# Parallel Plasma Simulation in High Performance Fortran

B. Di Martino*, S. Briguglio, G. Vlad**, and P. Sguazzero***

Associazione Euratom-ENEA sulla Fusione, C.R.E. Frascati,
C.P. 65 - 00044 - Frascati, Rome, Italy.

**Abstract.** Particle-in-cell (PIC) codes are suited for simulating kinetic effects relevant in determining the transport mechanisms in magnetically confined plasmas. A significant improvement of the simulation performance of such codes can be expected from parallelization, e.g., by distributing the particle population among several parallel processors. Parallelization of a hybrid MHD-girokinetic PIC code has been accomplished within the High Performance Fortran (HPF) framework, and tested on the IBM SP2 parallel system, using a "particle decomposition" technique. The adopted technique requires a moderate effort in porting the code in parallel form and results in intrinsic load balancing and modest inter-processor communication. The performance tests obtained confirm the hypothesis of high effectiveness of the strategy, if targeted towards moderately parallel architectures. Optimal use of resources is also discussed with reference to a specific physics problem.

## 1 Introduction

Particle-in-cell (PIC) codes are among the major candidates to yield a satisfactory description of the detail of kinetic effects, such as the resonant wave-particle interaction, relevant in determining the transport mechanisms in magnetically confined plasmas. PIC simulation techniques [1] consist in following the continuous phase-space evolution of a simulation-particle population, while taking into account the mutual interaction between each pair of simulation particles by means of an electromagnetic field, computed only at the points of a discrete spatial grid and then interpolated at each particle position. It can be shown that, to satisfy the physical condition of long-range interactions dominating over the particle-particle collisions, the average number of simulation particles per cell must be much larger than unity.

The high resolution required, both in physical and velocity space, by the need for realistic, accurate and noise-free simulations, has justified a large effort, in the last years, in implementing parallel versions of PIC codes. Aim of

* University of Vienna, Institute for Software Technology and Parallel Systems, Vienna, Austria *and* University of Naples "Federico II", Dipartimento di Informatica e Sistemistica, Naples, Italy

** e-mail: vlad@frascati.enea.it; tel.: +39 - 6 - 9400 5120; fax: +39 - 6 - 9400 5735

*** IBM , via Shanghai 53 , I-00144 Rome, Italy

this paper is to present a particular parallelization strategy for such codes. Different from other approaches, such strategy consists in particle decomposition, rather than domain decomposition. The whole spatial grid is therefore assigned to each processor, which however takes care only of a subset of the particle population. Partial contributions to the total pressure at the grid points, which is required to update the electromagnetic fields, are then communicated among processors and summed together. This strategy implies a *data parallel* execution scheme for the implementing code. The High Performance Fortran [2] (HPF) parallel model (which has already been adopted [3] to parallelize PIC codes using domain-decomposition schemes) is well suited for this strategy, and thus the parallelization of a sequential Fortran 77/Fortran 90 code can be carried out quite straightforwardly by translating it into a HPF program.

## 2 A Particle Decomposition Strategy for the Parallelization of the Hybrid MHD-gyrokinetic Code

A hybrid MHD-gyrokinetic initial-value code has been developed to integrate the coupled sets of reduced MHD equations for the electromagnetic fluctuating fields, and gyrokinetic equations of motion for the collision-free particle population. [4]

It is an approximatively-16,000 lines F77 code, distributed over more than 40 procedures. The core computation performs a typical PIC plasma simulation. At a very coarse grain level, the structure of the computation can be described as follows.

The physical domain is represented by a three-dimensional toroidal grid, for which quasi-cylindrical coordinates are adopted: the minor radius of the torus, $r$, and the poloidal and toroidal angles, $\vartheta$ and $\varphi$, respectively.

Phase-space coordinates and weights for the simulation particles are initially determined in such a way to yield a prescribed (e.g., Maxwellian) distribution function.

