Integration of Mobile Agents and OpenMP for programming heterogeneous clusters of Shared Memory Processors: a case study*

Rocco Aversa, Beniamino Di Martino, Nicola Mazzocca,
Massimiliano Rak and Salvatore Venticinque
Dip. Ingegneria dell'Informazione, Second University of Naples, Italy
beniamino.dimartino@unina.it

Abstract *In this paper we report our experience in integrating a Mobile agent framework and an OpenMP compiler for programming hierarchical distributed-shared memory multiprocessor architectures, in particular heterogeneous clusters of SMPs (and uniprocessor) nodes.*

We show, through a case-study, how the adoption of the mobile agent model (with added dynamic workload balancing feature) for the distributed parallelism and OpenMP parallelizing compiler technology for the inner shared memory parallelism, allows to yield a hierarchically distributed-shared memory implementation of an algorithm presenting multiple levels of parallelism, and to exploit the heterogeneity of the target computing architecture. The chosen application solves the well-known N-body problem.

1 Introduction

Clusters of Bus-based shared memory multiprocessor systems (SMPs), where moderately sized multiprocessor workstations and PCs are connected with a high-bandwidth interconnection network, are gaining more and more importance for High Performance Computing. They are increasingly established and used to provide high performance computing at a low cost.

*This work has been supported by the Italian Ministry for University and Research (MURST) (P.R.I.N. Project *ISIDE* - "Dependable reactive computing systems for industrial applications").

Current parallel programming models are not yet designed to take into account hierarchies of both distributed and shared memory parallelism into one single framework. Programming hierarchical distributed-shared memory systems is currently achieved by means of integration of environments/languages/libraries individually designed for either shared or distributed address space model. They range from explicit message-passing libraries such as MPI (for the distributed memory level) and explicit multithreaded programming (for the shared memory level), at a low abstraction level, to high-level parallel programming environments/languages, such as High Performance Fortran (HPF) [2] (for the distributed memory level), and OpenMP [3] (for the shared memory level).

A crucial issue in programming clusters of SMPs is the inherent heterogeneity of this architectural model.

The Mobile Agents technology [11] has the potential to provide a flexible framework to effectively program heterogeneous distributed architectures. Several characteristics of potential benefit for heterogeneous distributed architectures can be provided by the adoption of the mobile agent technology, as shown in the literature [6, 7, 8]: they range from network load reduction, heterogeneity, dynamic adaptivity, fault-tolerance to portability. Mobile agents meet the requirements of the heterogeneous distributed computing since the agents are naturally heterogeneous, they reduce the network load and overcome network latency by means of the mechanism of the migration, they adapt dynamically to the computing platform through the migration and cloning, and finally the different tasks of a sequential algorithm can be embedded into different mobile agents thus simplifying the parallelization of the sequential code.

In this paper we report our experience in integrating a Mobile agent framework and an OpenMP compiler for programming hierarchical distributed-shared memory multiprocessor architectures, in particular heterogeneous clusters of SMPs (and uniprocessor) nodes.

We show, through a case-study, how the restructuring of a sequential code implementing an irregular algorithm, with adoption of the mobile agent model for the distributed parallelism and OpenMP parallelizing compiler technology for the inner shared memory parallelism, allows to yield a hierarchically distributed-shared memory parallel version of the algorithm without completely rethinking its structure, reusing a great deal of the sequential code and trying to exploit the heterogeneity of the target computing architecture. The chosen application solves the well-known N-body problem.

We adopted a Mobile Agent programming framework developed by the IBM Japan research center, the *Aglet mobile agents Workbench* [9]. For the inner shared memory level parallelization we adopted the JOMP compiler developed by the EPCC research center of the University of Edinburgh [5]. We have augmented the Aglet Workbench by providing (1) an extension of the *multicast* communication primitive, and (2) by designing and implementing a *dynamic workload balancing* service. In this paper we describe how our technique for dynamical workload balancing is extended to handle multithreaded agents.

