EXPERIENCES ON PARALLELIZING COMPILATION FOR DEVELOPMENT AND PORTING OF LARGE SCALE APPLICATIONS ON DISTRIBUTED MEMORY PARALLEL SYSTEMS*

B. DI MARTINO†, S. BRIGUGLIO , M. CELINO‡, G. FOGACCIA , G. VLAD , V. ROSATO‡, AND M. BRISCOLINI§

Abstract.
High level environment, such as High Performance Fortran (HPF), supporting the development of parallel applications and porting of legacy codes to parallel architectures have not yet gained a broad acceptance and diffusion. Common objections claim difficulty of performance tuning, limitation of its application to regular, data parallel computations, and lack of robustness of parallelizing HPF compilers in handling large sized codes.

We have adopted the HPF approach in porting three different applications, performing plasma and molecular dynamics simulation, developed at the Italian National Agency for New Technology, Energy and the Environment (ENEA). We report in this paper our experiences gained during this effort, providing a case study for testing the suitability of the HPF approach to achieve the target of an easy and effective parallelization (or parallel development) and maintenance of real, large sized scientific applications.

Key words. Parallel Simulation, High Level Parallel Programming Environments, High Performance Fortran.

1. Introduction. The increased availability of parallel and distributed architectures has made urgent the need of effective and easy-to-use instruments for programming such systems.

It is generally accepted that the *message passing* 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 affects the portability of the resulting program.

An alternative approach to the programming of parallel architectures is based on the automatic transformation of sequential code, possibly augmented with parallelization directives, into explicitly parallel code, by means of *parallelizing compilers*. This method is not only useful for supporting the reuse of "dusty decks", but has also important advantages when developing an application from scratch:

1. easy design, development, verification and debugging of sequential code,
2. existence of several environments for the support of sequential programming, and
3. portability – the compiler performs the mapping from the sequential code to a code suited to a particular parallel target architecture, including machine-specific optimizations.

High Performance Fortran (HPF) is a programming language standard that adopts this approach. With HPF, the programmer is only responsible for defining, by means of (high level) directives, how the data are to be distributed to the processors. The HPF compiler is responsible for producing code to distribute the array elements on the available processors. The message-passing code produced by the compiler, ac-

*Associazione Euratom-ENEA sulla Fusione, C.R. Frascati, CP 65, 00044 Frascati, Roma (Italy)
†Second University of Naples, Dip. di Ingegneria dell' Informazione (Italy)
‡ENEA - HPCN Project, C.R. Casaccia, CP 2400, 00100 Roma AD (Italy)
§IBM Italia, Roma (Italy)

cording to the Single Program Multiple Data (SPMD) [1] execution model [1] instructs each processor to update the subset of the array elements which are stored in the local memory, possibly by getting values, corresponding to non-local accesses, owned by the other processors via communication primitives. HPF provides a standardized set of extensions to Fortran 90 (F90) and Fortran directives to enable the execution of a sequential program on (distributed memory) parallel computer systems. Compiler directives such as `PROCESSORS`, `ALIGN`, `DISTRIBUTE` and `TEMPLATE` are introduced to control the alignment and distribution of array elements on the (abstract) processors; directives such as `INDEPENDENT` and language constructs such as `FORALL` are introduced to express (data) parallelism. An extended set of intrinsic functions and a standard library provide parallel functionality at a high level of abstraction. `EXTRINSIC` procedures standardize the interface with other languages and sequential or parallel execution schemes, and there are directives to address sequence and storage association issues. See Refs. [2, 3, 4] for an extended overview of the language and its features.

The HPF approach eliminates the error-prone task of explicitly distributing the array data elements and explicitly programming how, where and when to pass messages between processors. The writing of efficient HPF programs is not, however, a trivial task. In general, the high-level nature of the language prevents the user from clearly understanding the behaviour of the parallel code being produced by the compiler, and this fact can frequently lead to inefficient codes. In particular, we outline in the following the three main issues that are commonly associated with the HPF approach.

First, the selection of an optimal data distribution layout is left to the programmer [2]. It is a complex task, on which the performance of the parallel code depends critically: the programmer has to find out (*i*) the portions of the code (*phases*) where the data distribution has to remain unchanged; (*ii*) a suitable redistributions among phases; (*iii*) the *alignments* among the array dimensions, for each phase, such that the cost of communications due to alignment conflicts is minimal; (*iv*) the best distributions for array dimensions, for each phase, such that data locality is maximized, thus minimizing the communication overhead.

Second, the adoption of a fixed parallel execution model, even a very flexible and general one like the SPMD model may prevent the selection of a parallelization strategy that is best suited to the characteristics of the code to be parallelized. For instance, in the context of irregular problems, execution models alternative to SPMD have been pointed out, and it is a common belief that the HPF approach cannot handle efficiently irregular computations.

Finally, the parallelization strategy performed by an HPF compiler needs a very sophisticated program analysis and code transformation technology. Despite the efforts and the relevant number of developed products from academia and industry, the state of the art in parallelizing compiler technology has not yet reached a maturity; thus a number of algorithms/codes which could in principle be handled efficiently within the HPF framework, produce instead unsatisfactory results if careful manual code restructuring is not applied before the HPF compiler action.

[1] Roughly: the same program, not necessarily the same instruction stream, is executed by each processor, each operating on a part of the data.

[2] However, this task has to be accomplished, of course, with the explicit message-passing coding as well, and HPF has a clear advantage over message-passing coding, because it allows for testing the efficiency attained with different data distributions by simply changing the directives in the program, instead of completely recoding a message-passing program.

In practice, the suitability of HPF for parallelization and porting of real, large-sized applications cannot be stated in a general way, but it has to be tested in specific, although representative, cases. Here we report the results obtained by adopting the HPF approach on three different physical problems/codes, developed at the Italian National Agency for New Technology, Energy and the Environment (ENEA), by the Enea researchers authoring this work.

The first two codes have been developed in the framework of controlled nuclear fusion research. Both the abundance of the combustible (deuterium and tritium – easy to extract from the sea water the former, to produce by *in situ* nuclear reactions the latter) and the intrinsic stability of the projected reactor make nuclear fusion one of the most promising energy source for the future. The development of economically convenient reactors then represents an appealing goal, which could, in the next decades, attract the interest of important sectors of the high-technology industry. At present, however, fusion research is still facing the target of demonstrating the scientific feasibility of a reactor prototype. One of the most relevant issues of such a research is related to the full comprehension of the turbulent phenomena that affect the magnetically confined deuterium-tritium plasmas, and, in particular, of the large energy and particle losses, experimentally observed and able to prevent the achieving of the reactor conditions. The difficulty in performing flexible and complete experimental campaigns and in treating the theoretical models by analytical methods induces a systematic recourse to computer simulations. Such simulations come out to be very heavy, because of the large ratio between the spatial and temporal size of the experiments and the corresponding scales that characterize turbulent fluctuations. Therefore, parallelization of existing numerical codes for the investigation of turbulent plasmas comes out to be a very important step in the qualitative progress of the overall research field.

One of the two plasma simulation codes considered here [5] adopts the *particle-in-cell* (PIC) technique, or, in a more advanced version, the *finite-size-particle* (FSP) one. The other one [6] addresses the same physical problem using the Lattice Boltzmann Equation (LBE) method, although it is, at present, in a pioneer stage and thus relies on much simpler models.

The third code simulates, using the Tight-Binding Molecular Dynamics (TBMD) technique, the atomistic evolution of semiconductor materials in several thermodynamic environments [7, 8]. TBMD has recently emerged as a useful method for studying the structural, dynamical and electronic properties of covalent materials. The method incorporates electronic structure calculation into molecular dynamics through an empirical tight-binding Hamiltonian and bridges the gap between *ab-initio* (fully quantum mechanical) [9] molecular dynamics and simulations using empirical classical potentials [10]. For these reasons the TBMD approach has gained great interest in the industrial applications: it can provide complementary informations to experimental measurements providing atomic scale information to the measured physical quantities. Furthermore the availability of parallel computers allows for studies of large systems on large time scales. For example this gives the possibility to study new materials for micro-electronics industry. Silicon and carbon, in their crystalline or amorphous structure, and their compounds are some of the materials that are crucial to develop new electronic devices [11, 12]. Several phenomena involving these materials can be reproduced at the atomic level: surface growth, fracture dynamics, melting, amorphization, etc.

