Master's Dissertation

Parallelization of the Treecode Algorithm for N-Body Simulation Using MPI, Hybrid, and GridRPC Programming Paradigms

Name : Truong Vinh Truong Duy

Student ID : 6124050303-3

Academic Adviser : Professor Katsuhiro Yamazaki

Graduate School of Science and Engineering, Ritsumeikan University

2007

Abstract

This dissertation describes the parallelization of the treecode algorithm for N-Body problem and performance comparison among three different parallel programming paradigms, MPI, hybrid MPI-OpenMP, and GridRPC. In N-Body simulation, the specific routine for calculating the forces on the bodies which accounts for upwards of 90% of the cycles in typical computations is eminently suitable for obtaining parallelism with light-weight OpenMP threads in the hybrid model and with multiple asynchronous GridRPC calls in the GridRPC model. Multiple levels of parallelism are achieved by the hybrid program: the workload of the force calculation is shared among OpenMP threads after ORB domain decomposition among MPI processes. In addition, redundant MPI intra-node communication is removed and loop scheduling of OpenMP threads is adopted with appropriate chunk size for better load balance in the hybrid model. Meanwhile, in the GridRPC implementation, this workload is divided among the compute nodes by simultaneously calling multiple GridRPC requests to them. A preliminary GridRPC computing system which consists of multiple clusters is constructed using NetSolve middleware for evaluating and comparing the performance of the GridRPC code and that of MPI and hybrid codes on individual clusters with data sets of 10,000, 50,000, and 100,000 bodies.

Experimental results demonstrate that no matter how many processors are used and how large the data set size is, the hybrid MPI-OpenMP implementation outperforms the corresponding pure MPI one by average of 30% on 4-way cluster and 20% on 2-way cluster. On the other hand, the performance of GridRPC program is determined by the larger, the better size of data sets, in other words, the computation time for remote calls. Even though the GridRPC code suffers a desperately poor performance with the execution time is almost twice as long as that of the hybrid code with the smallest data set of 10,000 bodies, the peak performance gained in case of 100,000 bodies is superior to the hybrid code’s performance, approximately 10% faster. The performance is further improved by an increased number of processors, a great merit brought by the GridRPC model which outweighs the cluster model in terms of maximizing the use of resources and producing higher throughput.

Acknowledgements

First and foremost I offer my sincerest gratitude to my supervisor, Professor Katsuhiro Yamazaki, who has supported me throughout my study with his knowledge, guidance and encouragement. Without his consistent help this dissertation would not have been completed or written. I would also like to thank Professor Shigeru Oyanagi, whose advice and comments were invaluable.

Besides, I am indebted to Japan International Cooperation Agency for generously providing me with the Japanese Grant Aid for Human Resource Development Scholarship. It was indispensable to help me become financially stable during my stay in Japan. I am grateful to Japan International Cooperation Center (JICE), JICE Osaka officials and the staff of International Affair Office and Graduate School of Science and Engineering for their guidelines to assist me in growing accustomed to living and studying in Japan.

Last but not least, I thank my family who endured this long process with me, always offering love, support and understanding. Thanks are also due to numerous friends, especially those at High Performance Computing Laboratory and Computer Systems Laboratory for their willingness to participate in challenging discussion and give help to tackle the language barrier in my daily life.

Table of Contents

Abstract ii
Acknowledgements iii
Table of Contents iv
List of Figures v
List of Tables vi
1 Introduction 1
2 Parallel programming paradigms 4
2.1 MPI and Hybrid MPI-OpenMP programming models 4
2.2 GridRPC programming model 6
2.3 Comparison among MPI, hybrid and GridRPC programming models 10
3 The treecode algorithm for N-Body simulation 12
3.1 The N-Body problem 12
3.2 The clustering approximation 15
3.3 Recursive division of space and the resulting tree 16
3.4 Example 18
3.5 Related work on parallelization of the treecode algorithm 19
4 Parallelization of the treecode algorithm 21
4.1 MPI parallelization 21
4.2 Multiple levels of parallelism with the hybrid model 25
4.3 Parallelization using asynchronous GridRPC calls 29
5 Performance evaluation and discussion 32
5.1 Computing platforms 32
5.2 Performance comparison between MPI and hybrid 33
5.3 Performance comparison between GridRPC and other versions 37
5.4 Discussion 40
6 Conclusion 43

List of Figures

Figure 1. Data transfer by message passing in MPI model. 4
Figure 2. Hybrid MPI-OpenMP programming model applied to a 2D array. 5
Figure 3. Communication pattern in MPI and hybrid programming models. 6
Figure 4. The GridRPC model. 7
Figure 5. The NetSolve system with full implementation of the GridRPC API. 9
Figure 6. The calculation of force between two bodies. 12
Figure 7. The clustering approximation approach. 15
Figure 8. Replacing clusters by their center of mass recursively. 16
Figure 9. Recursive division of 2D space and the resulting quad-tree. 17
Figure 10. The recursive distance check between the selected body and nodes. 17
Figure 11. The steps for calculating the force acting on body *a*. 18
Figure 12. ORB domain decomposition in 2D space on 16 processors. 21
Figure 13. ORB decomposition and the influence ring of a node. 22
Figure 14. The group-opening criterion for determining essential nodes. 23
Figure 15. The construction of P1's LET, assuming there are 4 processes P0, P1, P2, and P3. 23
Figure 16. An example of the parallel treecode with 2 processors. 25
Figure 17. Multiple levels of parallelism with the hybrid implementation. 26
Figure 18. Static, dynamic and guided loop scheduling methods of OpenMP threads. 27
Figure 19. Operation of the hybrid code with 2 MPI processes and 4 OpenMP threads. 28
Figure 20. Exploiting parallelism with simultaneous GridRPC calls. 29
Figure 21. Time chart of the client program with GridRPC parallelism. 31
Figure 22. The GridRPC computing system and a typical call from client. 33
Figure 23. Execution time of MPI and hybrid programs with 10,000 bodies. 34
Figure 24. Execution time of MPI and hybrid programs with 50,000 bodies. 35
Figure 25. Execution time of MPI and hybrid programs with 100,000 bodies. 36
Figure 26. Execution time of GridRPC, MPI, and hybrid with 10,000 bodies. 38
Figure 27. Execution time of GridRPC, MPI, and hybrid with 50,000 bodies. 39
Figure 28. Execution time of GridRPC, MPI, and hybrid with 100,000 bodies. 40

List of Tables

Table 1. GridRPC API functions 8
Table 2. MPI and hybrid vs. GridRPC 11
Table 3. System specification of three clusters for performance evaluation 32
Table 4. Execution time of MPI and hybrid programs with 10,000 bodies (seconds) 34
Table 5. Execution time of MPI and hybrid programs with 50,000 bodies (seconds) 35
Table 6. Execution time of MPI and hybrid programs with 100,000 bodies (minutes) 36
Table 7. Execution time with different schedules and chunk sizes (seconds) 37
Table 8. Execution time of GridRPC, MPI, and hybrid with 10,000 bodies (seconds) 38
Table 9. Execution time of GridRPC, MPI, and hybrid with 50,000 bodies (seconds) 38
Table 10. Execution time of GridRPC, MPI, and hybrid with 100,000 bodies (minutes) .. 39

1 Introduction

Large-scale highly parallel systems based on cluster of SMP architecture are today's dominant computing platforms which enable many different parallel programming paradigms. Optimal paradigms enable application developers to use the hardware architecture in the most efficient way, i.e., without any overhead induced by the programming paradigm. On distributed memory systems, MPI [23] is widely used for writing message passing programs across the nodes of a cluster while OpenMP [24] is a popular API for parallel programming on shared memory architecture. As a result, a combination of shared memory and message passing paradigms within the same application, hybrid programming, is expected to provide a more efficient parallelization strategy for clusters of SMP nodes. The hybrid MPI-OpenMP approach supports multiple levels of parallelism on an SMP cluster where MPI is used to handle parallelism across nodes and OpenMP is employed to exploit parallelism within a node.

There have been many efforts for porting message passing applications to hybrid applications, leaving both opportunities and challenges of getting higher performance with this model. The implementation, development and performance of hybrid program applications are discussed in [6]. The results demonstrate that this style of programming is not always be the most effective mechanism but can enjoy significant benefits from some situations. Similarly, the results from comparing MPI with MPI-OpenMP for the NAS benchmarks [17] are clearly application-dependent. The hybrid approach becomes better when processors make the communication performance considerable and the level of parallelization is sufficient. Bush et al. [19] also prove that hybrid MPI-OpenMP codes can give substantial performance on kernel algorithms such as Cannon's matrix multiply although it requires a large amount of work involved to succeed. Recently, the hierarchical image data structure of MPEG bit-stream was exploited by the hybrid model to accomplish a significant performance improvement of 18% compared to the MPI MPEG-2 encoder [4].

However, even if the hybrid model is likely to offer an improved performance for parallel solutions on SMP clusters, these separate clusters alone are not sufficient to perfectly satisfy the need to perform large numbers of complex computations. Rather, given the ability to integrate heterogeneous resources, Grid [7] is going to be a practical infrastructure for large-scale scientific applications. Therefore, gridifying existing applications to maximize the use of resources is an emerging trend coming with increasingly popular grids. One of the most attractive and practical programming model on grids is GridRPC [9] which is based on a Remote Procedure Call mechanism tailored for the grid. The GridRPC API is designed to address one of the factors that has hindered widespread acceptance of grid computing – the lack

of a standardized, portable and simple programming interface and provide remote library access and task-parallel programming model on the grid. Some representative GridRPC systems include GridSolve/NetSolve [25], and Ninf [26].

Providing simple, yet powerful, client-server-based frameworks for programming on the Grid, those systems have been successfully applied in various grid application projects. The design, implementation and performance evaluation of a suite of GridRPC programming middleware called Ninf-G2 is reported in [10]. The performance of Ninf-G2 evaluated using a weather forecasting system indicates that high performance can be attained even in relatively fine-grained task-parallel applications on hundreds of processors in a grid environment. A straightforward but effective scheme for parallel execution of SimSET, Monte Carlo simulation software for emission tomography, using NetSolve is described in [13]. When compute speeds and sustained workloads are taken into account, the speedup is essentially linear in the number of equivalent "maximum-service" processors. Also by using NetSolve, a framework of applications integration on the grid is proposed for efficiently solving a large-scaled and complicated optimization problem [14].

This research is aimed at parallelizing the treecode algorithm [22] for N-Body simulation and making performance comparison among three different parallel programming paradigms, MPI, hybrid MPI-OpenMP, and GridRPC. By so doing, some certain classes of applications well suited to the hybrid and GridRPC models are thought to be found. N-Body is a classical problem, and appears in many areas of science and engineering, including astrophysics, molecular dynamics, and graphics. In the treecode algorithm simulating the interaction among the bodies, the workload for calculating the forces on the bodies accounts for upwards of 90% of the cycles in typical computations. This specialty makes the treecode eminently suitable for exploiting multiple levels of parallelism with the hybrid model and obtaining parallelism with multiple asynchronous GridRPC calls in the GridRPC model. In the hybrid implementation, the workload of the force calculation is shared among light-weight OpenMP threads after ORB domain decomposition among MPI processes. Moreover, the removal of redundant MPI intra-node communication and adoption of loop scheduling for OpenMP threads with appropriate chunk size ensure load balance and enhance performance of the hybrid program. Meanwhile, in the GridRPC implementation, this workload is divided among the compute nodes by simultaneously calling multiple GridRPC requests to them. The NetSolve middleware is utilized to construct a preliminary GridRPC computing system which consists of multiple clusters for evaluating the performance of the GridRPC code. The performance of the GridRPC version is then compared to that of MPI and hybrid versions with data sets of 10,000, 50,000, and 100,000

bodies in 10 time-steps on 3 clusters in which there are 2 clusters of SMP architecture, one with 4 quad-processor nodes and the other with 16 dual-processor nodes.

The rest of the dissertation is structured as follows. Section 2 presents and compares the MPI, hybrid MPI-OpenMP and GridRPC parallel programming paradigms. The N-Body problem and treecode algorithm for solving this problem are described in Section 3. Section 4 outlines parallelism achieving methods and implementation of the treecode algorithm with MPI, hybrid and GridRPC models. Section 5 analyzes the experimental results and gives some discussions. Finally, the study is concluded in section 6.

2 Parallel programming paradigms

2.1 MPI and Hybrid MPI-OpenMP programming models

MPI has apparently become the most popular message passing library standard for parallel programming. The message passing model assumes that the underlying hardware is a collection of processors, each with its own local memory, and an interconnection network supporting message passing among processors. A processor has direct access only to the instructions and data stored in its local memory and processes pass messages both to communicate and to synchronize with each other. Processor 1 may send a message containing some of its local data values to processor 2, giving processor 2 indirect access to these values as displayed in Figure 1.

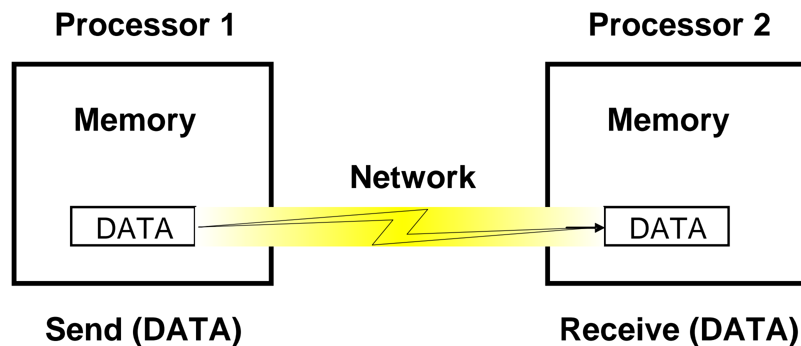

Figure 1. Data transfer by message passing in MPI model.

Message passing codes written in MPI are obviously portable and should transfer easily to clustered SMP systems. Although message passing may be necessary to communicate between nodes, it is not immediately clear that this is the most efficient parallelization technique within an SMP node. In theory, a shared memory model such as OpenMP should offer a more effective method within an SMP node. SMP clusters can be considered as a hierarchical two-level parallel architecture since they combine features of shared and distributed memory machines. Hence, a combination of both shared memory and message passing parallelization paradigms within the same application, hybrid MPI-OpenMP programming, may provide a strategy for exploiting distributed shared-memory architecture better than pure MPI.

Often, hybrid MPI-OpenMP model refers to a programming style in which communication between nodes is handled by MPI processes and each MPI process has several OpenMP threads running inside to occupy the CPUs of an SMP node. The number of OpenMP threads is equal to the number of CPUs in one SMP node and there are as many MPI processes as nodes in the

cluster. This programming style involves a hierarchical model: MPI parallelization occurring at the top level, and OpenMP parallelization occurring below. For example, Figure 2 shows a 2D grid which has been divided geometrically between four MPI processes. These sub-arrays have been further divided among three OpenMP threads and mapped to the architecture of an SMP cluster.

