Development of Large Scale High Performance Applications with a Parallelizing Compiler

B. DI MARTINO[2], S. BRIGUGLIO[1], M. CELINO[3], G. FOGACCIA[1], G. VLAD[1], V. ROSATO[3], M. BRISCOLINI[4]

[1] Associazione Euratom-ENEA sulla Fusione, CR Frascati, CP 65, 00044 Frascati, Rome, (Italy)

[2] Second University of Naples, Dip. di Ingegneria dell' Informazione (Italy)

[3] ENEA - HPCN Project, C.R. Casaccia, CP 2400, 00100 Roma AD (Italy)

[4] IBM Italia, Roma (Italy)

beniamino.dimartino@unina.it

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: easy design, development, verification and debugging of sequential code, existence of several environments for the support of sequential programming, and 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, according to the Single Program Multiple Data (SPMD) [1] execution model 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. See Ref. [2] 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, 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.

We have experimented 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. One of the most relevant issues of such a research, which is facing the target of demonstrating the scientific feasibility of a reactor prototype, 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 that, at present, 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, the parallelization of 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 [3] adopts the *particle-in-cell* (PIC) technique. The other one [4] 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 [5, 6]. 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) [7] molecular dynamics and simulations using empirical classical potentials [8]. 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 [9, 10]. 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, computational 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 [11] (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 parallelization effort.

2 Plasma particle simulation codes

Particle simulation codes [12] seem to be the most suited tool for the investigation of turbulent plasma behaviour. Particle simulation indeed consists in evolving the phase-space coordinates of a particle population in the electromagnetic fields selfconsistently computed, at each time step, in terms of the contribution yielded by

the particles themselves (e.g., pressure perturbation). Thus, it allows one to fully retain 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.

An accurate description of the plasma behaviour requires to deal with very large numbers of grid cells (N_{cell}) and particles (N_{part}), with an average number of particles per cell $N_{ppc} \equiv N_{part}/N_{cell} \gg 1$. Such a goal imposes to resort to parallelization techniques aimed to distributing the computational loads related to the particle population among several computational nodes. Here we consider the implementation of a parallel version of a specific PIC code, namely the Hybrid MHD-Gyrokinetic particle simulation Code [3] (HMGC), consisting of approximatively 16,000 F77 lines distributed over more than 40 procedures.

In order to avoid load-balancing problems, typical of standard *domain decomposition* [13] techniques, we adopt a *particle decomposition* [14] approach to the parallelization of HMGC. This approach 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 (e.g., $x(n_{part})$), 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 at the grid points presents two strictly linked problems: (*i*) such a quantity is replicated, and thus must be kept consistent among the processors; (*ii*) each element of the pressure array $press(n_x, n_y, n_z)$ takes contribution from particles that reside on different nodes. The strategy adopted to solve this problem relies on the associative and distributive properties of the updating laws for the pressure array with respect to the contributions given by every single particle: the computation for each update is split among the nodes into partial computations, involving the contribution of the local particles only; then the partial results are reduced into global results, which are broadcasted to all the nodes.

The scheme to handle with this "inhibitor of parallelism" within the loops over the particles, can be implemented in HPF by restructuring the code in the following way: (*i*) the data structure that store the values of the pressure, is replaced, within the bodies of the distributed loops, by a corresponding data structure augmented by one dimension ($press_{par}(n_x, n_y, n_z, :)$), with rank equal to (or greater than) the number of available processors; (*ii*) this temporary data structure is distributed, along the added dimension, over the processors; each of the distributed "pages" will store the partial computations of the pressure, which include the contributions of the particles that are local to each processor; (*iii*) at each iteration of the loop over the particles, the contribution of the corresponding particle to an element of the pressure array is added to the appropriate element of the distributed page; (*iv*) at the end of the iterations, the temporary data structure is reduced along the added and distributed dimension, and the result is assigned to the corresponding original data structure. This is implemented by using HPF intrinsic reduction functions such as `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 $press_{par}$ local to each particle, even though not complex, is anyway a non-linear function of the loop index. Moreover, each particle gives contribution only to the pressure of few neighbouring grid points; the array elements of each page of $press_{par}$ are then addressed by indirect references. These facts 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. Each local procedure executing on a given processor sees the portion of the arrays related to the particles, and the page of the "partial results" array $press_{par}$ assigned to that processor. It executes only the set of loop iterations that access the particles local to the processor, and updates the page of $press_{par}$ assigned to it. At the end of the execution of the local extrinsic procedure, all the partial updates of the components of $press_{par}$ are collected in the global-HPF-index-space $press_{par}$. Then, $press_{par}$ is reduced to *press* with the use of the SUM intrinsic function, as stated before.