At each time step,

1. a field solver computes the fluctuating electromagnetic fields at the $N_{grid}$ grid points of the toroidal domain.
2. Phase-space coordinates and weights of the particles are then evolved (*particle pushing*) in the fluctuating fields interpolated at each particle position.
3. The pressure-tensor components at the grid points are updated, in order to close the MHD equations for the next time step, by distributing the contribution of each particle among the vertices of the cell the particle belongs to.

Several quantities (both scalar and vectorial) – e.g., the number of particles that hit the wall ($r = a$) and are considered lost – are also calculated cumulating particle contributions in time, while performing the main computation.

The most widely explored approach [5, 6] consists in decomposing the physical domain among the different processors, assigning to each of them only the portion of particle population that resides into the subdomain of pertinence. At

each time step, particles that have migrated from a subdomain to another must be withdrawn from the original processor and assigned to the new one.

The most important merit of such a method is represented by the intrinsic scalability of the physical-space resolution with increasing number of processors: keeping the resolution in velocity space fixed, indeed, the maximum number of cells that a single processor can handle is determined by its memory (RAM) resources; adding further processors then allows for adding new cells to the physical domain and hence, once fixed the whole-domain size, increasing the spatial-resolution degree of the simulation. Unfortunately, this positive feature is balanced by at least the two following negative elements. First, parallelization of an existing serial code is not straightforward: tasks as updating each-processor population after particle pushing, for example, are peculiar of the parallel version and make it completely different from the original serial one. Second, serious load-balancing problems can take place: in principle, the whole particle population could be assigned, at a certain time step, to the same processor, causing the advantages of parallelization to vanish or requiring complicate dynamical redefinition of the subdomains.

On the basis of the previous considerations, it can be advantageous, in several practical cases, to adopt an alternative approach to parallelization, consisting in distributing statically the particle population among the processors, while assigning the whole domain to each processor. The respective merits of this particle decomposition are exactly complementary to those of the domain decomposition: the differences between the serial and the parallel version are expected to be very contained and load balancing is automatically ensured (a part from the typically small amount of lost particles); on the opposite side, the physical-space resolution is limited by the RAM resources of the single processor, and increasing the number of processors only allows for increasing the velocity-space resolution.

Anyway, if the number of particles assigned to each processor is much larger than the number of cells (the domain size), the single-node memory occupation of a replicated domain becomes negligible with respect to the memory occupation of a particle decomposition. This condition is equivalent to require that the number of particles per cell must be sensibly larger than the number of processors. Thus if the target architecture presents a moderate number of nodes, not exceeding the average number of particles per cell, the benefits of a domain decomposition on the memory occupation become negligible with respect to a particle decomposition. A large number of particles per cell represents anyhow, as stated above, the condition for the accuracy of the simulation (interactions dominated by long-range effects).

The condition of large number of particles per cell, which is needed for both the accuracy of the simulation and the effectiveness of the particle decomposition strategy, plays a role also in the definition of a strategy for the work distribution. In fact, such a condition means that the work related to the on-grid field-computation phase is negligible with respect to the one needed for the other two phases (particle pushing, and pressure-tensor updating), which involve scans over the particle population. The computational load is thus of the order

$O(N_{part})$, with $N_{part}$ being the number of simulation particles; it is then worth to distribute the second- and third-phase work, and to replicate the first-phase one, consistently with the replication of the domain.

Distributing work for the second and third phases means to distribute the computations that access (read and/or modify) the data structures related to the particles.

The second phase is thus inherently parallel, with no need of communications for non-local accesses (except for the computation of the time-cumulating quantities; see below): pushing of each particle is performed by updating the quantities related to the particle, making use of the previous-step computed values for that particle only (i.e., with no dependence on other-particle quantities), and of the replicated-field values interpolated at the particle position; all these quantities are locally available.

