1 A Grid based distributed simulation of Plasma Turbulence

Beniamino Di Martino and Salvatore Venticinque
Dip. Ingegneria dell'Informazione, Seconda University of Naples, Italy

Sergio Briguglio, Giuliana Fogaccia and Gregorio Vlad
Associazione EURATOM-ENEA sulla Fusione, C.R. Frascati, C.P. 65, 00044, Frascati, Rome, Italy

Grid technology is widespreading, but most grid-enabled applications just exploit shared storage resources rather than computational ones, or utilize static remote allocation mechanisms of Grid platforms. In this paper the porting on a Globus equipped platform of a hierarchically distributed-shared memory parallel version of an application for particle-in-cell (PIC) simulation of plasma turbulence is described, based on the hierarchical integration of MPI and OpenMP, and originally developed for generic (non Grid) clusters of SMP nodes.

1.1 INTRODUCTION

Grid technology is gaining more and more widespread diffusion within scientific community. Despite the original motivations behind many Grid initiatives, that is sharing and cooperatively using computational resources scattered through the globe in a transparent way with respect to their physical location, most grid-enabled applications just exploit shared storage resources rather than computational ones, or utilize static remote allocation mechanisms in order to transparently select and allocate sequential or parallel tasks, running, in any case, on a single grid node. Nowadays most Grid platforms are able to present computational and storage resources spread all over the world and managed by different entities, as they were a single, virtually homogeneous, parallel machine. Parallel tasks and applications could, at least in principle, be managed and executed over widespread, heterogeneous platforms.

Most of the existing Grid platforms are built by using the Globus Toolkit, an open source software toolkit developed by the Globus Alliance and many other groups all over the world [11]. The Globus Environment includes MPICH-G [9], a complete implementation of MPI for heterogeneous, wide-area environments, based on MPICH [12]. Built on top of Globus_io and native MPI, MPICH-G allows the communications among multiple machines connected across a wide-area network in order to couple them in a single execution.

Aim of this work is testing the viability of this approach on real large-scale parallel applications, by porting on a Globus-equipped platform a hierarchically distributed-shared memory parallel version of an application for particle-in-cell (PIC) simulation of plasma turbulence, based on the hierarchical integration of MPI and OpenMP [2], originally developed for generic (non Grid) clusters of SMP nodes. To this aim, a cluster of SMP workstations has been configured as a distributed system of single-node SMP Globus machines, representing a lowest-order approximation of a more complex geographically-distributed, heterogeneous Grid.

The paper proceeds as follows. In Sect. 1.2 we describe the inter-node, workload decomposition strategy, adopted in the distributed-memory context, along with its MPI implementation, while the integration of such inter-node strategy with the intra-node (shared-memory) one, implemented by OpenMP, is discussed in Sect. 1.3. In Sect. 1.4 the testing platform is described and experimental results are provided. Final conclusions are discussed in the last section.

1.2 MPI IMPLEMENTATION OF THE INTER-NODE DOMAIN DECOMPOSITION

Particle-in-cell simulation consists [1] in evolving the coordinates of a set of N_{part} particles in certain fluctuating fields computed (in terms of particle contributions) only at the points of a discrete spatial grid and then interpolated at each particle (continuous) position. Two main strategies have been developed for the workload decomposition related to porting PIC codes on parallel systems: the *particle decomposition* strategy [6] and the *domain decomposition* one [10, 7]. Domain decomposition consists in assigning different portions of the physical domain and the corresponding portions of the spatial grid to different processes, along with the particles that reside on them. Particle decomposition, instead, statically distributes the particle population among the processes, while assigning the whole domain (and the spatial grid) to each process. As a general fact, the particle decomposition is very efficient and yields a perfect load balancing, at the expenses of memory overheads. Conversely, the domain decomposition does not require a memory waste, while presenting particle migration between different portions of the domain, which causes communication overheads and the need for dynamic load balancing [4, 7].

