Workload decomposition strategies for shared memory parallel systems with OpenMP

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

A crucial issue in parallel programming (both for distributed and shared memory architectures) is work decomposition. Work decomposition task can be accomplished without large programming effort with use of high-level parallel programming languages, such as OpenMP. Anyway particular care must still be payed on achieving performance goals. In this paper we introduce and compare two decomposition strategies, in the framework of shared memory systems, as applied to a case study particle in cell application. A number of different implementations of them, based on the OpenMP language, are discussed with regard to time efficiency, memory occupancy, and program restructuring effort.

Keywords: OpenMP, shared memory, plasma simulation, particle in cell simulation, particle decomposition

1. Introduction

A crucial issue in parallel programming (both for distributed and shared memory architectures) is 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 or data access contention (for shared memory programming) and reducing the overhead due to managing the work decomposition itself.

Work decomposition task can be accomplished without large programming effort with use of high-level parallel programming environments/languages, such as High Performance Fortran (HPF) [13] (for distributed memory systems), and OpenMP [20] (for shared memory systems), expecially when the issue is porting large sequential codes to parallel architectures. While the developer is leveraged, with 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).

Here we focus on the problems related to porting typical particle in cell (PIC) applications on parallel architectures. The PIC simulation consists [4] 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 workload decomposition, in the context of distributed memory systems: the domain decomposition strategy and the particle decomposition one. Standard domain decomposition [12,16] 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. 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 complicate the parallel implementation of a serial code, expecially with high-level languages, besides introducing extra computational and communication overheads. On the opposite side, 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.

The particle decomposition [10] technique consists in statically distributing the particle population among processors, while replicating the data relative to grid

ISSN 1058-9244 / $8.00 © 2001, IOS Press. All rights reserved

quantities. It is apparent that load balancing is automatically enforced, because no particle has to be transferred (reassigned) from one processor to another. As a consequence, the implementation of such strategy with high-level languages is, in principle, relatively straightforward. This at the expense of an overhead on memory occupancy, given by the replication of data related to the domain, and a communication overhead related to the updating of the fields (each node manages only the partial updating associated to its portion of particle population). The former overhead forbids a good scalability of the maximum domain size with the number of processors; the latter one limits 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.

Aim of this paper is to compare the two decomposition strategies, in the framework of shared memory systems, as applied to the case study PIC application. A number of different implementations of them, based on the high-level language OpenMP, are discussed with regard to time efficiency, memory occupancy, and program restructuring effort.

The specific PIC code we deal with – the Hybrid MHD-Gyrokinetic Code (HMGC) [6] – includes all the main features of the codes developed for the investigation of, e.g., plasmas magnetically confined in toroidal devices. Porting of the same code in HPF for distributed memory architectures was presented in [10].

The paper is structured as follows. Section 2 describes the the main physical and computational aspects of the chosen application, also with respect to the parallelization of the corresponding codes. The different implementations in OpenMP of the particles decomposition strategy and the domain decomposition one for the shared memory context are described in Sections 3 and 4, respectively. The experimental results obtained with the HMGC PIC code are also reported. Conclusions on the validity of the proposed strategies are drawn in Section 5.

2. The plasma particle simulation application

Particle simulation codes [4] seem to be the most suited tool for the investigation of turbulent plasma behaviour. Particle simulation indeed consists in replacing the physical particle population by a simulation-particle one, with each particle representing – by its weight – a cloud (macroparticle) of non mutually interacting physical particles. By identifying the charge and the mass of each simulation particle with those of the whole cloud, and imposing that such a particle moves as its physical counterpart, all the relevant parameters (e.g., the Debye length, λ_D) of the simulation plasma coincide with the corresponding parameters of the physical plasma, notwithstanding that the simulation-particle density is much lower than the physical-particle one. The phase-space coordinates and the weight of the simulation particles are then evolved in the electromagnetic fields selfconsistently computed, at each time step, in terms of certain momenta of the particle distribution function (e.g., pressure), so retaining all the relevant kinetic effects.

The most widely used method for particle simulation is represented by the PIC approach. PIC simulation techniques consist in

- computing the electromagnetic fields only at the points of a discrete spatial grid (field solver phase);
- interpolating them at the (continuous) particle positions in order to evolve particle phase-space coordinates and weights (particle pushing phase);
- collecting particle contribution to 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 $N_{\mathrm{ppc}} \equiv N_{\mathrm{part}}/N_{\mathrm{cell}} \gg 1$, where N_{part} is the number of simulation particles, 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, 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.

2.1. Parallelization of particle in cell codes

Several contributions exist, in literature, on design and development of parallel particle in cell applications, targeted towards distributed memory architectures [1,9–11,16,17,19], shared memory [2,18], and hybrid distributed-shared memory ones [5].

Many of them [1,9,11,16,18,19] are based on the domain decomposition strategy, while the particle decomposition approach has been adopted in [10,17]. References [11,15] deal with the problem of dynamic load balancing for domain decomposition and provide a number of solutions.

Other works address the issue of a suitable composition of the two strategies, in order either to achieve a proper balance of respective merits and drawbacks [2] or to exploit hierarchical distributed-shared memory architectures, such as clusters of SMPs [5].

Several experiences are reported on utilization of different high level languages, programming supports and paradigms. References [1,10] present parallelization efforts carried out using High Performance Fortran. Reference [5] proposes an integration of High Performance Fortran and OpenMP for programming clusters of SMPs. References [7,19] propose the application of the Object Oriented programming paradigm; in particular, Ref. [7] discusses the utilization of Java, with JavaMPI as message passing support. Finally, Ref. [9] proposes an approach based on Programming Skeletons.