The paper is structured as follows. Section 2 describes the Aglet mobile agents Workbench, the JOMP compiler, their integrated usage, and a dynamical workload balancing technique, to be applied, via a proposed directive, to data parallel `for` constructs of multithreaded agents. In Section 3 we describe the integrated application of Agents and OpenMP, with dynamic balancing, to the case study N-body application. It is obtained by means of the integrated use of the Aglet workbench and JOMP compiler. Section 4 describes experimental results obtained with an heterogeneous cluster of three biprocessor Intel PCs and two 8-processor PowerPC nodes. Conclusions are drawn in section 5.

2 Integration of Mobile Agents and OpenMP

The Mobile Agents and high-level shared memory programming paradigms can be coupled in order to express a hierarchical (two-level) parallelism: an external distributed memory level, and an internal shared memory one. The interacting agents' execution model perform the external distributed memory parallelism; internal shared memory parallelism can be achieved within each agent's execution, and can be expressed through OpenMP directives.

We have obtained such a coupling through the integration of a framework for programming mobile agent applications and OpenMP compiler technology. In particular we have utilized the IBM Aglet workbench, (augmented with collective communication and dynamical load balancing features) and the JOMP Compiler.

The Aglet Workbench is a framework for programming mobile networks agents in Java developed by IBM Japan research group [9]. An *aglet* (agent applet) is a lightweight Java object that can move to any remote host that supports the Java Virtual Machine. An Aglet server program (*Tahiti*) provides the agents with an execution environment that allows for an aglet to be created and disposed, to be halted and dispatched to another host belonging to the computing environment, to be cloned, and, of course, to communicate with all the other aglets. We augmented the Aglet Workbench by providing (1) an extension of the *broadcast* communication primitive, and

(2) by designing and implementing a *dynamic workload balancing* service. These extensions are described in [4].

The JOMP Compiler [5] has been developed by the EPCC research center of the University of Edinburgh. It is an OpenMP compiler for the Java language, augmented with OpenMP like directives and methods, proposed by the authors. It provides with an implementation of most OpenMP directives for parallel regions, synchronization and mutual exclusion among threads. It provides a runtime library, in the form of a Java class library.

The integrated use of JOMP within the Aglet workbench is straightforward: The JOMP compiler is to be used just as a preprocessor for the Aglet Workbench, for the Java agent classes whose methods contains OMP annotations. More specifically, it is needed to:

- write the agent class containing OMP directives in a file `FileName.jomp`
- import in it the jomp.runtime library;
- compile the jomp file with the command: `java jomp.compiler.Jomp FileName`
- compile the file `FileName.java` obtained in the last step using the javac compiler;
- recall the FileName agent class in the thaiti console to create the agent.

The above described coupling mechanism can be applied without any limitation to the utilization of OpenMP directives. Nevertheless, its straigthforward application does not allow to apply the dynamic load balancing service offered by the augmented Aglet framework, at least during the execution of the parallel sections.

This because the workload assigned to each thread cannot be rebalanced, or anyhow modified, by the load balancing service. A more aggressive strategy, based on definition of an ad hoc directive and code preprocessing, is needed in order to obtain a dynamical workload reassignment among threads belonging to different agents.

In the following we describe how our technique for dynamical workload balancing is extended to handle multithreaded agents. We have explored at the moment the application of the technique only to data parallel computations, namely `for` constructs whose iterations do not present loop-carried dependencies (exactly the same conditions under which the OpenMP `parallel for` directives can be applied). A `BALANCED FOR` directive has been devised, which is applied by the user to non-nested `for` constructs:

```
// OMP BALANCED FOR
for(j=first_index;j<last_index;j++)
{
   ...
   A[j] = ...
   ...
}
```

The coordinator agent of the balancing framework manages a pool of loop iteration chunks of the loop iteration space. The coordinator agent assignes chunks to each worker agent, which either starts the computation of the loop, or asks for new workload. Each worker agent performs the computation corresponding to the assigned chunk, possibly in parallel through multiple threads; when it finishes, the framework asks for new workload to the coordinator, till workload is available.