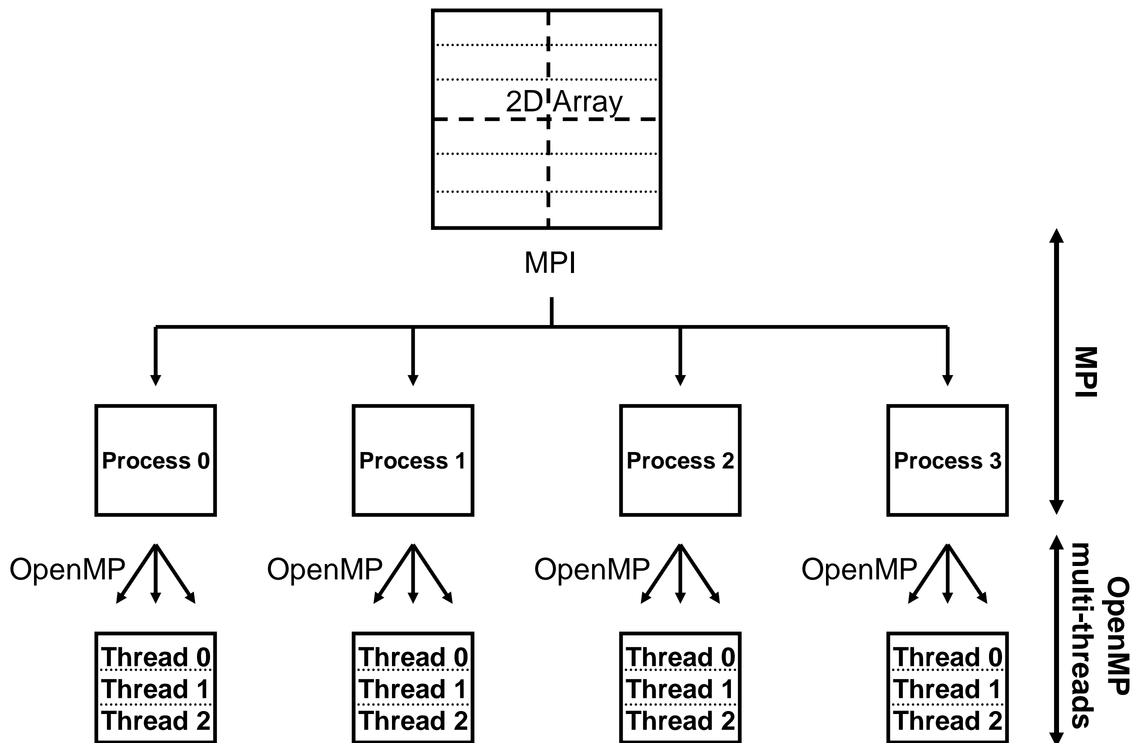

Figure 2. Hybrid MPI-OpenMP programming model applied to a 2D array.

Nonetheless, this style, called process-to-process communication method, is only one of two main different hybrid programming styles characterized by whether OpenMP threads take part in communication between nodes or not. In process-to-process communication, MPI routines are invoked outside OpenMP parallel regions, thus there is only MPI communication between nodes. On the other hand, in thread-to-thread communication, some MPI routines are placed inside OpenMP parallel regions, leading to OpenMP threads' involvement in inter-node communication. Each style has different merits and demerits, and appropriate for different classes of applications [5]. For the implementation of N-Body problem, the process-to-process model is suitable because parallelizing the specific time-consuming routines using lighter-weight OpenMP threads without having to communicate with each other is more effective.

In addition to the ability to support multiple levels of parallelism on an SMP cluster, the hybrid model outweighs the pure MPI model in terms of communication overhead. It requires

only inter-node communication between nodes since intra-node communication is substituted by direct access to the shared memory. Meanwhile, with pure MPI, additional intra-node communication is necessary within each node between MPI processes as illustrated in Figure 3. This benefit of hybrid programming is even more important, since the gap between intra-node communication and inter-node communication is significantly narrowed by rapid communication networks.

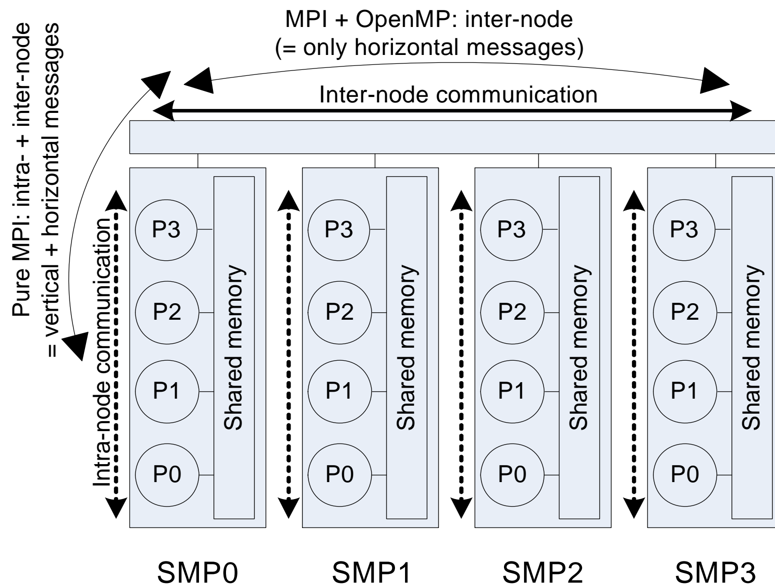

Figure 3. Communication pattern in MPI and hybrid programming models.

2.2 GridRPC programming model

GridRPC is a programming model based on Remote Procedure Call mechanism tailored for the grid. In 2005, the GridRPC API was published as a proposed recommendation by the Grid Remote Procedure Call Working Group [28]. Although when viewed at a very high level, the programming model provided by GridRPC is that of standard RPC plus asynchronous, coarse-grained parallel tasking, actually there are a variety of features that will largely hide the dynamic, insecure, and unstable aspects of the grid from programmers. This section introduces the GridRPC model, the functions that comprise the API, and how GridRPC applications run on the grid using NetSolve as the GridRPC programming middleware.

2.2.1 The GridRPC model

Figure 4 depicts the GridRPC model. The functions shown here are very fundamental and appear in a number of other systems. A service which can be either a computation service or a data service registers with a registry. A client subsequently contacts the registry to look up a

desired service and the registry returns a handle to the client. The client then uses the handle to call the service which eventually returns the results.

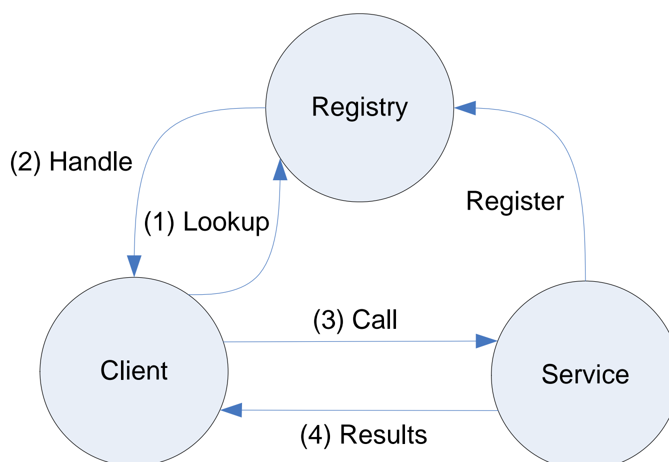

Figure 4. The GridRPC model.

Two fundamental objects in the GridRPC model are function handles and session IDs. The function handle represents a mapping from a function name to an instance of that function on a particular server. Once a particular function-to-server mapping has been established by initializing a function handle, all RPC calls using that function handle will be executed on the server specified in that binding. A session ID is an identifier representing a particular non-blocking RPC call. A session ID is used throughout the API to allow users to get the status of a previously submitted non-blocking call, to wait for a call to complete, to cancel a call, or to check the error code of a call.

2.2.2 The GridRPC API

The functions that comprise the GridRPC API fall into four main groups: initializing and finalizing functions, remote function handle management functions, GridRPC call functions, and asynchronous GridRPC control and wait functions. Table 1 gives an overview of the principal GridRPC API functions. The initialize and finalize functions are similar to the MPI initialize and finalize calls. GridRPC client calls before initialization or after finalization will fail. The function handle management group of functions allows creating and destroying function handles. The GridRPC call functions are available for end-users to call the desired service. Asynchronous GridRPC control and wait functions are applied to previously submitted non-blocking requests.

Table 1. GridRPC API functions

Initializing and finalizing functions	
grpc_initialize	reads the configuration file and initializes the required modules
grpc_finalize	releases any resources being used by GridRPC
Remote function handle management functions	
grpc_function_handle_default	creates a new function handle using the default server
grpc_function_handle_init	creates a new function handle with a server explicitly specified by the user
grpc_function_handle_destruct	releases the memory associated with the specified function handle
grpc_get_handle	returns the function handle corresponding to the given session ID
GridRPC call functions	
grpc_call	makes a blocking (synchronous) remote procedure call with a variable number of arguments
grpc_call_async	makes a non-blocking (asynchronous) remote procedure call with a variable number of arguments
grpc_call_argstack	makes a blocking call using the argument stack
grpc_call_argstack_async	makes a non-blocking call using the argument stack
Asynchronous GridRPC control and wait functions	
grpc_probe	checks if the asynchronous GridRPC call has completed
grpc_cancel	cancels the specified asynchronous GridRPC call
grpc_wait	waits for the specified session to end
grpc_wait_and	blocks until all of the specified non-blocking requests in a given set have completed
grpc_wait_or	blocks until any of the specified non-blocking requests in a given set has completed
grpc_wait_all	blocks until all previously issued non-blocking requests have completed
grpc_wait_any	blocks until any previously issued non-blocking request has completed

2.2.3 The NetSolve system with GridRPC API implementation

The GridRPC API is fully implemented on top of the NetSolve system [8] to bring together disparate computational resources with a view to using their aggregate power and dominating the rich supply of services supported by the emerging grid architecture. Basically, it is a RPC based client/agent/server system that allows users to remotely access both hardware and software components as shown in Figure 5. At the top tier, the client library is linked in with the user's application which then makes calls to GridRPC API for specific services. Through the API, the client application gains access to aggregate resources without having to know how remote resources are involved.

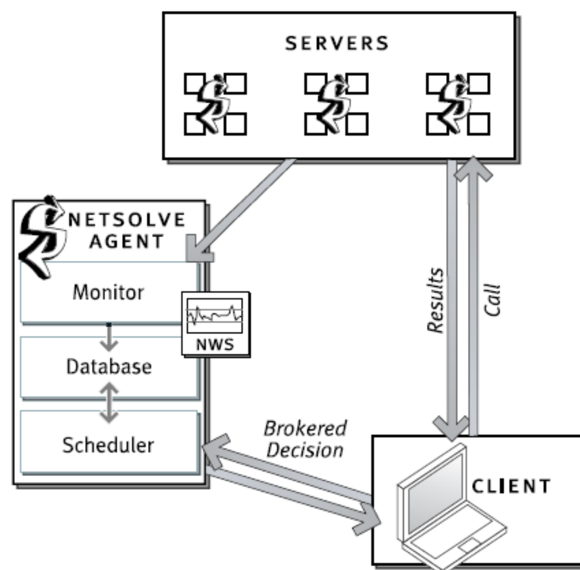

Figure 5. The NetSolve system with full implementation of the GridRPC API.

The agents maintain a database of all servers along with their capabilities and dynamic usage statistics which it uses to allocate server resources for client requests, ensuring load balancing and fault tolerance by keeping track of failed servers. NetSolve employs a load-balancing strategy that takes into account several system parameters, such as network bandwidth and latency, server workload and performance, and complexity of the function to be executed. To facilitate detection of server failures and network problems, the Network Weather Service (NWS) is integrated. The network authentication protocol Kerberos is also supported to provide strong authentication for client/server applications by using secret-key cryptography.

The server is a daemon process that awaits client requests and executes remote functions on behalf of clients. The server can run on single workstations, clusters of workstations, symmetric multi-processors or machines with massively parallel processors. A key component of the server

is a source code generator which parses a problem description file (PDF). This PDF contains information that allows the system to create new modules and incorporate new functionalities. In essence, the PDF defines a wrapper that the system uses to call functions being incorporated.

In reality, from the user's perspective the mechanisms employed by NetSolve system make the GridRPC calls fairly transparent. Nevertheless, behind the scenes, a typical call to the system involves a number of phases, as follows.

1. Client contacts the agent for a list of capable servers that can execute the desired function.
2. The agent returns a list of available servers.
3. Client contacts server and sends input parameters.
4. Server runs appropriate service and returns output parameters or error status to client.

2.3 Comparison among MPI, hybrid and GridRPC programming models

MPI is undoubtedly the most well known programming model for data-parallel applications using message passing. Besides, the hybrid model comes with the benefit of exploiting multiple levels of parallelism, MPI level and OpenMP level. Even so, both MPI and hybrid have some difficulties with programming and weak capability for tolerance to faults. When compared with MPI and hybrid, GridRPC has several advantages such as it provides programmers with an easier way to obtain parallelism, it can use the resources on the grids, it has better capability for tolerance to faults, and it can adapt heterogeneity and dynamic behavior of grids. But on the other hand, there exist a number of problems in the GridRPC model. The heterogeneous environment may include networks with small communication capacity, causing a delay in communicating among processing elements. The preparation time needed for calling and returning results, i.e. the initialization time and finalization time, is much higher than the message passing model. Also, additional effort is required for deploying modules responding to remote calls on all the servers of a GridRPC computing system, i.e., the installation and registration of a service to the registry in the system to make it available to clients. Table 2 summarizes and compares some important characteristics between MPI, hybrid and GridRPC programming models.

Table 2. MPI and hybrid vs. GridRPC

	MPI and hybrid	**GridRPC**
Parallelism	Mainly data parallel	Mainly task parallel
Model	Single program multiple data	Client/server
API	MPI, plus OpenMP for hybrid	GridRPC API
Ease of programming	Difficult to parallelize	Easy to implement from existing sequential codes
Fault tolerance	Poor	Good
Load balancing	Totally done by programmers if schedulers are unavailable	Integrated self-schedulers
Communication	Usually fast	Dependent upon the environment
Preparation time	Low initialization and finalization time	Overhead by high initialization and finalization time
Deployment	Unnecessary	Necessary
Security	Not a problem in case of clusters	An important issue in case of grids
Resources	Intended for clusters	Intended for grids
Reputation	Popular	Emerging and evolving

3 The treecode algorithm for N-Body simulation

3.1 The N-Body problem

The N-Body problem is concerned with determining the effects of forces between "bodies" and appears in many areas, including molecular dynamics, fluid dynamics and graphics. Let us examine the problem in terms of astronomical systems called the gravitational N-Body problems. The objective is to find the positions and movements of the bodies in space (say planets), i.e. their subsequent motions given the initial positions, masses, and velocities that are subject to gravitational forces from other bodies as identified by Newtonian laws of physics. In 1687 Isaac Newton formulated the principles governing the motion of two bodies under the influence of their mutual gravitational attraction.

We review the equations governing the motion of the bodies according to Newton's laws of motion and gravitation [29]. We assume for now that the mass m, position (r_x, r_y) and velocity (v_x, v_y) of each body is known. In order to model the dynamics of the system, we must know the total force exerted on each body. Newton's law of universal gravitation asserts that the strength of the gravitational force between two bodies is given by the product of their masses divided by the square of the distance between them, scaled by the gravitational constant G, which is 6.67×10^{-11} Nm^2 / kg^2. The pull of one body towards another acts on the line between them. Since we will be using Cartesian coordinates to represent the position of a body, it is convenient to break up the force into its x and y components (F_x, F_y) as can be seen in Figure 6.