These codes present different characteristics, in terms of code size, computation-

al structure, data size. They represent a balanced mix of applications for testing the suitability of the HPF language to achieve the target of an easy and effective parallelization (or parallel development) of real scientific packages, computation and memory demanding.

The developed HPF codes have been tested on a IBM SP parallel system, with 16 Power2 RISC processors, each with clock frequency of 160 MHz and equipped with 512 MB RAM and 9.1GB HD. The HPF codes have been compiled by the IBM *xlhpf* compiler [13] (an optimized native compiler for IBM SP systems).

In the next sections we describe in details the main physical aspects, the parallelization strategy adopted, the main issues arisen and the solutions found, for each of the applications considered. In the conclusions we summarize the main experiences gained with this porting effort.

2. Plasma particle simulation codes. Particle simulation codes [14] 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 (*particle pushing*) 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 *particle-in-cell* (PIC) approach. It consists in (*i*) computing the electromagnetic fields only at the points of a discrete spatial grid, then (*ii*) interpolating them at the (continuous) particle positions in order to perform particle pushing, and (*iii*) collecting particle contribution to pressure at the grid points to close the field equations.

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 plasma condition (corresponding to long range particle interactions dominating over the short range ones) results in a much more relaxed requirement. Indeed, it comes out to be satisfied if $n_0 L_c^3 \gg 1$, with n_0 being the density of simulation particles, even if the usual condition, $n_0 \lambda_D^3 \gg 1$, is not.

The condition $n_0 L_c^3 \gg 1$ can be written as $N_{part}/N_{cell} \gg 1$, where N_{part} is the number of simulation particles and N_{cell} that of grid cells. 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 computational nodes.

2.1. Parallelization strategy. The standard approach to the parallelization of PIC codes is based on the *domain decomposition* [15]: different portions of the physical domain are assigned to different processors, together with the particles that reside on them. In this way, both the number of operations per time step and the memory space required to each computational node are reduced, in an average sense,

by a factor approximately equal to the number of processors, n_{proc} (here we restrict our analysis to the case of distributed-memory architectures).

The main advantage of the scheme consists in the almost linear scaling of the attainable physical-space resolution (more precisely, the maximum number of spatial cells) with the number of processors. Inter-processor communication are however necessary to transfer particle data from one computational node to another when some particle moves from one portion of the grid to a different one; this particle migration can result in severe load-balancing problems and, eventually, in a very inefficient parallelization. In order to avoid such problems, a dynamical redistribution of grid and particle quantities is required, which makes the parallel implementation of a PIC code very complicate.

An alternative approach to the parallelization of PIC codes is based on *particle decomposition* [16]. It consists in statically distributing the particle population among processors, while replicating the data relative to grid quantities. Before updating the electromagnetic fields, at each time step, partial contributions to particle pressure coming from different portions of the population must be summed together. It is apparent that load balancing is automatically enforced, because no particle has to be transferred from one processor to another. As a consequence, the implementation of parallelization in HPF is, in principle, relatively straightforward.

In particular, HPF directives for (cyclic) data distribution can be applied to all the data structures related to the particle quantities, and the HPF `INDEPENDENT` directive can be used to distribute the loop iterations over the particles, related to the particle-pushing and pressure-updating phases. The underlying HPF compiler will distribute those iterations by following the *owner computes* rule applied to the distributed data.

The updating of particle pressure presents two strictly linked problems: (*i*) these quantities are replicated, and thus must be kept consistent among the processors; (*ii*) each quantity 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. In addition, some other quantities, which cumulate with particles and time, present these same properties; we can apply the same strategy to the task of computing such quantities. The scheme to handle with these "inhibitors of parallelism" within the loops over the particles, can be implemented in HPF by restructuring the code in the following way:

1. the data structures – scalars and (multi-dimensional) arrays – that 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 to (or greater than) the number of available processors;

2. 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 processor;

3. at each iteration of the loops over the particles, the contribution of the corresponding particle to an element of the quantities under consideration is added to the appropriate element of the distributed page;

4. 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 by using HPF intrinsic reduction functions such as `SUM`.

As an example, we consider the following skeletonized F90 code excerpt, maintaining the main features of the corresponding portion of the specific PIC code considered, namely the Hybrid MHD-Gyrokinetic particle simulation Code [5] (HMGC), consisting of approximatively 16,000 F77 lines distributed over more than 40 procedures. Each element of the three-dimensional array `pressure` is updated by the contribution of the particles falling in the neighbours of the corresponding grid point.

```
       real*8, dimension (n_r,n_theta,n_phi):: pressure
       real*8, dimension (n_part) ::  weight,r,theta,phi
       pressure = 0.
       do l=1,n_part
         weight_l = weight(l)
         r_l      = r(l)
         theta_l  = theta(l)
         phi_l    = phi(l)
         j_r      = f_1(r_l)
         j_theta  = f_2(theta_l)
         j_phi    = f_3(phi_l)
         pressure(j_r,j_theta,j_phi) =
      &       pressure(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 particle pressure on the $n_r \times n_\vartheta \times n_\varphi$ grid points, by trilinear weighting of the contribution of each particle among the vertices of the cell. The computation of the array `pressure` 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 (n_r,n_theta,n_phi):: pressure
       real*8, dimension (n_r,n_theta,n_phi,
      &                   number_of_processors()):: pressure_par
       real*8, dimension (n_part) ::  weight,r,theta,phi
!HPF$ DISTRIBUTE (CYCLIC) :: weight,r,theta,phi
!HPF$ ALIGN WITH weight(:) :: pressure_par(*,*,*,:)

       n_proc = number_of_processors()
       pressure_par = 0.

!HPF$ INDEPENDENT, NEW(l,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(l)
         r_l      = r(l)
         theta_l  = theta(l)
         phi_l    = phi(l)
```

```
      j_r      = f_1(r_l)
      j_theta  = f_2(theta_l)
      j_phi    = f_3(phi_l)
      if (mod(l,n_proc).eq.0) then
i_proc = n_proc
      else
i_proc = mod(l,n_proc)
      endif
      pressure_par(j_r,j_theta,j_phi,i_proc)=
   &           pressure_par(j_r,j_theta,j_phi,i_proc) +
   &           f_4(weight_l,r_l,theta_l,phi_l)
    enddo
    pressure(:,:,:) = SUM(pressure_par(:,:,:,:),dim=4)
```

The pages of the structure `pressure_par` are distributed to the processors. The code restructuring within the loop is limited to the computation (`if ...`) of the page of `pressure_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 `pressure_par` in `pressure` (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 intrinsic procedures for distributed parameters, these communications are performed as vectorized and collective minimum-cost communications.

The computation for the selection of the page of `pressure_par` local to each particle, even though 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 `pressure_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 extrinsic procedures `HPF_LOCAL`.