The third phase presents two strictly linked problems: (1) the quantities to be updated (the pressure-tensor components at the grid points), are replicated, and thus must be kept consistent among the processors, and (2) each quantity takes contribution from a number of particles that can reside on different nodes. The solution relies on the associative and distributive properties of the updating laws for the pressure terms, 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, and the partial results are then reduced into global results, which are broadcasted to all the nodes.

In addition, all the quantities that cumulate with particles and time present these same properties; thus we can apply the same strategy.

## 3 Implementation of the Parallelization Strategy in HPF

The strategy depicted in the previous section, both for data and work distribution, can be implemented quite straightforwardly in the framework of the HPF paradigm [2, 7, 8]. The particle decomposition is implemented by applying HPF directives for (cyclic) data distribution to all the data structures related to the particle quantities. The above-described work distribution for the second and third phases, is implemented in HPF by distributing the loop iterations over the particles, with use of HPF `INDEPENDENT` directives. The underlying HPF compiler will distribute those iterations by following the "owner-computes rule" applied to the distributed data. The scheme to handle with the "inhibitors of parallelism" within the loops over the particles (the pressure tensor components, and the quantities that cumulate with particles and time), can be implemented in HPF by restructuring the code in the following way:

- the data structures, [scalars and (multi-dimensional) arrays] which store the values of these quantities, are replaced, within the bodies of the distributed loops, by corresponding data structures augmented by one dimension; their rank must be equal or greater than the number of available processors;

– these data structures are distributed, along the added dimension, over the processors; each of the distributed "pages" will store the partial computations of the quantities, which include the contributions of the particles that are local to each of the processors;
– at each iteration of the loops over the particles, the computed contribution of the corresponding particle to an element of the quantities under consideration, is added to the corresponding element of the distributed page;
– at the end of the iterations, the temporary data structures are reduced along the added and distributed dimension, and the results are assigned to the corresponding original data structures. This is implemented with the use of HPF intrinsic reduction functions such as `SUM`.

As an example, we consider the following skeletonized F90 code excerpt, where each element of the three-dimensional array `press` is updated by the contribution of the particles falling in the closeness of the corresponding grid point.

```
      press = 0.
      do l=1,n_part
        weight_l = weight_part(l)
        r_l      = r_part(l)
        theta_l  = theta_part(l)
        phi_l    = phi_part(l)
        j_r      = f_1(r_l)
        j_theta  = f_2(theta_l)
        j_phi    = f_3(phi_l)
        press(j_r,j_theta,j_phi) = press(j_r,j_theta,j_phi) +
     &       f_4(weight_l,r_l,theta_l,phi_l)
      enddo
```

Here `f_1`, `f_2`, `f_3` and `f_4` are suited nonlinear functions of the particle weight and/or phase-space coordinates. This excerpt represents, very schematically, the update of the pressure tensor components, by trilinear weighting of the contribution of each particle among the vertices of the cells. The computation of the array `press` is anyway representative of any computation of all the quantities that inhibit the parallel execution of the loops over the particles.

The code, restructured according to the above guidelines, looks like the following:

```
      real*8, dimension (0:n_r,n_theta,n_phi,
     &                   number_of_processors()):: press_par
      real*8, dimension (n_part) ::  weight_part, r_part,
     &                               theta_part, phi_part
!HPF$ DISTRIBUTE (CYCLIC) :: weight_part, r_part,
!HPF$&                       theta_part, phi_part
!HPF$ ALIGN WITH weight_part(:) :: press_par(*,*,*,:)
      nprocs = number_of_processors()
```

```
      press_par = 0.
!HPF$ INDEPENDENT, NEW(weight_l,r_l,theta_l,phi_l,j_r,j_theta,
!HPF$&                j_phi,i_proc)
      do l=1,n_part
        weight_l = weight_part(l)
        r_l      = r_part(l)
        theta_l  = theta_part(l)
        phi_l    = phi_part(l)
        j_r      = f_1(r_l)
        j_theta  = f_2(theta_l)
        j_phi    = f_3(phi_l)
        if (mod(l,nprocs) = 0) then i_proc = nprocs
        else i_proc = mod(l,nprocs)
        endif
        press_par(j_r,j_theta,j_phi,i_proc)=
     &           press_par(j_r,j_theta,j_phi,i_proc) +
     &           f_4(weight_l,r_l,theta_l,phi_l)
      enddo
      press(0:n_r,n_theta,n_phi) =
     &      SUM(press_par(0:n_r,n_theta,n_phi,:),dim=4)
```

