Mobile Agents Based Collective Communication: An Application to a Parallel Plasma Simulation

Salvatore Venticinque[1], Beniamino Di Martino[1], Rocco Aversa[1], Gregorio Vlad[2], and Sergio Briguglio[2]

[1] Dipartimento di Ingegneria dell' Informazione - Second University of Naples - Italy
Real Casa dell'Annunziata - via Roma, 29 81031 Aversa (CE) - Italy
[2] Associazione EURATOM-ENEA sulla Fusione, C.R. Frascati, C.P. 65, 00044, Frascati, Rome, Italy

Abstract. Collective communication libraries are widely developed and used in scientific community to support parallel and Grid programming. On the other side they often lack in Mobile Agents systems even if message passing is always supported to grant communication ability to the agents. Collective communication primitives can help to develop agents based parallel application. They can also benefit social ability and interactions of collaborative agents. Here we present a collective communication service implemented in the Jade agent platform. Furthermore we propose its exploitation to interface transparently heterogeneous executions instances of a scientific parallel application that runs in a distributed environment.

1 Introduction

Collective communication is a communication activity that involves more entities belonging to a group and sharing a common goal or a common interest. In commercial application it can be exploited to enhance the way internet users have today to communicate by Internet. In high performance computing it represents a parallel programming paradigm that allows communication and synchronization of processes by message passing. Collective communication primitives are constructs of the language which exploit the underlaying middleware to perform specified schemes of message exchange. They can help to develop agents based parallel applications. They can also benefit social ability and interactions of collaborative agents. Mobile Agents technology has been widely addressed in the scientific community to provide a flexible programming approach in parallel and distributed environments. Some experimental results can be found in [1,2,3,4]. A mobile agent is a Software Agent with an added feature: the capability to migrate across the network together with its own code and execution state. This paradigm allows both a pull and a push execution model [5]. In fact, the user can choose to download an agent or to move it to another host. Mobility can provide many advantages when we aim at developing distributed applications. System reconfiguration by agent migration can help to optimize the execution time by reducing network traffic and interactions with remote systems. Furthermore, statefull migration allows to redistribute dynamically the agents for load balancing purposes. Several different criteria can guide agent distribution, such as moving the execution near to the data, exploiting new idle nodes, or allocating agents on the nodes in such a way that communications are optimized. Many different implementations of the mobile agent programming paradigm are available. A list of more than 60 known agent platforms can be found at `http://labe.felk.cvut.cz/~bendap1/agentsoftware/list.html`.

However existing Mobile Agents platforms do not provide collective communication primitives which can help the programmer to develop parallel applications. Here we present a collective communication service implemented in the Jade agent platform and we propose its exploitation in a scientific parallel application to interface transparently heterogeneous executions which run in parallel in a distributed environment. Next section introduces message passing, collective communication concepts and technology together with agents systems. Section three presents the collective communication service and its implementation. Section four describes the parallel application we are going to use as case study. Finally Conclusions summarize the current status of our research and ongoing work.

2 Message Passing and Collective Communication: Concepts and Technology Together with Agents Systems

Message passing libraries and collective communication primitives to support parallel and Grid programming are widely developed and used in scientific community. As an example, we cite MPI (Message Passing Interface), the best known standard [6] which defines more the 300 routines. Its implementations for fortran and C languages are available on different architecture by countless developers. JavaMPI [7] is a pure java implementation of the standard, while MPIJ [8] is a java interface to a native code implementation of the supported methods. Other parallel environments such as DECK [9] provide similar facilities. On the other side collective communication primitives can be rarely found implemented in agents platforms. The FIPA standard [10] provides specifications to design an agent platform. It describes the software architecture of an agent platform, the basic services which must be provided, the life cycle of an agent, but the most relevant part of the standard concerns the agent communication language. In fact, FIPA original effort aims, above all, at supporting interoperability among autonomous agents belonging to the same platform or to different ones. Regarding agent communication FIPA defines communicative acts and their requirements [FIPA00037], grammatical structure of Agent Communication Language, agent interaction protocols, their ontology, semantic and requirements. The standard do not deal with the transport communication protocol and the way to exchange messages. Nowadays many platforms are compliant with the FIPA specifications and implement many message transport protocols (MTPs) to support interoperability with other agent systems. An example is JADE, developed by TILAB (Telecom Italia laboratory) which implements HTTP MTP, IIOP MTP, Orbacus MTP for inter platform communication and RMI for intra platform communication. Regarding the communication primitives supported by a mobile agents platform different message passing models can be available: synchronous, asynchronous, future messaging, etc. They differ according to the way the sender behaves when sending a message. It can wait for the end of the message handling at destination, it can wait or check later for a response. The Aglet Workbench, developed by IBM Japan research group [11] supports all the mechanisms described above.

