even when referred

```
      ...
      integer, allocatable :: j_r_upper(:)
      integer l_index(n_part)
      integer n_part_lower(n_r)
      ...
      n_procs=omp_get_max_threads()
      allocate(j_r_upper(0:n_procs))
c
c   assignment of balanced global portions of the domain
c   to each processor
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   sorting 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
```

Fig. 9.

only to the pressure-updating procedure). In agreement with Eq. (16), we observe, from Fig. 4 (left), that the speed-up values depart from the linear scaling with $n_{\rm proc}$ only for $n_{\rm proc}$ greater than a certain value, which is higher, the higher the average number of particles per cell, $N_{\rm ppc}$, is. On the other side, a significant departure from the linear scaling with $n_{\rm node}$ is observed, in Fig. 4 (right), only for the lowest values of $N_{\rm ppc}$, because of the small number of nodes involved.

It is interesting to compare such results with the model ones reported in Fig. 2. From Fig. 2 (left) we expect, for $N_{\rm cell} = 4096$, $n_{\rm node} = 2$ and $n_{\rm proc} = 8$, that the cases with $N_{\rm ppc} = 16$, $N_{\rm ppc} = 64$ and $N_{\rm ppc} = 256$ are characterized, respectively, by $\eta < 50\%$, $\eta \approx 50\%$ and $\eta > 50\%$. Moreover, from Figs 2 (right), which refers to $N_{\rm ppc} = 64$, we see that the point $(n_{\rm node} = 2, N_{\rm cell} = 4096)$ falls in the high-efficiency region both for $n_{\rm proc} = 2$ and $n_{\rm proc} = 4$, while it is close to the $\eta = 50\%$ curve for $n_{\rm proc} = 8$. Finally, from the same Figure we note that the high-efficiency region for $n_{\rm proc} = 2$ extends up to $n_{\rm node} \approx 10$: we can then expect that cases with $n_{\rm node} \leqslant 4$ are all contained in that region. All these previsions are confirmed by the experimental results shown in Fig. 11.

Figure 12 (left) shows the scaling of the speed-up with respect to $n_{\rm proc}$, at fixed number ($n_{\rm node} = 2$) of 8-processor nodes, obtained by the *particle-domain decomposition* version, *v2a*, of the *pressure updating* procedure. Figure 12 (right) reports the values of $s_u$ for the same procedure versus the number of (2-processor) nodes, at $n_{\rm proc} = 2$. We note that, at least for the specific application here considered, this *particle-domain decomposition* strategy can be an interesting compromise between maximizing efficiency and minimizing memory. The bottle-neck, with regard to the efficiency performances, is represented by the critical section associated to the sorting procedure. A significant improvement of the efficiency can be obtained by limiting the sorting phase (and then the *critical* computation) to those particles that have changed domain portion in the last step. This will indeed produce a reduction of the related computation by a factor $(n_{\rm node} n_{\rm proc}/N_{\rm cell})^{\frac{1}{3}}$,

```
      ...
      p_par = 0.
c
c   pressure loop
c
!$OMP parallel do private(i_proc,j_r,i,l0,l,j_theta,j_phi)
      do i_proc=1,n_procs
       do j_r=j_r_upper(i_proc-1)+1,j_r_upper(i_proc)
        do i=1,n_part_r(j_r)
         l0=n_part_lower(j_r)+i
         l=l_index(l0)
         j_theta = f_theta(theta(l))
         j_phi = f_phi(phi(l))
         p_par(j_r,j_theta,j_phi,1) =
     &                           p_par(j_r,j_theta,j_phi,1)
     &                           + h(r(l),...,w(l))
        enddo
       enddo
      enddo
!$OMP end parallel do
```

Fig. 10.