The efficiency η, defined as the speed-up factor divided by the number of processors (n_{proc}), is plotted, in Fig. 1, versus n_{proc}/N_{ppc}, for typical parallel PIC simulations obtained by HMGC, with values of n_{proc} up to $n_{proc} = 14$. Different cases, corresponding to increasing values of the typical mode number n retained in the (linear) simulation, are considered; note that the spatial resolution (the number of cells N_{cell}) scales as n^3. The efficiency, which is at its ideal value ($\eta \approx 1$) at low values of n_{proc}/N_{ppc}, fast decreases above a certain threshold[1]. 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 $n_{proc} \geq N_{ppc}$, it breaks down, and the computational effort of each processor is mainly devoted to the replicated calculation of grid quantities. 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.

Such bottle-necks in efficiency and performance associated to grid quantities can be removed [15] by re-

[1] 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.

sorting to a gridless finite-size-particle method [16].

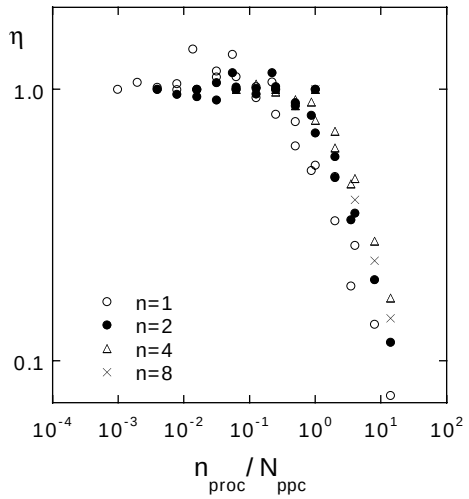

FIG. 1: Efficiency η, defined as the speed-up factor divided by the number of processors, versus n_{proc}/N_{ppc}, for parallel PIC simulations corresponding to different values of the typical mode number n retained.

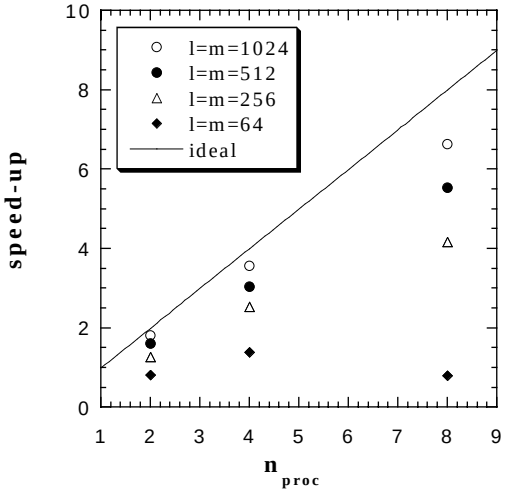

FIG. 2: Speed-up of the LBE algorithm versus the number of processors, for the same values of $l_s = m_s = 10$ and different values of $l = m$. The continuous line represents the ideal speed-up.

3 Plasma LBE Simulation Code

In order to perform realistic two-dimensional plasma-turbulence simulations, a LBE code [17] has been developed, consisting of approximatively 1000 F77 lines distributed over 14 procedures [18]. The numerical evolution of the particle populations can be summarized in the following steps: (*i*) computation of fluid moments: the values of the fluid moments are calculated as linear combinations of particle populations; (*ii*) smoothing phase: fluid-moment values and their spatial derivatives are used to compute the local-equilibrium distribution function; a spatial smoothing on the derivative terms is introduced, in order to eliminate numerical instabilities; (*iii*) collision phase: on each lattice site, the particle distribution functions are advanced, by retaining the effect of collisions; (*iv*) propagation phase: each particle propagates along its velocity direction toward the next neighbor site.

In the two-dimensional LBE simulations, particle populations and fluid moments are defined on a two-dimensional spatial lattice $l \times m$ and are expressed as two-dimensional 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 neighbor 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 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). 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 compute* 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 no communication aggregation and pre-loop optimization of communications are performed. A different strategy must then be adopted. This has been obtained, as in section 2, by the use of `EXTRINSIC(HPF_LOCAL)` procedures. Communication occurs before the average computation, and the HPF compiler transfers data between processors all at once, using only one temporary buffer to send the m_s data columns. Moreover, the average along the column index, will run simultaneously on multiple processors, using, for every processor, the locally distributed array sections.