Such workload decomposition strategies can be applied both for distributed-memory parallel systems [7, 6] and shared-memory ones [5]. They can also be combined, when porting a PIC code on a hierarchical distributed-shared memory system (e.g., a cluster of SMPs), in two-level strategies: a distributed-memory level decomposition (among the n_{node} computational nodes), and a shared-memory one (among the n_{proc} processors of each node).

In this paper we consider the combination of the domain-decomposition strategy at the inter-node level (described in the present Section) with the particle-decomposition strategy at the intra-node level (cfr. Sect. 1.3).

The typical structure of a PIC code for plasma particle simulation can be represented as follows. At each time step, the code

1. computes the electromagnetic fields only at the N_{cell} points of a discrete spatial grid (*field solver* phase);
2. interpolates the fields at the (continuous) particle positions in order to evolve particle phase-space coordinates (*particle pushing* phase);
3. collects particle contribution to the pressure field at the spatial-grid points to close the field equations (*pressure computation* phase).

We can schematically represent the structure of this time iteration by the following code excerpt:

```
call field_solver(pressure,field)
call pushing(field,x_part)
call compute_pressure(x_part,pressure)
```

Here, `pressure(1:n_cell)`, `field(1:n_cell)` and `x_part(1:n_part)` (with `n_cell`$= N_{cell}$ and `n_part`$= N_{part}$) represent pressure, electromagnetic-field and particle-position arrays, respectively. In order to simplify the notation, we will refer, in the pseudo-code excerpts, to a one-dimensional case, while the experimental results reported in the following refer to a three-dimensional (3-D) application.

In implementing a parallel version of the code, according to the distributed-memory domain-decomposition strategy, different portions of the physical domain and of the corresponding spatial grid are assigned to the n_{node} different nodes, along with the particles that reside on them. This approach yields benefits and problems that are complementary to those yielded by the particle-decomposition one [6]: on the one hand, the memory resources required to each node are approximately reduced by the number of nodes (`n_part`$\sim N_{part}/n_{node}$, `n_cell`$\sim N_{cell}/n_{node}$); an almost linear scaling of the attainable physical-space resolution (i.e., the maximum size of the spatial grid) with the number of nodes is then obtained. On the other hand, inter-node communication is required to update the fields at the boundary between two different portions of the domain, as well as to transfer those particles that migrate from one domain portion to another. Such a particle migration possibly determines a severe load unbalancing of the different processes, then requiring a dynamic balancing, at the expenses of further computations and communications.

Three additional procedures then characterize the structure of the parallel code: at each time step

- the number of particles managed by a process has to be checked, in order to avoid excessive load unbalancing among the processes (if such an unbalancing is verified, the load-balancing procedure must be invoked);

- particles that moved from one subdomain to another because of particle pushing must be transferred from the original process to the new one;
- the values of the pressure array at the boundaries between two neighbor subdomains must be corrected, because their local computation takes into account only those particles which belong to the subdomain, neglecting the contribution of neighbor subdomain's particles.

Let us report here the schematic representation of the time iteration performed by each process, before giving some detail on the implementation of such procedures:

```
 call field_solver(pressure,field)
 call check_loads(i_check,n_part,n_part_left_v,
&    n_part_right_v)
 if(i_check.eq.1)then
  call load_balancing(n_part_left_v,n_part_right_v,
&     n_cell_left,n_cell_right,n_part_left,n_part_right)
  n_cell_new=n_cell+n_cell_left+n_cell_right
  if(n_cell_new.gt.n_cell)then
   allocate(field_aux(n_cell))
   field_aux=field
   deallocate(field)
   allocate(field(n_cell_new))
   field(1:n_cell)=field_aux(1:n_cell)
   deallocate(field_aux)
  endif
  n_cell=max(n_cell,n_cell_new)
  n_cell_old=n_cell
  call send_receive_cells(field,x_part,
&     n_cell_left,n_cell_right,n_part_left,n_part_right)
  if(n_cell_new.lt.n_cell_old)then
   allocate(field_aux(n_cell_old))
   field_aux=field
   deallocate(field)
   allocate(field(n_cell_new))
   field(1:n_cell_new)=field_aux(1:n_cell_new)
   deallocate(field_aux)
  endif
  n_cell=n_cell_new
  n_part=n_part+n_part_left+n_part_right
 endif
 call pushing(field,x_part)
 call transfer_particles(x_part,n_part)
 allocate(pressure(n_cell))
 call compute_pressure(x_part,pressure)
 call correct_pressure(pressure)
```

