Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

Parsley: A Scalable Framework for Dependence-Driven Task Scheduling in Distributed-Memory Multiprocessor Systems

MASAKAZU SEKIJIMA[1] SHINYA TAKASAKI[2] SHUGO NAKAMURA[1] MITSUNORI IKEGUCHI[1]
KENTARO SHIMIZU[1,2]

{*masakazu,takasaki,shugo,ike,shimizu*}*@bi.a.u-tokyo.ac.jp*
Department of Biotechnology, The University of Tokyo[1]
Department of Information Science, The University of Tokyo[2]
1-1-1 Yayoi, Bunkyo-ku, Tokyo 113-8657, JAPAN

Abstract

This paper describes the design and implementation of a new parallel programming environment called *Parsley*, which provides fine-grained scheduling services based on the structure of the application programs. In Parsley, application programs are divided into subtasks which may run serially or in parallel. Parsley provides a programming interface that allows a user to define subtasks and to specify the precedence constraints among them. According to this specification, the Parsley system schedules subtasks and allocates processors. Thus, the subtasks are executed in a dependence-driven manner. We developed a parallel molecular dynamics simulation program based on the Parsley mechanism and executed it on a scalable multiprocessor system. We achieved good scalability and showed that our system is efficient for large-scale molecular dynamics simulation.

Key Words: parallel programming environments, resource management, scheduling, molecular dynamics simulation, computational chemistry

1 INTRODUCTION

There are many parallel computing environments that support resource management services, which lack in the existing message-passing libraries such as PVM and MPI. For example, distributed job-management and process-management systems such as Condor [7], Utopia [12], and PRM [1] provide flexible load-sharing services for heterogeneous, large-scale distributed systems. CARMI [8] provides a general framework for resource management. It provides a flexible structure that can support a variety of policies and aims at scalable resource management. DRMS [14] provides the means for application programs to specify their resource requirements and performs run-time program migration and data redistribution according to the specified requirements. Although some of these systems provide resource management facilities based on the applications' resource requirements, their resource allocation policies are restricted and load balancing is performed for tasks or processes that are almost independent.

VDCE [2] and HeNCE [5] use the dependencies of tasks for the resource allocation. They each provide a graphical interface that lets the user specify the dependencies used to control the allocation of resources. They differ from Parsley with regard to the granularity of tasks: VDCE and HeNCE provide only task-level (process-level) scheduling, whereas Parsley can perform subtask-level (described later) scheduling for the single-program, multiple-data (SPMD) environment. More finely-grained parallelism can be achieved by an approach that exploits application-specific parallelism specified by a language compiler or an application program. DUDE [6] and Chores [4] are run-time systems that schedule nested loops on multiprocessors. They perform dependence-driven scheduling by using information about data dependencies or control dependencies between iterations passed from compilers. These run-time systems are able to provide fine-grained, dynamic resource allocation according to the application program structure, but they need language compiler cooperation and are applicable only to specific program structures (e.g., nested loops). In MARS [13], a dependency graph is built from an application program and is used for the process-level scheduling. MARS also gathers statistical data on the behavior of the application programs by monitoring the CPU work load and communication time. Such historical information is effective in improving the scheduling policies and is also used in our system.

In this paper we describe the design and implementation of a new parallel programming environment called *Parsley*, which provides fine-grained resource management services that reflect the structure of the application programs. In Parsley, application programs are divided into subtasks which may run serially or in parallel. A user can define subtasks according to the application's structure without considering the size of subtasks for load balancing. Parsley provides a programming interface that allows a user to define subtasks and to specify the precedence constraints among them. Parsley uses these constraint's to determine the order in which subtasks are executed (schedules subtasks) and assigns subtasks to the processors. The subtasks are thus, executed in a dependence-driven manner. Dependence-driven execution is similar to the data-driven execution used in data-flow models except that the control dependencies as well as the data dependencies are used in determining the execution sequence. Its asynchrony results in a high degree of concurrency. Parsley also makes it easy to develop parallel programs because it performs load balancing automatically at execution time. We developed a parallel molecular dynamics (MD) simulation program for the dependence-driven execution mechanism of Parsley and ran it on a scalable multiprocessor system, the HITACHI SR2201.

2 SYSTEM DESIGN

2.1 System Architecture

Figure 1 shows the architecture of our current implementation of the Parsley system. It is implemented on the MPI, which provides the interprocess communication facilities. In NOW (network of workstation) it exchanges state information between workstations, but in parallel computers its function is very restricted: it simply monitors the execution time of subtasks and provides resource use statistics useful for higher-layer policy modules. The policy modules provide resource management policies as well as the resource allocation policies. The task management service (TMS) provides the mechanisms of resource allocation for the MPI tasks. This module is useful for resource sharing in distributed systems. The subtask management service (SMS) is a key module for the fine-grained, dependence-driven scheduling described in this paper. This module is used for multiprocessors that employ SPMD style parallel processing as well as for conventional distributed systems. Its functions are described in the following sections.

