

Programming Shared Memory Architectures with OpenMP: a case study

Beniamino Di Martino

Dip. Ingegneria dell'Informazione, Second University of Naples, Italy
beniamino.dimartino@unina.it

Sergio Briguglio, Giuliana Fogaccia, Gregorio Vlad

Associazione Euratom-ENEA sulla Fusione, Frascati, Rome, Italy
{briguglio,fogaccia,vlad}@frascati.enea.it

Abstract *High level languages and environments such as OpenMP, based on parallelizing compiler technology, provide for easy design, development, verification and debugging of parallel applications and porting of legacy codes to parallel architectures based either on shared or distributed memory model.*

In this paper we discuss the particle-decomposition parallelization of a particle in cell code targeted to shared memory parallel architectures, following the parallelizing compilation approach, in particular utilizing the OpenMP language extensions.

1 Introduction

Several approaches exist to programming of parallel architectures, based either on shared or distributed memory model. Low-level approaches, based on library function calls and manual partition of data, include the *Message Passing* environments (e.g., PVM [10] and MPI [11]) for the distributed memory architectural model, and *multithreaded programming* for the shared memory model (a standard library exists, designed by POSIX and commonly referred as *Pthreads*[5]). It is generally accepted that these low level environments are very complicated to use, 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 sequen-

tial code, possibly augmented with parallelization directives, into explicitly parallel code, by means of *parallelizing compilers*. Languages/environments following this approach include the High Performance Fortran (HPF) [7, 6, 13] language (or, at a higher level, PCN [9], Fortran M [8], *P³L* [2] and SCL [3]) for the distributed memory architectural model, and the OpenMP [12] language, which is the counterpart of HPF for the shared memory model.

OpenMP has recently emerged as an industry standard interface for Shared Memory programming of parallel applications. Using OpenMP, it is possible to write applications with a shared memory programming model, portable to a wide range of parallel computers. The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on a wide range of SMP architectures. It augments C and Fortran with a set of directives to express task and data parallelism, synchronization and control of scope on access of data, and work balancing.

In this paper we discuss the parallelization of a skeleton PIC simulation code, which includes all the main features of the codes developed for the investigation of, e.g., plasmas magnetically confined in toroidal devices. The parallelization is targeted to shared memory parallel architectures, by using the OpenMP language extensions.

The paper is structured as follows. Section

2 describes the the main physical and computational aspects of the application chosen. Section 3 describes the parallelization strategy adopted for the shared memory model and implemented in OpenMP. Section 4 reports the experimental results obtained with a skeletonized version of a real PIC code.

2 The plasma particle simulation application

Particle simulation codes [1] 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 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.

PIC simulation techniques consist in

- computing the electromagnetic fields only at the points of a discrete spatial grid (*field solver phase*);
- interpolating them at the (continuous) particle positions in order to perform particle pushing (*particle pushing phase*);
- collecting particle contribution to pressure at the grid points to close the field equations (*pressure computation phase*).

The presence of a discrete grid, with spacing L_c between grid points, leaves the physically relevant dynamics related to the scales larger than L_c unaffected. At the same time, the 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 $n_0 \lambda_D^3 \gg 1$, with λ_D being the Debye length, 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.

As an example, we consider the following skeletonized F90 code excerpt, which maintains the main features of the corresponding portion of real PIC codes for the simulation of magnetically confined plasmas. Each element of the three-dimensional pressure array \mathbf{p} is updated by the contribution of the particles falling in the neighbours of the corresponding grid point.

```

p = 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)
  p(j_r,j_theta,j_phi) =
&      p(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 non-linear functions of the particle weight and/or phase-space coordinates. This excerpt represents, very schematically, the update of the particle pressure, by trilinear weighting of the contribution of each particle among the vertices of the cells. The computation of the array \mathbf{p} is anyway representative of any computation of all the quantities that inhibit the parallel execution of the loops over the particles.

3 The Parallelization Strategy for Shared Memory architectures with OpenMP

The parallelization strategy we have designed for shared memory architectures consists in distributing to different threads the work needed to update the quantities related to different particles (in the particle pushing phase), or needed to update the pressure fields using those quantities.

Using OpenMP, the code parallelization is relatively straightforward: in particular, the `parallel do` directive can be used to distribute the loop iterations over the particles, related to the particle-pushing and pressure-updating phases. Attention must be paid to protect the *critical sections* of code from race conditions, that is to ensure *mutual exclusion* among threads accessing shared data. An example of critical section is represented by the update of the particle pressure. The `critical` directive is used at this purpose to mutually exclude the different threads from the shared access to the array `p`.