With regard to the contributions [2,18] specifically targeted to shared memory architectures, it is worth mentioning that Ref. [18] addresses the issue of porting 3D PIC codes on CC-NUMA shared memory architectures, namely a Convex Exemplar machine. The hierarchical memory structure of the architecture solicits minimization of accesses to remote memory and maximization of cache reuse; these demands induced those Authors to adopt a domain decomposition, with particles and fields data storage in nodes' private memory; minimization of cache misses was achieved by periodically rebalancing particles among processors through sorting.

Reference [2] proposes a hierarchical combination of particle and domain decomposition (hybrid partitioning) for execution on NUMA shared memory architectures, in order to obtain the best compromise between memory occupancy and non-local accesses. In this approach, the set of possible data partitioning schemes ranges from a standard domain decomposition (the domain is decomposed in portions, each assigned to a processor) to a standard particle decomposition (the whole domain is replicated among processors) passing through schemes where subgroups of processors share portions of the domain (with each portion assigned to a subgroup) and where the number of processor per group (and thus the number of domain portions) characterize the partitioning; the best partitioning is then selected heuristically on the basis of empirical results.

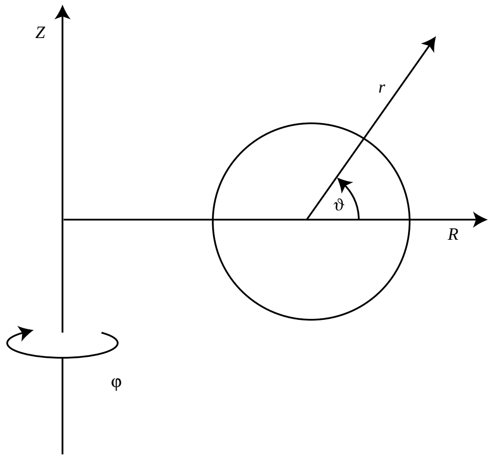

Fig. 1. Toroidal coordinate system (r, ϑ, φ) for a tokamak plasma equilibrium.

2.2. *Parallelization of HMGC on shared memory architectures*

Here we consider the parallelization of a specific PIC code, HMGC, 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 [8]. The code consists of approximately 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:

```
do l = 1, n_part
 r_l = r(l)
 ...
 w_l = w(l)
 r(l) = r_l + g_r(r_l,...)
 ...
```

```
  w(l) = w_l + g_w(r_l,...)
 enddo
```

with n_part= N_part being the number of particles and g_r, ..., g_w 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 phase-space coordinates.