In order to avoid continuous reallocation of particle arrays (here represented by x_part) because of the particle migration from one subdomain to another, we overdimension (e.g., +20%) such arrays with respect to the initial optimal-balance size, N_{part}/n_{node}. Fluctuations of n_part around this optimal size are allowed within a certain band of oscillation (e.g., ± 10%). This band is defined in such a way to prevent, under normal conditions, index overflows and, at the same time, to avoid excessive load unbalancing. One of the processes (the MPI rank-0 process) collects, in subroutine check_loads, the values related to the occupation level of the other processes and checks whether the band boundaries are exceeded on any process. If this is the case, the "virtual" number of particles (n_part_left_v, n_part_right_v) each process should send to the neighbor processes to recover the optimal-balance level is calculated (negative values means that the process has to receive particles), and i_check is set equal to 1. Then, such informations are scattered to the other processes. These communications are easily performed with MPI by means of the collective communication primitives MPI_Gather, MPI_Scatter and MPI_Bcast. Load balancing is then performed as follows.

Particles are labelled (subroutine load_balancing) by each process according to their belonging to the units (e.g., the n_cell spatial-grid cells) of a finer subdivision of the corresponding subdomain. The portion of the subdomain (that is, the number of elementary units) the process has to release, along with the hosted particles, to neighbor subdomains in order to best approximate those virtual numbers (if positive) is then identified. Communication between neighbor processes allows each process to get the information related to the portion of subdomain it has to receive (in case of negative "virtual" numbers). Net transfer information is finally put into the variables n_cell_left, n_cell_right, n_part_left, n_part_right. Series of MPI_Sendrecv are suited to a deadlock-free implementation of the above described communication pattern.

As each process could be requested, in principle, to host (almost) the whole domain, overdimensioning the spatial-grid arrays (pressure and field) would cause losing of the desired memory scalability (there would be, indeed, no distribution of the memory-storage loads related to such arrays). We then have to resort to dynamical allocation of the spatial-grid arrays, possibly using auxiliary back-up arrays (field_aux), when their size is modified.

Portions of the array field have now to be exchanged between neighbor processes, along with the elements of the array x_part related to the particles residing in the corresponding cells. This is done in subroutine send_receive_cells by means of MPI_Send and MPI_Recv calls. The elements of the spatial-grid array to be sent are copied in suited buffers, and the remaining elements are shifted, if needed, in order to be able to receive the new elements or to fill possibly occurring holes. After sending and/or receiving the buffers to/from the neighbor processes, the array field comes out to be densely filled in the range 1:n_cell_new. Analogously, the elements of x_part corresponding to particles to be transferred are identified on the basis of the labelling procedure performed in subroutine load_balancing and copied into auxiliary buffers; the residual array is then compacted in order to avoid the presence of "holes" in the particle-index space. Buffers sent by the neighbor processes can then be stored in the higher-index part of the `x_part` (remember that such an array is overdimensioned).

After rearranging the subdomain, subroutine `pushing` is executed, producing the new particle coordinates, `x_part`. Particles whose new position falls outside the original subdomain have to be transferred to a different process. This is done by subroutine `transfer_particles`. First, particles to be transferred are identified, and the corresponding elements of `x_part` are copied into an auxiliary buffer, ordered by the destination process; the remaining elements of `x_part` are compacted in order to fill holes. Each process sends to the other processes the corresponding chunks of the auxiliary buffer, and receives the new-particle coordinates in the higher-index portion of the array `x_part`. This is a typical all-to-all communication; the fact that the chunk size is different for each destination process makes the `MPI_Alltoallv` call the tool of choice.