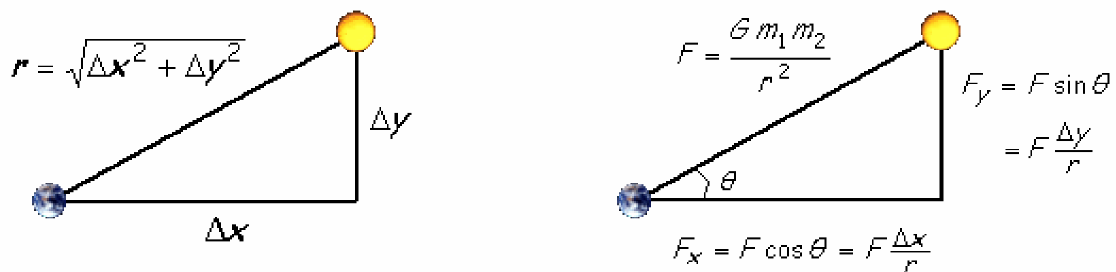

Figure 6. The calculation of force between two bodies.

Each body will feel the influence of each of the other bodies based on the law of universal gravitation above, that is the gravity interaction between all pairs of bodies except of self-interaction, and the forces will sum together. The principle of superposition says that the total force acting on a body in the x or y direction is the sum of the pair-wise forces acting on the body in that direction.

$$F_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} F_{ij}$$

The new position and new velocity of i-th body must be calculated from the known acceleration. Subject to the forces, the acceleration of a body according to Newton's second law of motion is given by:

$$a = \frac{F}{m}$$

Hence, all the bodies will move to new positions due to these forces and have new velocities.

For a computer simulation, we use values at particular times, t_0, t_1, t_2, etc., with the time interval Δt . Then, for a body of mass m, the force is given by:

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

and a new velocity

$$v^{t+1} = v^t + \frac{F \Delta t}{m}$$

where v^{t+1} is the velocity of the body at time $t + 1$ and v^t is the velocity of the body at time t. If a body is moving at a velocity v over the time interval Δt , its position changes by:

$$r^{t+1} - r^t = v \Delta t$$

where r^t is its position at time t. Once bodies move to new positions, the forces change and the computation has to be repeated.

However, the velocity is not actually constant over the time interval Δt , and thus only an approximate answer is obtained. For a more accurate solution, we can use the leapfrog finite difference approximation scheme to numerically integrate the above equations. In the leapfrog scheme, we maintain the position and velocity of each body, but they are half a time step out of phase. The velocity and position are computed alternatively, i.e.,

$$F^t = \frac{m(v^{t+1/2} - v^{t-1/2})}{\Delta t}$$

and

$$r^{t+1} - r^t = v^{t+1/2} \Delta t$$

In a three-dimensional space having a coordinate system (x, y, z), the distance between the bodies at (x_1, y_1, z_1) and (x_2, y_2, z_2) with the masses m_1, m_2 is given by

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

The forces are resolved in the three directions, using:

$$F_x = \frac{Gm_1m_2}{r^2}\left(\frac{x_2 - x_1}{r}\right)$$

$$F_y = \frac{Gm_1m_2}{r^2}\left(\frac{y_2 - y_1}{r}\right)$$

$$F_z = \frac{Gm_1m_2}{r^2}\left(\frac{z_2 - z_1}{r}\right)$$

The basic algorithm for N-Body simulation is outlined below.

```
 t = 0
 while t < t_final
  for i = 1 to n          // n = number_of_bodies
   // compute force on body i
   f(i) = sum[ j=1,...,n,  j != i ] f(i,j)
   //Calculate its acceleration (ax, ay) at time t using its force at time t
   a_x = F_x / m
   a_y = F_y / m
   // Calculate the updated velocity based on the accelerations
   v_x = v_x + dt a_x
   v_y = v_y + dt a_y
   // The resulting positions are given by
   r_x = r_x + dt v_x
   r_y = r_y + dt v_y
   // move body i under force f(i) for time dt
  end for
  t = t + dt              // dt = time interval
end while
```

3.2 The clustering approximation

The basic algorithm outlined above actually models a dynamical evolution of bodies with a direct N-Body simulation. In the framework of this direct algorithm, each body interacts with (N – 1) other bodies. The interactions must be carried out for each of the N bodies. In order to evaluate force interactions of the system composed of the N bodies, $N * (N - 1)$ computations are needed; that is nearly N^2 computations. The direct computation is an $O(N^2)$ algorithm and not feasible to be applied for most interesting N-Body problems when N is very large. Fortunately, it turns out that there are clever divide-and-conquer algorithms which only take $O(N\log N)$ time. Among them, the treecode, or Barnes-Hut algorithm is the most widely used algorithm for N-Body simulation.

The crucial idea used in the treecode algorithm to reduce the time complexity is to group nearby bodies and approximate them as a single body. If the group is sufficiently far away, we can approximate its gravitational effects by using its center of mass. More mathematically, if the ratio $\frac{D}{r}$ ($\frac{\text{size of box containing bodies of the group}}{\text{distance to center of mass}}$) is small enough, the group is treated as a single point, located at the center of mass and with a mass equal to the total mass of all the bodies of the group as illustrated in the case of viewing the Andromeda galaxy from the earth in Figure 7.

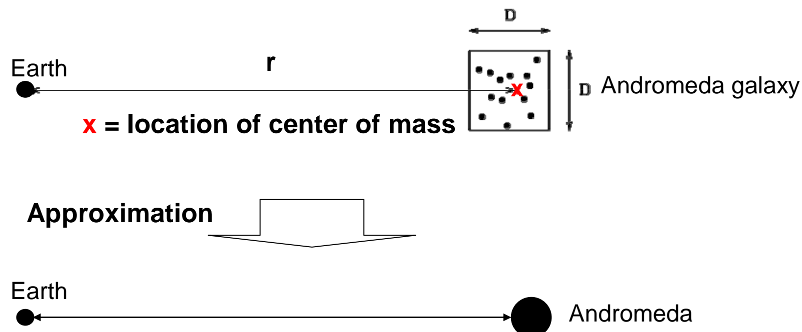

Figure 7. The clustering approximation approach.

The center of mass of a group of bodies is the average position of a body in that group, weighted by mass. Formally, if two bodies have positions (x_1, y_1) and (x_2, y_2), and masses m_1 and m_2, their total mass and center of mass (x, y) are given by:

$$m = m_1 + m_2$$

$$x = \frac{x_1 m_1 + x_2 m_2}{m}$$

$$y = \frac{y_1 m_1 + y_2 m_2}{m}$$

This clustering approximation idea is hardly new. Indeed, Newton modeled the earth as a single point located at its center of mass in order to calculate the attracting force on the falling apple, rather than treating each tiny particle making up the earth separately. What is new and more important is applying this idea recursively. Within the Andromeda galaxy, this geometric picture repeats itself as shown in Figure 8. As long as the ratio *D1/r1* is also small, the bodies inside the smaller box can be replaced by their center of mass in order to compute the gravitational force on, say, the planet Vulcan. This nesting of boxes within boxes can be repeated recursively.

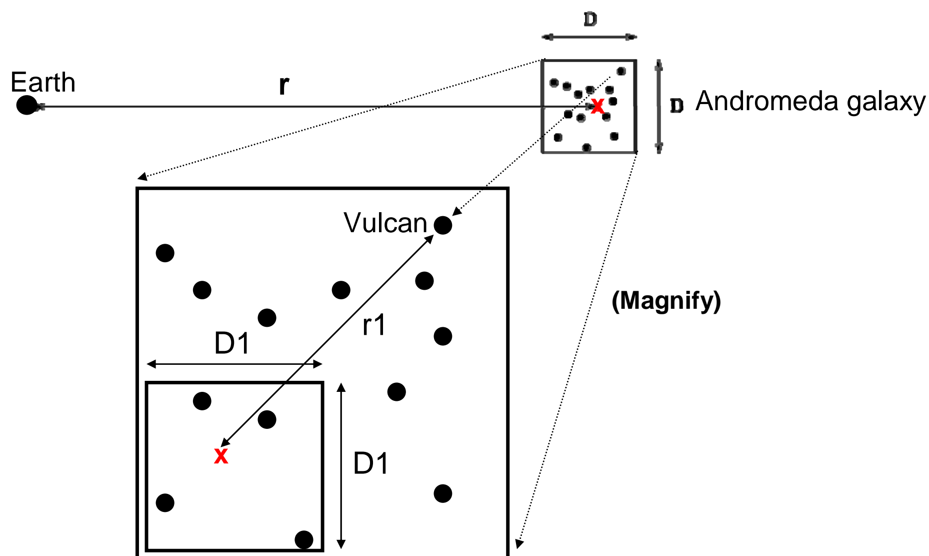

Figure 8. Replacing clusters by their center of mass recursively.

3.3 Recursive division of space and the resulting tree

What we need is a data structure to subdivide space that makes this recursion easy. The treecode algorithm is a clever scheme for grouping together bodies that are sufficiently nearby by recursively dividing the set of bodies into groups and storing them in a quad-tree for 2D space or oct-tree for 3D space. As described in Figure 9, the root (the topmost node) represents the whole space which can be broken into four smaller squares of half the perimeter and a quarter the area each; these are the four children of the root. Each child can in turn recursively be broken into quadrants to get its children until each contains 0 or 1 body. If there is only one body in the square left, it is stored as a leaf in the tree structure. When the square is empty, it is ignored. Hence, some internal nodes may have fewer than 4 non-empty children. Each leaf

represents a single body. Each internal node represents the group of bodies beneath it, and stores the center of mass and the total mass of all its children bodies.

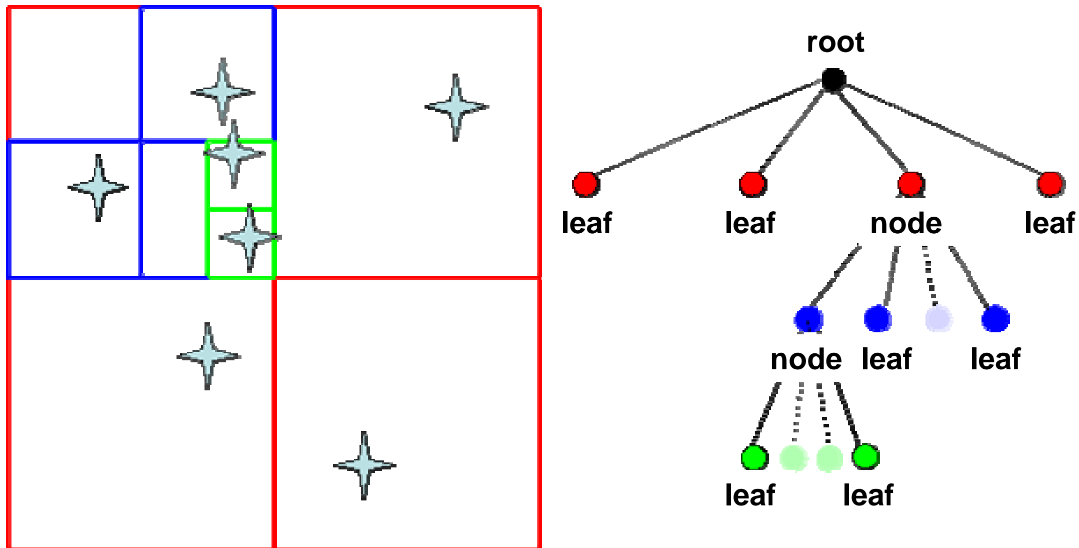

Figure 9. Recursive division of 2D space and the resulting quad-tree.

Once the tree construction phase is finished, we come to the core of the algorithm, computing the force acting on every single body in the system by traversing the tree from its root. If the center of mass of the current node is sufficiently distant from the selected body, the force is computed as the force acting between the selected individual body and the node. If the distance between the selected body and the node (its center of mass) is not sufficient then the node is "opened" and a distance check is performed between all of its leaves or sub-nodes and the selected body (Figure 10). This is executed repeatedly.

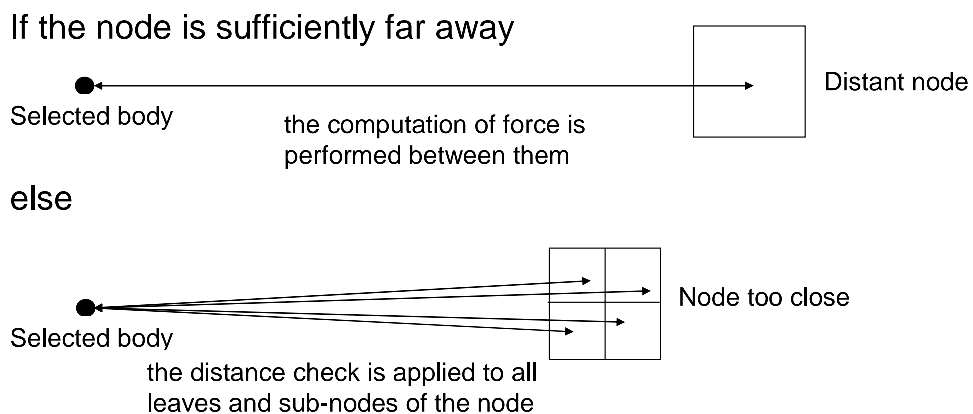

Figure 10. The recursive distance check between the selected body and nodes.

In order to determine if a node is sufficiently far away, the idea for clustering approximation presented earlier is applied to compute the ratio $\frac{D}{r}$, where D is the size of the region represented by the internal node, and r is the distance between the body and the node's center of mass. If the ratio $\frac{D}{r}$ is less than θ, then the internal node is sufficiently far away, where θ is a constant called the opening angle, $0 \leq \theta \leq 1$.

3.4 Example

As a concrete example for force calculation in this algorithm [30], let us consider the 5 bodies and the corresponding tree in Figure 11. The root node contains the center of mass and total mass of all five bodies *a*, *b*, *c*, *d*, and *e*, which have masses 1, 2, 3, 4, and 5 kg, respectively. With two other internal nodes, each contains the center of mass and total mass of the bodies *b*, *c*, and *d*. We start at the root node to calculate the force acting on body *a,* for instance. The force calculation proceeds through a couple of steps:

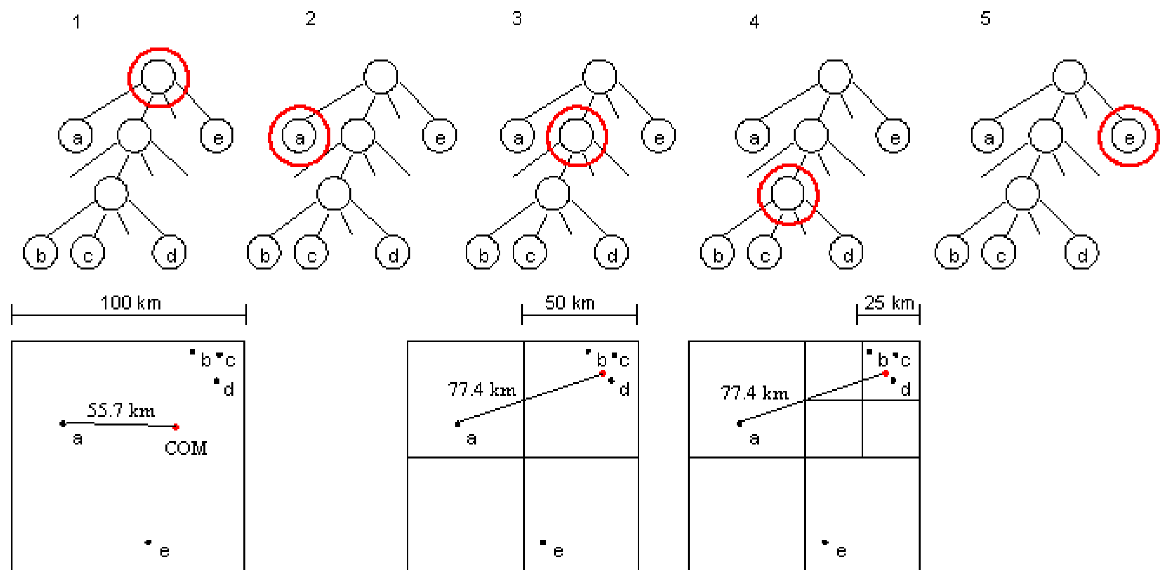

Figure 11. The steps for calculating the force acting on body *a.*

Step 1: The first node examined is the root. The ratio D/r = 100/55.7 > θ = 0.5, so we perform the process recursively on each of the root's children.