The pressure loop can be schematized by the following one:

```
 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 spatial grid point. 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.

It can be seen 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 to different threads (and, then, processors) the work needed to update particle coordinates and weights. 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. As no particular problem arises for the pushing phase, it will be neglected in the following.

The immediate parallelization of the pressure loop is inhibited by the updating of the array p. Such a computation is indeed an example of irregular array-reduction operation (cf., e.g. [14]), 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). 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 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 next Sections we discuss some possible parallelization strategies for the pressure updating loop, which present close analogies to the strategies developed in the context of distributed memory models: namely, the particle decomposition strategy, and the domain decomposition one.

3. Particle decomposition strategy

As stated above, the most immediate parallelization strategy for shared memory architectures consists in distributing the particle loop iterations among different threads, 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 using the `critical` directive, in the following way:

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

The optional name, `p_lock`, given to the critical section is intended to distinguish such a section from other analogous critical ones: in the real code (HMGC), indeed, other arrays, besides the pressure `p`, must be computed, playing the same role of inhibitors of parallelism. The assignment of a different name to each critical section avoids to apply the mutual exclusion to threads accessing different shared arrays.

Unfortunately, the serialization induced by the protected critical section on the shared access to the array `p` represents a bottleneck that can affect performance.

We have tested this strategy (and the other ones presented in the following), by running the corresponding OpenMP version ($v1a$) of HMGC on a IBM SP parallel system, equipped with, among the others, two 8-processors SMP RISC processors, with clock frequency of 160 MHz, 512 MB Random Access Memory and 9.1 GB Hard Disk. The OpenMP codes 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. Several cases have been considered, with different sizes of the grid and the particle population. To be specific, executions on a grid with `n_r` $= 32n$ cells in the radial direction, `n_theta` $= 16n$ cells in the poloidal direction, and `n_phi` $= 8n$ cells in the toroidal one have been considered (i.e., $N_{\text{cell}} = 4096n^3$), with n ranging from 1 to 4. Moreover, the average number of particles per cell has been varied, ranging from $N_{\text{ppc}} = 1$ to a maximum value depending on the grid size and corresponding to $N_{\text{part}} \equiv N_{\text{cell}} \times N_{\text{ppc}} = 4194304$.

Figure 2 shows the scaling of the speed-up (s_u) with respect to the number of processors. The results are shown for $n = 1$ and five different values of the average number of particles per cell: from $N_{\text{ppc}} = 4$ to $N_{\text{ppc}} = 1024$. Speed-up values refer only to the execution of the section related to the updating of the pressure (and analogous quantities). The speed-up has been defined as the ratio between the wall-clock time, t_s, obtained by the serial execution (`-qsmp=omp` option suppressed) of the OpenMP version of the code and the one, t_{v1a}, obtained by the parallel execution. In practice, such times are computed as

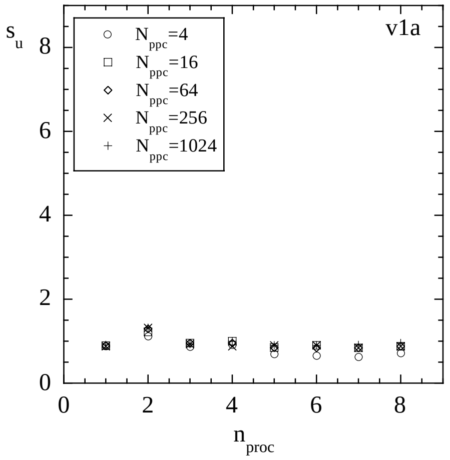

Fig. 2. Speed-up of the pressure-updating phase versus the number of processors, for the critical section version, $v1a$. Here $n = 1$ and five different values of the average number of particles per cell have been considered: from $N_{\text{ppc}} = 4$ to $N_{\text{ppc}} = 1024$.

$$t_{v1a} \equiv t_{\text{reset}} + t_{\text{loop}}, \tag{1}$$

$$t_s \equiv t_{v1a}|_{\text{serial}}, \tag{2}$$

where t_{reset} is the time needed to reset the values of the grid arrays elements to zero, and t_{loop} is the time required by the execution of the particle loop. The results for the case $n_{\text{proc}} = 1$ have been also reported, corresponding to executions on a single processor of the parallel versions.

It can be seen that, for all the considered cases, the speed-up comes out to be very poor (nearly fixed to 1). In fact, the synchronization imposed by the `critical` directive makes the computation substantially serial. We observe that the caution imposed by such a directive is likely to be too conservative: the race condition, although possible in principle (and, in fact, unpredictable), probably occurs for a very few loop iterations. On the basis of this observation, we have developed a different OpenMP implementation ($v1b$) of the pressure computation, whose schematic representation is reported in the Appendix. During each loop iteration, after computing the indices `j_r`, `j_theta` and

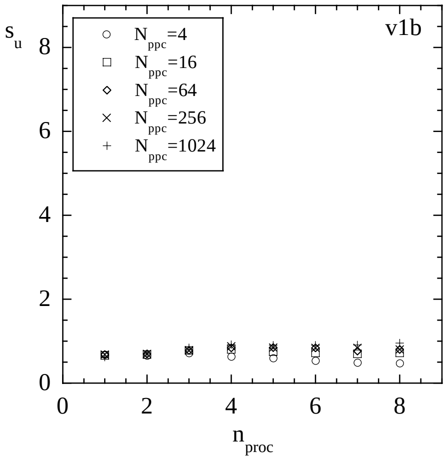

Fig. 3. Speed-up versus the number of processors, for the element locking version, $v1b$. The same parameters of Fig. 2 have been considered.

j_phi, a check is executed on the status of the corresponding element of the array p. If the element is currently addressed (in reading and updating operations) by a concurrent loop iteration, the current iteration is skipped, and the processing of the corresponding particle is postponed to a further loop execution. If, on the contrary, the element is "free", it is immediately locked by the current iteration; it will be freed again after its updating has been completed. Further executions of the loop (within the same time step) are limited to the residual (not processed) portion of particle population. The instructions needed for locking and releasing the array p element, as well as for checking its status, are executed in critical sections labelled by the same name. In principle, if the relative weight of the computation protected by such critical sections came out to be lower than that of the p-element updating (protected by the critical section of the version $v1a$), the version $v1b$ could yield, in spite of the computation overhead, better results than the previous one.

In fact, for the specific application we consider in this paper, the $v1b$ implementation does not improve the speed-up results, as it can be seen from Fig. 3. Here the whole wall-clock time is given by

$$t_{v1b} \equiv t_{\rm reset} + t_{\rm loops}, \tag{3}$$

where $t_{\rm loops}$ is the sum of the wall-clock times related to the different executions of the particle loop (if the race condition effectively occurs quite rarely, such time is dominated by the first execution).

The bottleneck represented by the protection of critical sections of the particle loop can be eliminated, at the expenses of memory occupation, by means of an alternative strategy, 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.

Here we consider two different ways to implement such a strategy. In the first one ($v2a$), the splitting is obtained by introducing an auxiliary array, p_aux, defined as a private variable with the same dimensions as p. Each processor works on a separate copy of the array, 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; the critical directive can be used to perform such a sum. The code section then reads as follows:

```
  p = 0.
!$OMP parallel private(l,j_r,
j_theta,j_phi,p_aux)
  p_aux=0.
!$OMP do
  do l=1,n_part
   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 = p + p_aux
!$OMP end critical (p_lock)
!$OMP end parallel
```

The second way ($v2b$) to implement the alternative strategy consists in giving the auxiliary array p_aux a shared-variable character, while augmenting its rank by one dimension, sized as the number of processors. Each page of the augmented dimension of the array p_aux will be updated, within the bodies of the distributed loops, by a different thread. At the end of the distributed loop, the reduction of the pages of p_aux into the array p is very easily performed by using the intrinsic function SUM. The section assumes the following form:

```
  real*8, allocatable ::
  p_aux(:,:,:,:)
  n_threads=omp_get_max_threads()
  allocate(p_aux(n_r,n_theta,n_phi,
  n_threads))
  p_aux=0.
  p = 0.
!$OMP parallel private(l,j_r,
j_theta,j_phi,i_thread)
  i_thread=omp_get_thread_num()+1
!$OMP do
  do l=1,n_part
   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,i_thread)=
 &  p_aux(j_r,j_theta,j_phi,i_thread)
 &  + h(r(l),...,w(l))
  enddo
!$OMP end do
!$OMP end parallel
  p = sum(p_aux,dim=4)
```

Here, `n_r`, `n_theta` and `n_phi` are the number of grid points in the radial, poloidal and toroidal direction, respectively. The execution environment integer function `omp_get_max_threads` has been used to identify the maximum number of threads that can constitute the team executing the parallel region.

Note that this alternative strategy (based on the introduction of auxiliary arrays), makes the execution of the $N_{\rm part}$ iterations of the loop perfectly parallel. The serial portion of the computation is limited to the reduction of the different copies (or pages) of `p_aux` into `p`. Then, its size scales with $N_{\rm cell} \times n_{\rm proc}$, with $N_{\rm cell}$ and $n_{\rm proc}$ being the number of grid points and processors (equal to the number of threads), respectively. Such product is much smaller than $N_{\rm part}$, as long as the $N_{\rm ppc} \gg n_{\rm proc}$. The price payed to obtain such an improvement is represented by the increased memory requirement: $N_{\rm cell} \times n_{\rm proc}$ more real*8 elements must be stored. 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. Under the above condition, $N_{\rm ppc} \gg n_{\rm proc}$, the whole memory requirement is not significantly affected. This conclusion can however break down if many other arrays, besides `p`, need to be copied (or augmented) in the concrete case (cf. the comment above concerning the use of an optional name for the critical section). We can represent such a feature by giving

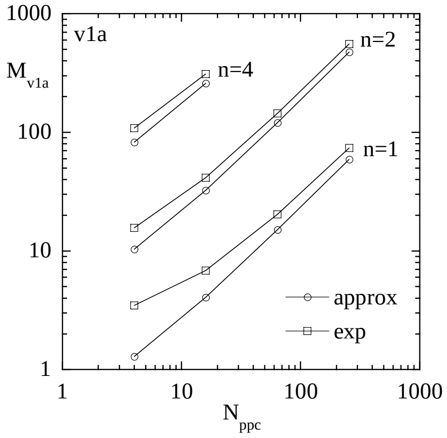

Fig. 4. Memory-occupation data measured (squares) during executions with $n_{\rm proc} = 4$ for the version $v1a$, compared, at different values of n and $N_{\rm ppc}$, with the values corresponding to the approximations (circles) given by Eq. (6).

the following approximate expressions of the memory requirement for, e.g., the versions $v1a$ and $v2a$:

$$M_{v1a} \approx m_{\rm cell}N_{\rm cell} + m_{\rm part}N_{\rm ppc}N_{\rm cell}, \qquad (4)$$

$$M_{v2a} \approx m_{\rm cell}N_{\rm cell} + \delta m_{\rm cell}N_{\rm cell}n_{\rm proc} + m_{\rm part}N_{\rm ppc}N_{\rm cell}, \qquad (5)$$

where the term proportional to $m_{\rm cell}$ refers the shared grid arrays, that proportional to $\delta m_{\rm cell}$ corresponds to the different copies (or pages) of the pressure-like arrays, that proportional to $m_{\rm part}$ is related to the $N_{\rm ppc} \times N_{\rm cell}$ particles. For the specific case of HMGC, and the particular choice of the dependence of $N_{\rm cell}$ on the parameter n, the asymptotic (large n) behaviour of the memory requirements (in Megabytes) can be approximated as follows

$$M_{v1a}|_{HMGC} \approx (0.37 + 0.23N_{\rm ppc})n^3, \qquad (6)$$

$$M_{v2a}|_{HMGC} \approx (0.37 + 0.11n_{\rm proc} + 0.23N_{\rm ppc})n^3. \qquad (7)$$

Such approximations are compared with the experimental data in Figs 4 and 5, where the memory-occupation data measured during executions with $n_{\rm proc} = 4$ are reported, for different values of n and $N_{\rm ppc}$, along with the values corresponding to the approximations given by Eqs (6) and (7), respectively.

From Eqs (5) and (7) we see that the memory enhancement is effectively negligible for

$$N_{\rm ppc}/n_{\rm proc} \gg \delta m_{\rm cell}/m_{\rm part} \simeq 0.5.$$

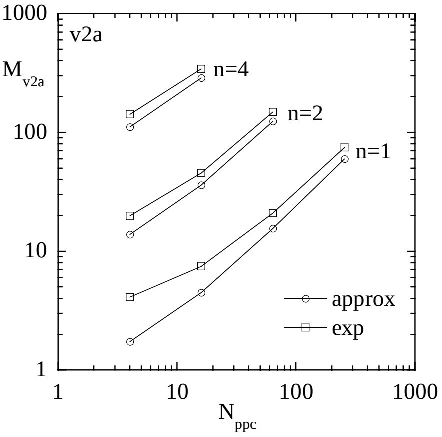

Fig. 5. Comparison between the memory-occupation data measured (squares) during executions with $n_{\text{proc}} = 4$ for the version $v2a$, compared, at different values of n and N_{ppc}, and the approximations (circles) given by Eq. (8).

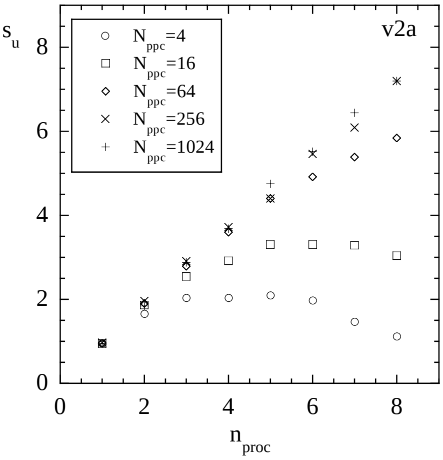

Fig. 6. Speed-up versus the number of processors, for the private array version, $v2a$. Parameters are chosen as in the previous Figures.

Figures 6 and 7 show the speed-up values obtained by version $v2a$ and version $v2b$, respectively. The corresponding times are defined as follows:

$$t_{v2a} \equiv t_{\text{reset}} + t_{\text{parallel}}, \tag{8}$$

$$t_{v2b} \equiv t_{\text{reset}} + t_{\text{parallel}} + t_{\text{sum}}, \tag{9}$$

where t_{parallel} and t_{sum} are the times required by the execution of the whole `parallel` section and the `sum` operation, respectively. Note that, for the version $v2a$, t_{parallel} includes the time needed for the reduction operation, while, for the version $v2b$, t_{reset} includes the time needed for resetting of the elements of the augmented arrays.

We observe that, in these cases, the speed-up values depart from the linear scaling only for n_{proc} greater than a certain value, which is higher, the higher the average number of particles per cell, N_{ppc}, is. These findings can be qualitatively explained considering that the following approximations hold:

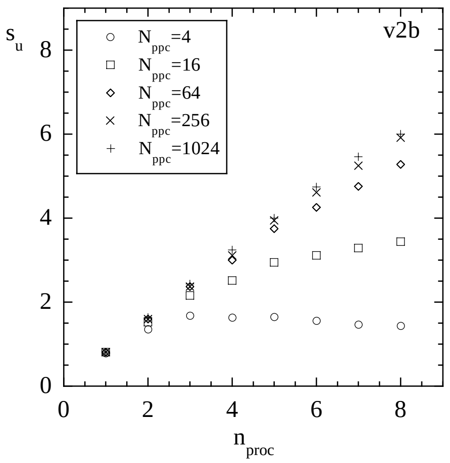

Fig. 7. Speed-up versus the number of processors, for the augmented array version, $v2b$. Parameters are chosen as in the previous Figures.

$$t_s \approx t_{\text{loop}} \approx \alpha_{\text{loop}} N_{\text{ppc}} N_{\text{cell}}, \tag{10}$$

$$t_{v2a} \approx t_{v2b} \approx \alpha_{\text{loop}} \frac{N_{\text{ppc}} N_{\text{cell}}}{n_{\text{proc}}} + \alpha_{\text{red}} n_{\text{proc}} N_{\text{cell}}. \tag{11}$$

Here we neglect the time required to reset the arrays to zero, create and terminate threads and distribute the work among them. We also neglect the different scaling with n_{proc} between the time required by the reduction of the private copies of the arrays and the one required by the optimized `sum` of the different pages of the augmented arrays. From the above approximations, we expect a speed-up approximately given by

$$s_u \approx \frac{n_{\text{proc}}}{1 + \dfrac{\alpha_{\text{red}}}{\alpha_{\text{loop}}} \dfrac{n_{\text{proc}}^2}{N_{\text{ppc}}}}. \tag{12}$$