HPF programs may call non-HPF subprograms as *extrinsic procedures* [2]. 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 processor. HPF 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 an HPF compiler should 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 physical processor sees an array containing just those elements of the global array that are mapped to that physical processor. A call to an extrinsic procedure results in

a separate invocation of a local procedure on each processor. The execution of an extrinsic procedure consists of the concurrent execution of a local procedure on each executing processor. 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 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 effectively independent loop iterations. To this purpose, the loops over the particles become the bodies of the extrinsic procedures, as exemplified, for the example under examination, in the following:

```
       INTERFACE
       EXTRINSIC(HPF_LOCAL)
      &subroutine extr_pressure(weight,r,theta,phi,pressure_par)
       real*8, dimension(:), intent(in) :: weight,r,theta,phi
       real*8, dimension(:,:,:,:), intent(out) :: pressure_par
!HPF$ DISTRIBUTE (CYCLIC) :: weight,r,theta,phi
!HPF$ ALIGN WITH weight(:) :: pressure_par(*,*,*,:)
       end subroutine extr_pressure
       END INTERFACE
       ...
       call extr_pressure(weight,r,theta,phi,pressure_par)
       pressure(:,:,:) = SUM(pressure_par(:,:,:,:),dim=4)
       ...
       EXTRINSIC(HPF_LOCAL)
      &subroutine extr_pressure(weight,r,theta,phi,pressure_par)
       real*8, dimension(:), intent(in) :: weight,r,theta,phi
       real*8, dimension(:,:,:,:), intent(out) :: pressure_par

       pressure_par = 0.
       do l=1, UBOUND(weight,dim=1)
         weight_l = weight(l)
         r_l      = r(l)
         theta_l  = theta(l)
         phi_l    = phi(l)
         j_r      = f_1(r_l)
         j_theta  = f_2(theta_l)
         j_phi    = f_3(phi_l)
         pressure_par(j_r,j_theta,j_phi,1)=
      &           pressure_par(j_r,j_theta,j_phi,1) +
      &           f_4(weight_l,r_l,theta_l,phi_l)
       enddo
       end subroutine extr_pressure
```

Here each local procedure executing on a given processor sees the portion of the arrays related to the particles (`weight`, `r`, `theta` and `phi`), and the page of the "partial results" array `pressure_par` assigned to that processor. It executes only the set of loop iterations that access the particles local to the processor (`l=1,UBOUND(weight, dim=1)`), and updates the page of `pressure_par` assigned to it. At the end of the execution of the local extrinsic procedure, all the partial updates of the components

of `pressure_par` are collected in the global-HPF-index-space `pressure_par`. Then, `pressure_par` is reduced to `pressure` with the use of the `SUM` intrinsic function, as seen before.

The efficiency η, defined as the speed-up factor divided by the number of processors, is plotted, in Fig. 2.1, versus n_{proc}/N_{ppc} (with $N_{ppc} \equiv N_{part}/N_{cell}$ being the average number of particle per cell), for typical parallel PIC simulations obtained by HMGC. Different cases, corresponding to increasing values of the typical mode number n retained in the (linear) simulation, are considered; note that the spatial-resolution level (i.e., the number of cells N_{cell}) scales as n^3. The efficiency, which is

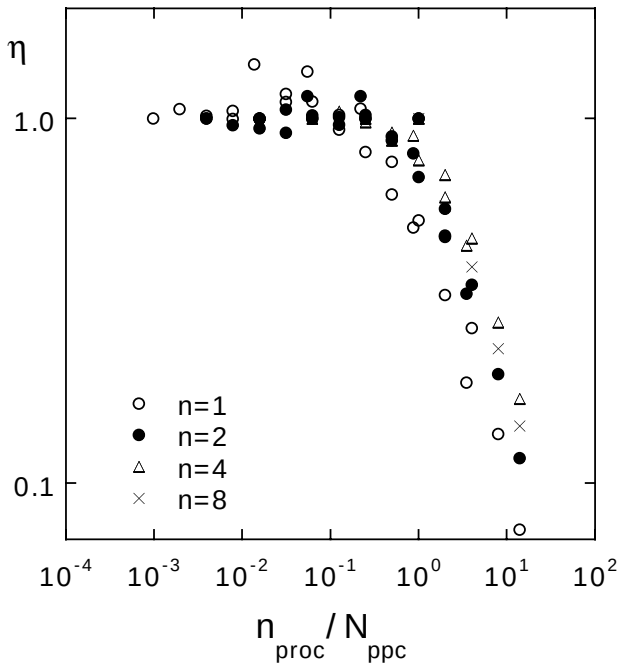

FIG. 2.1. *Efficiency η, defined as the speed-up factor divided by the number of processors, versus n_{proc}/N_{ppc}, for parallel PIC simulations corresponding to different values of the typical mode number n retained.*

at its ideal value ($\eta \approx 1$) at low values of n_{proc}/N_{ppc}, fast decreases above a certain threshold[3].

The reason for the efficiency decrease in the framework of *particle decomposition* parallelization of PIC codes is that grid calculations do not take advantage from such a parallelization, because each processor has to handle the whole spatial domain. Indeed, particle decomposition comes out to be very efficient as far as the computational load related to the particle population dominates, for each processor, the one related to the grid. This condition corresponds to require a large average number of particles per cell on each processor, and it is surely satisfied for moderately parallel machines. However, for $n_{proc} \gtrsim N_{ppc}$, it breaks down, and the computational effort of each processor is mainly devoted to the replicated calculation of grid quantities.

[3]Note that for the largest N_{ppc} simulations, superlinear results are obtained, which can be possibly traced back to memory and/or cache effects and compiler options.

Moreover, even neglecting efficiency problems, the implementation of the code on a very large number of processors does not allow one to reach high spatial-resolution levels, due to the offset in the memory requests represented by the grid data: the highest spatial resolution that can be reached in a simulation without requiring *memory paging* depends on the Random Access Memory resources of the single computational node, whereas increasing the number of processors only allows to increase the average number of particles per cell and, hence, the velocity-space resolution.

2.2. Gridless simulation. The existence, in the framework of the more appealing *particle decomposition* approach, of bottle-necks in efficiency and performance associated to grid quantities induces one to by-pass the introduction of a spatial grid, so resorting to the gridless FSP simulation [17]. Such a method can be applied in the (quite general) case of periodic spatial domains, which allows one to solve the equations for the fields in Fourier space. It consists in computing the electromagnetic fields by transforming them from the Fourier space back to the real one directly at the particle position; correspondingly, particle contribution to pressure are first Fourier transformed and then summed together. No intermediate grid calculation is then required. The benefic smoothing of the unphysically strong short-range interactions (enforced, in the PIC case, by the finite grid-point spacing) is replaced by an analogous smoothing obtained by replacing the singular particle contribution to the density, $\delta(\mathbf{x}-\mathbf{x}_l)$ (with $\mathbf{x}_l$ being the l-th particle position) by a shaped regular (e.g., Gaussian) one, $S(\mathbf{x}-\mathbf{x}_l)$. The FSP method is expected to yield the same qualitative findings of the PIC one if $L_c \approx L_s$, where L_s is the width of the function S. Such a width can be interpreted as the typical size of the numerical particles (or, more appropriately, *charge clouds*).

In the framework of serial simulations, the FSP method is more expensive than the PIC one, without presenting any significant advantage in terms of memory requests. The fields have indeed to be transformed back and forth, from the Fourier space to the real one, for each particle, rather than for each grid point, and the memory required to store the Fourier harmonics of such fields is of the same order of magnitude as that needed to store field values on the grid. Moreover, in the FSP case, it is not possible to use the Fast Fourier Transform (FFT).

However, very often the interest is focused, in practice, on simplified simulations in which, both because of the relevance of linear studies and the weak coupling between modes in nonlinear ones, only few field harmonics are evolved, in spite of the high mode numbers and the corresponding high spatial resolution considered. In such few-harmonic limit, the FFT algorithm is not efficient, even in the PIC frame. Moreover, as far as parallel simulations are considered, the calculations related to the particle Fourier transforms, in the FSP case, are distributed among the processors, and the bottle-neck related to the replicated field calculations is drastically reduced. In such situations the FSP approach can become more convenient than the PIC one.

In the FSP version of HMGC, the Fourier analysis is performed along the two periodic coordinates ϑ and φ (the poloidal and toroidal angles, respectively, used to describe the plasma confined in a toroidal chamber). The spatial grid along the radial (non periodic) coordinate r is instead maintained.