3 Mobile Agents Collective Communication

In a previous paper we have presented MAGDA - Mobile Agents based Grid Architecture - [14]. It is conceived in order to provide secure access to a wide class of services in an heterogeneous system, locally and geographically distributed. MAGDA is a layered architecture, which strictly adheres to the Layered Grid Model[15]. We exploited MAGDA to develop Mobile Agents based parallel applications in [12,4,13]. We extended the multicast communication mechanism supported locally in the Aglet Workbench in order to make it working in a distributed environment. Now we aim at providing at agent level a collective communication library and at developing an agent gateway to extend the communication ability of legacy applications (such as classical MPI applications) to heterogeneous and distributed environments. The conceived model defines two communication layers, as shown in Figure 1. The first one is implemented by a native parallel middleware; the second one, at upper level, is implemented by agents. An agent gateway is able to perform as a bridge forwarding messages from an environment to the other. A similar approach is exploited by MPICH_G2 [16] in the Globus middleware among proprietary implementation of the MPI standard. In our architecture, a collective communication manager coordinates groups, collects and delivers messages being able to migrate across the network, in order to optimize the mean latency and the traffic according to the physical location of the communicating parties.

3.1 Designing the Collective Communication Service

We designed the collective communication facility as a service provided by an agent inside the Jade platform. The provider is both a group and a message manager. It can be federated with other providers in order to distribute the handling of groups and of messages. Furthermore it is mobile and can migrate to a best suited node according to the physical distribution of group members in order to minimize latencies and traffic which affect the network. The communication manager accepts subscriptions by agents to new or to already available groups. Afterwards, agents can send to it messages which must be handled according to the selected communication mechanism. In the starting phase the provider

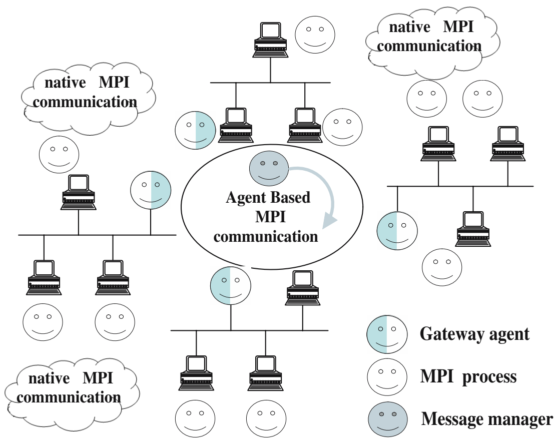
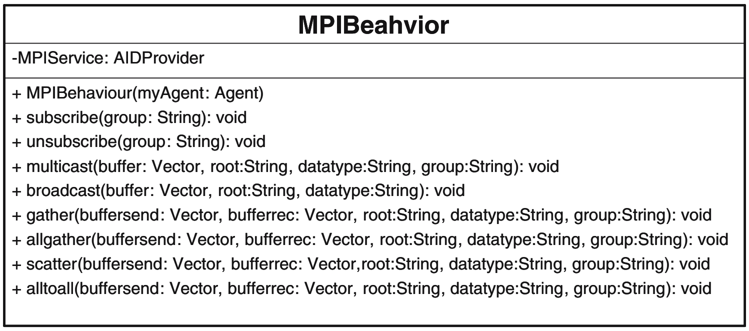