Finally, after reallocating the array `pressure`, subroutine `compute_pressure` is called. Pressure values at the boundary of the subdomain are then corrected by exchanging the locally-computed value with the neighbor process (subroutine `correct_pressure`), by means of `MPI_Send` and `MPI_Recv` calls. The true value is obtained by adding the two partial values. The array `pressure` can now be yielded to the subroutine `field_solver` for the next time iteration.

Note that, for the sake of simplicity, we referred, in the above description, to one-dimensional field arrays. In the real case we have to represent field informations by means of multi-dimensional arrays. This requires us to use MPI derived datatypes as arguments of MPI calls in order to communicate blocks of pages of such arrays.

1.3 INTEGRATION OF THE INTER-NODE DOMAIN DECOMPOSITION WITH INTRA-NODE PARTICLE DECOMPOSITION STRATEGIES

The implementation of the particle decomposition strategy for a PIC code at the shared-memory level in a high-level parallel programming environment like OpenMP has been discussed in Refs. [5, 3]. We refer the reader to those papers for the details of such implementation. Let us just recall the main features of this intra-node approach, keeping in mind the inter-node domain-decomposition context. It is easy to see that the particle pushing loop is suited for trivial work distribution among different processors. The natural parallelization strategy for shared memory architectures consists in distributing the work needed to update particle coordinates among different threads (and, then, processors), with no respect to the portion of the domain in which each particles resides. OpenMP allows for a straightforward implementation of this strategy: the `parallel do` directive can be used to distribute the loop iterations over the particles. With regard to the pressure loop, the computation for each update is split among the threads into partial computations, each of them involving only the contribution of the particles managed by the responsible thread; then the partial results are reduced into global ones. The easiest way to implement such a strategy consists in introducing an auxiliary array, `pressure_aux`, defined as a `private` variable with the same dimensions and extent as `pressure`. Each processor works on a separate copy of the array and there is no conflict between processors updating the same element of the array. At the end of the loop, however, each copy of `pressure_aux` contains only the partial pressure due to the particles managed by the owner processor. Each processor must then add its contribution, outside the loop, to the global, shared, array `pressure` in order to obtain the whole-node contribution; the `critical` directive can be used to perform such a sum. The corresponding code section then reads as follows:

```
       pressure(1:n_cell) = 0.
!$OMP parallel private(l,j_x,pressure_aux)
       pressure_aux(1:n_cell) = 0.
!$OMP do
       do l=1,n_part
        j_x = f_x(x_part(l))
        pressure_aux(j_x) = pressure_aux(j_x) + h(x_part(l))
       enddo
!$OMP end do
!$OMP critical (p_lock)
       pressure(:) = pressure(:) + pressure_aux(:)
!$OMP end critical (p_lock)
!$OMP end parallel
```

with `f_x` being the function which relates the (continuous) particle position to the cell index and `h` the pressure updating function which has associative and distributive properties with respect to the contributions given by every single particle. Note that this strategy, based on the introduction of an auxiliary array, makes the execution of the n_{part} iterations of the loop perfectly parallel. The serial portion of the computation is limited to the reduction of the different copies of `pressure_aux` into `pressure`.

The integration of the inter-node domain-decomposition strategy with the intra-node particle-decomposition one does not present any relevant problem. The only fact that should be noted is that, though the identification of particles to be transferred from one subdomain to the others can be performed, in subroutine `transfer_particles`, in a parallel fashion, race conditions can occur in updating the counters related to such migrating particles and their destination subdomains. Also the updating has then to be protected within `critical` sections.