The HPF parallelization strategy adopted for the FSP simulation is equivalent to the strategy adopted for the PIC one, where the Fourier harmonics of the particle pressure take the place of the pressure itself. The HPF code excerpt performing the update of the n_h Fourier harmonics of the particle pressure, `pressure(1:n_h,1:n_r)`, with the introduction of the `HPF_LOCAL` extrinsic procedure, takes the form:

```
       INTERFACE
       EXTRINSIC(HPF_LOCAL)
      &subroutine extr_pressure(weight,r,pressure_par)
       real*8, dimension(:), intent(in) :: weight,r
       real*8, dimension(:,:,:), intent(out) :: pressure_par
!HPF$ DISTRIBUTE (CYCLIC) :: weight,r
!HPF$ ALIGN WITH weight(:) :: pressure_par(*,*,:)
       end subroutine extr_pressure
       END INTERFACE
       ...
       call extr_pressure(weight,r,pressure_par)
       pressure(:,:) = SUM(pressure_par(:,:,:),dim=3)
       ...
       EXTRINSIC(HPF_LOCAL)
      &subroutine extr_pressure(weight,r,pressure_par)
       real*8, dimension(:), intent(in) :: weight,r
       real*8, dimension(:,:,:), intent(out) :: pressure_par

       pressure_par = 0.
       do l=1, UBOUND(weight,dim=1)
         weight_l = weight(l)
         r_l      = r(l)
         j_r      = f_1(r_l)
         do i=1,n_h
           pressure_par(i,j_r,1) =
      &      pressure_par(i,j_r,1) + f_5(weight_l,r_l,i)
         enddo
       enddo
       end subroutine extr_pressure
```

where `f_5` is a suited nonlinear function of the particle weight and radial coordinate.

Figure 2.2 shows the efficiency values versus n_{proc}/N_{ppc} for parallel simulations performed by the FSP version of HMGC. It can be seen that, according to what we expect from the reduction of the distributed-computation bottle-necks, the particle-decomposition parallelization of FSP codes is very efficient even for high values of n_{proc}/N_{ppc} and is then suited, in principle, for implementation on massively parallel architectures.

3. Plasma LBE Simulation Code. The Lattice Boltzmann Equation method is a particularly promising numerical method to perform fluid-dynamics high-resolution simulations [18]. The macroscopic dynamics is simulated starting from a *fictitious* system of particle populations $\{N_j(\vec{r},t) \in [0,1]\Re, j = 1,\ldots,b\}$, moving on a discrete lattice with discrete velocities $\{\vec{c}_j, j = 1,\ldots,b\}$, with $\vec{r}$ being the position vector, t the time variable and b the number of neighbour sites.

Such particle system is not related to the real microscopic system; the numerical LBE method is able to reproduce only the macroscopic dynamics of the physical system, differing from the plasma particle-simulation approach, discussed in the previous section, which, instead, is able to reproduce also kinetic effects such as wave-particle resonances.

In the LBE method, simulation particles that arrive at the same site undergo a collision, with the collision operator chosen in such a way to locally conserve the

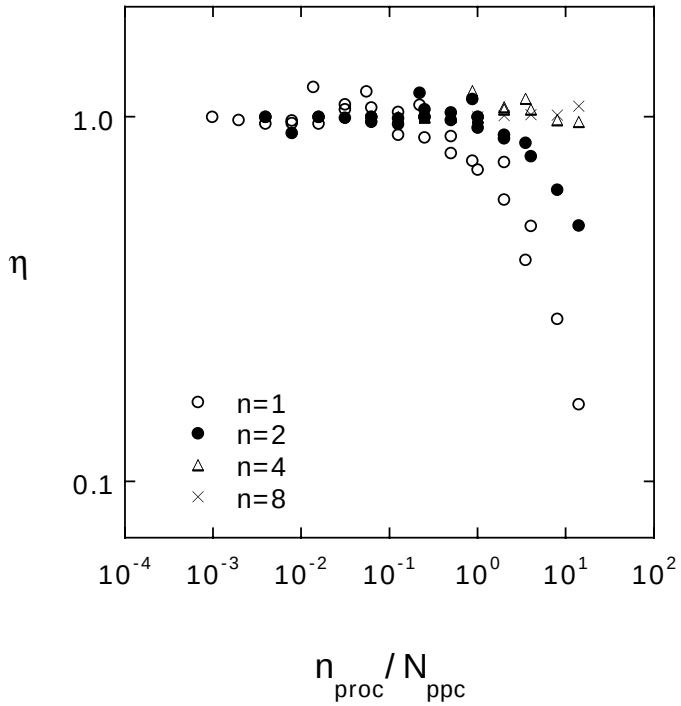

FIG. 2.2. *Efficiency versus n_{proc}/N_{ppc}, for parallel FSP simulations.*

particle number and the total momentum. After the collision phase, each particle propagates toward one of the neighbour sites along the direction of its velocity.

Each fluid moment (mass density, momentum density and pressure) is expressed as a linear combination of particle populations.

The LBE describes the evolution of this particle system, assuming that, under the effect of collisions, the system relaxes toward a local-equilibrium distribution function $N_j^{eq}(\vec{r},t)$, and linearizing the collision operator around such distribution function. The basic step of the LBE method is the choice of the local-equilibrium distribution function $N_j^{eq}(\vec{r},t)$, which must be written as a suitable combination of fluid moments [19] to reproduce, by a multiscale expansion of the LBE [18], the macroscopic equations of the plasma turbulence.

In order to perform realistic two-dimensional plasma-turbulence simulations, a LBE code has been developed, consisting of approximatively 1000 F77 lines distributed over 14 procedures [19]. The minimum number of populations comes out to be $b = 18$. The numerical evolution of these populations $\{N_j(\vec{r},t), j = 1,\ldots,18\}$, from time t to time $t+\Delta t$, on a domain of $l \times m$ lattice spacings can be summarized in the following steps [20]:

1. **Computation of fluid moments**

The values of the fluid moments at time t and lattice site $\vec{r}$ are calculated as linear combinations of particle populations.

2. **Smoothing phase**

Fluid-moment values and their spatial derivatives are used to compute the local-equilibrium distribution function $N_j^{eq}(\vec{r},t)$ [19]. The spatial-derivative terms can give raise to numerical instabilities, characterized by wavelengths of the order of the lattice

spacing [6, 19]; in order to eliminate these instabilities, the fluid moments used in the spatial-derivative terms are smoothed over a spatial region of $l_s \times m_s$ lattice spacings, with l_s and m_s much smaller than the maximum number of sites in the respective directions [19].

3. **Collision phase**

On each lattice site $\vec{r}$, the particle distribution functions are advanced, by half a time step, by retaining the effect of collisions:

$$N_j(\vec{r}, t + \frac{\Delta t}{2}) = N_j(\vec{r}, t) + \sum_{k=1}^{18} A_{jk} \left[N_k(\vec{r}, t) - N_k^{eq}(\vec{r}, t) \right]$$

$$+B_0(\vec{r}) \sum_{k=1}^{18} B_{jk} N_k(\vec{r}, t) - \frac{B_0^2(\vec{r})}{24} \sum_{k=1}^{18} s_k (c_{jx} c_{kx} + c_{jy} c_{ky}) N_k(\vec{r}, t) \ , \tag{3.1}$$

with $j = 1, \ldots, 18$. The collision matrix, A_{jk}, and the magnetic-field one, B_{jk}, are 18×18 constant matrices, which can be written, respectively, in terms of the kinematic viscosity and thermal conductivity [18], and of the magnetic field B_0 [19]. The local-equilibrium distribution function $N_k^{eq}(\vec{r}, t)$ uses the fluid-moment values computed in the first step of the algorithm, except for the spatial-derivative terms, which are obtained using the smoothed fluid moments, computed in the second step.

4. **Propagation phase**

In the next half time step, each particle propagates along its velocity direction toward the next neighbour site:

$$N_j(\vec{r} + \vec{c}_j \Delta r, t + \Delta t) = N_j(\vec{r}, t + \frac{\Delta t}{2}) \qquad j = 1, \ldots, 18. \tag{3.2}$$

3.1. Parallelization Strategy. In the two-dimensional LBE simulations, particle populations $N_{j=1,\ldots,18}$ and fluid moments are expressed as $l \times m$ arrays, which are updated at every time step. When using a regular domain decomposition, data-parallel implementation of the code is immediate. Each processor operates on the local data relative to its own sub-domain and communicates with the other processors when data belonging to different sub-domains are required. In the LBE algorithm, data must be communicated to the neighbour sites during the propagation and smoothing phases, whilst the fluid-moment computation and the collision one are completely local. A decomposition of the $l \times m$ lattice into contiguous blocks of dimension $l \times [m/n_{proc}]$ seems to be the most convenient one, where $[m/n_{proc}]$ is the integer upper bound on the division and n_{proc} the number of processors.