More specifically, the `for` construct is preprocessed and transformed, within the worker's code, as follows:

```
do while(balance( ))
{
  // OMP PARALLEL FOR
  for(j=my_first_index;j<my_last_index;j++)
  {
    ...
    A[j] = ...
    ...
  }
  broadcast(A[my_first_index:my_last_index])
}
```

When the `balance` procedure is called:

- the agent is stopped;
- the framework asks for an iteration chunk to the coordinator;

- the framework wait for the coordinator answer;
- if there is load available the coordinator send a new iteration chunk; the framework updates `my_first_index` and `my_last_index` values;
- the framework restarts the worker;
- the balance procedure returns true if new load was available from the coordinator, false otherwise.

Note that the inner `for` loop is annotated with a standard OpenMP `parallel for` directive, in order to be possibly computed in a multithreaded way by the worker agent.

3 A case-study: the parallelization of the N-body application using Mobile Agents and OpenMP

The chosen case-study is a sequential algorithm that solves the N-body problem by computing, during a fixed time interval, the positions of N bodies moving under their mutual attraction. The algorithm is based on a simple approximation: the force on each particle is computed by agglomerating distant particles into groups and using their total mass and centre of mass as a single particle. The program repeats, during a fixed time interval, three main steps: to build an octal tree (*octree*) whose nodes represents groups of nearby bodies; to compute the forces aging on each particle through a visit in the octree; to update the velocities and the positions of the N particles.

During the first step, the algorithm, starting from the N particles, builds the *octree* storing its nodes in an array using a depth-first strategy. At each level in the *octree* all nodes have the same number of particles until each leaf remains with a single particle. The subsequent level in the *octree* can be obtained by repeatedly splitting in two halves the particles in each node on the basis of the three spatial coordinates (x,y,z). At the end of this stage a tree node represents a box in the space characterized by its position (the two distant corners of the box) and by the total mass and the center of mass of all the particles enclosed in it. A leaf of the tree obviously coincides with a box containing a single particle.

The second stage of the algorithm computes the forces on each of the N particles by visiting the above built *octree*. Each particle traverses the tree. If the nearest corner of the box representing the current node is sufficiently distant, the force due to all particles is computed from the total mass and centre of mass of the box. Otherwise the search comes along the lower levels of the tree, if necessary, till the leaves. Thus, distant particles contributions are taken in account by looking at nodes high in the octree; nearby particles contributions by looking at leaves. Once obtained the force components affecting the particles, it's possible to update their velocities and positions. The last step computes respectively the new velocities and the new positions of the N particles using a simple Euler step. A first analysis of the sequential algorithm suggests some preliminary considerations that can drive the parallelizzation strategy:

- The first step of the algorithm (the building of the *octree*) appears to be a typical *master-slave* computation: a master reads input data and starts building the first levels of the octal tree until the elaboration can proceed in parallel;
- The forces computation step can be classified as a *data parallel* computation since each particle can calculate the force aging on it in an independent way. However it's worth while noting that this task, that is even the heaviest one of the algorithm, cannot be statically balanced since the elaboration requirements are strongly data-dependent (the visit of each particle can stop to different levels of the octree);
- Finally, the computation steps that updates the particles velocities and posi-

tions can be still recognized as a *data parallel* portion of the code but that can be parallelizzable and statically balanced in a trivial way.

The considerations about the parallel nature of the different phases of the sequential algorithm and the functionalities of the programming environment, described above, leads us to obtain in a straightforward manner a cost-effective distributed version of the N-body application (see the pseudo code of the Aglet in 3).

The parallelization of the code relative to the construction of the *octree*, needs that the reading of the input data and the production of the first level in the octree is carried out by a single *master* agent. As soon as the nodes in the current level of the *octree* exceeds in number the computing nodes, the *master* agent cloned itself and dispatches the clones to each host making up the computing environment. Every agent, the *master* and the *slaves*, is responsible for the building of the subtrees assigned to it, an operation that can be carried out in parallel. At the end of this stage every agent has filled up a slice of the complete octree data structure and can send a remote *multicast* message to all the other agents so that each of them is able to get a complete copy of the *octree* (*join_subtree*). In the not trivial parallelization of this step of the algorithm we exploit the flexibility of explicit programming using agents. In this way, for example, it's simple to take in account the heterogeneity of the target architecture assigning to each clone a number of subtrees proportional to the computing power of the node that hosts it.