Step 2: The first child is body *a* itself. A node does not exert force on itself, so we don't do anything.

Step 3: This child represents the northeast quadrant of the space, and contains the center of mass of bodies *b*, *c*, and *d*. Now $D/r = 50/77.4 > \theta$ so we recursively calculate the force exerted by the node's only child.

Step 4: This is also an internal node, representing the northeast quadrant of its parent, and containing the center of mass of bodies *b*, *c*, and *d*. Now $D/r = 25/77.4 < \theta$. Treating the internal node as a single body whose mass is the sum of the masses of *b*, *c*, and *d*, we calculate the force exerted on body *a*, and add this value to the total force exerted on *a*. Since the parent of this node has no more children, we continue examining the other children of the root.

Step 5: The next child is the one containing body *e*. This is a leaf, so we calculate the pair-wise force between *a* and *e*, and add this to *a*'s total force.

The treecode algorithm is fast because we don't need to individually examine any of the bodies in a group after the clustering approximation is applied. Just like in the example, when the internal node containing *b*, *c*, and *d* is approximated as a single body, we only need to calculate the force between *a* and this body instead of every body in that node. This approach leads to the reduction of required interactions and to the decrease of computational complexity to $O(N\log N)$. Constructing the tree requires a time of $O(N\log N)$, and so does computing all the forces, so that the overall time complexity of the method is $O(N\log N)$. It is a substantial improvement over the direct summation method with $O(N^2)$ complexity.

3.5 Related work on parallelization of the treecode algorithm

During the past decades, N-Body treecode has been applied successfully to various problems in galaxy dynamics, galaxy formation and cosmological structure formation. Owing to a strong desire to increase the size and speed of both galactic and cosmological simulations, there have been many efforts to parallelize the treecode algorithm using different programming models on several diverse parallel machines. A parallel treecode is introduced in [11] with a domain decomposition based on the orthogonal recursive bisection of the N-Body volume into rectangular sub-volumes for the purpose of load balance. The code is written with MPI to handle the message passing on the Cray T3D, Paragon and the IBM PS/2. A PC-based parallel treecode has also been developed [31]. In this implementation, another well-known message passing library software, PVM (Parallel Virtual Machine), serves as the background for processors on clusters of personal computers to communicate with each other. The performance of a PGHPF (Portland Group High Performance Fortran) program running on a Cray T3E computer is reported in [15]. It is based on a dynamic and adaptive method for the domain decomposition, which makes use of the hierarchical data arrangement by the treecode.

Despite the fact that a wide variety of programming models have been utilized to parallelize the treecode algorithm, the hybrid MPI-OpenMP and GridRPC paradigms have been surprisingly excluded. The hybrid model is an emerging trend for fully exploiting SMP clusters while the GridRPC is evolving to provide a model for gridifying applications to maximize the use of resources. More importantly, the sequential treecode has one specialty which should be carefully taken into account in parallelization of the algorithm. That is, the specific routine for calculating the forces on the bodies accounts for upwards of 90% of the cycles in typical computations. It is deemed eminently suitable for obtaining higher level of parallelism with light-weight OpenMP threads in the hybrid MPI-OpenMP programming model and with multiple asynchronous GridRPC calls in the GridRPC programming model. This aspect is thought to have a positive effect on improving performance of the hybrid and GridRPC solutions to the treecode.

4 Parallelization of the treecode algorithm

4.1 MPI parallelization

4.1.1 ORB domain decomposition

The sequential treecode algorithm works well but the parallelization of the treecode is not immediately obvious since the tree is very unbalanced when the bodies are not uniformly distributed in their bounding box. The main difficulty is that both the bodies and the tree structure must somehow be distributed in a balanced way among many independent processors. Hence, it is important to divide space into domains with equal workloads to avoid load imbalance. As a result, the Orthogonal Recursive Bisection (ORB) domain decomposition [11] is adopted to divide the space into as many non-overlapping subspaces as processors, each of which contains an approximately equal number of bodies, and assign each subspace to a processor. It works as follows. First, a vertical line is found that divides the area into two areas each with an approximately equal number of bodies. Next, for each area, a horizontal line is found that divides it into two areas each with an equal number of bodies. This is repeated until there are as many areas as processors, and after that one processor is assigned to each area. Figure 12 shows an example of the ORB division in 2D space on 16 processors.

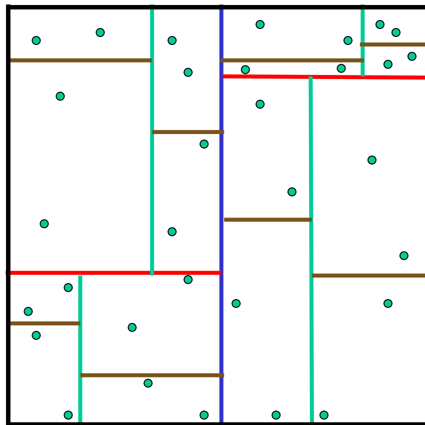

Figure 12. ORB domain decomposition in 2D space on 16 processors.

4.1.2 Construction of locally essential trees

Nonetheless, after the domain decomposition, each process has only the local tree for local bodies. In principle, they need the global tree to determine the forces due to the effect of influence ring along the borders. For example, node *n* belonging to process 0 has influence on bodies along the borders with P1, P2, and P3 as displayed in Figure 13, recall that θ is a constant called the opening angle. Thus node *n*, as well as other necessary nodes which

represent clusters of bodies, are called essential and must be known by P1, P2, and P3 to compute the forces of bodies in the influence ring of *n*. Those bodies that are not in the influence ring are either too close to node *n* to apply the clustering approximation, or far away enough to use *n*'s parent's information, therefore *n* will be essential to only bodies within its influence ring. Because it is too expensive to import an entire copy of the global tree to all the processors, a Locally Essential Tree (LET) for each processor is built instead.

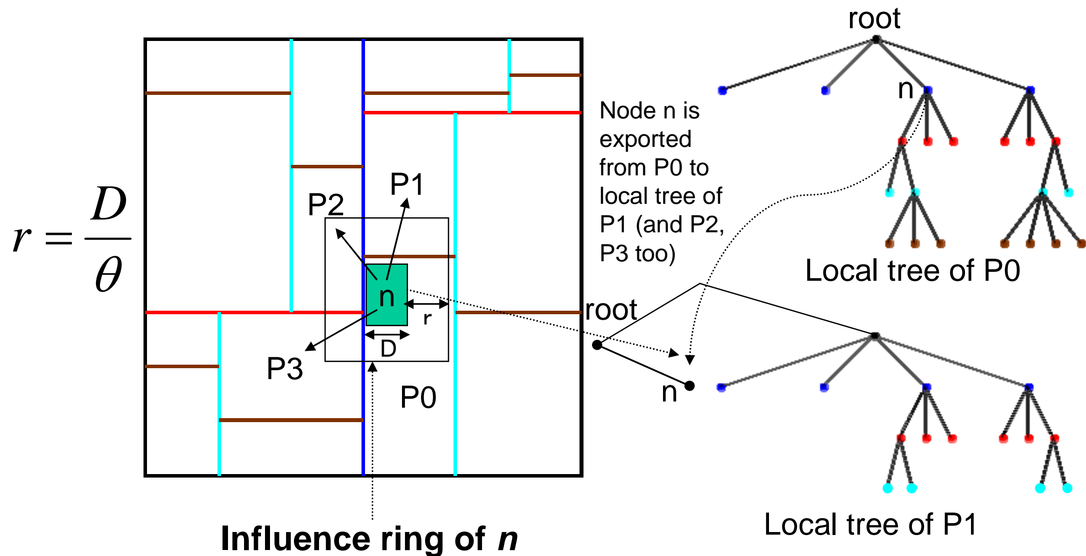

Figure 13. ORB decomposition and the influence ring of a node.

A LET of a processor is actually an extension of the local tree for its set of bodies which contains all the nodes of the global tree that are essential for the bodies contained within that processor, allowing the processor to run the sequential treecode for computing forces of its own bodies independently without requiring the global tree. Consequently, a processor only needs to import sub-trees from distant processors which it can insert on to its existing structure to create the locally essential tree. The criterion applied to determine if a node from a distant domain is essential and should be imported to the current domain is called the group-opening criterion. We let *n* be a node in the domain of processor 0, $D(n)$ be the length of a side of the square corresponding to *n*, and $r(n)$ be the shortest distance from *n* to the domain owned by processor 1 as depicted in Figure 14. Then if the ratio $\frac{D(n)}{r(n)}$ is greater than or equal to θ, *n* is essential to processor 1 and be a part of the sub-tree exported from processor 0 to processor 1.

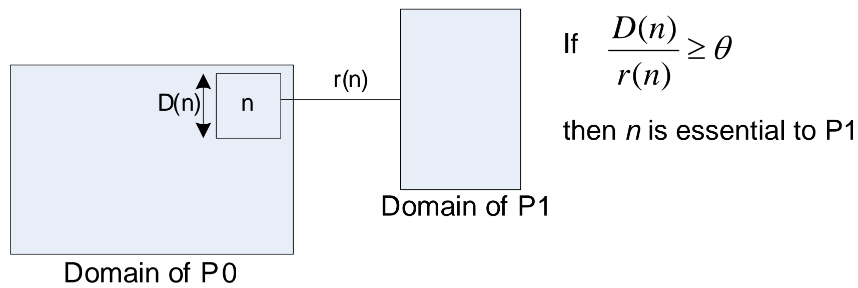

Figure 14. The group-opening criterion for determining essential nodes.

Practically, the locally essential trees are constructed in the following manner. After building the local tree, each processor imports the root nodes of the trees from all of the others. A local binary tree is built from the base up in each processor using the imported root nodes (Figure 15a). A walk through the binary tree using a group opening-criterion determines which processors must be examined further to gather more tree nodes if necessary. Tree walks are performed in the needed processors using the group opening criterion of the requesting processors. The essential nodes which satisfy the criterion are gathered together and exported to the calling processor. The sub-trees are inserted at the appropriate node of the binary tree (Figure 15b). The result of this procedure is the locally essential tree which contains the binary tree structure with the local tree and the sub-trees imported from other processors (Figure 15c). Once each processor has its own LET, they can proceed exactly as in the sequential case.

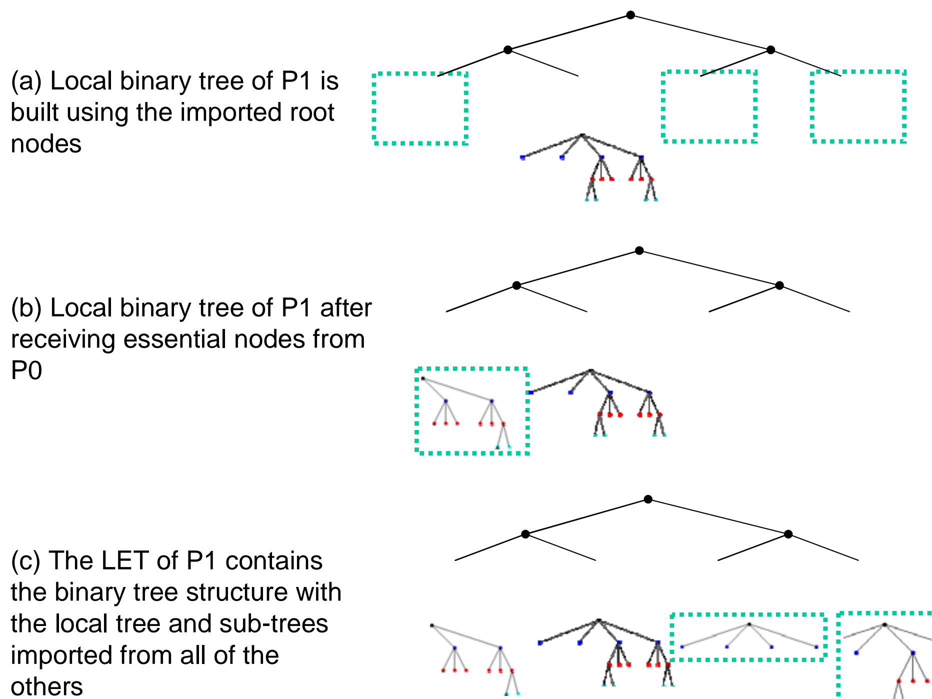

Figure 15. The construction of P1's LET, assuming there are 4 processes P0, P1, P2, and P3.

4.1.3 Summary and an example

In summary, the parallel treecode involves several steps as follows.

1. ORB domain decomposition across processors.
2. Construct the local trees.
3. Exchange tree nodes to construct the locally essential trees.
4. Walk through trees to calculate forces (sequential treecode algorithm from this step).

Figure 16 illustrates an example of the parallel treecode described previously with 2 processors in use. 20 bodies in the original domain are divided equally to these two processes so that each has 10 bodies. Further on, each process constructs the local tree for its own bodies. After that each process collects all the nodes in its domain deemed essential to the other based on the group-opening criterion, and exchanges these nodes directly with each other afterwards. Once two processes have received and inserted essential nodes into the local tree, they have their own LET and can carry out the sequential treecode for force calculation. The force on a body represented by a leaf in the system is evaluated by traversing down the tree from root. At each level, a node is added to an interaction list if it is distant enough for a force evaluation. Otherwise, the traversal continues recursively with the children. This procedure results in a list of interactive nodes for each body, for instance the body number 1 has 3 nodes in its list that contribute to the total force. The accumulated list of interactive nodes and bodies is then looped through to calculate the force on the given body.