The two-dimensional population and fluid-moment arrays are distributed (`BLOCK`) among processors along their second dimension.

The distribution of the loop iterations is immediate for the fluid-moment computation and the collision phase.

The smoothing phase performs averages of the fluid moments, both along the row index (over l_s lattice spacings) and the column index (over m_s lattice spacings). The average along the row index is parallelized by the HPF compiler automatically: every processor executes independently this computation on the own data section. For the average along the column index, each processor needs data of the m_s columns adjacent to its own sub-domain, requiring communication with the other processors. Using the standard HPF features to parallelize this operation, the computation and communication established by compiler are not optimized. In particular, the loop over the column index is not efficiently distributed by the HPF compiler: after the application

of the *owner computes* rule (i.e., in this case, all processors perform the whole loop for $j = 1, m$, evaluating for each j whether the index belongs to their own sub-domain (and executing the computation only in this case), and whether data communication between processors is necessary to calculate the average), the compiler does not apply a *loop tiling* optimization (consisting in rearranging the iteration chunks belonging to each process, and restructuring the loop header), thus leading to a non-negligible computation overhead with respect to the serial case. Moreover, the communication of the m_s data columns belonging to each neighbour sub-domain comes out to be inefficient because no communication aggregation, and pre-loop optimization of communications is performed. Thus, columns are communicated individually rather than all at once, and each column is communicated several times, every time the average relative to a different value of the sub-domain index j requires it. A different strategy must then be adopted. This has been obtained, as in § 2, by the use of `EXTRINSIC(HPF_LOCAL)` procedures. Two local arrays, `my_left_lay(l,m_s,i_proc)` and `my_right_lay(l,m_s,i_proc)`, containing, respectively, the first and the last m_s columns of the sub-domain are built-up, for each processor `i_proc`. Then, data are communicated between the processors that evolve neighbouring sub-domains, by copying the two arrays in two different arrays, characterized by shifted processor index, `right_lay(l,m_s,i_proc-1)` and `left_lay(l,m_s,i_proc+1)`, respectively. In such a way, communication occurs before the average computation, and the H-PF compiler transfers data between processors all at once, using only one temporary buffer to send the m_s data columns. Finally, a second `EXTRINSIC(HPF_LOCAL)` procedure, which performs the average along the column index, will run simultaneously on multiple processors, using, for every processor `i_proc`, the locally distributed array sections.

During the propagation phase a communication between processors will be established to calculate the propagated values of particle populations on the lattice sites belonging to the first and last column of each sub-domain. The particle-population propagation can be easily expressed by the use of the Fortran90 array intrinsic procedure `CSHIFT`. The HPF compiler parallelizes automatically the propagation phase and establishes the necessary communication between processors. In particular, before the particle populations propagate, each processor exchanges the data located in the extreme columns of its own sub-domain with the (topologically) adjacent processors. This communication phase is optimized by the compiler putting all the data to be sent (or received) on a temporal buffer and sending (or receiving) them, all at once, by a single send (or receive) call.

3.2. Performance figures. In Fig. 3.1 the speed-up factor of the parallel LBE code is plotted versus the number of processors, for several parallel executions with different grid sizes, keeping the same value of $l_s = m_s = 10$. For the smallest-size case $l = m = 64$, the communication phase results quite heavy in comparison with the computational one. Indeed, for $n_{procs} > 4$ the parallel-version performance decreases with the number of processors; the overhead associated to the communication makes useless the time reduction related to the parallel computation. As the lattice dimensions are increased, however, the computational phase becomes more and more relevant and the parallel-code efficiency increases, approaching the ideal speed-up (the continuous line).

4. Tight-Binding Molecular Dynamics code. The goal of a generic Molecular Dynamics (MD) code is to generate time trajectories of a system constituted by N atoms by solving their equations of motion depending on the forces acting on each

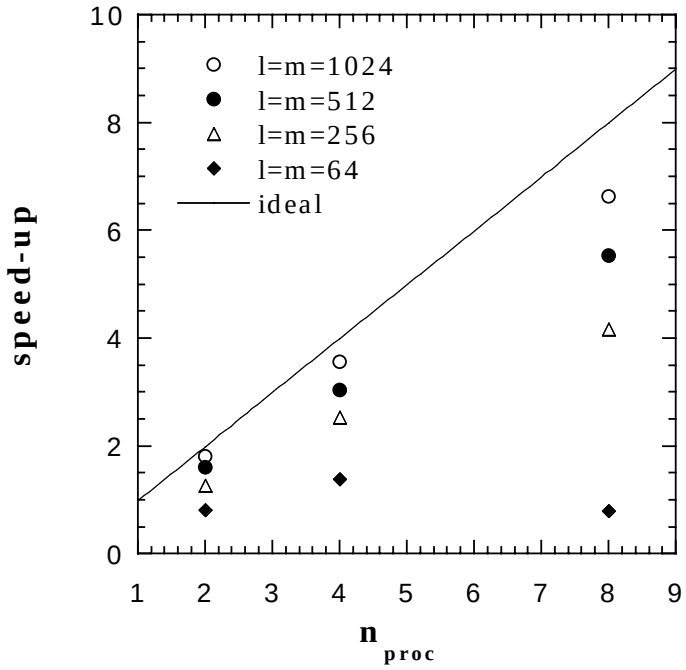

FIG. 3.1. *Speed-up of the LBE algorithm versus the number of processors, for the same values of* $l_s = m_s = 10$ *and different values of* $l = m$. *The continuous line represents the ideal speed-up.*

atom. The specific TBMD code considered here consists of approximatively 5000 F77 lines distributed over 14 procedures. Example of a carbon material simulated by the TBMD code is reported in Fig. 4.1, where new carbon structures are studied in predefined thermodynamic conditions. This kind of simulations allows testing the stability of new carbon structures that can have a large employ in combustion cells and microelectronic devices.

The main computational task of a generic MD code is represented by the evaluation of the forces. In the classical representation, such evaluation is essentially related to the calculation of the interparticle distances, and the corresponding computational cost scales as N^2. Differently from the classical case, the Tight-Binding (TB) formulation is based on the adiabatic approximation of the Hamiltonian H_{tot} of a system of ions and electrons in a solid [7],

$$H_{tot} = T_i + T_e + U_{ee} + U_{ei} + U_{ii} \ , \tag{4.1}$$

where T_i and T_e are the kinetic energy of ions and electrons, respectively, and U_{ee}, U_{ei}, U_{ii} are the electron-electron, electron-ion and ion-ion interactions, respectively. Referring to the theory of one electron moving in the presence of the average field due to the other valence electrons and to the ions, the reduced one-electron Hamiltonian can be written as

$$h = T_e + U_{ee} + U_{ei} \ , \tag{4.2}$$

giving the eigenvalues (energy levels), ϵ_j, and the eigenfunctions, $|\Psi_j>$, where $j = 1, \ldots, n$ and n is the rank of the matrix h. In the TB scheme, the eigenfunctions are