On the contrary, the computation stage of the force acting on a single particle (*compute_forces*()), as well as the update of its velocity and position (*compute_new_speed&position*()) results to be completely independent. Each of the N particles, in fact, computes the force aging on it by traversing separately the *octree*, just as the updating phase of the particles velocities and the positions requires the same computation on different data. So, a parallelization can be obtained by simply distributing the particles among the agents. The only critical difference between the two data-parallel portion of the code is that the force computation stage can be nowise balanced by statically assigning an appropriate number of particles to the nodes. In this case it's immediate to write a multithread version of forces computation stage by inserting the `BALANCED FOR` directive inside the agent code(see fig. 3), so to exploit the availability of nodes with multiple processors, without giving up the benefits of the dynamic workload balancing infrastructure available in our programming framework As side effect, the `BALANCED FOR` directive distribute the results of the computation (the *forces* array) among all the agents. On the contrary, a multithread version of the positions and velocities updating stage of the agent code can be simply obtained using the standard OMP directive (see fig. 3). Finally. at the end of their work the agents, by means of a new *all-to-all* multicast message, obtain an updated copy of the particles velocities and positions so that the next iteration step of the algorithm can start (*join_subsets()*). It is worth while underlining that the original sequential routines with the appropriate input parameters can be completely reused becoming different methods of the agent code.

4 Experimental results

The tests we report in this section have been carried out on two different systems: an IBM SP parallel system, equipped with two 8-processors SMP PowerPC nodes, with clock frequency of 200 MHz and 2 GB RAM and an heterogeneous cluster of three 2-processors SMP composed by two Pentium II with clock frequency of 350 MHz and 256 MB RAM and a Pentium III, 733 MHz and 64MB RAM (the biprocessor PCs cluster). In order to understand the effectiveness, under a performance aspect of the OpenMP/Mobile agents integration methodology, the case-study application has been tested, both for balanced and unbalanced versions, comparing the resulting speed-ups with the ideal maximum speed-up.

```
import com.ibm.aglets*;
import jomp.runtime.*;

public class NbodyAgletpar extend Aglet{

public void onCreation()
{
Split_agents();
      Dispatch_agents();
}

public void run()
{
   OMP.setNumThreads(n);
   for(i=0;i<n_iterations;i++)
   {
     create_subtree_of_particles();
     join_subtree();
     compute_subsets();
     compute_new_speed&positions();
     join_ subsets();
   }
}
}
```

Figure 1: The pseudo code of the Agent.

```
compute_subset_of_forces()
{
// OMP BALANCED FOR
for(int i=first_particle;
    i<last_particle;i++)
  {
     ...
  }

}

compute_new_speed&positions ()
{
  // OMP PARALLEL FOR
for(int i=my_first_particle;
    i<my_last_particle;i++)
  {
     ...
  }

}
```

Figure 2: The data parallel portion of the Agent code.

Node Name	Power Index
Platone	1
Hegel	1
Leibnitz	2.77

Table 1: Biprocessor PCs cluster's power indices.

We limited our performance analysis only to the force computing phase that is the most CPU-consuming portion of the algorithm and at the same time the most significant to our purposes.

The value of the 'ideal maximum speed-up' (S) is valuated on the homogeneus system (IBM SP) as $S(N) = N$, where N is the number of processors used. For the heterogeneous cluster the ideal maximum speed-up evaluation is more complex, in fact it is not simply equal to the number of the processors, but depends on the process mapping and on the characteristics of each computational node, as was reported in [10]. The correct speed-up can be evaluated using a "power index" p_i, that is the ratio of the rates at which the task of the application is executed on node i and on the sequential processor respectively. This power index can be evaluated using a suitable benchmark program; in this case we used the same case study application, executed with one thread and one agent on each node, (see the measured power indices in table 4). The maximum speed-up is given by $S^*(N) = \sum_{i=1}^{N} p_i$.