Figure 8 shows the parallelization efficiency ($\eta \equiv s_u / n_{\text{proc}}$) versus the quantity $n_{\text{proc}}^2 / N_{\text{ppc}}$, for the two versions, $v2a$ and $v2b$. It can be seen that both the ideal efficiency regime (at low values of $n_{\text{proc}}^2 / N_{\text{ppc}}$) and the decreasing efficiency one (at higher $n_{\text{proc}}^2 / N_{\text{ppc}}$) follow the approximate scaling obtained from Eq. (12). Values for executions with $n = 1$, 2 and 4 are reported. The slight differences observed at different n (and, then, N_{cell}) in the low-efficiency regime can be explained in terms of corrections to the approximate expressions of t_{v2a} and t_{v2b} proportional to n_{proc}; such corrections may be related to the management of threads.

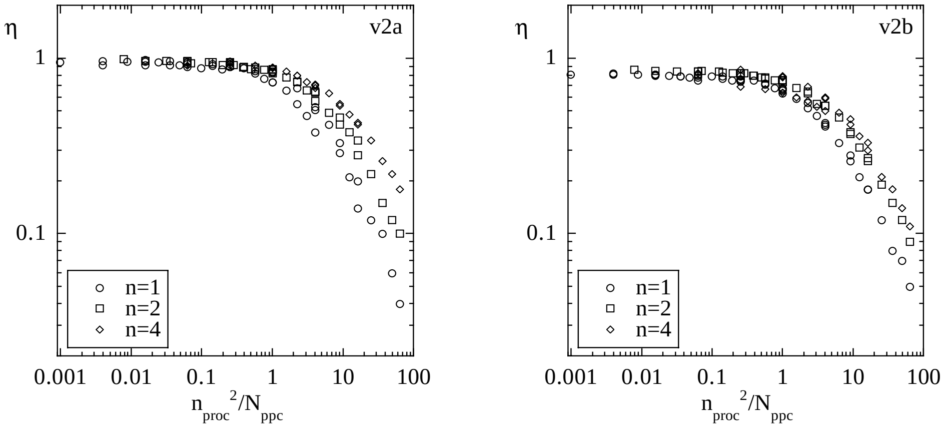

Fig. 8. Efficiency values versus $n_{\mathrm{proc}}^2/N_{\mathrm{ppc}}$, for the private ($v2a$) and the augmented array ($v2b$) versions. Values for executions with $n = 1$, 2 and 4 are reported.

We can conclude that the two alternative parallel versions that avoid the insertion of `critical` sections in the particle loop are both rather efficient (with a slight prevalence of version $v2a$ over version $v2b$) as far as $n_{\mathrm{proc}}^2/N_{\mathrm{ppc}}$ is lower than a certain threshold, which, for the specific code considered in this paper, comes out to be approximately equal to 1.

4. Domain decomposition strategy

In the previous Section, we have discussed several different implementations of what we can indicate as a particle decomposition strategy. Indeed, the work distribution consists in assigning the particle loop iterations to different threads, 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 threads 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: namely, the domain decomposition strategy. 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 thread. 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 threads, however, no further race condition occurs in updating the pressure array element, as loop iterations that could, in principle, concur to the updating of the same element are executed by the same thread.

A possible implementation of this strategy (version $v3a$, 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 thread. Load balancing is enforced by adding elementary portions to a given-thread load until the number of particles assigned to the thread approximately equals the average number of particles per threads, $N_{\mathrm{part}}/n_{\mathrm{proc}}$. Particles are then reordered according to their thread belonging.
- The pressure loop is executed in the form of a parallel loop over threads in which a loop over the particle belonging to the thread is nested. Race conditions are automatically avoided.

Note that the load balancing is implemented within a loop over threads. It then causes negligible com-

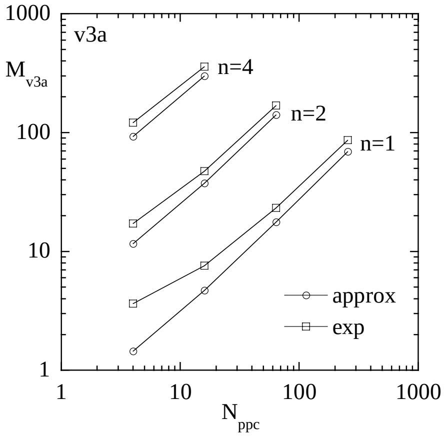

Fig. 9. Memory-occupation data measured (squares) during executions with $n_{\rm proc} = 4$ for the version $v3a$, compared, at different values of n and $N_{\rm ppc}$, with the approximations (circles) given by Eq. (14).

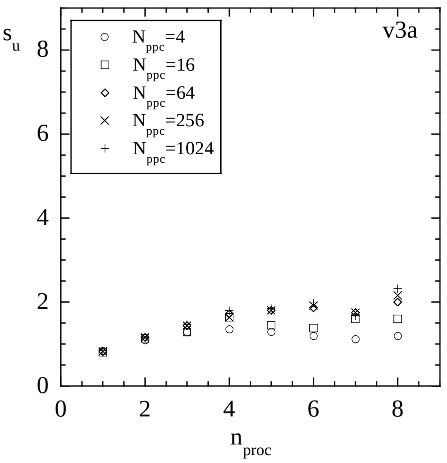

Fig. 10. Speed-up versus the number of processors, for the domain decomposition version, $v3a$. Parameters are chosen as in the previous Figures.

putation overheads. Moreover, different from the distributed memory context, it does not require 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 threads (processors). The approximate expression for such a requirement can indeed be written as

$$M_{v3a} \approx m_{\rm cell} N_{\rm cell} + (m_{\rm part} + \delta m_{\rm part}) N_{\rm ppc} N_{\rm cell}, \tag{13}$$

with the quantity $\delta m_{\rm part}$ being related to the integer-label particle array.

In the case of HMGC, the asymptotic (large n) behaviour of the memory requirement (in Megabytes) can be approximated by

$$M_{v3a}|_{-HMGC} \approx (0.37 + 0.27 N_{\rm ppc}) n^3, \tag{14}$$

which has to be compared with the expressions given by Eqs (6) and (7). Figure 9 shows the satisfactory agreement between this approximation and the experimental results obtained, with $n_{\rm proc} = 4$, at different values of n and $N_{\rm ppc}$.

Figure 10 shows the speed-up values obtained by this domain decomposition version, $v3a$. The wall-clock time is computed, in this case, as follows:

$$t_{v3a} \equiv t_{\rm reset} + t_{\rm pre-loop} + t_{\rm assign} + t_{\rm reorder} + t_{\rm loop}. \tag{15}$$

Here $t_{\rm pre-loop}$ refers to the particle loop needed to identify the domain portion in which each particle falls, $t_{\rm assign}$ is the time required by the balanced assignment loop (over threads), $t_{\rm reorder}$ and $t_{\rm loop}$ 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 respect to the efficiency performances, is still represented by the critical section, although this bottleneck is not so penalizing as in the particle-decomposition, critical section versions, $v1a$ and $v1b$.

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 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 Fig. 11, which shows a comparison between the results obtained by the version $v3a$ and a companion version, $v3b$, which implements such a selective reordering. The results of the most efficient particle decomposition implementation, $v2a$, are also shown for reference. The case $n = 1$ and $N_{\rm ppc} = 64$ is considered, for example.

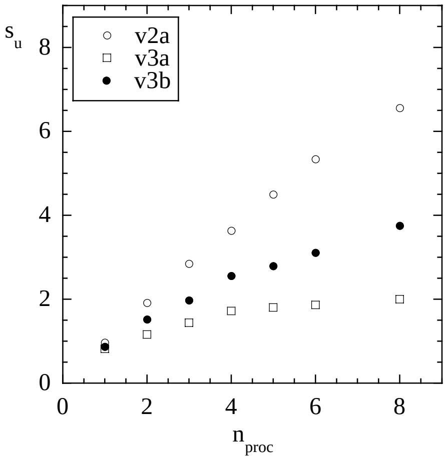

Fig. 11. Comparison between the speed-up obtained, at different number of processors, by the domain decomposition version, $v3a$, and a companion selective reordering version, $v3a$. The results of the most efficient particle decomposition implementation, $v2a$, are also shown for reference. The case $n = 1$ and $N_{\mathrm{ppc}} = 64$ is considered.

5. Concluding remarks

We have presented two work decomposition strategies in the framework of shared memory systems, with application to a case study PIC application. A number of different implementations of them, based on the high-level language OpenMP, have been discussed with regard to time efficiency, memory occupancy, and program restructuring effort.

The computation we dealt with is a particular example of an irregular array reduction. Extensions to OpenMP able to handle irregular array reductions have been proposed by J. Labarta et al. [14], but neither the present version of OpenMP nor the upcoming one (2.0) present such capability.

With the indirect clause of the parallel directive proposed in Ref. [14], the pressure computation can be (very simply) recoded as follows:

```
  p = 0.
!$OMP parallel do private(l,j_r,
j_theta,j_phi)
!$OMP& reduction(+:p)
!$OMP& indirect(j_r,j_theta,j_phi)
  do l = 1,n_part
   j_r = f_r(r(l))
   j_theta = f_theta(theta(l))
   j_phi = f_phi(phi(l))
   p(j_r,j_theta,j_phi) = p(j_r,
   j_theta,j_phi)
  & + h(r(l),...,w(l))
  enddo
!$OMP end parallel do
```

In the automatic parallelizing compilation area the issue of irregular array reductions, and in general indirect and irregular array accesses, has been addressed, and a technique called *inspector-executor* has been proposed. This technique was originally developed by Mehrotra, Koelbel, and Saltz for distributed memory architectures [21], and redesigned and extended for inclusion in HPF compilers by Benkner et al. [3]. A version of this technique, targeted to shared memory systems, is presented in Ref. [14].

Experimental results show, in the above mentioned and other related work, that inspector-executor techniques, although quite generally applicable to a broad range of irregular application, allow for achieving only a moderate efficiency, in the general case. Our results show that an high efficiency can be achieved, only if “ad hoc” solutions are adopted, which take into account the peculiar characteristics of the application (or application class, under consideration (such as domain geometry, nature of particle movement, and so on); in addition, goals such as efficiency, low memory occupancy overhead, and low code restructuring effort are very much interrelated, being matter of trade-off, and, for a given parallel implementation, very much sensitive to the ratio domain size vs. particle size of the application under consideration.

We can conclude that, even if general-purpose compiler techniques will not succeed in attaining, in general, high efficiency over the whole range of the “irregular array reduction operation” applications, “ad hoc” solutions (yet still generally applicable to an application category rather than to a single specific application) can achieve this goal, taking also into account other goals (such as memory occupancy) and requiring a moderate programming/restructuring effort, if high-level languages, such as OpenMP, are adopted.

Appendix: Schematic representation of the versions $v1b$ and $v3a$

In Section 3, we have described a particular implementation (version $v1b$) of the particle decomposition strategy that consists in executing, for each iteration of the particle loop a critical check on the status of the array element that should be updated during the iteration itself. If the check gives a negative result, the element is critically locked. It will be (critically) released as soon as its updating has been completed. In case of positive result (element already locked), the rest of the iteration is skipped and postponed to a further loop execution, limited to the residual portion of not completed iterations. Such implementation can be schematically represented as follows:

```
  integer l_check_part(n_part),
  l_check_part_0(n_part)
  integer i_check_address(n_r,
  n_theta,n_phi)
  l_check_part(:)=0
  l_check_part_0(:)=0
  p = 0.
  n_loop=n_part
  i_loop_exec=0
 1000 continue
  n_race=0
  i_check_address(:,:,:)=0
!$OMP parallel do private(igoto,
l_loop,l,j_r,j_theta,j_phi)

  do l_loop=1,n_loop
  if(i_loop_exec.gt.0)then
    l=l_check_part(l_loop)
    else
     l=l_loop
    endif
    j_r = f_r(r(l))
    j_theta = f_theta(theta(l))
    j_phi = f_phi(phi(l))

    igoto=0
!$OMP critical (address_lock)
    if(i_check_address(j_r,j_theta,
    j_phi).eq.1)then
     igoto=1
    else
     i_check_address(j_r,j_theta,
     j_phi)=1
    endif
!$OMP end critical (address_lock)
    if(igoto.eq.1)then
     go to 10
    else
     igoto=2
    endif

    p(j_r,j_theta,j_phi) = p(j_r,
    j_theta,j_phi)
    & + h(r(l),...,w(l))
    go to 20

 10 continue
!$OMP critical (race_lock)
  n_race=n_race+1
  l_check_part_0(n_race)=l
!$OMP end critical (race_lock)