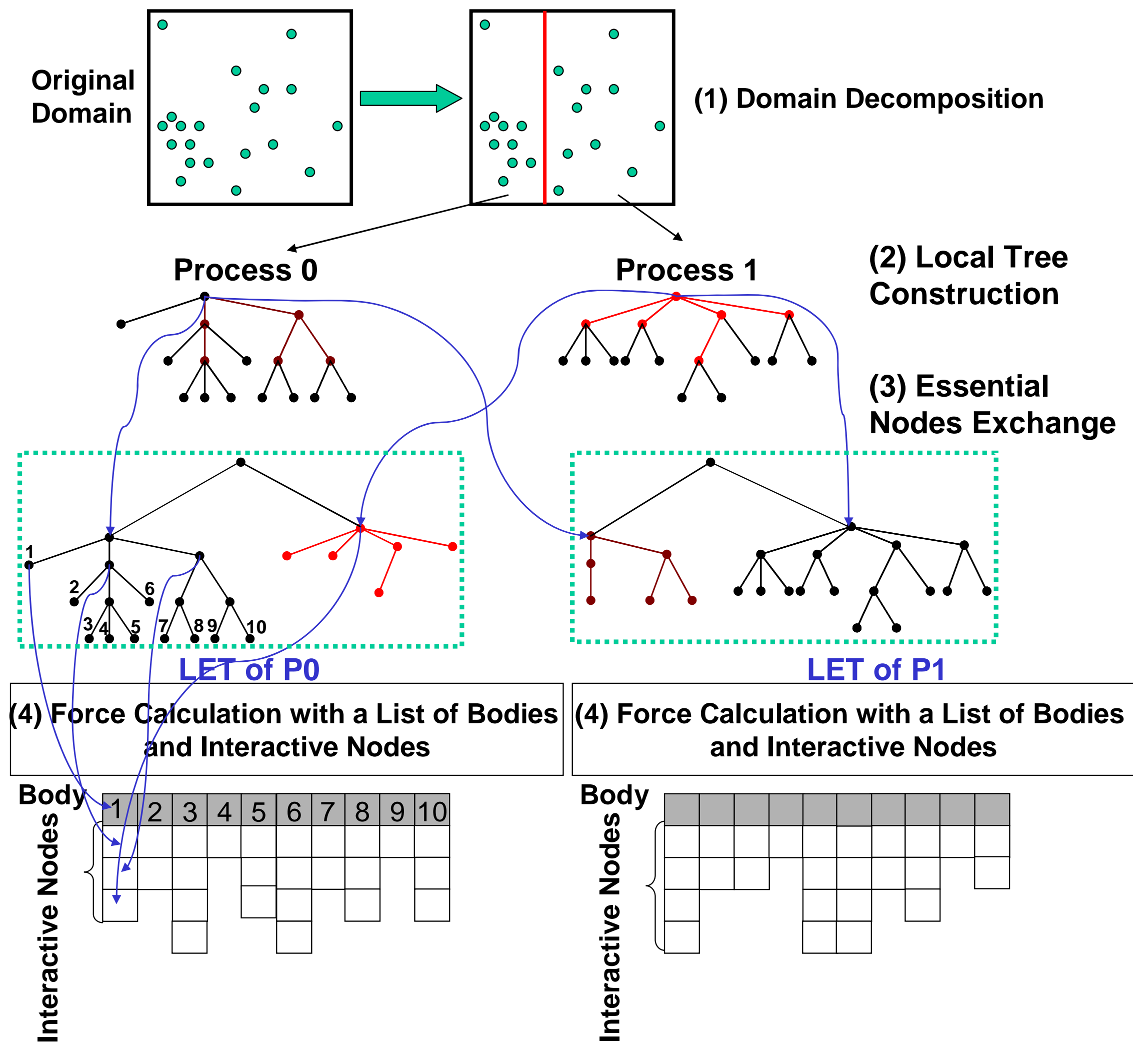

Figure 16. An example of the parallel treecode with 2 processors.

4.2 Multiple levels of parallelism with the hybrid model

4.2.1 MPI level and OpenMP level

In the hybrid implementation, multiple levels of parallelism are achieved as shown in Figure 17. For the first level using MPI parallelization, the hybrid program works exactly like the MPI one. The bodies are distributed in a balanced way among the MPI processes using ORB domain decomposition. After the local trees have been constructed, the processes collect and exchange essential nodes to each other to insert into and expand the local trees to LETs. Each process then walks through its own tree to create a list of interactive nodes for each body similar to the case of sequential algorithm. For the second level by OpenMP parallelization, the force calculation which accounts for upwards of 90% of the cycles in typical computations is eminently suitable for obtaining parallelism with OpenMP work-sharing threads running in each MPI process. The

bodies and their corresponding list of interactive nodes are assigned to different threads for calculating the force on each body. Hence, the hybrid program is expected to speed up the performance.

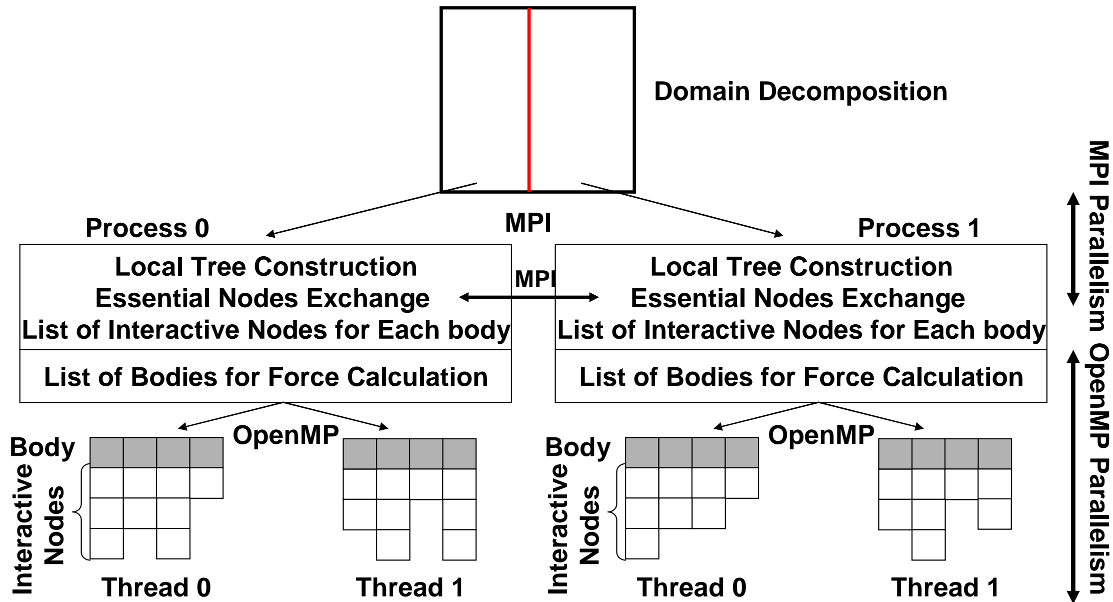

Figure 17. Multiple levels of parallelism with the hybrid implementation.

4.2.2 Loop scheduling methods of OpenMP threads

Moreover, in OpenMP loop parallelization of hybrid MPI-OpenMP program, there is no guarantee that just because a loop has been correctly parallelized, its performance will improve. In fact, in some circumstances parallelizing the wrong loop can slow the program down. Even when the choice of loop is reasonable, some performance tuning may be necessary to make the loop run acceptably fast. Thus, among several mechanisms for controlling this factor provided by OpenMP, loop scheduling with static, dynamic and guided is employed to ensure sufficient work with better load balance for OpenMP threads.

In static scheduling, iterations are divided into chunks of size *chunk* until fewer than *chunk* remain. Chunks are statically assigned to processors in a round-robin fashion: the first thread gets the first chunk; the second thread gets the second chunk, and so on, until no more chunks remain. The iterations are also divided into chunks of size *chunk* in dynamic scheduling, similarly to a static scheduling. Nevertheless, chunks are assigned to threads dynamically on a "first come, first do" basis; when a thread finishes one chunk, it is dynamically assigned another. Lastly, in the guided scheduling, for a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a

chunk size with value *k* greater than 1, the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than *k* iterations (except for the last chunk to be assigned, which may have fewer than *k* iterations). Chunks are dynamically assigned to threads just like dynamic scheduling. Figure 18 summarizes these 3 types of loop scheduling.

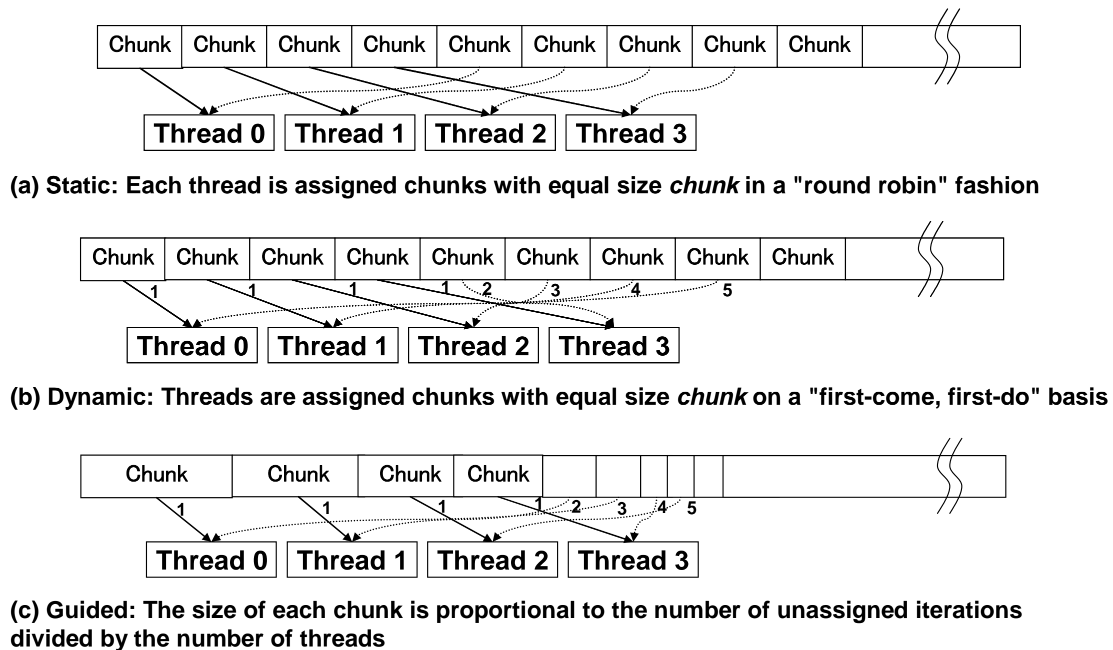

Figure 18. Static, dynamic and guided loop scheduling methods of OpenMP threads.

4.2.3 Operation of the hybrid implementation

The following section briefly presents pseudo code of the hybrid program.

```
{
…
MPI_Initialize();
ORB_domain_decomposition(MPI_processes, bodies);
Constructs_the_local_tree_code(my_bodies);
Build_the_LET(MPI_processes);
#pragma omp parallel for private (n) schedule (type)
//The workload here is divided among OpenMP threads
For body#0 to body#n in list of bodies {
Calculate_forces(interaction_list);
}
Move_bodies(my_bodies);
MPI_Finalize();
…
}
```

The operation of the hybrid implementation is demonstrated in Figure 19. Similar to the example for MPI program, two MPI processes are assigned two sub-domains of the original domain, each with an equal number of bodies. Next, they construct the local trees and exchange essential nodes with each other to build LETs. Finally, in each MPI process, the bodies and their corresponding list of interactive nodes are assigned to two different OpenMP threads for calculating the force on each body using static/dynamic/guided scheduling with appropriate chunk size.

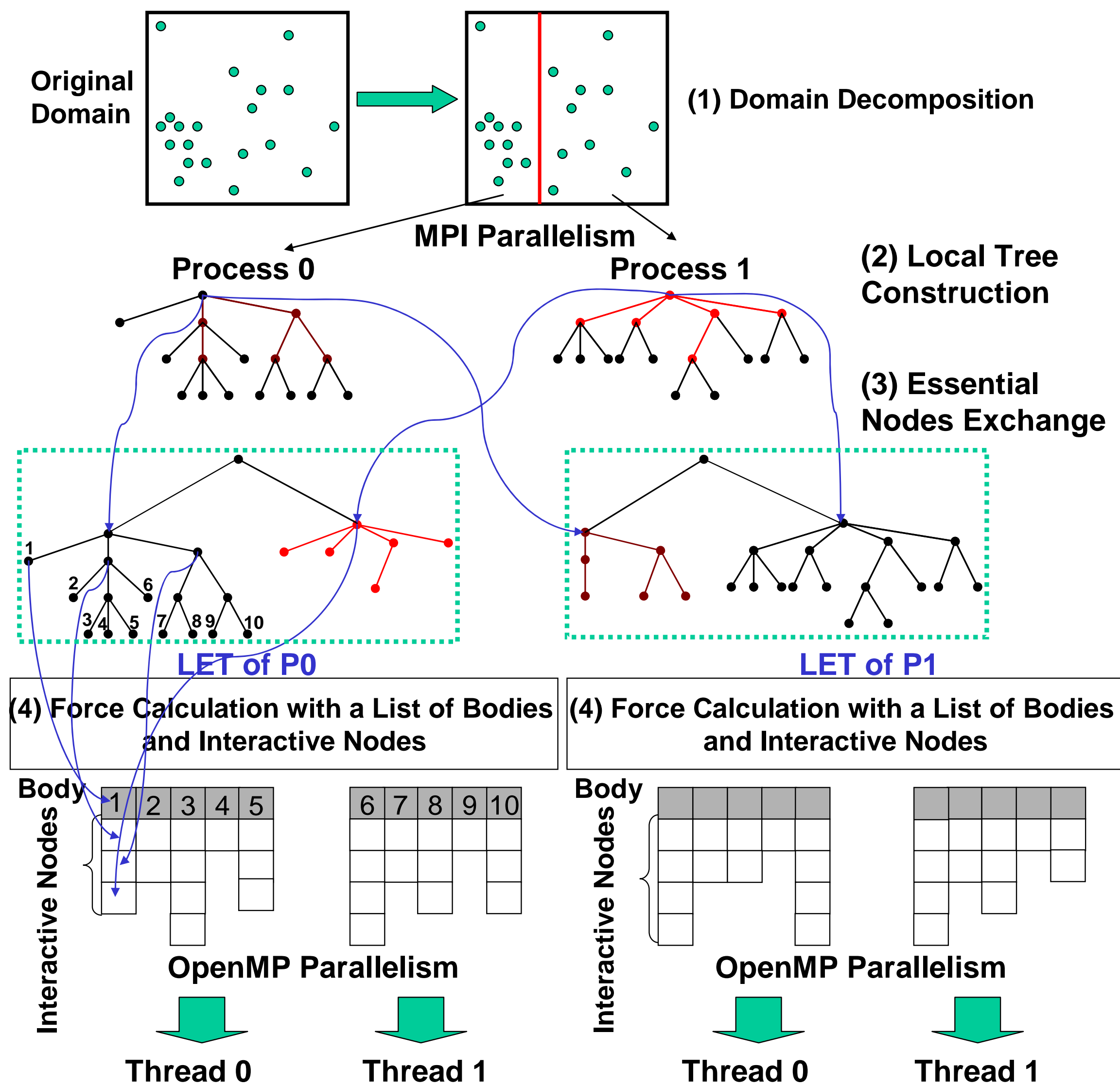

Figure 19. Operation of the hybrid code with 2 MPI processes and 4 OpenMP threads.

4.3 Parallelization using asynchronous GridRPC calls

So far, we have discussed the parallelization of treecode algorithm using MPI and hybrid MPI-OpenMP to develop implementations which are intended for clusters. With the existence of increasingly popular grids, gridifying existing applications to port them to a grid computing environment is an emerging trend for maximizing the use of resources. Fortunately, the workload of force calculation which accounts for most of the cycles is also perfect for obtaining parallelism with GridRPC calls and exploiting multiple clusters. A simplified description of the GridRPC implementation is illustrated in Figure 20.