1.4 THE MPICH-G2 IMPLEMENTATION

In this Section, we want to compare the parallel porting of the described PIC application on a "classical" hierarchical distributed-shared memory architecture by simple integration of MPI and OpenMP [2] and the parallel porting of the same application on a Grid environment. The system used as testbed for this comparison is the Cygnus cluster of the Parsec Laboratory, at the Second University of Naples. This is a Linux cluster with four SMP nodes equipped with two Pentium Xeon 1 Ghz, 512 MB RAM and a 40 GB HD, and a dual-Pentium Xeon 3.2GHz front end.

While the parallel porting on the "classical" architecture can be performed quite easily by resorting to the Argonne MPICH implementation of MPI (Gropp and Lusk) [12], a large effort is necessary to build up and configure a Grid programming environment in order to support the vendor's compiler and facilities. The most important among these facilities is represented by the Globus Toolkit, an open source software toolkit developed by the Globus Alliance and many other groups all over the world, widely used for building Grids [11]. The Globus Environment includes MPICH-G [9], a complete implementation of MPI for heterogeneous, wide area environments, based on MPICH. Built on top of Globus_io and native MPI, MPICH-G allows the communications among multiple machines connected across a wide area network in order to couple them in a single execution. It uses Globus_io services in order to convert data in messages sent between machines of different architectures, and selects TCP for wide area messaging and vendor-supplied MPI for cluster-of-workstations messaging. Existing parallel programs written for MPICH can be executed over the Globus infrastructure just after recompilation.

MPICH-G has been recently completely rewritten, giving rise to an upgraded version, MPICH-G2 [14], with greatly improved performances. While the communication functionality of MPICH is based on a communication device having a common Abstract Device Interface (ADI), MPICH-G2 uses a new device named globus2. As MPICH-G2 does not support shared memory communication, in order to exploit the shared-memory and multithreading features of our architecture, we integrate, also in the Grid-porting framework, the MPI facility with the OpenMP support provided by the Intel compiler.

For the experiments we report in this Section, we have installed, on our system, the ROCKS LINUX distribution supported by NPACI and the Globus Toolkit 2.0, integrated with MPCH-G2 v. 1.5.2. Building and installing the GT2 implementation of the Globus toolkit along with the Intel compiler, needed to exploit the OpenMP resource for SMP programming, did not prove to be a trivial task.

We considered three different configurations of the system:

1. a *classic* configuration: the SMP cluster, with no Globus environment;

2. a *single Globus machine* configuration, with four two-way nodes;

3. a *set of Globus machines* configuration, each equipped with a single two-way node.

The first of such configurations will represent the reference case of a parallel porting of the PIC application on a hierarchical distributed-shared memory architecture [2]: the application is executed by using MPICH built with the ch_p4 device. The second configuration uses a front end as a single Globus node for the submission of a single MPIch_p4 job on the 4-node machine. It corresponds to the entry-level exploitation of the Grid facilities for scientific computing: the assignment of a parallel job to a single remote Globus parallel machine. In the third configuration each of the four nodes hosts a Globus gatekeeper, which is able to run MPICH-G2 jobs. The MPICH-G2 request, submitted from the front end node, spreads one process per Globus node. This configuration is the most ambitious one, in the present investigation, as it allows

Table 1.1 Elapsed times (in seconds) for 3-D skeleton-code implementation at different pairs n_{node}/n_{proc} for the *classic* configuration.

	1/1	*1/2*	*2/1*	*2/2*	*3/1*	*3/2*	*4/1*	*4/2*
Load balancing $\times 10^{-2}$	0.000	0.001	7.50	13.22	18.27	30.14	10	16.14
Pushing	2.379	0.978	1.763	0.721	0.744	0.298	0.558	0.224
Compute pressure	1.967	1.043	0.905	0.479	0.577	0.309	0.526	0.223
Correct pressure $\times 10^{-2}$	0.007	0.007	1.15	0.943	0.999	0.975	0.783	0.52
Transfer particles	0.225	0.130	0.134	0.084	0.112	0.078	0.093	0.069