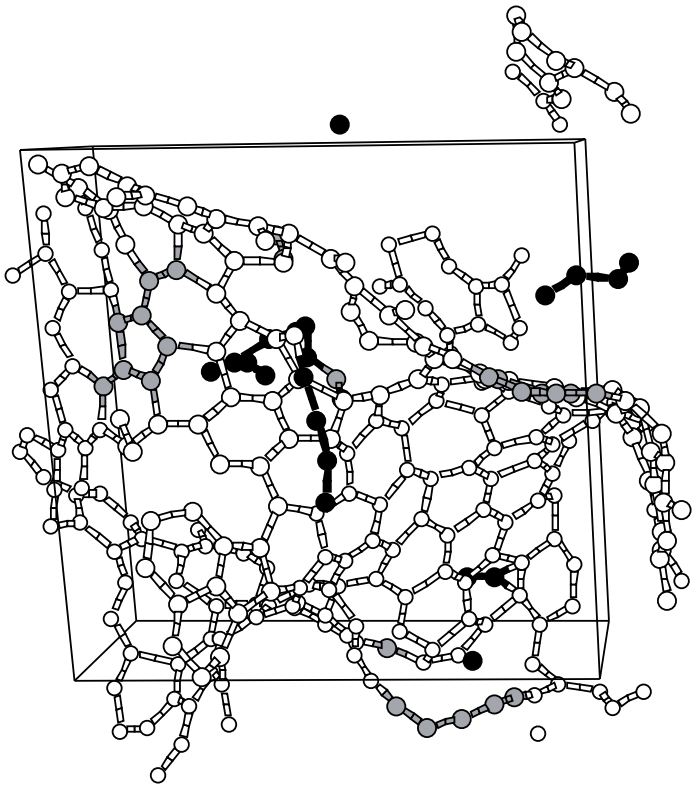

FIG. 4.1. *High temperature carbon system that unfolds till the appearance of graphite-like sheet. Molecular dynamics code allows the simulation of real materials at the atomic level in predefined thermodynamic conditions.*

represented as linear combinations of atomic orbitals $|\phi_{l\alpha}>$,

$$|\Psi_j>=\sum_{l\alpha} c^j_{l\alpha}|\phi_{l\alpha}> , \tag{4.3}$$

where l is the quantum number indexing the orbital and α labels the atom. The expansion coefficients $c^j_{l\alpha}$ represent the occupancy of the l-th orbital located of the α-th atom. In the present TB scheme, the elements of the matrix h, $<\phi_{l'\beta}|h|\phi_{l\alpha}>$, related to the nearest neighbour atoms α and β, can be factorized as

$$<\phi_{l'\alpha}|h|\phi_{l\beta}>= a_{l'l} f(r_{\alpha\beta}) , \tag{4.4}$$

where $a_{l'l}$ are parameters fitted on *ab-initio* calculation or experimental results, and $f(r_{\alpha\beta})$ is a scaling function dependent on the interparticle distance $r_{\alpha\beta}$. As a further approximation, a minimal basis set (i.e., the minimal set of electronic orbitals for each atom) is usually adopted: four basis functions (s, p_x, p_y, p_z) per atom are known to be sufficient for a satisfactory description of the valence bands in the case of elemental semiconductors (silicon and carbon). In order to obtain the eigenvalues (the single-particle energies) ϵ_j and the coefficients $c^j_{l\alpha}$, it is necessary to fully diagonalize the matrix h at each MD time-step. The rank n of the matrix is given by $n = N \times k$, where k is the dimension of the basis set ($k = 4$ in the simplest case of elemental

semiconductors). This implies a N^3 scaling for the computational cost of the force evaluation and, as a consequence, of the overall TB calculations. Once the eigenvalues and the eigenvectors are known, the (overall attractive) electron contribution to the atomic forces can be computed as a sum over the eigenstates. The force calculation is completed by adding the (repulsive) ion-ion contribution derived from a many-body approach [21], which depends on the mutual ion-ion distances and takes into account the overlap interaction originated by the non-orthogonality of the basis orbitals and the possible charge transfer.

The Hamiltonian of Eq.(4.1) refers only to the atomic coordinates describing the internal degrees of freedom of the system. In order to simulate the interaction with the surroundings, represented by external temperature and pressure, further degrees of freedom are added to the Hamiltonian H_{tot}. Thus the MD simulation in the isothermal-isobaric ensemble requires the presence of 10 extra variables, which are coupled to the internal degrees of freedom via suitable parameters which can be adjusted to ensure the fastest convergence to equilibrium (Parrinello-Rahman and Nosé algorithm [22]).

After the calculation of the force on each atom, the equations of motion can be integrated by using a finite difference scheme. In the code here considered, a sixth-order predictor-corrector scheme (VI-order Gear algorithm) [23] has been implemented because of its good accuracy.

4.1. Parallelization Strategy in HPF. The layout of the generic iteration step of the sequential TBMD code consists in the following steps:

1. Predictor part of the integration of the dynamical equations for the evolution of the $3N + 10$ degrees of freedom;

2. Computation of the interparticle distances $r_{\alpha\beta}$: this is used to fill the matrix of nearest neighbours and the array used to store the number of nearest neighbours of each atom;

3. Computation of the matrix elements $< \phi_{l'\alpha}|h|\phi_{l\beta} >$;

4. Diagonalization of the real skew matrix h for the computation of the half spectrum of eigenvectors and eigenvalues. This is the most computationally intensive part of the code; furthermore every t_{phys} time steps the whole spectrum of eigenvectors and eigenvalues must be computed for post processing study of the electronic properties of the material;

5. Computation of the attractive part of the atomic forces (Hellman-Feynman routine) by means of the eigenvalues and the eigenvectors obtained in the previous step;

6. Computation of the repulsive part of the atomic forces, depending only on the interparticle distances: the nearest neighbours matrix is thus used;

7. Corrector part of the integration of the dynamical equations for the evolution of the $3N + 10$ degrees of freedom;

8. All the physical quantities are evaluated; every t_{phys} time steps, their values are stored on disk.

The predictor-corrector scheme requires the knowledge of the atomic coordinates up to the fifth order derivative: the whole atomic system is thus described by 3×6 arrays of length N for the coordinates and by further 6 arrays of the same length to store the forces (3 for the attractive part and 3 for the repulsive one). A $N \times N$ matrix is requested to store the interparticle distances. For minimizing the computation of the TB h matrix, the components of the forces and of the distances are also stored in 9 arrays of dimension N. The matrix h must be allocated, with dimensions at least

$4N \times 4N$.

The main issue that arises in defining the parallelization strategy is the selection of the layout for the data structures, most of them involved in more than one (or all) computational tasks. In most of the schemes for data distribution, the task of minimizing the amount of communication (by improving data locality) conflicts with the task of maximizing the degree of parallelization (by maximizing the number of iterations in the loop distribution). This issue is complicated by the introduction of a call to an optimized parallel library for the diagonalization of the real skew matrix h. We have chosen a library that can perform the matrix diagonalization and that can be used jointly with the HPF directives: the Parallel ESSL (PESSL) [24]. The PESSL is a parallel mathematical library specifically tuned for the IBM SP computers and can be called both from message passing routines and HPF framework. In the latter case the efficiency of the matrix distribution and of the start up of the diagonalization are lower than in the message passing framework, but the implementation is simpler. This allows a sharp tuning of the parameters needed for the parallel routine. The linear algebra section of the PESSL is based on the ScaLAPACK [25] public domain software. Also in this case PESSL assumes that the application program is using the SPMD programming model. For communication, PESSL includes the Basic Linear Algebra Communications Subprograms (BLACS), which use the Parallel Environment (PE) and the Message Passing Interface (MPI) communication library.

The different tasks in which the computation is decomposed are all suitable to be parallelized in the HPF data parallel framework.

Steps (1) and (7) (the predictor and corrector steps) involve sweeps over the monodimensional arrays `x, y, z` (the particle coordinates), `x1, y1, z1, ...x5, y5, z5` (the derivatives of the particle coordinates up to the fifth order), `fx,fy,fz` (the total forces acting on each particle).

The do loops implementing those sweeps do not present loop-carried dependencies; thus their iterations can be distributed among the processors, by following the optimal data layout, which corresponds to an alignment, and cyclic distribution, of the above structures:

```
DISTRIBUTE (CYCLIC) :: x,y,z
ALIGN WITH x :: x1,y1,z1,... x5,y5,z5
ALIGN WITH x :: fx,fy,fz
```

The data parallel structure of the computation and the absence of *stencil effects* imply a total absence of communications.

The step (2) computes the distances `xx/yy/zz/d(N,N)` (two-dimensional arrays) from `x,y,z`; it involves the construction of the *Verlet lists* `list/list1(N,N)` (along with the filling marker vectors `nlst/nlst1(N)`), which keep track of the symmetry and the sparsity of the values stored in `xx/yy/zz/d` and are consequently used to reduce the amount of computations when accessing the latter arrays.