The pages of the structure `press_par` are distributed to the processors. The code restructuring within the loop is limited to the computation (`if ...`) of the page of `press_par` that is local to the particle considered in that iteration, and the update of the proper element belonging to that page.

The reduction of the (distributed) pages of `press_par` in `press` (replicated) is very easily performed with the use of the HPF intrinsic 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 intrinsics procedures for distributed parameters, these communications are performed as vectorized and collective minimum-cost communications.

The computation for the selection of the page of `press_par` local to each particle, even if not complex, is anyway a non-linear function of the loop index (`l`). This, together with the presence of indirect references (through the elements of an array) to the elements of each page of `press_par`, could represent a problem if the target HPF compiler is able to perform data-dependence analysis for array elements, and actually performs an unrequested check about the assertion of independence of the loop iterations, provided by the user with the `INDEPENDENT` directive. In this case, such a compiler would not distribute the loop iterations, because the nonlinearity and the indirect character of the indexing expressions prevent any state-of-the art dependence test from proving the actual independence of the loop iterations, and would make worst-case assumption of dependence. This problem can be anyway quite easily bypassed with the help of the HPF extrinsics `HPF_LOCAL`.

We use the extrinsic mechanism to achieve the same effect as the INDEPENDENT directive, i.e., the distribution of the execution of loop iterations over the processors, when this latter cannot be enabled due to the complex and/or indirect references to distributed arrays within the yet independent loop iterations. To this purpose, the loops over the particles become the bodies of the extrinsics, as exemplified, for the example under examination, in the following:

```
      INTERFACE
      EXTRINSIC(HPF_LOCAL)
     &subroutine extr_press(weight_part,r_part,theta_part,
     &                      phi_part, press_par)
      real*8, dimension(:), intent(in) :: weight_part,r_part,
     &                                    theta_part,phi_part
      real*8, dimension(:,:,:,:), intent(out) :: press_par
!HPF$ DISTRIBUTE (CYCLIC) :: weight_part,r_part,
!HPF$&                       theta_part,phi_part
!HPF$ ALIGN WITH weight_part(:) :: press_par(*,*,*,:)
      end subroutine extr_press
      END INTERFACE
      ...
      call extr_press(weight_part,r_part,theta_part,phi_part,
     &                press_par)
      press(0:n_r,n_theta,n_phi) =
     &      SUM(press_par(0:n_r,n_theta,n_phi,:),dim=4)
      ...
      EXTRINSIC(HPF_LOCAL)
     &subroutine extr_press(weight_part,r_part,theta_part,
     &                      phi_part, press_par)
      real*8, dimension(:), intent(in) :: weight_part,r_part,
     &                                    theta_part,phi_part
      real*8, dimension(:,:,:,:), intent(out) :: press_par
      press_par = 0.
      do l=1, UBOUND(weight_part,dim=1)
        weight_l = weight_part(l)
        r_l      = r_part(l)
        theta_l  = theta_part(l)
        phi_l    = phi_part(l)
        j_r      = f_1(r_l)
        j_theta  = f_2(theta_l)
        j_phi    = f_3(phi_l)
        press_par(j_r,j_theta,j_phi,1)=
     &           press_par(j_r,j_theta,j_phi,1) +
     &           f_4(weight_l,r_l,theta_l,phi_l)
      enddo
      end subroutine extr_press
```

Here each local procedure executing on a given processor sees the portion of the arrays related to the particles (`weight_part`, `r_part`, `theta_part` and `p_part`), and the page of the "partial results" array `press_par` assigned to that processor. It executes only the set of loop iterations which access the particles local to that processor (`L=1,UBOUND(weight_part,dim=1)`), and updates the page of `press_par` (`press_par(j_r,j_theta,j_phi,1)`) assigned to it. At the end of all the executions of the local extrinsics, all the partial updates of the components of `press` are collected in the global-HPF-index-space `press_par`. `press_par` is then reduced to `press` with the use of the `SUM` intrinsic function, as seen before.