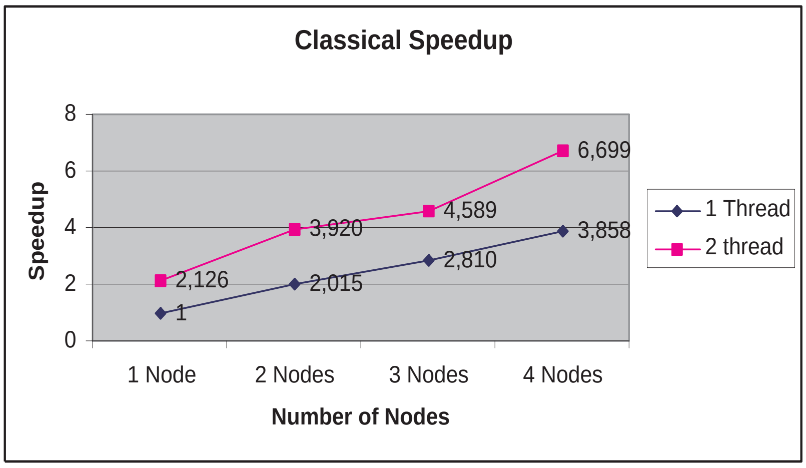

Fig. 1.1 Speedup for the *classical* configuration.

us to test the effective capability of the Grid environment in supporting distributed (not just remote) computation.

Table 1.1 shows, for the *classic*-configuration experiment, the elapsed time for the different phase of a single iteration (time step, in Sect. 1.2) of our application. The results corresponding to cases with different number of nodes and/or processors per node are reported. Here and in the following, we consider a moderate-size case, with a 3-D spatial grid of $128 \times 32 \times 16$ points and $128 \times 32 \times 16 \times 16$ particles.

Figure 1.1 shows the corresponding speed-up values measured with respect to the whole single iteration. It can be seen that the parallel porting obtained with integration of MPI and OpenMP is very efficient, as already found in Ref. [2]. The experiments performed with the system configured as a *single Globus machine* yield, as expected, very close figures for the single-iteration speed-up values. We also observe, however, in the whole simulation elapsed time, a contribution related to the time needed to

Table 1.2 Elapsed times (in seconds) for 3-D skeleton-code implementation at different pairs n_{node}/n_{proc} for the *set of Globus machines* configuration.

	1/1	*1/2*	*2/1*	*2/2*	*3/1*	*3/2*	*4/1*	*4/2*
Load balancing $\times 10^{-2}$	0.002	1,55	7,57	13,50	17,08	29.98	98.05	15.99
Pushing	2.376	1.11	1.114	0.525	0.749	0.342	0.556	0.255
Compute pressure	1.967	1.062	0.902	0.478	0.582	0.307	0.426	0.222
Correct pressure $\times 10^{-2}$	0.008	0.007	1.51	0.96	1.09	0.70	0.95	0.40
Transfer particles	4.567	2.31	2.27	1.238	1.617	1.032	1.18	0.709

process the request submitted from the front end. Such a contribution is contained in 8-10 sec., with no significant dependence on the simulation size or the number of nodes effectively used by the single Globus machine. The results obtained with the *set of Globus machines* configuration are reported, for the single iteration time and speed-up values in Tab. 1.2 and Fig. 1.2 respectively. The qualitative features observed with the previous configurations (efficient use of the SMP architecture by means of OpenMP and drop of efficiency in the less symmetric cases) are obtained also in this case.