The optimal layout for these data structures is their alignment with the arrays of coordinates, and thus their distribution (`CYCLIC`) over the first coordinate:

```
ALIGN list(:,*) with x(:)
ALIGN list1 with list
ALIGN nlst(:) with list(:,*)
ALIGN nlst1(:) with list1(:,*)
ALIGN with list :: xx,yy,zz,d
```

The iterations of the double loop nests performing the computation of distances and Verlet lists would be distributed accordingly. Unfortunately, this data layout is

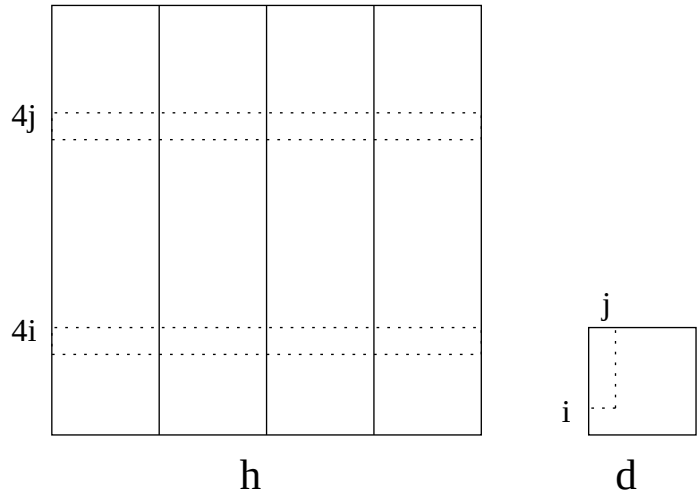

FIG. 4.2. *Schematic structure of the matrices* `h` *and* `d`.

in conflict with the optimal data layout for step (5), the Hellman-Feynman forces computation. This step updates the data structures `fhfx, fhfy, fhfz(N)` (monodimensional arrays) representing the Hellmann-Feynman forces, and uses the auxiliary two-dimensional arrays `fhffx, fhffy, fhffz(N,N)`. The elements of these arrays are computed by sweeping over the two-dimensional distance arrays `d(i,j)` and, for each couple (i, j), by reducing the $4i$-th to $(4i + 4)$-th and $4j$-th to $(4j + 4)$-th rows of the Hamiltonian matrix `h(n,n)`:

```
do i=1,n
  do j= 1,n
  ....
    do 30 im=1,4
      do 40 jm=1,4
        i1= 4*(i-1)+im
        j1= 4*(j-1)+jm
        summa = 0.d0
        do k=1,n4/2
          summa=  summa + occup(k) * h(i1,k) * h(j1,k)
        enddo
        ....
      enddo
    enddo
    fhffx(i,j) = ...(summa)
    ....
  enddo
enddo
```

The sweep of the rows of `h` for each distance `d(i,j)` is graphically visualized in Fig. 4.2. For each sweep of a row of the matrix `d`, there is a sweep of the whole matrix `h`. If we follow the data layout prescribed above for the matrix `d` (i.e., cyclic distribution of the rows over the processors), we should replicate the whole matrix `h` over the processors to reduce the amount of communications. This is unfeasible, as the matrix `h` is, by far, the largest structure of the whole data set, being 16 times larger (in the case of

a minimal basis set) than the matrix `d`.

The only way to cope with this constraint is to give up on distributing the matrix `d` (thus replicating it over the processors). The Hamiltonian `h` is instead distributed over the second dimension (see Fig. 4.2). The sweep over the elements of the matrix `d` is thus replicated, but the reduction of the 8 rows of `h`, for each pair (i, j), can be performed in parallel: each processor sweeps the portion of rows of `h` assigned to it, and performs a partial reduction of those elements. The results of the partial reduction operated from each processor are then composed, and the result of the final reduction is broadcasted. The amount of distribution of the iterations of the overall loop nest (and thus the degree of parallelization) remains the same as in the case of the alternative data layout (distributed distances, and replicated Hamiltonian). With the latter scheme, there is an additional communication overhead, due to the communication of the partial results of the reduction. This overhead, however, turns out to be negligible with respect to the computation overhead, and thus an efficiency gain is yielded by parallel execution.

Instead of redistributing the matrix `d` from the optimal data layout of step (2) to the whole replication needed in this step, we preferred to keep these matrices always replicated (also during step (2)): although, in this way, the computation is replicated among the processors during that phase, the communication overhead occurring during the redistribution of those matrices would be much more costly than the computation overhead of the replication of step (2).

In steps (3) and (4) the computation and diagonalization of the Hamiltonian two-dimensional matrix `h(n,n)`, and computation of its eigenvalues in the monodimensional array `eval(n)` are performed. The PESSL routine SYEVX is called for this task. The parameters in the tuning process are those needed (a) to control the absolute tolerance on the eigenvalues (in order to improve the efficiency of the algorithm for the orthogonalization of the eigenvectors) and (b) to balance the workspace and the memory requested for the eigenvectors calculations.

The use of the PESSL routine introduces extra inter-node communications because the SYEVX routine needs the matrix to be distributed in both dimensions as cyclic. We thus need to redistribute the Hamiltonian matrix `h` during this step and to restore its distribution over the columns as needed in the Hellmann-Feynman routine.

In Fig. 4.3 the speed-up of the TBMD code is reported, showing the suitability of the HPF approach, which allows reducing the computational time even if the problem is intrinsically irregular.

The package of the code in HPF version, compilable for SP systems with the `xlhpf` compiler, can be found in the CPC program library [26].

5. Conclusions. In this paper we reported our experiences gained in porting medium and large sized numerical codes to a distributed memory parallel architecture, following the parallelizing compilation approach, and, in particular, the HPF one.

The main features and problems of such codes, with respect to their parallelization within the HPF framework, can be summarized as follows. The first code, consisting of approximatively 16,000 lines F77 code and distributed over more than 40 procedures, is a typical particle-in-cell code. The application of a *domain decomposition* technique, usually applied for the parallelization of PIC codes, was discarded because of the severe load-balancing problems it presents. In order to avoid such problems, within this approach, a dynamical redistribution of grid and particle quantities is needed, which requires an extensive code restructuring and makes the parallel implementation of a PIC code very complicate. This is expecially true in our case, due to the code size,

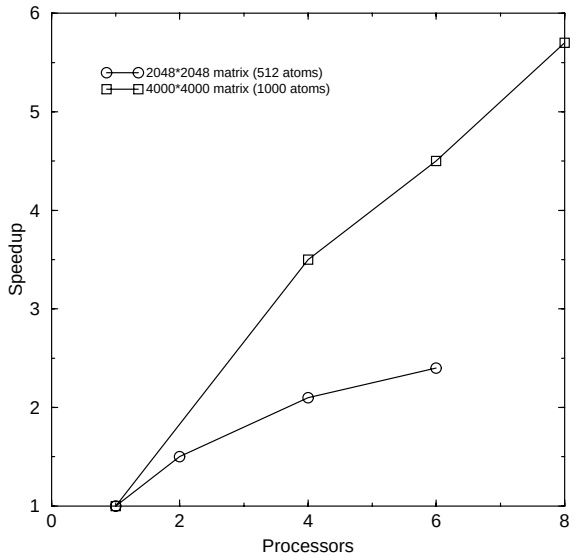

FIG. 4.3. *Speed-up of the Tight-Binding Molecular Dynamics code obtained by using H-PF+PESSL parallel routines.*

the utilization of programming techniques (like the frequent usage of common blocks, and the implicit reshaping of arrays at procedure boundaries) that complicate the code restructuring, and the need of an easy portability and maintenance of the parallelized code. We have thus devised a *particle decomposition* technique, consisting in statically distributing the particle population among processors, while replicating the data relative to grid quantities. Load balancing is in this case automatically enforced, because no particle has to be transferred from one processor to another, thus enabling a very efficient and relatively straightforward implementation of the parallelization in HPF. We experienced problems with the parallelization of loop bodies, due to the huge number of lines of the loop bodies to be parallelized, and the complexity of the references to the distributed array elements. Such problems where solved by resorting to the `HPF_LOCAL` extrinsics mechanism provided by HPF.