Fig. 1. The two layer communication hierarchy

publishes its service as Collective Communication Service in the directory facilitator which is the FIPA standard directory where agents look for available services (according to the standard, it must be implemented by an agent and must run in each platform). Each agent requestor is able to look for the service and to subscribe dynamically to a collective communication group identified by the provided identification label. If the required group is unknown a new group is built with the received label otherwise the agent is added to the already existing group. The agent subscription is delayed till any open collective communication involving that group will be successfully completed. Each agent requestor is able to subscribe to a group, to unsubscribe to the same group and to take part in a collective communication by sending a message to the provider and waiting for the response. According to the selected communications mechanism, the response is returned when all the other group members have joined the same communication instance. We support the following communication mechanisms:

- *broadcast*: the message is forwarded to all the members of all existing group registered by a provider
- *multicast*: the message is forwarded to all the agents of the specified group.
- *scatter*: the content of the message received by a specified agent (the root) is distributed to the other group members.
- *gather*: the content of the received messages are collected in a vector that is sent to the specified root member of the specified group. All the others agent receive an acknowledge when the communication is completed.
- *allgather*: the content of the received messages are collected in a vector that is sent to all the members of the specified group.
- *alltoall*: content of the received message from each agent is divided among the others group members in the way that each one receives part of the content from all messages.

We implemented the described service in the Jade platform. We provide a set of APIs at client side that allows the programmer to create a MPI message and to invoke the desired collective communication primitive. In the jade programming model the developer can define an agent behavior extending a basic *Behavior* class whose instances will be scheduled by the agent itself. As it is shown in Figure 2 we implemented a MPIBehaviour that looks for an available collective communication service and provide some final methods which allow the agent to subscribe/unsubscribe to communication groups and to take part in the supported collective communication actions. The datatype parameter identifies



```
MPIBeahvior
-MPIService: AIDProvider
+ MPIBehaviour(myAgent: Agent)
+ subscribe(group: String): void
+ unsubscribe(group: String): void
+ multicast(buffer: Vector, root:String, datatype:String, group:String): void
+ broadcast(buffer: Vector, root:String, datatype:String): void
+ gather(buffersend: Vector, bufferrec: Vector, root:String, datatype:String, group:String): void
+ allgather(buffersend: Vector, bufferrec: Vector, root:String, datatype:String, group:String): void
+ scatter(buffersend: Vector, bufferrec: Vector,root:String, datatype:String, group:String): void
+ alltoall(buffersend: Vector, bufferrec: Vector, root:String, datatype:String, group:String): void
```

Fig. 2. The MPIBehaviour class diagram

the object class inserted in the buffer. The programmer can send or receive any datatype and any amount of object by a Vector instance. The root parameter is the agent identifier of the receiver/sender in the specific communication pattern. The group parameter defines who will participate to the communication.

3.2 Mobile Agent Communication Gateway

In order to interface the agent based communication service described in the previous section with legacy MPI applications we have designed an agent gateway. The agent will start the MPI application and will be considered by the MPI processes as a process itself. When a communication primitive is invoked, the agent transparently will act as message gateway forwarding the information to other agent gateways which subscribed to a collective communication group of higher level. Hence, the agent gateway represents a virtual process which is physically allocated on a remote heterogenous node or distributed among more nodes. It will be implemented by a real MPI process that will take part in the communication at two different levels: at the lower one it exploits native MPI, at the higher one the agent technology. We are still evaluating its implementation details so we are not able to provide a final design of the component of the communication middleware. In [17] we have implemented in a Grid environment a parallel application to study plasma turbulence. Starting from this previous experience, we are going to apply the presented approach to the case study described in the next section. As shown previously, we have provided all the necessary communication primitives: scatter, broadcast and alltoall implementations. Note that all the MPI primitives are handled by MPI daemons running on each node. We will have to extends these daemons to support the hierarchical communication presented above.

4 Plasma Turbulence Simulation

Particle-in-cell simulation consists [18] 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 [19] and the *domain decomposition* one [20,21]. 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 [22,21]. 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 real code refers 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 [19]: 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. 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. 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.