## 4 Results

In this section, we present the results obtained from the described HPF parallel version of the Hybrid MHD-Gyrokinetic Code (making use of the extrinsic mechanism) compiled with the IBM *xlhpf* compiler [9] (an optimized native compiler for IBM SP systems) under the *-O3* and *-qhot* options, and running it on an IBM Scalable Power system (SP2) at the ENEA Research Centre of Frascati (Rome).

The results presented here refer to executions on a domain grid with `n_r=32` points in the radial direction, `n_theta=16` points in the poloidal direction, and `n_phi=16` points in the toroidal one; we performed several executions by varying both the number of processors `nprocs` (ranging from `nprocs=2` to `nprocs=8`) and the average number of particle per cell `NPPC` $\equiv N_{part}/N_{grid}$ (ranging from `NPPC=2` to `NPPC=1024`).

In Fig. 1 the efficiency ($\equiv$ speed-up/number-of-processors) values are plotted against `NPPC/nprocs`. The speed-up is defined as the ratio between the sequential-simulation cpu time and the corresponding parallel-simulation one. Sequential simulations have been performed after compiling the source code with the *xlhpf* compiler under the *-qnohpf* option (the use of the *-qstrict* option gives no appreciable differences). We can observe that the efficiency tends to saturate at larger values of `NPPC` the greater the number of processors is. This feature is due to the fact that, increasing the number of processors, the average number of particles per cell assigned to each processor decreases, and the importance of the grid-related calculation tends to overcome the particle-related one. In fact, when plotted against `NPPC/nprocs`, the efficiency values approximatively fall on a "universal" curve.

Note that for the largest simulations, superlinear results are obtained, which can possibly be traced back to memory and/or cache effects and compiler options.

## 5 Validity of the Particle-Decomposition Approach

Figure 2 reports the findings of Hybrid-MHD-Gyrokinetic-Code simulations corresponding to different values of `NPPC`. In particular, the growth rate $\gamma$ of an Energetic Particle Mode [10], normalized to the inverse characteristic time of

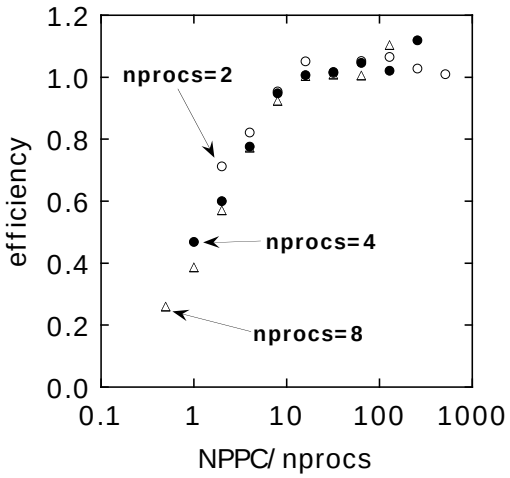


**Fig. 1.** Scaling of the efficiency with the average number of particles per cell per processor `NPPC/nprocs`.

Alfvén-mode propagation, is shown. Such a quantity corresponds to the rate at which a small electromagnetic perturbation to a given equilibrium grows exponentially in time. In the present case an equilibrium characterized by a ratio between the kinetic pressure of a destabilizing energetic population and the equilibrium magnetic pressure equal to 0.03 has been considered. It can be observed that, by increasing `NPPC` (and thus the resolution in the velocity space), the growth rate converges to a limiting value. Defining conventionally the accuracy of the simulation as the relative deviation from this value, the requirement of a certain accuracy fixes the minimum needed resolution, i.e., the minimum `NPPC` value (`NPPC_min`). Note that the determination of such a value may depend strongly on the different physical scenarios considered.