The second code, consisting of approximatively 1000 lines F77 code, distributed over 14 procedures, is a two-dimensional LBE code, studying the evolution of a *fictitious* particle system on a two-dimensional discrete spatial domain. Data-parallel implementation is immediate by a regular domain decomposition, and communication of data owned by different processors is efficiently established by the HPF compiler in most part of the code. Nevertheless, the HPF compiler does not optimize the specific computation and communication phases of the simulation: a non-negligible computation overhead is added by the loop iteration distribution restructuring performed by the compiler, and the communication of the "shade boundary" array elements stored in the non-local memories is not optimized. Also for this code, the problem has been solved using the `HPF_LOCAL` extrinsic procedures.

The main issue arisen with the third code, consisting of approximatively-5000

lines F77 code distributed over 14 procedures, was the selection of the optimal distribution (and dynamic redistribution) for the data structures. In most of the schemes for data distribution, the target of minimizing the amount of communication conflicts with the target of maximizing the degree of parallelization. The introduction of dynamic redistributions, needed to align the distributions with the different access patterns occurring in the several different computational tasks, added further complexity in the evaluation of the best data layout (with respect to communication overhead and memory occupation). The ease of data distribution specification in H-PF was an invaluable feature we exploited for this task. We had anyway to perform code restructuring at a certain extent, in order to decouple some data accesses, thus obtaining better performance results. The second important issue we faced was the introduction of a call to an optimized parallel library for matrix diagonalization. We utilized the Parallel ESSL (PESSL) package, a parallel mathematical library specifically tuned for the IBM SP computers and designed to be integrated with the HPF framework. We had anyway to choose a suitable data layout for the data structure involved in the PESSL call, and thus we had to redistribute the corresponding arrays at the call boundary.

We can conclude the following. Porting large scale legacy codes to parallel architectures by HPF, and obtaining effective results both in performance and easiness of maintenance is an achievable task, although not as easy as it is often claimed to be. Such an efficient porting comes out to be possible not only when the computation exhibits regular data-parallel behaviour, but even when it presents characteristics of irregularity. Many aspects must however be carefully considered when planning the parallelization strategy, all of them dependent on each other. The most important is the unchangeability of the execution scheme adopted by HPF, the SPMD one: in the presence of irregular computations, this prevents us from adopting execution schemes typically suggested in the literature. As shown for the plasma particle simulation application, this does not necessarily lead to inefficient parallelization: a careful study of the computational characteristics of the application showed the opportunity for an alternative strategy, with data parallel characteristics, which leads to almost ideal efficiency behaviour (within the range of data sizes of interest).

The second important aspect is the choice of the data distribution: it must be carefully planned, expecially when the application is composed of several different computations occurring on the same data structures (as it was the case for the molecular dynamics application). Any change of data layout for any of the data structures involved severely affects the efficiency of all the code portions accessing it, in a very unpredictable way. The utilization of optimized parallel procedures to perform some of the computations and the presence of features for redistributing data at run-time add further degrees of complexity to the selection of the optimal data layout.

The third aspect is the state of the technology of the HPF compilers: despite the tremendous amount of complexity involved in the analysis and automatic restructuring they perform, they currently make a good job in handling large-size legacy codes. Nevertheless, a number of pitfalls are still present, such as the limited power of the dependency analysis in the presence of indirect accesses (cf. the first application), or the limited capability for aggregation and pre-loop factorization of communications, in the presence of complex loop nest and data accesses (cf. the second application). The `HPF_LOCAL` extrinsic mechanism provided by HPF comes out to be the *panacea* to solve this sort of problems.

The main benefits we gained from adopting the HPF approach were in the ra-

pidity of porting (around 20 person-days for the first application, 5 for the second application and 10 for the third one, considering all the aspects of design and code writing/restructuring) and the subsequent easy maintenance of the parallel code. We underline the important characteristic of an HPF code consisting in the coincidence of the sequential and parallel versions, which eliminates the need for caring for their mutual consistency.

REFERENCES

[1] F. Darema et al., *A Single Program Multiple Data computational Model for EPEX/Fortran*, Parallel Computing, 7 (1) (1988), pp. 11–24.

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

[3] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Scientific Programming, 2 (1-2) (1993), pp. 1–170.

[4] H. Richardson, *High Performance Fortran: history, overview and current developments*, Tech. Rep. TMC-261, Thinking Machines Corporation, 1996.

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

[6] G. Fogaccia, R. Benzi and F. Romanelli, *Lattice Boltzmann simulations of electrostatic plasma turbulence*, in High Performance Computing and Networking. Proceedings, LNCS Vol. 1067, Springer, Berlin (1996), pp. 276–282.

[7] L. Colombo, *Tight-Binding Molecular Dynamics*, in Annual Review of Computational Physics IV, edited by D. Stauffer, World Scientific, Singapore (1996), p. 147.

[8] C. Z. Wang, K. M. Ho, *Tight Binding Molecular Dynamics studies of covalent systems* in New methods in Computational Quantum Mechanics, I. Prigogine, S.A. Rice eds., vol. XCIII in Advances in Chemical Physics, John Wiley & Sons, 1996.

[9] R. Car and M. Parrinello, *Unified approach for molecular dynamics and density-functional theory*, Phys. Rev. Lett., 55 (1985), pp. 2471–2474.

[10] M. P. Allen and D. J. Tildesley, *Computer simulation of liquids*, Clarendon Press, Oxford, 1987.

[11] L. Miglio, V. Meregalli, F. Tavazza, M. Celino, *Analysis of the metal-semiconductor structural phase transition in $FeSi_2$ by tight-binding molecular dynamics*, Europhysics Letters, 37 (2) (1997), pp. 415–420.

[12] V. Rosato, M. Celino and L. Colombo, *On the effect of quench-rate on the structure of amorphous carbon*, Computational Materials Science, 10 (1998), p. 67.

[13] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Y. Wang, W. M. Ching, T. Ngo, *A HPF Compiler for the IBM SP2*, in Proc. SuperComputing '95, ACM, 1995.

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

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

[16] B. Di Martino, S. Briguglio, G. Vlad and P. Sguazzero, *Parallel Plasma Simulation in High Performance Fortran*, in High Performance Computing and Networking, Springer, Berlin, 1998, pp. 203–212.

[17] A. B. Langdon and C. K. Birdsall, *Theory of plasma simulation using finite-size particles*, Phys. of Fluids, 13 (1970), pp. 2115–2122.

[18] R. Benzi, S. Succi and M. Vergassola, *The Lattice Boltzmann-equation - theory and applications*. Phys. Rep., 222 (1992), pp. 145–197.

[19] G. Fogaccia, R. Benzi and F. Romanelli, *A Lattice Boltzmann algorithm for 3-D simulations of plasma turbulence*, Phys. Rev. E, 54 (1996), pp. 4384–4393.

[20] G. Fogaccia, *Parallel Implementation of a Lattice Boltzmann Algorithm for the Electrostatic Plasma Turbulence*, High Performance Computing and Networking. Proceedings, LNCS Vol. 1401, Springer, Berlin, 1998, pp. 213–222.

[21] I. Kwon, R. Biswas, C. Z. Wang, K. M. Ho, C. M. Soukoulis, *Transferable tight-binding models for silicon*, Phys. Rev. B, 49 (1994), pp. 7242–7249.

[22] M. Parrinello and A. Rahman, *Polymorphic transitions in single crystals: a new molecular dynamics method*, J. Appl. Phys., 52 (1981), pp. 7182–7190; S. Nosé, *Constant Temperature Molecular Dynamics Methods*, Progr. Theoretical Phys. Suppl., 103 (1991), pp. 1–46.

[23] C. W. Gear, *The numerical integration of ordinary differential equations of various orders*,

Report ANL 7126, Argonne National Laboratory (1966); C. W. Gear, *Numerical initial value problems in ordinary differential equations*, Prentice-Hall, Englewood Cliffs, NJ (1971).

[24] *Parallel Engineering and Scientific Subroutine - Guide and Reference, Release 2*, IBM (1996).

[25] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, R. C. Whaley, *ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance*, in Proceedings of Supercomputing '96, Sponsored by ACM SIGARCH and IEEE Computer Society, 1996. (http://www.supercomp.org/sc96/proceedings/).

[26] B. Di Martino, M. Celino, V. Rosato, *An High Performance Fortran Implementation of a Tight-Binding Molecular Dynamics simulation*, Computer Physics Communications, 120 (1999), pp. 255–268.