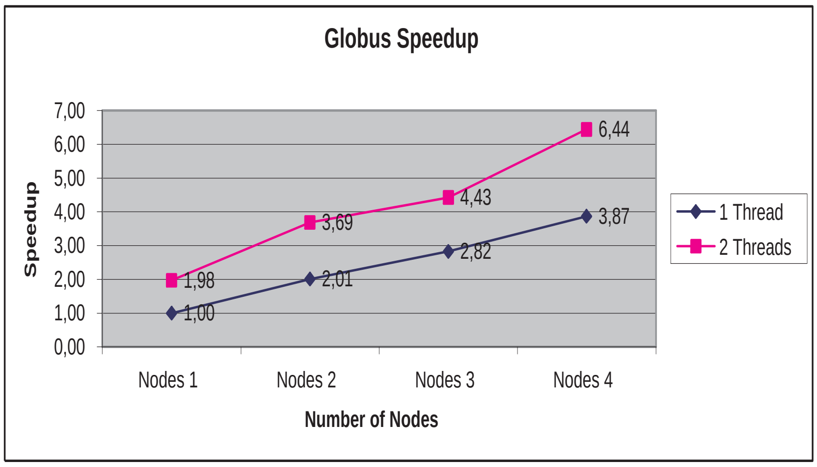

Fig. 1.2 Speedup for the *set of Globus machines* configuration.

From a quantitative point of view, the efficiency of this Grid parallel porting is still fairly good, though it has to be stressed that we are treating with a rather ideal Grid (no geographical spreading, no network problems). Similarly to the *single Globus machine* case, an offset is revealed in the elapsed whole-simulation time, related to the submission of the request from the front end to each gatekeeper, as well as to

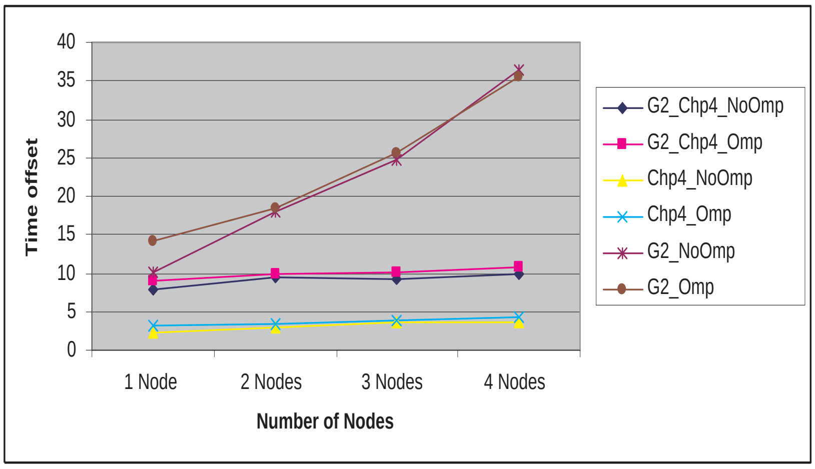

Fig. 1.3 The time offset versus the number of Globus nodes

starting all the processes and, later on, releasing the reserved resources. As before, the offset does not show any relevant dependence on the simulation size; in this case, however, it increases with the number n_G of Globus machines, as it is shown in Fig. 1.3. Note that, for realistic simulations (thousands of iterations and much larger number of particles and spatial-grid points), such offset would be in fact negligible, as far as not too large sets of Globus machines (and not too large number of nodes per machine) are considered.

Finally, we check how worth is the effort aimed to allow the integration of OpenMp and MPICH-G2 (through the resorting to the Intel compiler) in this *set of Globus machines* configuration. Instead of exploiting the SMP facility, we can execute two MPI processes on each node. For this eight MPICH-G2 process case, we get a speed-up of 6.07, to be compared with the value (6.44) obtained in the OpenMP case.

1.5 CONCLUSIONS

In this paper we have addressed the issue of distributing parallel computation among the Globus machines of a Grid environment. The case of a scientific application – namely, a particle-in-cell one – has been considered. A cluster of workstation was configured as a distributed system of four single-node SMP Globus machines, as a lowest-order approximation of a more complex geographically-distributed, heterogeneous Grid. Installing the MPICH-G2 library and the Intel Fortran compiler on the cluster allowed us to port a parallel version of the application, based on the hierarchical integration of MPI and OpenMP, originally developed for generic (non Grid) clusters of SMP nodes. We were then able to compare the performances obtained on the Grid environment with those obtained with the simple cluster of SMP nodes (same hardware). We find that the parallelization efficiency is slightly reduced, though maintaining a fairly high level.