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 Fig. 2 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 $n_{proc} > 4$ 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 classical equations of motion on the bases of the evaluation of the forces acting on each atom.

The Tight-Binding Molecular Dynamics (TBMD) formulation allows to compute the quantum forces acting on each atom. The method is based on the adiabatic approximation of the total Hamiltonian of a system of ions and electrons in a solid [5]. 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 h gives the eigenvalues ε_j, and the eigenfunctions, Ψ_j describing the electronic states in the system. In the TBMD scheme, the eigenfunctions are represented as linear combinations of atomic orbitals $\phi_{l\alpha}$, 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. Furthermore, the elements of the matrix h, $< \phi_{l'\beta}|h|\phi_{l\alpha} >$, related to the nearest neighbor atoms α and β, can be factorized as

$$< \phi_{l'\alpha}|h|\phi_{l\beta} >= a_{l'l} f(r_{\alpha\beta}) \,, \tag{1}$$

where $a_{l'l}$ are parameters fitted on *ab-initio* or experimental results, and $f(r_{\alpha\beta})$ is a scaling function dependent on the particle 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 $n = N * k$, 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. 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 [19], 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.

In order to simulate the interaction with the surroundings, represented by external temperature and pressure, further degrees of freedom are added to the total Hamiltonian (Parrinello-Rahman and Nosé algorithm [20]).

After the calculation of the force on each atom, the equations of motion can be integrated by using a finite difference scheme: a sixth-order predictor-corrector scheme (VI order Gear algorithm) [21] has been implemented in view of its good accuracy.

The layout of the generic iteration step of the sequential TBMD code consists in the following steps: (*i*) Predictor part of the integration of the dynamical equations for the evolution of the $3N + 10$ degrees of freedom (10 is the number of the extra degrees of freedom inserted to cope with pressure and temperature scaling); (*ii*) Computation of the interparticle distances $r_{\alpha\beta}$: this is used to fill the matrix of nearest neighbors and the array used to store the number of nearest neighbors of each atom; (*iii*) Computation of the matrix elements $< \phi_{l'\alpha}|h|\phi_{l\beta} >$; (*iv*) 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; moreover, each t_{phys} time steps, the whole spectrum of eigenvectors and eigenvalues is computed for the complete analysis of the electronic properties of the material; (*v*) 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; (*vi*) Computation of the repulsive part of the atomic forces, depending only on the distances among the atoms: the nearest neighbours matrix is thus used; (*vii*) Corrector part of the integration of the dynamical equations for the evolution of the $3N + 10$ degrees of freedom; (*viii*) All the physical quantities are evaluated; every t_{phys} time steps, their values are stored on disk.

The adopted predictor-corrector scheme requires the knowledge of the atomic coordinates till 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 $N * N$ 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 $4N * 4N$.

To perform the diagonalization of h, we have chosen the Parallel ESSL (PESSL) [22]: a parallel mathematical library specifically tuned for IBM SP computers. This library 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 [23] public domain software. 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 amenable to be parallelized in the HPF data parallel framework. Steps (*i*) and (*vii*) (the predictor and corrector steps) involve sweeps over 21 monodimensional arrays: the particle coordinates, their derivatives up to the fifth order and 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 a cyclic distribution. The data parallel structure of the computation and the absence of *stencil effects* imply a total absence of communications.

The step (*ii*) computes the inter-atomic distances (3 two-dimensional arrays); it involves the construction of *Verlet lists* which keep track of the symmetry and the sparsity of the values stored in the distances matrices and are 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.

The iterations of the double loop nests which perform the computation of distances and Verlet lists would be distributed accordingly. Unfortunately, this data layout is in conflict with the optimal data layout for

step (*v*), the Hellman-Feynman forces computation. This step updates the data structures representing the Hellmann-Feynman forces, and uses three auxiliary two-dimensional arrays. 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)`. Thus, 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 replicate the matrix `d` over the processors. The Hamiltonian `h` is instead distributed over the second dimension. The sweep over the elements of the matrix `d` is thus replicated, but the reduction of the 8 rows of `h`, for each couple (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 then broadcasted. This communication, however, turns out to be negligible with respect to the computation overhead, and thus the efficiency is gained by the parallel execution.