```

p = 0.
!$OMP parallel do private(weight_l,r_l,
& theta_l,phi_l,j_r,j_theta,j_phi)
shared(p)
do l=1,n_part
weight_l = weight(l)
r_l      = r(l)
theta_l  = theta(l)
phi_l    = phi(l)
j_r      = f_1(r_l)
j_theta  = f_2(theta_l)
j_phi    = f_3(phi_l)
!$OMP critical
p(j_r,j_theta,j_phi) =
& p(j_r,j_theta,j_phi) +
& f_4(weight_l,r_l,theta_l,phi_l)
!$OMP end critical
enddo

```

Nevertheless the serialization induced by the protected critical section on the shared access to the array `p` represents a bottleneck which could affect performances. This bottleneck can be eliminated, at the expenses of memory occupation, by means of the following strategy, which relies on the associative

and distributive properties of the updating laws for the pressure array with respect to the contributions given by every single particle: the computation for each update is split among the threads into partial computations, involving the contribution of the local particles only; then the partial results are reduced into global results.

The array `p` is replaced, within the bodies of the distributed loops, by the array `p_par` augmented by one dimension with respect to `p`. Each page of the augmented dimension of the array `p_par` will be updated by a different thread. At the end of the distributed loop, the reduction of the pages of `p_par` in `p` is very easily performed with the use of the intrinsic function `SUM`.

```

real*8, allocatable :: p_par(:, :, :, :)
allocate(p_par(0:n_r,n_theta,n_phi,
& 1:num_parthds()))