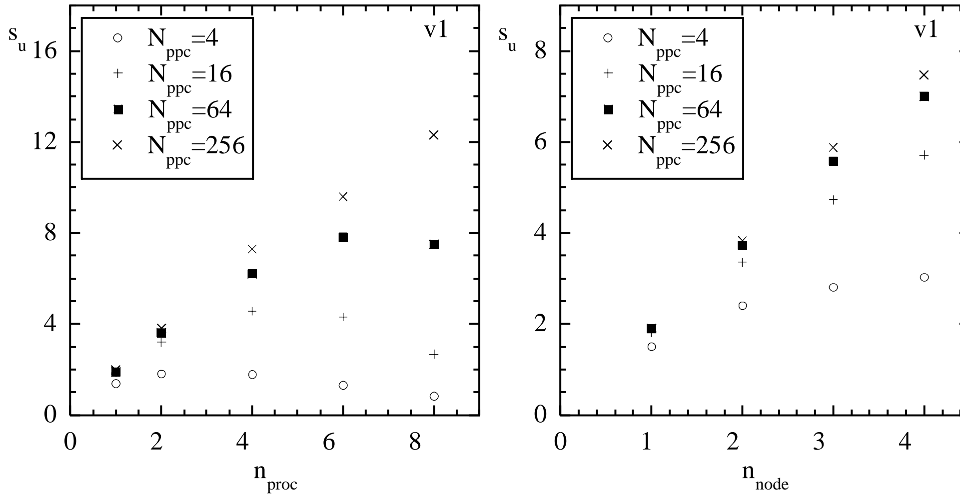


Fig. 11. Speed-up of the *pressure updating* procedure for the *particle-particle decomposition* version (*v1*), at different values of the average number of particles per cell, $N_{\mathrm{ppc}}$. Such quantity is plotted (left) versus the number of processors per node, at fixed number of (8-processor) nodes, $n_{\mathrm{node}} = 2$, and (right) versus the number of 2-processor nodes, at fixed number of processors per node, $n_{\mathrm{proc}} = 2$.

as noted with regard to Eq. (21). Figure 13 shows the same scalings as Fig. 12 for a modified *particle-domain decomposition* version, *v2b*, which implements such a selective sorting. Such results can be compared with the model ones reported in Fig. 3. The agreement is very satisfactory. In particular, the experimental results show that, differently from the *particle-particle* framework, all the cases with $N_{\mathrm{cell}} = 4096$, $n_{\mathrm{node}} = 2$, $n_{\mathrm{proc}} = 8$ are characterized by low efficiency, regardless, in this respect, to the number of particle per cell, $N_{\mathrm{ppc}}$. This is consistent with the model findings, Eq. (25) and Fig. 3 (left), which predict that, for $n_{\mathrm{node}} = 2$ and $n_{\mathrm{proc}} = 8$, $N_{\mathrm{cell}} = 4096$ is below the lower limit for efficient parallelization, independently of $N_{\mathrm{ppc}}$. On the opposite, the cases with $n_{\mathrm{node}} = 2$, $N_{\mathrm{ppc}} = 64$ and $n_{\mathrm{proc}} = 2$ or 4 are high-efficiency ones, as predicted on the basis of the model results shown in Fig. 3 (right). High efficiency is in fact obtained, consistently with the same model results, in the case with $N_{\mathrm{ppc}} = 64$ and $n_{\mathrm{proc}} = 2$, for all the considered values of $n_{\mathrm{node}}$ ($n_{\mathrm{node}} \leqslant 4$).

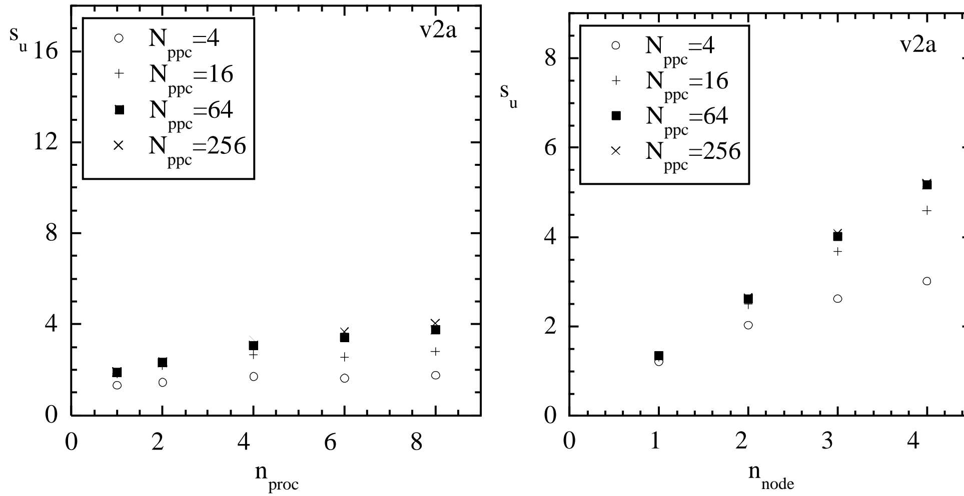


Fig. 12. Speed-up of the *pressure updating* procedure for the *particle-domain decomposition* version, *v2a*, at different values of $N_{\rm ppc}$. The left frame shows this quantity versus the number of processors per node, at fixed number of (8-processor) nodes, $n_{\rm node} = 2$. The right frame considers the effect of varying the number of 2-processor nodes, at fixed number of processors per node, $n_{\rm proc} = 2$.