In steps (*iii*) and (*iv*) the computation and diagonalization of the Hamiltonian two-dimensional matrix `h(n,n)`, and computation of its eigenvalues 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 (which has been tuned to obtain the minimum of tolerance 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.

The good speed-up, till 12 processors (about 9), of the TBMD code shows the suitability of the HPF approach, which allows the reduction of 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 [24].

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.

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. It has been parallelized by 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 an implementation of parallelization in HPF which has proved to be very efficient and, at least in principle, relatively straightforward. 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. We solved those problems with the utilization of 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 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 in order 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 HPF 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 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 those 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, we cannot adopt execution schemes typically suggested in the literature, but we must stick to the SPMD scheme. 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 behaviour of all the code portions accessing it, in an 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 big sized 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 these problems.

The main benefits we gained from adopting the HPF approach were in the rapidity 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*, Vol. 7, No. 1, 1988, pp. 11-24.

[2] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, Rice University, 1997.
[3] S. Briguglio, G. Vlad, F. Zonca, and C. Kar, Hybrid magnetohydrodynamic-gyrokinetic simulation of toroidal Alfvén modes, *Phys. Plasmas*, Vol. 2, 1995, pp. 3711-3723.
[4] 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.
[5] L. Colombo, Tight-Binding Molecular Dynamics, in *Annual Review of Computational Physics IV*, ed. D. Stauffer, World Scientific, Singapore, 1996, pp. 147-178.
[6] C.Z. Wang, K.M. Ho, Tight Binding Molecular Dynamics studies of covalent systems in *New methods in Computational Quantum Mechanics*, ed. I. Prigogine, S.A. Rice, vol. XCIII in *Advances in Chemical Physics*, John Wiley & Sons, 1996, pp. 651-702.
[7] R. Car and M. Parrinello, Unified approach for molecular dynamics and density-functional theory, *Phys. Rev. Lett.*, Vol. 55, 1985, pp. 2471-2474.
[8] M.P. Allen and D.J. Tildesley, *Computer simulation of liquids*, Clarendon Press, Oxford, 1987.
[9] 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*, Vol. 37, 1997, pp. 415-420.
[10] V. Rosato, M. Celino and L. Colombo, On the effect of quench-rate on the structure of amorphous carbon, *Computational Materials Science*, Vol. 10, 1998, pp. 67-74.
[11] 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.
[12] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation*, McGraw-Hill, New York, 1985.
[13] P. C. Liewer and V. K. Decyk, A General Concurrent Algorithm for Plasma Particle-in-Cell Codes, *J. Computational Phys.*, Vol. 85, 1989, pp. 302-322.
[14] B. Di Martino, S. Briguglio, G. Vlad and P. Sguazzero, Parallel Plasma Simulation in High Performance Fortran, in *High Performance Computing and Networking. Proceedings*, LNCS Vol. 1401, Springer, Berlin, 1998, p. 203-212.
[15] S. Briguglio, G. Vlad, G. Fogaccia, and B. Di Martino, Parallelelization of Gridless Finite-Size-Particle Plasma Simulation Codes, in *High Performance Computing and Networking. Proceedings*, LNCS Vol. 1593, Springer, Berlin, 1999, p. 241-250.
[16] A. B. Langdon and C. K. Birdsall, Theory of plasma simulation using finite-size particles, *Phys. of Fluids*, Vol. 13, 1970, pp. 2115-2122.
[17] R. Benzi, S. Succi and M. Vergassola, *Phys. Rep.*, Vol. 222, 1992, p. 145.
[18] 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.
[19] I. Kwon, R. Biswas, C.Z. Wang, K.M. Ho, C.M. Soukoulis, *Phys. Rev. B*, Vol. 49, 1994, p. 7242.
[20] M. Parrinello and A. Rahman, Polymorphic transitions in single crystals: a new molecular dynamics method, *J. Appl. Phys.* Vol. 52, 1981, pp. 7182-7190; S. Nosé, Constant Temperature Molecular Dynamics Methods, *Progr. Theoretical Phys. Suppl.*, Vol. 103, 1991, pp. 1-46.
[21] 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.
[22] *Parallel Engineering and Scientific Subroutine - Guide and Reference, Release 2*, IBM, 1996.
[23] 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/).
[24] B. Di Martino, M. Celino, V. Rosato, An High Performance Fortran Implementation of a Tight-Binding Molecular Dynamics simulation, *Computer Physics Communications*, Vol. 120, 1999, pp. 255-268.