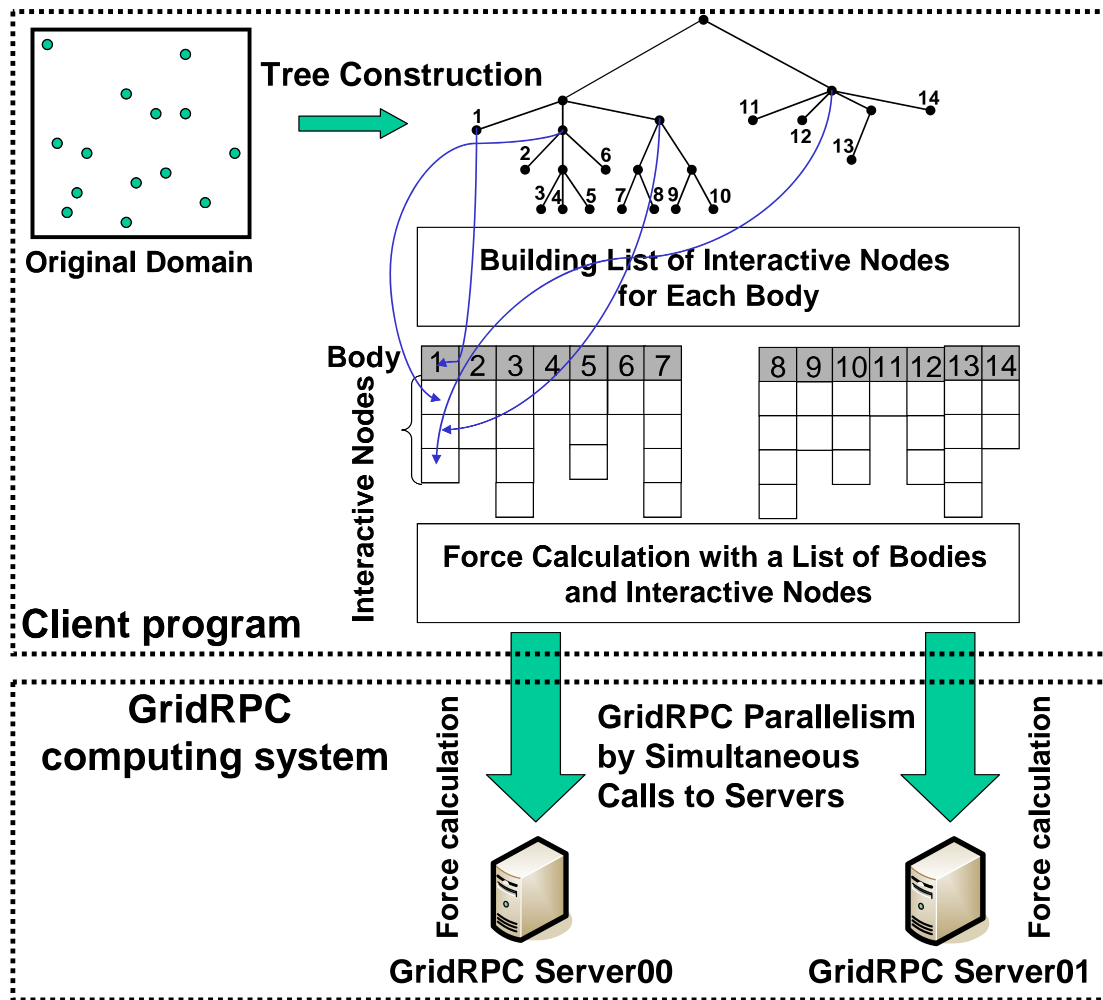

Figure 20. Exploiting parallelism with simultaneous GridRPC calls.

After the tree construction phase for the original domain, the list of interactive nodes for each body is built by walking through the tree. Since the calculation of the summation force on each body with its list of interactive nodes is completely independent from each other, the bodies and their corresponding lists of interactive nodes are divided among the servers by simultaneously

calling multiple asynchronous GridRPC requests to them. The list of bodies can be assigned to the servers dynamically by the agents or statically by the users. If the agents are responsible for the assignment, they will decide the number of bodies which the servers should execute automatically. Otherwise, the number of bodies for each server is explicitly specified by the users. In the figure, two GridRPC calls are initiated to run on two GridRPC servers at the same time; each is explicitly given 7 in a total of 14 bodies for force calculation. Actually parallelism achieving method in GridRPC system is quite similar to OpenMP parallelism in the hybrid MPI-OpenMP programming model as presented in the preceding section, but utilizing simultaneous GridRPC calls over separate servers instead of parallel OpenMP threads on different processors.

In practice, there are two modules to be implemented, the server module for computing the forces and the client module for calling the server module. The server module receives input data, in particular the bodies and lists of interactive nodes, from the client module and returns the list of bodies with the updated force, position and velocity after each calculation. The implementation of the server module is deployed on all the servers together with the interface description file written in Interface Definition Language (IDL) to build the library. IDL is a language for describing interfaces for server functions and methods which are to be called by clients. The interface description file is then referred by the client program to learn how to correctly call the server module.

The client program controls the execution of the whole computation. In the client module, because GridRPC client calls before initialization or after finalization will fail, they are surrounded by grpc_initialize and grpc_finalize for initializing and releasing resources. Once the initialization phase has been finished, grpc_function_handle_default is called for creating a new function handle to use the servers. Later on, the grpc_call_async API for asynchronous GridRPC calls is used in order to call the server module on servers remotely and concurrently with the handle. Consequently, the grpc_wait_all API is inserted after the asynchronous calls to wait for the completion of all of them for synchronization, followed by the grpc_function_handle_destruct to release the function handle after calling the remote function successfully.

Furthermore, since the GridSolve/NetSolve middleware integrates a self-scheduler running on the agents which automatically allocates remaining jobs to idle servers and keeps track of failed ones, it is unnecessary to explicitly implement the load balancing portion in the client module. The client program which assigns the bodies and corresponding lists of interactive nodes to execute on servers for force calculation using GridRPC calls is outlined as follows.

```
  {
  …
  Constructs_the_tree(bodies);
  Build_lists_of_interactive_nodes(bodies);
  …
  grpc_initialize();
  grpc_function_handle_default("ComputeForces");
  /* calling multiple asynchronous GridRPC requests for force calculation on
servers */
  foreach body
   grpc_call_async(handle,current_body,current_list_of_interactive_nodes);
  grpc_wait_all();
  grpc_function_handle_destruct(handle);
  grpc_finalize();
  Move_bodies(bodies);
  …
  }
```

Figure 21 displays the time chart for the operation of the client program in conjunction with GridRPC calls to server module on the servers.

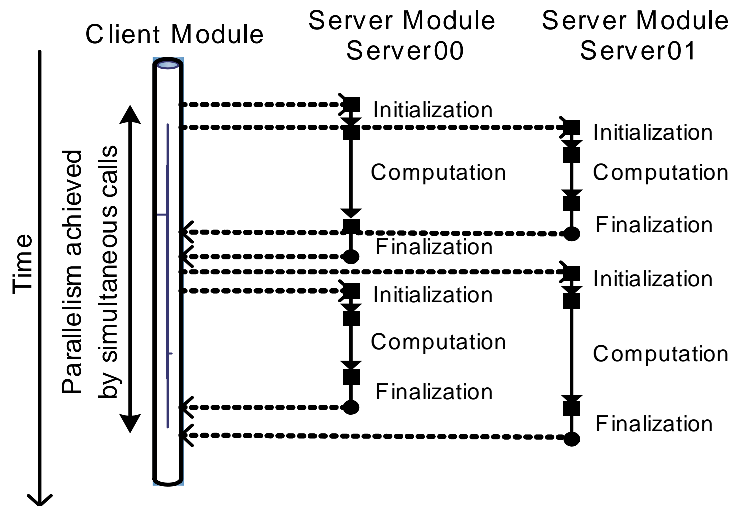

Figure 21. Time chart of the client program with GridRPC parallelism.

Clearly, the initialization time and finalization time also contribute to the total computing time in addition to the computation time. However, they remain almost constant throughout the computation. Thus, the longer computation time is, the lower related overhead is, resulting in the higher parallelism acquired. This is one of the key aspects having an enormous impact on the performance of the GridRPC implementation.

5 Performance evaluation and discussion

5.1 Computing platforms

System specifications of 3 clusters used for evaluating the parallel solutions are detailed in Table 3. The first 2 clusters, Diplo and Atlantis, are SMP clusters and hence they were exploited for performance evaluation of the MPI and hybrid programs.

Table 3. System specification of three clusters for performance evaluation

Name	Node	# of Nodes	# of CPUs	Memory per Node	OS	Network
Diplo	Quad Xeon 3GHz	4	16	4GB	CentOS4.4 with Rocks4.2	Gigabit Ethernet
Atlantis	Dual Xeon 2.8GHz	16	32	2GB	RedHat8.0 with SCore5.8	Gigabit Ethernet
Raptor	Pentium IV 3GHz	8	8	2GB	FedoraCore3 with SCore5.8.3	Gigabit Ethernet

With a view to providing a platform for the GridRPC implementation, a preliminary GridRPC computing system which consists of 2 clusters, Diplo and Raptor, has been constructed utilizing NetSolve middleware as depicted in Figure 22. The system comprises 2 agents, 12 servers with 24 processors. Basically, it is a RPC based client/agent/server system that allows users to remotely access both hardware and software components. At the top tier, the client library is linked in with the user's application which makes calls to GridRPC API for specific services. The tarbo and spino front-end nodes are designated as the primary and secondary agents, respectively. The agents monitor and maintain a database of servers along with a number of system parameters, which it uses to allocate server resources for client requests. The 4 compute nodes of Diplo and 8 compute nodes of Raptor operate as the servers of the system, executing remote functions on behalf of clients. A typical call from a client to the system involving several steps is illustrated in the figure too.

The MPI and hybrid codes are compiled using Intel C compiler version 9 and MPI library 1.2 of MPICH implementation. The GridRPC client program and server module are built by GCC compiler with GridRPC library supplied by the NetSolve system. All of them, MPI, hybrid, and GridRPC implementations, were tested with 3 data sets of 10,000, 50,000 and 100,000 bodies in 10 time-steps. On these systems, we repeated the experiments 5 times and observed small performance variations with acceptable standard deviation for 5 measures. The average of these 5 measures is presented.

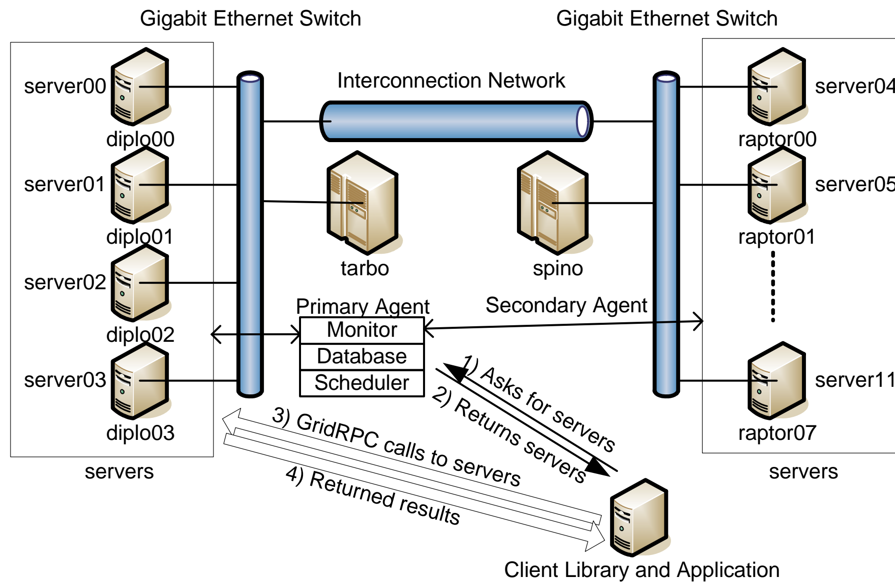

Figure 22. The GridRPC computing system and a typical call from client.

5.2 Performance comparison between MPI and hybrid

The timing runs of the MPI and hybrid MPI-OpenMP codes with the data sets of 10,000, 50,000, and 100,000 bodies for a fixed number (1, 2, 4, 8, 16, and 32) of CPUs on 2 SMP clusters, 4-way Diplo and 2-way Atlantis, are displayed in Tables 4, 5, 6 and Figures 23, 24, 25 respectively.

In both clusters, it is easy to recognize that the hybrid implementation outperforms the corresponding pure MPI one at all times whatever processors and data sets are used. The benefits of multiple levels of parallelism offer a significant performance improvement of approximately 30% on 4-way Diplo cluster and 20% on 2-way Atlantis cluster for the hybrid program. The average difference between the hybrid and MPI on Diplo is higher than on Atlantis, resulted from a greater number of OpenMP threads on Diplo in contrast to Atlantis because there are 4 OpenMP threads to be created on a 4-way compute node, and only 2 OpenMP threads on a 2-way compute node for each MPI process to best suit the system architecture. As a result, it is thought that this factor would be even higher in 8 or 16-way clusters although the codes have had no opportunity to be evaluated on such systems.

On the other hand, the performance of both the hybrid and MPI programs on 4-way cluster is quite different as against that on 2-way cluster. Given the same size of data set, there are times the experimental results obtained on 4-way cluster are better than on 2-way cluster for both programs, and vice versa, the results on 4-way cluster are sometimes worse than that on 2-way cluster. With 10,000 bodies to be tested, even though the hybrid program on Diplo is faster in

case of 2 and 4 CPUs, it begins to become inferior and slower than on Atlantis when the number of processors is increased to 8 and 16. Similar outcome is gained with the MPI program; its performance on Diplo is better than on Atlantis with 2 and 4 CPUs, but gets worse if more than 4 CPUs are employed. Since the number of bodies is small, resulting in short computation time and hence communication time becomes sizeable and comparable to computation time. Because Atlantis is built with the SCore Cluster System Software on top of the PM low level communication library which has been proven to provide higher performance [27], communication between Atlantis nodes is faster than between Diplo nodes. Thus with 2 and 4 CPUs in use, i.e., these processors are in the same node, meaning no communication is required between Diplo nodes, the performance on Diplo is better. Nonetheless, when communication between Diplo nodes takes place in case of 8 and 16 CPUs, the merit of SCore begins to take effect and raises the Atlantis's performance.

Table 4. Execution time of MPI and hybrid programs with 10,000 bodies (seconds)

	# OF CPUS	1	2	4	8	16	32
Diplo	Hybrid	-	22.3	11.4	6.7	3.4	-
	MPI	42	23.1	13.1	7.9	3.9	-
Atlantis	Hybrid	-	23	11.6	5.8	3	1.6
	MPI	46.8	23.5	13.5	6	3.1	1.7

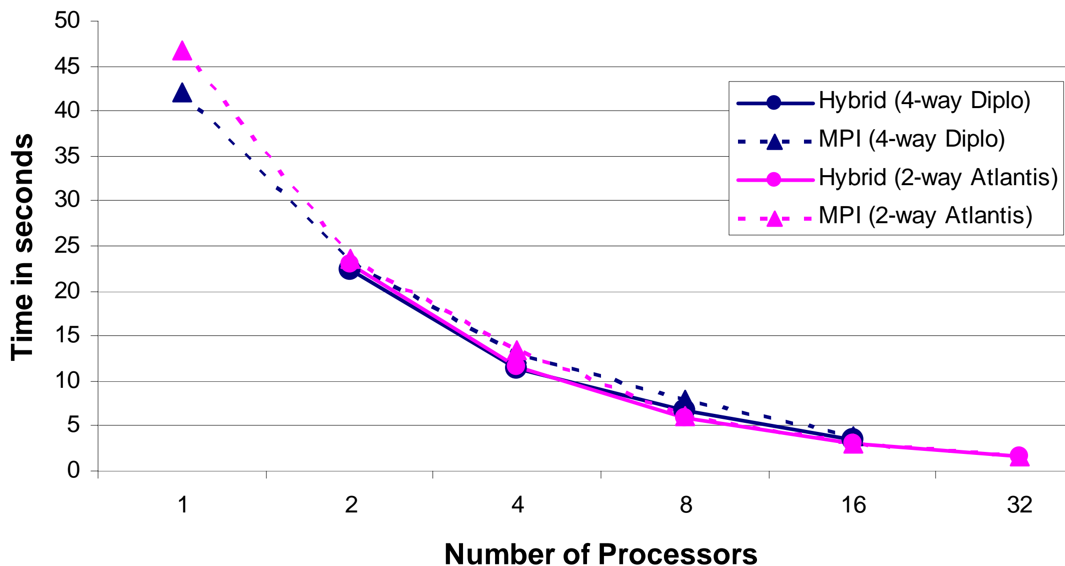

Figure 23. Execution time of MPI and hybrid programs with 10,000 bodies.