The tests on the SP parallel system show the obtained speed-up with one or two nodes. All the tests are obtained with one agent for each node, all the agents have the same number of threads. The total number of threads is reported on the x-axis, the evaluated speed-up on the y-axis; this mean that if we report the speed-up evaluated for 4 threads on two nodes, there are two agents with two threads each one.

Note that the balancing of the workload can be done only in presence of different agents, so while figure 4 reports both balanced

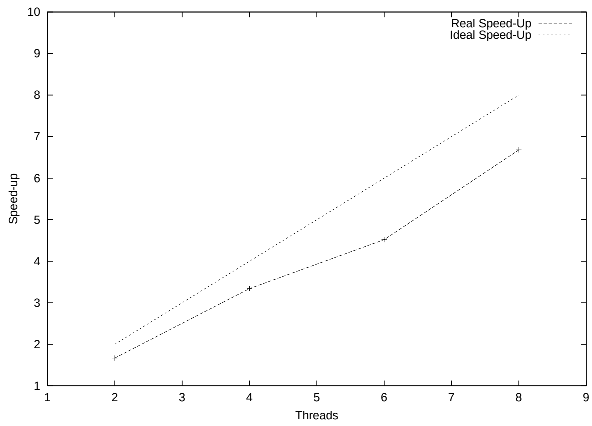

Figure 3: Speed-up of the Nbody application on IBM SP Parallel System, 1 Node, 8 processors.

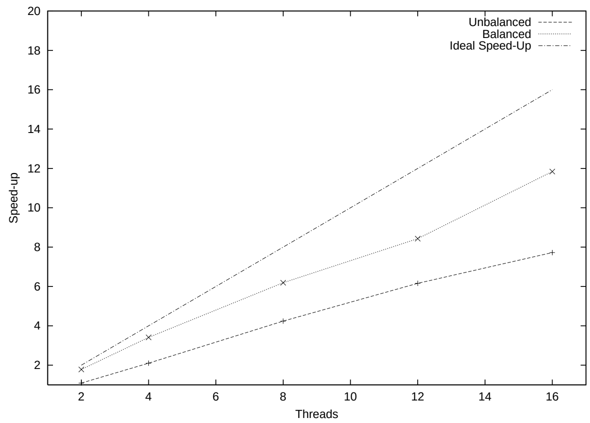

Figure 4: Speed-up of the Nbody application on IBM SP Parallel System, 2 Nodes, 8 processors.

and unbalanced versions of the case-study, the diagram reported in 4, reports only the parallel program speed-up varying the number of threads.

The tests on the biprocessor PCs cluster, instead, report the different speed-up with a different number of nodes, when each agent is composed by one, or two threads.

Note that the Leibnitz node, that adds heterogeneity to the cluster, is used only for the test with 3 nodes.

To summarize, the speed-up of the balanced version is always better of the unbalanced one; in particular, for low number of

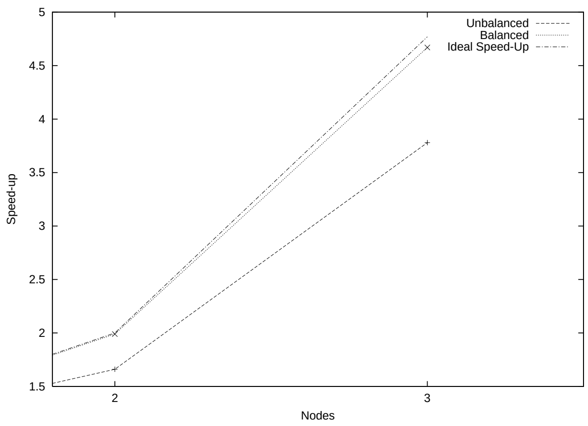

Figure 5: Speed-up of the Nbody application on biprocessor PCs Cluster using only one thread for each agent.