p_par = 0.
!$OMP parallel do private(weight_l,r_l,
& theta_l,phi_l,j_r,j_theta,
& j_phi,il)
& shared(p_par)
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)
il = mod(l,num_parthds())
if(il.eq.0) il=num_parthds()
p_par(j_r,j_theta,j_phi,il)=
& p_par(j_r,j_theta,j_phi,il) +
& f_4(weight_l,r_l,theta_l,phi_l)
enddo
p(0:n_r,n_theta,n_phi) =
& SUM(p_par(0:n_r,n_theta,n_phi,:),)
& dim=4)

```

In this way the update of the shared elements of `p` by each thread is decoupled into accesses to data (the pages of `p_par`), which, even though shared in principle, are in fact exclusively accessed by each thread. There is thus no need to protect the critical section, with obvious benefits in terms of performance. The strategy has, as a counterpart, drawbacks in terms of memory occupancy.

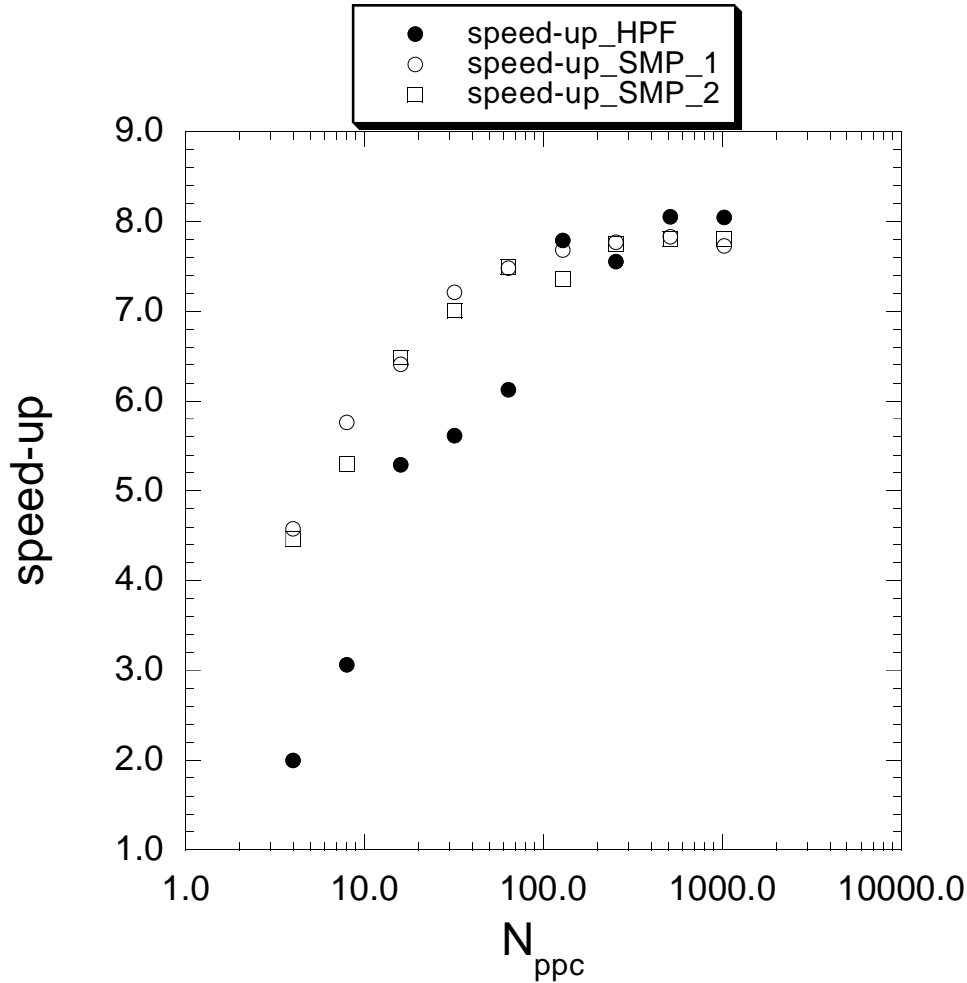


Figure 1: Speed up obtained for the three different parallelizations (distributed memory (HPF), shared memory with critical regions (SMP1) and with replicated pressure quantities (SMP2), at fixed number of processors (= 8) and at varying the average number of particle per cell N_{ppc} .

4 Experimental Results

We have tested the parallelization strategy presented above on a skeletonized version of a PIC code, which exhibits approximately the relative weight among the three computational phases as the Hybrid MHD-Gyrokinetic Code [4] – a code for the investigation of Alfvénic turbulence in plasmas magnetically confined in toroidal devices. The skeletonized version implements all the features relevant for the parallelization issues described above. We have developed an OpenMP version, and compared it with a version for distributed memory architectures, developed in High Per-

formance Fortran [4]. The developed HPF and OpenMP codes have been tested on a IBM SP parallel system, equipped with 2 8-nodes SMP RISC processors and with 16 Power2 RISC processors, each with clock frequency of 160 MHz, 512 MB RAM and 9.1GB HD.

The OpenMP code has been compiled by the IBM *xlf* (ver. 5.1) compiler (an optimized native compiler for Fortran95 with OpenMP extensions for IBM SMP systems).

The HPF code has been compiled by the IBM *xlhpfc* compiler (an optimized native compiler for IBM SP systems).

The results refer to executions on a spatial

grid with `n_r=32` cells in the radial direction, `n_theta=16` cells in the poloidal direction, and `n_phi=8` cells in the toroidal one. We performed several executions by varying the average number of particles per cell N_{ppc} (ranging from $N_{ppc} = 4$ to $N_{ppc} = 1024$, corresponding to a total number of particles ranging from $16K$ to $4M$).

Figure 1 shows the scaling of the speed-up with respect to the different values of the average number of particles per cell, for the parallel codes implementing the three different parallelization strategies depicted above: the one for distributed memory implemented using HPF, and the two versions for shared memory implemented using OpenMP (with critical regions (SMP1) and with replicated pressure quantities (SMP2)). The number of processors has been fixed in all cases to 8.

Both the scalability and the absolute value of the speed-up improve with increasing N_{ppc} , for all the cases. The two versions of the shared memory implementation do present only negligible differences for all the values of N_{ppc} ; this result shows that the use of critical sections in the first version does not actually represent a real bottleneck for this computation. One possible reason for this behaviour is the low frequency of concurrent writing accesses to the pressure quantities by the computing threads. This result motivates us to utilize the first strategy, based on critical sections, for the porting of the real code, due to the disadvantages of the second strategy in terms of code restructuring and of memory occupancy.

A comparison of the speed-up plots of the distributed and shared memory versions shows that, for low values of N_{ppc} , the shared memory version performs much better, while the speed-ups are substantially equivalent for high values of N_{ppc} .

This behaviour is explainable by the communication overhead, and the replicated computation of the grid quantities, in the distributed memory version; this effect is relevant for small data sizes, while becomes negligible for large data sizes.

References

- [1] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation* (McGraw-Hill, New York, 1985).
- [2] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, M. Vanneschi, "The P^3L Language: an Introduction", Technical Report HPL-PSC-91-29, Hewlett-Packard Laboratories, Pisa Science Centre, Dec. 1991.
- [3] J. Darlington, A.J.Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu and R.L.Whie, "Parallel Programming Using Skeleton Functions", in PARLE'93, LNCS 694, pp. 146-160, Springer-Verlag, 1993.
- [4] B. Di Martino, S. Briguglio, G. Vlad, and P. Sguazzero, "Parallel plasma simulation in High Performance Fortran", In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking, International Conference and Exhibition, Amsterdam, The Netherlands, April 1998, Proceedings, Lecture Notes in Computer Science 1401*, page 203. Springer, 1998.
- [5] Extension for threads (1003.1c-1995) in: "Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface (API)", IEEE/ANSI Std 1003.1, 1996 Edition (ISBN 1-55937-573-6).
- [6] High Performance Fortran Forum, *High Performance Fortran Language Specification, Scientific Programming 2 (1-2) (1993) 1-170*.
- [7] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, Rice University, 1997.
- [8] I. Foster and K. M. Chandy, "Fortran M: a Language for Modular Parallel Programming", *Journal of Parallel and Distributed Computing*, 25(1), 1995.

- [9] I. Foster, R. Olson and S. Tuecke, "Productive Parallel Programming: the PCN Approach", *Scientific Programming*, 1(1), 1992.
- [10] A. Geist et al.. PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, 1994.
- [11] Message Passing Interface Forum. Document for a Standard Message-Passing Interface. Technical Report No. CS-93-214, University of Tennessee, 1994.
- [12] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface" ver. 1.0, October 1997.
- [13] H. Richardson, High Performance Fortran: history, overview and current developments, Tech. Rep. TMC-261, Thinking Machines Corporation, 1996.