Once fixed `NPPC` $\approx$ `NPPC_min`, we see from Fig. 1 that the requirement of high efficiency ($\approx 1$) puts an upper bound on the number of processors that can be used in a simulation: `nprocs` $\leq$ `nprocs_max` $\approx$ `NPPC_min`$/10$. For the particular case considered in Fig.2, an accuracy level of 5% in the determination of the growth rate of the Alfvén mode corresponds to `NPPC_min` $\approx 120$ and `nprocs_max` $\approx 12$. Different conclusions, corresponding to a large number of processors can be motivated by the need for a higher value of the speed-up (which monotonically increases with `nprocs`) and/or larger global-memory resources, in spite of a lower efficiency value. With reference to the latter point, we indeed observe that the optimal-accuracy-end-efficiency choice corresponds to a number of particles per processor given by $N_{grid} \times$ `NPPC_min/nprocs_max` $\approx 10 N_{grid}$; thus, requiring higher spatial resolution (i.e., higher values of $N_{grid}$) would eventually exceed the single-node memory resources. Such a limitation can be, of course, overcome by using a higher number of processors.

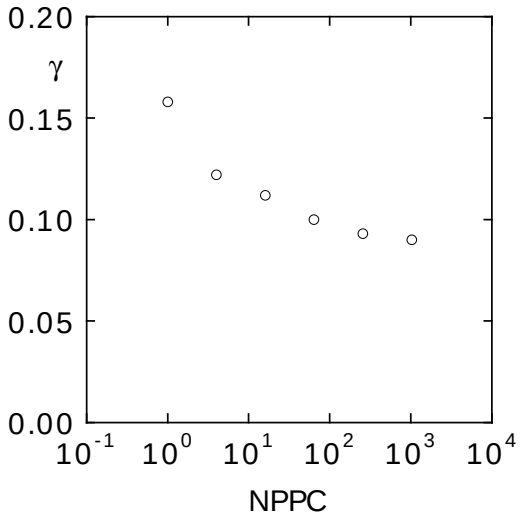


**Fig. 2.** Growth rate of an Energetic Particle Mode normalized to the inverse characteristic time of Alfvén-mode propagation versus the average number of particles per cell `NPPC`.

## References


1. C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation* (McGraw-Hill, New York, 1985).
2. High Performance Fortran Forum, "High Performance Fortran Language Specification", Version 1.1, 1994 (copying is by permission of Rice University).
3. E. Akarsu et al., "Particle-in-Cell Simulation Codes in High Performance Fortran", Proc. of *SuperComputing '96*, Nov. 1996.
4. S. Briguglio, G. Vlad, F. Zonca, and C. Kar, "Hybrid magnetohydrodynamic-gyrokinetic simulation of toroidal Alfvén modes", Phys. Plasmas **2**, 3711 (1995).
5. P. C. Liewer and V. K. Decyk, "A General Concurrent Algorithm for Plasma Particle-in-Cell Codes", J. Computational Phys. 85, 302 (1989).
6. G.C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, *Solving Problems on Concurrent Processors*, Prentice Hall, Englewood Cliffs, NJ, 1988.
7. High Performance Fortran Forum, "High Performance Fortran Language Specification", *Scientific Programming*, **2**(1-2), 1-170 (1993).
8. H. Richardson, "High Performance Fortran: history, overview and current developments", Tech. Rep. TMC-261, Thinking Machines Corporation, April 1996.
9. M. Gupta et al., "An HPF Compiler for the IBM SP2", Proc. of *SuperComputing '95*, Nov. 1995.
10. Liu Chen, "Theory of magnetohydrodynamic instabilities excited by energetic particles in tokamaks", Phys. Plasmas **1**, 1519 (1994).