5 Conclusions

We presented a collective communication service for mobile agents system. The mobility feature allows to dynamically optimize the dispatching of messages by migrating the service itself or, if it is possible, the communication parties. A first implementation in the Jade platform supporting limited communication primitives has been presented. We still lack to provide optimized implementation of messaging forwarding. Also strategies to move the provider to the best nodes and federation of multiple providers need to be implemented. We described how this service is going to be exploited for interfacing heterogeneous parallel applications which interact by collective communication primitives such as the ones defined in the MPI standard. We have finally introduced a real parallel application, used in our ongoing work, which has being used to test our proposed architecture for collective communication service for mobile agents. Due to lack of time we are not able to provide stable results, but performance analysis will be discussed on the conference event.

References

1. F. Rana and L. Moreau: Issues in Building Agent-Based Computational Grids.O. UK Multi-Agent Systems Workshop, Oxford, December 2000
2. H. Tianfield and R. Unland: Towards self-organization in multi-agent systems and Grid computing, Multiagent and Grid Systems Journal, IOS Pres , Volume 1, Number 2 / 2005 89 - 95

3. Z. Li and M. Parashar, Rudder: An agent-based infrastructure for autonomic composition of Grid applications, Multiagent and Grid Systems Journal, IOS Pres, Volume 1, Number 3 / 2005 Pages: 183 - 195
4. R. Aversa, B. Di Martino, N. Mazzocca, S. Venticinque: Mobile Agent Programming for Clusters with Parallel Skeletons. VECPAR'2002. Lecture Notes in computer Science, vol. 2565, pp. 614-627, Springer- Verlag, 2003. (ISBN 3-540-00852-7)
5. A. Grama, V. Kumar and A. Sameh: Scalable parallel formulations of the Barnes-Hut method for n-body simulations. Parallel Computing 24, No. 5-6 (1998) 797-822.
6. Message Passing Interface Forum, MPI: A message-passing interface standard. International Journal of Supercomputer Application, 8((3/4)): 165-416, 1994.
7. S. Mitchev and V. Getov. Towards portable message passing in Java: Binding MPI. In Recent Advance in PVM and MPI, col. 1332 of Lecture Notes in Computer Science. Springer Werlag, 1997.
8. B. Carpenter, V. Getov, G. Judd et Al.: MPJ: MPI like Message Passing for Java., Concurrency: Practice and Experience, 12(11):1019–1038, 2000.
9. R. Silva, D. Picinin, M, Barreto et Al.: Performance Analysis of DECK Collective Communication Service.
10. A. Grama, V. Kumar, and A. Sameh.: Foundation for intelligent physical agents. *www.http://www.fipa.org*, 2000.
11. D. Lange and M. Oshima: Programming and Deploying Java Mobile Agents with Aglets. Addison Wesley, 1998.
12. R. Aversa, B. Di Martino, N. Mazzocca, M. Rak, S. Venticinque: Integration of Mobile Agents and OpenMP for programming clusters of Shared Memory Processors: a case study. In proc. of EWOMP (European Workshop on OpenMP), 2001, 8-12 Sept. 2001, Barcelona, Spain.
13. R. Aversa, B. Di Martino, N. Mazzocca, S. Venticinque: Mobile Agents for Distribute and Dynamically Balanced Optimization Applications. On High-Performance Computing and Networking (Lecture Notes in Computer Science vol.2110), ed. By B. Hertzberger et al. (Springer, Berlin, 2001).
14. R. Aversa, B. Di Martino, N. Mazzocca and S. Venticinque: MAGDA: A Mobile Agent based Grid Architecture. In Journal of Grid Computing, Springer Netherlands, 2006
15. Foster, I. (2001). The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science 2150*
16. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. N. Karonis, B. Toonen, and I. Foster. Journal of Parallel and Distributed Computing, Vol. 63, No. 5, pp. 551-563, May 2003.
17. B. Di Martino, S. Venticinque, S. Briguglio, G. Fogaccia, G. Vlad: A Grid based distributed simulation of Plasma Turbulence. In: L.T. Yang and M. Guo (Eds.), High Performance Computing: Paradigm and Infrastructure, Wiley eds., 2004
18. Birdsall, C.K., Langdon, A.B.: Plasma Physics via Computer Simulation. (McGraw-Hill, New York, 1985).
19. 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.
20. 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).
21. 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.
22. Cybenko, G.: Dynamic Load Balancing for Distributed Memory Multiprocessors. J. Parallel and Distributed Comput., **7**, (1989) 279–391.