When the size of data set rises from 10,000 to 50,000 bodies, the performance of 4-way cluster is also improved together with the longer computation time. In this case, the hybrid and MPI programs on Diplo only lose to those on Atlantis with 16 CPUs. For the remaining combination of processors (2, 4, and 8), they show the performance of a higher quality on 4-way cluster than on 2-way cluster. The possible explanation for this consequence is much the same as in the previous data set. Based on that reason, all 4 nodes of Diplo need to communicate to each other when 16 processors are utilized, causing a negative effect on the overall performance of Diplo contrasted with Atlantis.

Table 5. Execution time of MPI and hybrid programs with 50,000 bodies (seconds)

	# OF CPUS	1	2	4	8	16	32
Diplo	Hybrid	-	508	258.7	134.5	81.1	-
	MPI	1151.4	575	287.9	144.2	85.6	-
Atlantis	Hybrid	-	545.2	272.9	137.5	70.8	42.1
	MPI	1192.2	596.5	300.4	146.5	73.9	44.1

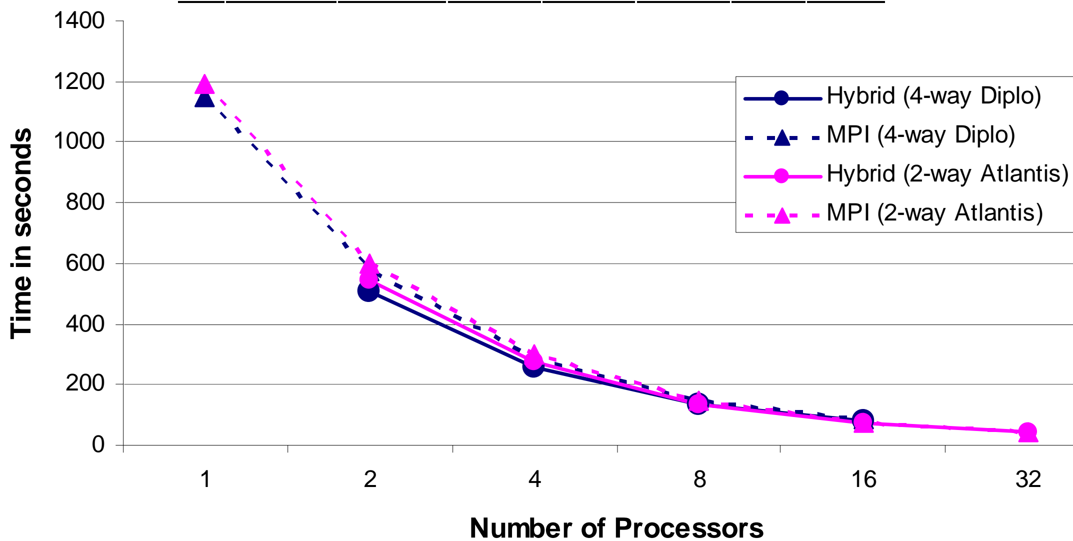

Figure 24. Execution time of MPI and hybrid programs with 50,000 bodies.

The biggest data set of 100,000 bodies presents the most interesting results. Since the computation time is greatly lengthened with an increase in the number of bodies, the hybrid implementation has a chance to fully exploit the use of OpenMP threads for sharing the workload on each SMP node. It is obvious that the Diplo cluster with 4 processors in each node has an advantage over the 2-way Atlantis cluster. Consequently, the hybrid program's performance on 4-way cluster is always superior to that on 2-way cluster. Nevertheless, the

execution time of the MPI implementation on Diplo, by contrast, is longer than on Atlantis at all times. The outcome is likely caused by the communication overhead generated among compute nodes of Diplo. Even though the communication time is not substantial beside computation time, it still grows with increased interactions among bodies and slows down the performance of MPI code on Diplo. As a result, the difference factor between the hybrid and MPI on Diplo in this case is the largest among 3 data sets.

Table 6. Execution time of MPI and hybrid programs with 100,000 bodies (minutes)

	# OF CPUs	1	2	4	8	16	32
Diplo	Hybrid	-	48.6	24.9	12.6	6.4	-
	MPI	134.5	71.6	37.8	19.7	10.4	-
Atlantis	Hybrid	-	54.3	27.1	14.8	7.6	4
	MPI	124.3	63	31.9	16.2	8.3	4.3

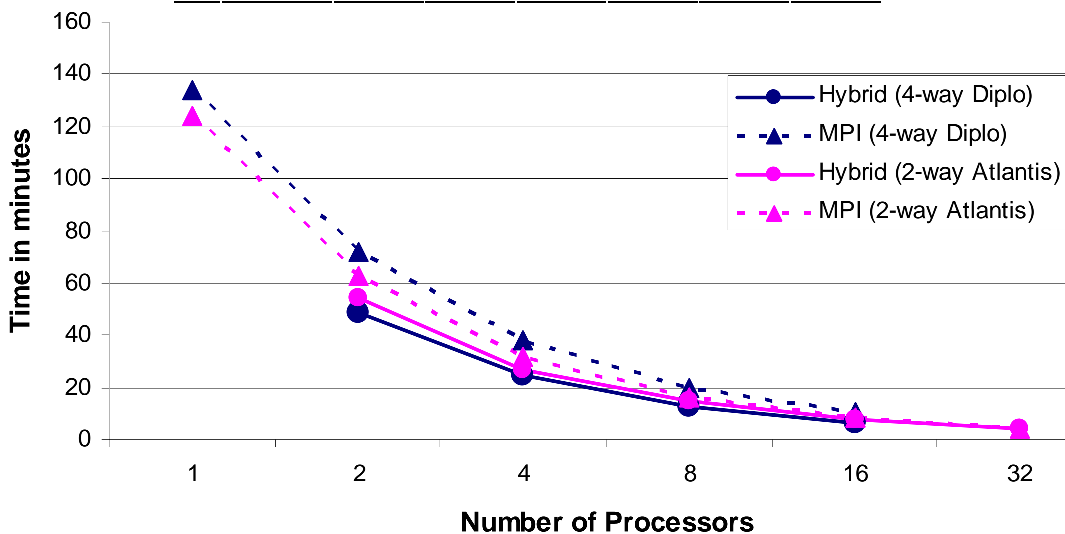

Figure 25. Execution time of MPI and hybrid programs with 100,000 bodies.

The performance of the hybrid program with a combination of different scheduling methods and chunk sizes on 4-way cluster was also evaluated using 8 and 16 CPUs with the data set of 100,000 bodies. The timing results obtained by running the hybrid code are listed in Table 7.

From this table, schedule dynamic evidently outdoes schedules static and guided in most cases, and the chunk size has an important impact on the performance. With chunk sizes equal to or less than 1000 (16 CPUs) and 2000 (8 CPUs), execution time has not changed much with all schedule types. The chunk sizes of 500 and 1000 are the best with dynamic scheduling in case of 8 and 16 CPUs respectively. The quality of load balance drops with increasing chunk

size and running time grows steadily with chunk size greater than 1000 (16 CPUs) and 2000 (8 CPUs). As the bodies are not uniformly distributed in their bounding box, the force computation time varies enormously from one body to another. Therefore, a chunk size which is too large easily leads to load imbalance. The experimental results of the hybrid implementation reported earlier are collected with the schedule type of dynamic and the appropriate chunk size depending on the size of data sets to get the possibly highest performance, that is, 50 for 10,000, 200 for 50,000, and 500 for 100,000.

Table 7. Execution time with different schedules and chunk sizes (seconds)

	CHUNK SIZE	1	100	500	1000	2000	5000
8 CPUs	Static	756.7	755.7	754.8	757	752.3	906.8
	Dynamic	755.2	754.3	751.8	756.2	752.1	906.8
	Guided	754.1	758	765.5	756.3	767.7	908.8
16 CPUs	Static	385.9	389.9	387.8	385.7	409.8	447.4
	Dynamic	385.2	388.2	386.3	384.1	408.1	445.3
	Guided	385.6	387.1	388.7	390.6	407.9	445.6

5.3 Performance comparison between GridRPC and other versions

The GridRPC N-Body implementation was executed on the GridRPC computing environment which consists of the Diplo and Raptor clusters with the aim of evaluating and comparing the performance with the MPI and hybrid ones. The execution time of the GridRPC code on GridRPC system, MPI and hybrid codes on 4-way Diplo cluster with the data sets of 10,000, 50,000, and 100,000 bodies are listed in Tables 8, 9, 10 and Figures 26, 27, 28, respectively.

Unlike the hybrid implementation which enjoys the benefit of multiple level parallelism regardless of data set sizes, the GridRPC program suffers a desperately poor performance with the execution time is almost twice as high as that of the hybrid program in case of the smallest data set of 10,000 bodies. The execution time is reduced gradually from 1 to 8 processors. However, this time rises when the number of CPUs is increased from 8 to 16 and from 16 to 24, leading to the best execution time conducted with 8 processors. The cause for this result is that the computation time for each GridRPC call on the servers is too short to take advantage of simultaneous calls and compensate the cost for initialization, communication and finalization, i.e., the parallel portion is not large enough compared to the sequential one of the program.

Table 8. Execution time of GridRPC, MPI, and hybrid with 10,000 bodies (seconds)

	# OF CPUS	1	2	4	8	16	24
Diplo	Hybrid	-	22.3	11.4	6.7	3.4	-
Diplo	MPI	42	23.1	13.1	7.9	3.9	-
Grid	GridRPC	73.1	47	29.2	21.8	25.1	27.5

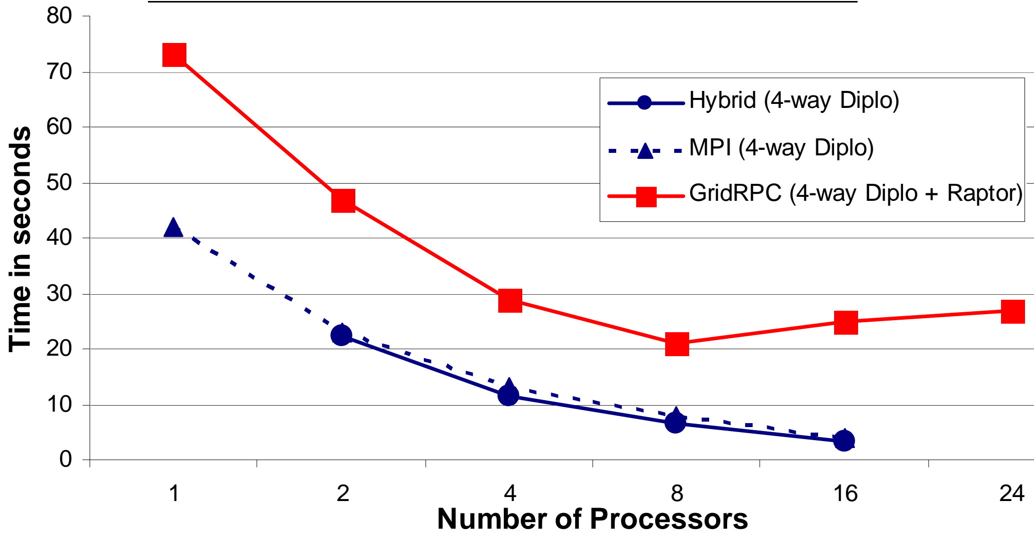

Figure 26. Execution time of GridRPC, MPI, and hybrid with 10,000 bodies.

When the data set grows to 50,000 bodies, more bodies require longer computation time for the GridRPC calls, causing an improved result of the GridRPC code. Even though the MPI and hybrid versions still strongly exhibit a higher performance, the gap between them and the GridRPC implementation is narrowed. The GridRPC program is about 30% slower, but scales well at all times. The more processors are used, the lower execution time is. The peak performance is gained by 24 CPUs, giving rise to an execution time nearly equivalent to that of the MPI version with 16 CPUs.

Table 9. Execution time of GridRPC, MPI, and hybrid with 50,000 bodies (seconds)

	# OF CPUS	1	2	4	8	16	24
Diplo	Hybrid	-	508	258.7	134.5	81.1	-
Diplo	MPI	1151.4	575	287.9	144.2	85.6	-
Grid	GridRPC	1667.5	840	431.3	224.1	130	115

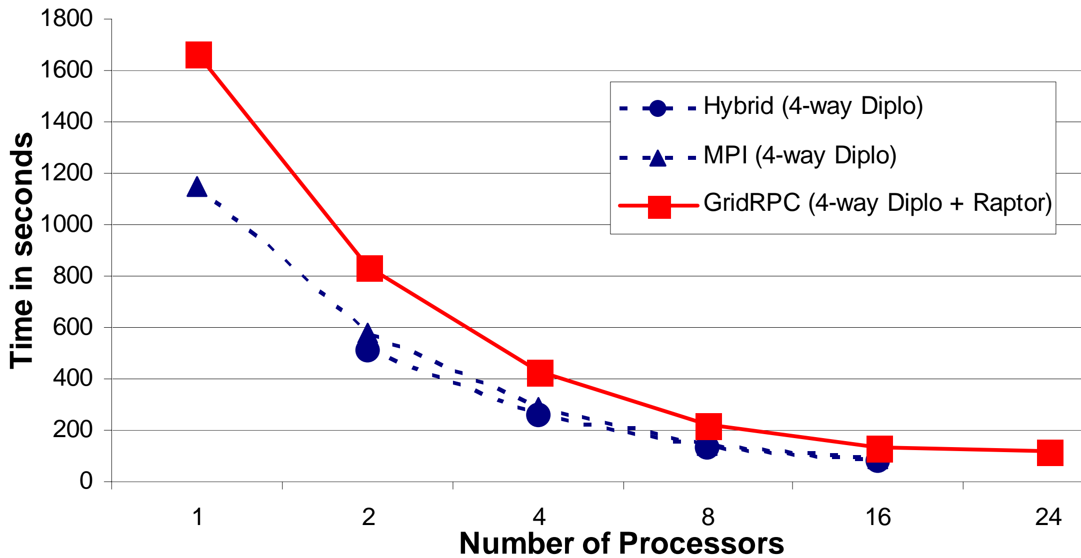

Figure 27. Execution time of GridRPC, MPI, and hybrid with 50,000 bodies.