 20 continue
  if(igoto.eq.2)then
!$OMP critical (address_lock)
    i_check_address(j_r,
    j_theta,j_phi)=0
!$OMP end critical (address_lock)
  endif

  enddo
!$OMP end parallel do

  i_loop_exec=i_loop_exec+1
  n_loop=n_race
  if(n_race.ne.0)then
  l_check_part(1:n_loop)=
  l_check_part_0(1:n_loop)
  go to 1000
  endif
```

The domain decomposition strategy can be implemented, e.g., according to the scheme proposed in Section 4. The domain is decomposed along one of its dimensions. The elementary portion each particle belongs to is identified by a loop over particles. The number of particles belonging to each elementary portion is updated inside a critical section of the loop (in order to avoid race conditions). The relative order of the particle within the population residing in the same portion is defined inside the critical section too. A global portion, composed by several consecutive elementary portions is assigned to each thread. The size of each global portion is chosen in such a way to ensure an approximate load balancing: each thread will manage a number of particles approximately equal to $N_{\rm part}/n_{\rm proc}$. Particles are then reordered according to their thread belonging. Finally, the pressure loop, rewritten as a loop over threads in which a loop over the particle belonging to the thread is nested, can be distributed among threads without concerns for the occurrence of race conditions.

This scheme can be represented as follows:

```
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_threads=omp_get_max_threads()
allocate(j_r_upper(0:n_threads))
p = 0.
n_part_r = 0
```

```
!$OMP parallel do private(l,j_r)
 do l = 1,n_part
  j_r = f_r(r(l))
  j_r_part(l)=j_r
!$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
 n_part_average=float(n_part)/
 float(n_threads)
 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_thread=n_part_portion/
  n_part_average+1
  if(i_thread.gt.n_threads)
  i_thread=n_threads
  j_r_upper(i_thread)=j_r
 enddo
  j_r_upper(0)=0
!$OMP parallel do private(l,j_r,l0)
 do l = 1,n_part
  j_r=j_r_part(l)
  l0=n_part_lower(j_r)+i_r(l)
  l_index(l0)=l
 enddo
!$OMP end parallel do
!$OMP parallel do private(i_thread,
j_r,i,l0,l,j_theta,j_phi)
 do i_thread=1,n_threads
  do j_r=j_r_upper(i_thread-1)+1,
  j_r_upper(i_thread)
   do i=1,n_part_r(j_r)
    l0=n_part_lower(j_r)+i
    l=l_index(l0)
    j_theta = f_theta(theta(l))
    j_phi = f_phi(phi(l))
    p(j_r,j_theta,j_phi) =
    p(j_r,j_theta,j_phi)
    & + h(r(l),...,w(l))
 enddo
!$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] N.G. Azari and S.-Y. Lee, Hybrid partitioning for particle-in-cell simulation on shared memory systems, *Proc. of 11th International Conference on Distributed Computing Systems*, 20–24 May 1991, pp. 526–533.

[3] S. Benkner, K. Sanjari, V. Sipkova and Bob Velkov, Parallelizing Irregular Applications with Vienna HPF+ Compiler VFC, in: *Proc. HPCN 98,* Amsterdam, Netherlands, April 1998.

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

[5] S. Briguglio, B. Di Martino and G. Vlad, Workload Decomposition for Particle Simulation Applications on Hierarchical Distributed-Shared Memory Parallel Systems with integration of HPF and OpenMP, *Proc. of 15th ACM International Conference on Supercomputing* (*ICS"20001*), Sorrento, Italy, June 16–21, 2001.

[6] S. Briguglio, G. Vlad, F. Zonca and C. Kar, Hybrid magnetohydrodynamic-gyrokinetic simulation of toroidal Alfvén modes, *Phys. Plasmas* **2** (1995), 3711–3723.

[7] D.S. Cai and Q.M. Lu, Parallel PIC code using Java on PC cluster, *Proc. of Fourth International Conference on High Performance Computing in the Asia-Pacific Region*, 14–17 May 2000, pp. 495–500.

[8] L. Chen, Theory of magnetohydrodynamic instabilities excited by energetic particles in tokamaks, *Phys. Plasmas* **1** (1994), 1519–1522.

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

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

[11] 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.

[12] 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.

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

[14] J. Labarta, E. Ayguadè, J. Oliver and D. Henty, New OpenMP Directives for Irregular Data Access Loops, Proc. of *2nd European Workshop on OpenMP – EWOMP"2000*, Edinburgh, UK, 14–15 September 2000.

[15] W. Liao, C. Ou and S. Ranka, Dynamic alignment and distribution of irregularly coupled data arrays for scalable parallelization of particle-in-cell problems, *Proc. of IPPS '96, Parallel Processing Symposium*, 15–19 April 1996, pp. 57–61.

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

[17] O. Lubeck and V. Faber, Modeling the Performance of Hypercubes: a case study using the Particle in Cell application, Elsevier, *Parallel Computing* **9** (1988), 37–52.

[18] P. MacNeice, C.M. Mobarry, J. Crawford and T.L. Sterling, Preliminary insights on shared memory PIC code performance on the Convex Exemplar SPP1000, *Proc. of Sixth Symposium on the Frontiers of Massively Parallel Computing,* 27–31 Oct. 1996, pp. 214–222.

[19] 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.

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

[21] J. Saltz, R. Ponnusamy, S. Sharma, B. Moon, Y.-S. Hwang, M. Uysal and R. Das, A Manual for the CHAOS Runtime Library, Technical Report UMIACS TR CS-TR-3437, Univ. of Maryland, March 1995.