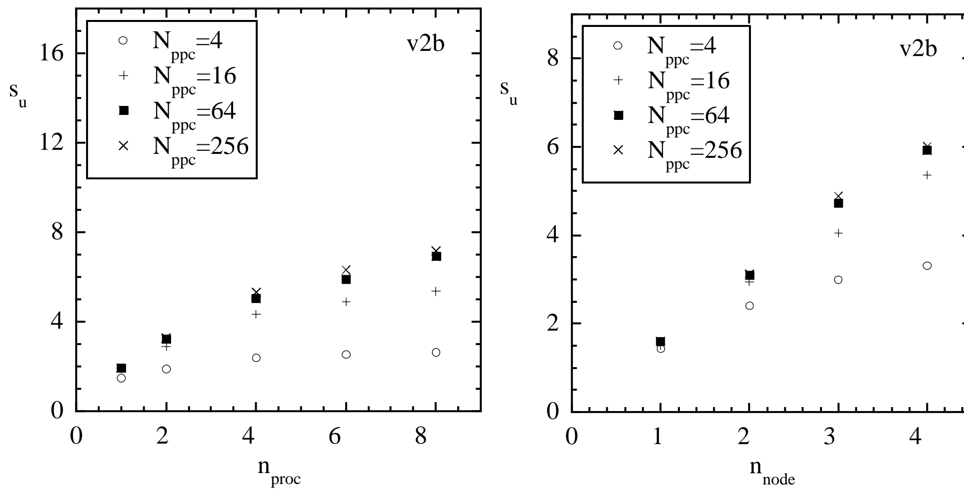


Fig. 13. Speed-up for the *selective sorting* version, *v2b*, of the *pressure updating* procedure, at different values of $N_{\rm ppc}$: versus the number of processors per node, at fixed number of (8-processor) nodes, $n_{\rm node} = 2$ (left); versus the number of 2-processor nodes, at fixed number of processors per node, $n_{\rm proc} = 2$ (right).

Finally, it is worth noting that only speed-up values related to the pressure-updating loop have been reported in the present Section. As the parallelization of the particle-pushing loop is trivial (and common to all the strategies here considered), the overall results would correspond to better (and less spread) performances.

## 5. Summary

We have studied the problem of porting large-scale PIC codes on hierarchical distributed-shared memory systems in a high-level language programming framework. The devised workload decomposition 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 choice of a high-level language parallelization (which reduces the programming efforts and increases the portability of the resulting code) forces the adoption of a *particle decomposition* strategy at the inter-node level. It is indeed suited to be implemented in HPF, differently from the *domain decomposition* strategy, preferable, in principle, because of the reduced mem-

ory requirements. At the intra-node, shared-memory, level, both *particle* and *domain decomposition* can be implemented in a high-level language such as OpenMP and integrated with the *particle decomposition* strategy devised at the distributed-memory level. Then, two distinct composed strategies – a *particle-particle* decomposition and a *particle-domain* one – can be envisaged for the overall parallelization. The respective merits of such different strategies have been examined, on the basis of simple estimates, with respect to the memory occupancy, and the parallelization efficiency.

The implementation of each strategy has been described in detail. It is based on integrating the HPF and OpenMP programming environments. This task can be accomplished, with negligible programming efforts, by means of the HPF extrinsic procedures `HPF_LOCAL`.

The experimental measurements of the speed-up factors obtained by each version of the code (corresponding to the various decomposition strategies) at different values of the number of particles, nodes and processors, qualitatively confirm the model predictions, although a more complete comparison would require tests on a much larger cluster of SMP nodes.

It comes out that each of the composed strategy presents merits and defects. More precisely, the *particle-particle* decomposition 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 the spatial grid. This imposes, as the number of processors is increased, a cut-off on the maximum size of the spatial grid that can be simulated (that is, the maximum value of $N_{\mathrm{cell}}$). On the opposite, the *particle-domain* decomposition does not introduce such a supplementary need for memory, but it requires a relevant programming effort and produces lower speed-up values. Moreover, the efficiency condition imposes, for this strategy, a lower limit on $N_{\mathrm{cell}}$, because the demand for (serial) particle-sorting computation increases with decreasing size (in terms of number of cells) of each domain portion. Finally, both strategies yield better performances in term of parallelization efficiency at high values of the number of particles per cell, $N_{\mathrm{ppc}}$, and not too large values of the number of processors per node, $n_{\mathrm{proc}}$.

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

[3] S. Briguglio, G. Vlad, F. Zonca and C. Kar, Hybrid Magnetohydrodynamic-Gyrokinetic Simulation of Toroidal Alfvén Modes, *Phys. Plasmas* **2** (1995), 3711–3723.

[4] B. Di Martino, S. Briguglio, G. Vlad and G. Fogaccia, *Workload Decomposition Strategies for Shared Memory Parallel Systems with OpenMP*, to appear on Scientific Programming, 2002.

[5] B. Di Martino, S. Briguglio, G. Vlad and P. Sguazzero, Parallel PIC Plasma Simulation through Particle Decomposition Techniques, *Parallel Computing* **27**(3) (2001), 295-314.

[6] 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.

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

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

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

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

[11] 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.

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