The best performance of the GridRPC program is achieved with the largest data set of 100,000 bodies. Finally, this time it leaves both the MPI and hybrid versions behind with a significantly reduced execution time, approximately 40% and 10% faster, respectively. With the increase in the number of bodies, the parallel portion executed by the GridRPC servers now becomes large enough to effectively exploit the use of simultaneous calls. Meanwhile, the sequential time is trivial and does not contribute to the total computation time as much as in the previous data sets. In addition, all the existing 24 CPUs of both clusters in the preliminary GridRPC computing system, Diplo and Raptor, are employed to again successfully acquire the best result, a rise in the number of processors which can never be accomplished by individual clusters.

Table 10. Execution time of GridRPC, MPI, and hybrid with 100,000 bodies (minutes)

	# OF CPUS	1	2	4	8	16	24
Diplo	Hybrid	-	48.6	24.9	12.6	6.4	-
	MPI	134.5	71.6	37.8	19.7	10.4	-
Grid	GridRPC	86.8	43.7	21.9	11.1	5.9	5.2

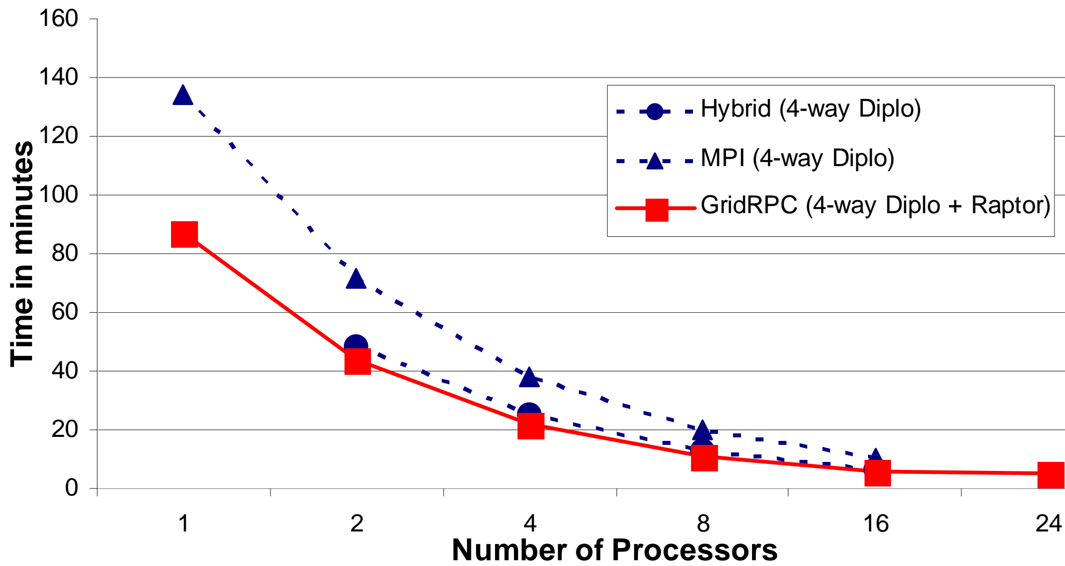

Figure 28. Execution time of GridRPC, MPI, and hybrid with 100,000 bodies.

5.4 Discussion

(1) Performance of MPI and hybrid

The performance evaluation and comparison show quite interesting results. Given the ability to obtain multiple levels of parallelism, the hybrid program outclasses the corresponding pure MPI program whatever processors and data sets are used. Based on the fact that performance of the hybrid program rises with the number of created OpenMP threads, the hybrid program is thought to even further lengthen the difference with the MPI one in 8 or 16-way clusters.

Furthermore, another big advantage of the hybrid model compared to pure MPI model is that it lowers the number of sub-domains in ORB domain decomposition. For instance, only 4 sub-domains are needed to create for the hybrid program while 16 sub-domains are necessary for the MPI program on 4-way Diplo cluster. As the number of sub-domains increases, the shapes of domains have a larger range of aspect ratios forcing tree walks to proceed to deeper levels. The complexity involved in determining locally essential data rises with the number of sub-domains as well. The number of node interactions grows with the number of sub-domains because of these effects. As a result, the hybrid model helps reduce this node interaction overhead.

(2) OpenMP loop schedulers

The adoption of different loop scheduling methods confirms that schedule dynamic is the best. As mentioned earlier, the routine calculating the force of bodies accounts for the vast majority of the cycles in typical calculations in this simulation. Synchronization overhead incurred by dynamic scheduling is trivial beside this computation time. Consequently, schedule dynamic is

always better than schedule static with all chunk sizes, and provides better load balance than schedule guided in most cases even though the difference among them is slim. Not only the schedule type but also the chunk size has an effect on the performance of the hybrid implementation. Therefore, choosing a chunk size is a trade-off between the quality of load balancing and the synchronization and computation costs.

(3) SCore system software

The size of the data set, in particular the communication time, affects the performance of Diplo cluster and Atlantis cluster too. Both the computation time and communication time are proportional to the data set size. By taking advantage of the benefit provided by SCore, Atlantis deals with communication overhead much better than Diplo which is vulnerable and can not tolerate that overhead. The result makes it clear why SCore has been widely chosen for building high performance computing cluster systems.

(4) GridRPC performance

Similarly, the size of the data set, i.e., the computation time of the GridRPC calls executed on the servers, was found to make a big difference to the performance of the GridRPC implementation. Most of the time the GridRPC code scales well with an increase in the number of employed processors. Nonetheless, the larger the data set is, the higher computation time is, bringing about the enhanced performance of the GridRPC program. With the largest data set of 100,000 bodies tested, the GridRPC performance is superior to that of MPI and hybrid codes with approximately 10% faster than the hybrid. Hence, it should come as no surprise that the implementation would demonstrate even higher performance with larger data sets. The consequences once again prove that the GridRPC programming model is well suited to problems with data intensive functions.

(5) The exploitation of resources

The lowest execution time is gained by an increased number of processors, 24, a great merit brought by the GridRPC model which outweighs the cluster model in terms of exploiting all the existing resources and producing higher throughput. Based on a RPC mechanism tailored for the grid, the GridRPC paradigm has successfully addressed the issues of stability, heterogeneity and security of grid computing and provided an attractive and simpler programming model on the grid. For solving large-scale scientific problems, the GridRPC model has given the impression of a potential candidate.

(6) Speedup

On the speedup front, the highest speedup factors of the MPI code are 29 on Atlantis when running with 32 processors for 100,000 bodies and 13.5 on Diplo with 16 processors for 50,000 bodies. The speedup of the hybrid implementation is not as obvious to calculate since it is inapplicable to the sequential case executing on a single processor. Still, providing that the speedup is estimated based on the execution time on 2 processors treated as having the factor of 2, the hybrid implementation presents the best speedup factors of 29 on Atlantis with 32 CPUs for 10,000 bodies and 15 on Diplo with 16 CPUs in case of 100,000 bodies. Regarding speedup, the GridRPC program appears to be inferior when compared to MPI and hybrid. In the worst case of 10,000 bodies, it can only give the best possible factor of 3.4 with 8 CPUs. Then again, as well as the performance, the speedup is gradually improved and the GridRPC achieves the highest factors of 14.5 for 50,000 bodies and 17 for 100,000 bodies, both with 24 processors.

(7) Memory capacity

Generally, there exists a great challenge that faced all the MPI, hybrid and GridRPC programming paradigms, arising from the fact that the memory requirements of the parallel treecode are quite large. Nearly all of the memory is assigned to the body and node arrays for the local set of bodies with additional arrays for bodies and nodes imported for the locally essential trees in the MPI and hybrid codes. Fortunately, on modern computing platforms with large amount of memory, it seems no longer a problem. No problem linked with memory was encountered on the systems used for evaluating the performance.

6 Conclusion

This dissertation reports the parallelization of the treecode algorithm for N-Body problem using MPI, hybrid, and GridRPC parallel programming models. With the hybrid model, after ORB domain decomposition among MPI processes, the workload of time-consuming routine for calculating forces of the bodies is shared among OpenMP threads. Besides, loop scheduling of OpenMP threads is employed with appropriate chunk size for better load balance in the hybrid program, resulting in enhanced performance. Given these abilities, the hybrid program outdoes the corresponding pure MPI program by average of 30% on 4-way cluster and 20% on 2-way cluster in terms of execution time whatever processors and data sets are used. With respect to the results, it is believed that for some certain classes of problems, the hybrid paradigm provides the most efficient mechanism to fully exploit clusters of SMP nodes.

Meanwhile, in the GridRPC code, this workload for calculating the forces on the bodies is parallelized using multiple asynchronous calls on servers. The performance of GridRPC program is determined by the larger, the better size of data sets, that is, the computation time of the GridRPC calls executed on the servers. In case of the largest data set of 100,000 bodies, the GridRPC version overtakes the MPI and hybrid versions with approximately 10% faster than the hybrid implementation. Consequently, porting to a multiple cluster computing environment using GridRPC is a proper approach to gain high performance and maximize the use of resources.

Nevertheless, the preliminary GridRPC computing system built for evaluating the GridRPC program has been made up of only 2 out of 3 clusters, Diplo and Raptor, lacking Atlantis due to networking-related problems. Thus joining Atlantis to the existing GridRPC system to provide higher throughput is one of the top priorities. On the other hand, adding graphics to the implementations for drawing the movement of the bodies in space and producing much more interesting effects is worth considering. In addition to MPI, hybrid and GridRPC models presented in the dissertation, shared-memory parallel programming is another noted paradigm with OpenMP emerged as the de facto standard. Making performance comparison among almost all of the popular programming models, message passing, shared-memory, hybrid, and GridRPC, is an appealing plan to offer a complete parallel solution to the treecode. Also, for solving the N-Body problem, a number of methods have been introduced to further reduce the time complexity in which the Fast Multipole Method (FMM) algorithm [16] has been shown to be $O(N)$. Adapting the parallel implementations using FMM algorithm is expected to greatly speed up the performance.

References

[1] T.V.T Duy, K. Yamazaki, and S. Oyanagi, "Performance Evaluation of Treecode Algorithm for N-Body Simulation Using GridRPC System", IPSJ and IEICE, Forum of Information Technology, 2007.

[2] T.V.T Duy, K. Yamazaki, and S. Oyanagi, "Performance Improvement of Treecode Algorithm for N-Body Problem with Hybrid Parallel Programming", The 69th IPSJ National Convention, 1A-6, 2007.

[3] T.V.T Duy, K. Yamazaki, K. Ikegami, and S. Oyanagi, "Hybrid MPI-OpenMP Paradigm on SMP clusters: MPEG-2 Encoder and N-Body Simulation", The 5th International Conference on Research, Innovation & Vision for the Future, pp. 157-158, 2007.

[4] K. Yamazaki, K. Ikegami, and S. Oyanagi, "Speed Improvement of MPEG-2 Encoding using Hybrid Parallel Programming", IPSJ and IEICE, FIT 2006, Information Technology Letters, LC-005, Vol.5, 2006.

[5] R. Rabenseifner, "Hybrid Parallel Programming: Performance Problems and Chances", Proc. of the 45th Cray User Group Conference, 2003.

[6] L.Smith, M.Bulk, "Development of Mixed Mode MPI/OpenMP Applications", Scientific Programming, Vol. 9, Numbers 2-3, pp. 83-98, 2001.

[7] I. Foster and C. Kesselman, "The Grid: Blueprint for a New Computing Infrastructure", Morgan Kaufmann, pp. 259-278, 1999.

[8] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra, "NetSolve: Grid Enabling Scientific Computing Environments", Grid Computing and New Frontiers of High Performance Processing Journal, Advances in Parallel Computing, 14, Article. 276, 2005.

[9] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "Overview of GridRPC: A Remote Procedure Call API for Grid Computing", Proceedings of the Third International Workshop on Grid Computing, LNCS 2536, pp. 274-278, 2002.

[10] Y. Tanaka, H. Takemiya, H. Nakada, and S. Sekiguchi, "Design, implementation and performance evaluation of GridRPC programming middleware for a large-scale computational Grid", Proc. of 5th IEEE/ACM International Workshop on Grid Computing, pp. 298-305, 2004.

[11] J. Dubinski, "A parallel tree code", New Astronomy 1, pp. 133-147, 1996.

[12] T. Hiroyasu, M. Miki, H. Shimosaka, M. Sano, Y. Tanimura, Y. Mimura, S. Yoshimura, and J. Dongarra, "Truss Structural Optimization Using NetSolve System", Meeting of the Japan Society of Mechanical Engineers, Vol. 2002, No.5 (20021004), pp. 141-146, 2002.

[13] M. G. Thomason, R. F. Longton, J. Gregor, G. T. Smith, R. K. Hutson, "Simulation of Emission Tomography with NetSolve", Technical Report, TR-CS-02-491, University of Tennessee, 2002.

[14] H. Shimosaka, T. Hiroyasu, and M. Miki, "Optimization Problem Solving System using GridRPC", Transactions of the Japan Society of Mechanical Engineers, Vol.72, No.716 (20060425), pp. 1207-1214, 2006.

[15] P. Mioochi, "Simulation of the dynamics of globular clusters: an efficient parallelization of a tree-code", Proc. of the conference "Dynamics of Star Clusters and the Milky Way", Vol. 228, pp. 526-528, 2001.

[16] J. Makino, "Yet another fast multipole method without multipoles --- pseudo-particle multipole method", Journal of Computational Physics, Vol.151, pp. 910-920, 1999.

[17] F. Cappello, and D. Etiemble, "MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks", Proc. of Supercomputing, Article No. 12, 2000.

[18] D. S. Henty, "Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling", Proc. of Supercomputing, Article No. 10, 2000.

[19] I.J. Bush, C.J. Noble, and R.J. Allan, "Mixed OpenMP and MPI for Parallel Fortran Applications", European Workshop on OpenMP, 2000.

[20] A. Kneer, “Industrial Hybrid OpenMP/MPI CFD application for Practical Use in Free-surface Flow Calculations”, Workshop on OpenMP Applications and Tools, 2000.
[21] Yun He, and C. HQ Ding, “MPI and OpenMP paradigms on cluster of SMP architectures: the vacancy tracking algorithm for multi-dimensional array transposition”, Scalable Computing: Practice and Experience, Vol. 5, No. 2, pp. 117-128, 2002.
[22] Treecode, treecode guide, http://ifa.hawaii.edu/~barnes/treecode/treeguide.html
[23] MPI, MPI: A Message-Passing Interface standard. Message Passing Interface Forum, June 1995, http://www.mpiforum.org/.
[24] OpenMP, The OpenMP ARB, http://www.openmp.org/
[25] GridSolve/NetSolve, http://icl.cs.utk.edu/netsolve/
[26] Ninf, http://ninf.apgrid.org/
[27] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato, "PM: An Operating System Coordinated High Performance Communication Library", High-Performance Computing and Networking, Vol. 1225, pp. 708-717, 1997.
[28] GridRPC Working Group, https://forge.gridforum.org/projects/gridrpc-wg/
[29] B. Wilkinson, and M. Allen, "Parallel Programming: Techniques and Applications Using Network of Workstations and Parallel Computers", 2nd Edition, Prentice-Hall, 2004.
[30] Barnes-Hut Galaxy Simulator, http://www.cs.princeton.edu/courses/archive/fall04/cos126/assignments/barnes-hut.html
[31] H.R. Viturro, and D.D. Carpintero, “Parallelized tree-code for clusters of personal computers”, Astronomy & Astrophysics, Supplement Series 142, pp. 157-163, 2000.