Though the integration of the Globus toolkit and the Intel compiler was not a straightforward task, the related effort is a valuable one, as the Intel compiler allows us to use the OpenMP facility and to fully exploit the SMP architecture of each node. Replacing OpenMP programming with the execution of as many MPI processes as the processors owned by each node yields indeed lower efficiency values.

Finally, the time offsets due to the submission of the request to the gatekeepers, to the activation of the processes and to releasing of the resources, though scaling with the number of Globus machines, appear to be negligible in comparison to realistic-simulation elapsed times.

These conclusions are partly affected, of course, by the almost-ideal character of the Grid environment considered (close, homogeneous, dedicated nodes): real Grid porting of the same application, which will be the subject of future work, could reveal much less efficient. Nonetheless, such a porting, whose feasibility we have here demonstrated, would maintain its value with respect to the task of increasing memory resources (and, then, the achievable simulation size).

References

1. Birdsall, C.K., Langdon, A.B.: Plasma Physics via Computer Simulation. (McGraw-Hill, New York, 1985).

2. Briguglio, S., Di Martino, B., Fogaccia, G., Vlad, G.: Hierarchical MPI+OpenMP implementation of parallel PIC applications on clusters of Symmetric MultiProcessors. Lecture Notes in Computer Science 2840, pages 180-187. Springer, 2003.

3. Briguglio, S., Di Martino, B., Vlad, G.: Workload Decomposition Strategies for Hierarchical Distributed-Shared Memory Parallel Systems and their Implementation with Integration of High Level Parallel Languages. Concurrency and Computation: Practice and Experience, Wiley, Vol. **14**, n. 11, (2002) 933–956.

4. Cybenko, G.: Dynamic Load Balancing for Distributed Memory Multiprocessors. J. Parallel and Distributed Comput., **7**, (1989) 279–391.

5. Di Martino, B., Briguglio, S., Vlad, G., Fogaccia, G.: Workload Decomposition Strategies for Shared Memory Parallel Systems with OpenMP. Scientific Programming, IOS Press, **9**, n. 2-3, (2001) 109–122.

6. Di Martino, B., Briguglio, S., Vlad, G., Sguazzero, P.: Parallel PIC Plasma Simulation through Particle Decomposition Techniques. Parallel Computing **27**, n. 3, (2001) 295–314.

7. Ferraro, R.D., Liewer, P., Decyk, V.K.: Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code, J. Comput. Phys. **109**, (1993) 329–341.

8. Foster, I. (2001). The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science 2150*

9. Foster, I. and Karonis, N. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In Proc. SC'98, 1998.

10. Fox, G.C., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D.: Solving Problems on Concurrent Processors (Prentice Hall, Englewood Cliffs, New Jersey, 1988).

11. Globus: A Metacomputing Infrastructure Toolkit. I. Foster, C. Kesselman. Intl J. Supercomputer Applications, 11(2):115-128, 1997. Provides an overview of the Globus project and toolkit.

12. Gropp, B., Lusk, R., Skjellum, T., and Doss, N. "Portable MPI Model Implementation," Argonne National Laboratory, July 1994.

13. MPI: A message-passing interface standard. International Journal of Supercomputer Application, 8((3/4)): 165-416, 1994.

14. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. N. Karonis, B. Toonen, and I. Foster. Journal of Parallel and Distributed Computing, 2003.

15. OpenMP Architecture Review Board,"OpenMP Fortran Application Program Interface" ver. 1.0, October 1997.

16. The PVM Concurrent Computing System: Evolution, Experiences and Trends, V. Sunderam, J. Dongarra, A. Geist, and R Manchek, Parallel Computing, Vol. 20, No. 4, April 1994, pp 531-547.