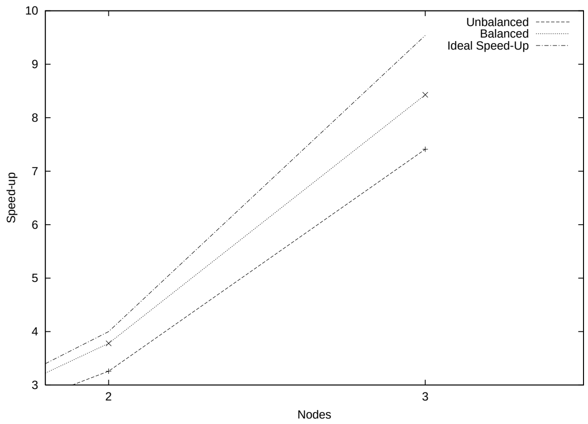

Figure 6: Speed-up of the Nbody application on biprocessor PCs Cluster, two thread for each agents.

threads the speed-up of the balanced version is near to the ideal speed-up value. We can conclude that, under the tested conditions, the dynamic workload balancing service together with the multithreaded execution appear to be precious to obtain a parallel version well suited for the heterogeneous target computing architecture.

5 Conclusions

In this paper we have addressed the problem of an effective utilization of the mobile agent model and parallelizing compiler technology for High Performance Distributed Computing, targetted in particular to hierarchical distributed-shared memory architectures, namely clusters of SMP nodes.

We have shown, through a case-study, how the adoption of the mobile agent model for the distributed parallelism and OpenMP parallelizing compiler technology for the inner shared memory parallelism, allows to yield a hierarchically distributed-shared memory implementation of an algorithm presenting multiple levels of parallelism. The addition to the Mobile Agent framework of a dynamic workload balancing service, and its extension to handle dynamical workload reassignment among threads belonging to different agents allows to exploit the heterogeneity and/or load unbalance of the target computing architectures.

Acknowledgement. We wish to thank Marco Briscolini and Antonio De Gaetano for the utilization of the IBM SP system at ENEA Computing Center in Frascati, Italy.

References

[1] 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).

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

[3] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface" ver. 1.0, October 1997.

[4] R. Aversa, B. Di Martino, N. Mazzocca, S. Venticinque, "Mobile Agents for Distributed and Dynamically Balanced Optimization Applications", in: B. Hertzberger, A. Hoekstra, R. Williams (Eds.), *High Performance Computing and Networking*, Lecture Notes in Computer Science, n. 2110, pp. 161-170, Springer-Verlag, 2001.

[5] M. Bull, M. Westhead, M. Kambites, J. Obdrzalek, "Towards OpenMP for Java", Proc. of 2^{nd} *European Workshop on OpenMP - EWOMP'2000*, 14-15 September 2000, Edinburgh (UK).

[6] T. Drashansky, E. Houstis, N. Ramakrishnan, J. Rice, "Networked Agents for Scientific Computing", Communications of the ACM, vol. 42, n. 3, March 1999.

[7] Gray R., Kotz D., Nog S., Rus D., Cybenko G., "Mobile agents: the next generation in distributed computing" Proc. of Int. Symposium on Parallel Algorithms/Architecture Synthesis, 1997.

[8] H. Kuang, L.F. Bic, M. Dillencourt, "Paradigm-oriented distributed computing using mobile agents", Proc. of. 20th Int. Conf. on Distributed Computing Systems, 2000.

[9] D. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, Reading (MA), 1998.

[10] A.Mazzeo, N. Mazzocca,U.Villano, "Efficiency Measurements in heterogeneous Distributed Computing Systems: from Theory to Practice", Concurrency Practice and Experience, vol.10 n. 4 pp 285-313, 1998.

[11] V. A. Pham, A. Karmouch, "Mobile software agents: an overview", IEEE Communications Magazine, Vol. 36(7), July 1998, pp. 26 -37.