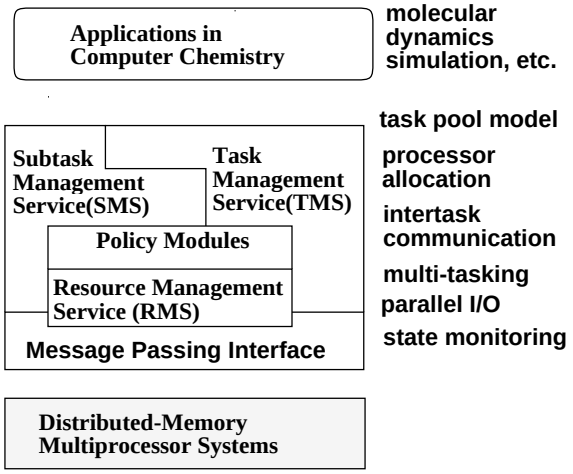

Figure 1: System architecture.

2.2 Programming Models

Application programs are divided into multiple subtasks that may run serially or in parallel. Normally, there are some precedence constraints that must be enforced. A typical precedence constraint is that one subtask needs to use the results of one or more other subtasks. Parsley provides a programming interface that lets the user defines subtasks and specify their precedence constraints.

The programming model is based on a master-slave model (Figure 2). A master manages a pool of subtasks, called the subtask pool, and assigns subtasks in the pool to available slave processors according to the specified precedence constraints. A subtask becomes ready for execution when its immediate predecessors have already been executed; and if it does not have any predecessors, it is always ready for execution. When a processor completes a subtask, it notifies the master, and the master assigns to it the next ready subtask in the subtask pool. Since the processor allocation is performed dynamically, the definition of each subtask is independent of physical processors.

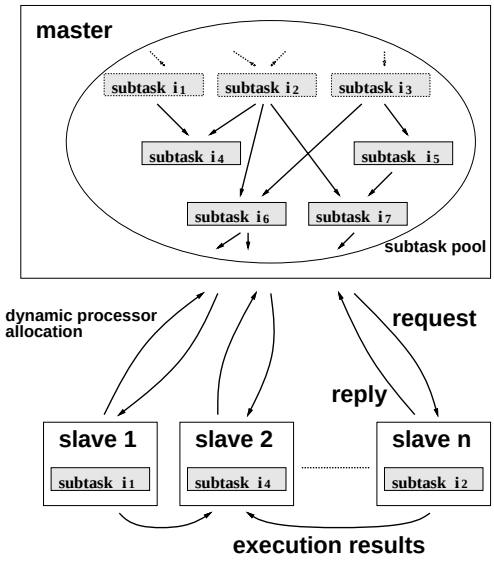

Figure 2: Subtask execution based on a master-slave model.

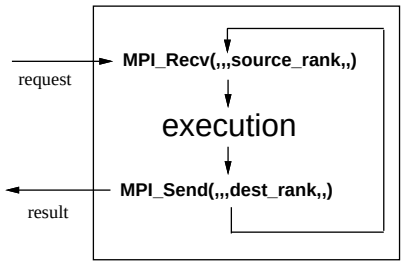

Figure 3: Slave-program structure in MPI.

Figure 3 and 4 show the slave program structures for SPMD programming. In both, a master processor sends the request, which consists of an operation code and parameters, to a slave processor. The program in slave processors has a loop structure: It receives the request, executes it, and then replies to the master. In this context a subtask is defined as a predefined set of operations that each slave processor executes in accordance with the request from the master processor. In Parsley, `Parsley_Send/Parsley_Recv` primitives are used to send/receive messages. These primitives correspond to `MPI_Send/MPI_Recv` primitives but they specify the subtask ID, not the rank of MPI processes, as the destination of the message passing. Inter-subtask communication is translated into MPI message-passing primitives at execution time. Note that the execution mechanism of Parsley itself can also be applied to conventional tasks or processes. In this paper, however, we apply it to an SPMD programming model for implementing efficient parallel processing on a massively parallel computer.

Figure 5 shows the example of how the user register subtasks and specifies subtask dependencies. A user registers a subtask by using the `Create_Subtask()` primitive and the system assigns a subtask ID that uniquely identifies a subtask. A user specifies the dependency between subtasks by using `Set_Subtask_Dependency()` to specifying these subtask IDs.

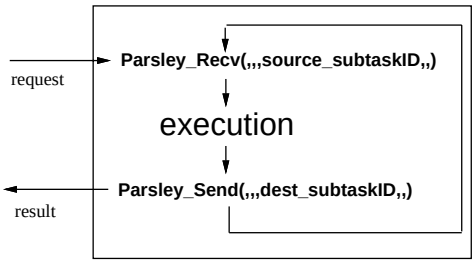

Figure 4: Slave-program structure in Parsley.

The Parsley systems maintains the following data structure for each subtask:

- a predecessors list (PL) containing the subtask IDs of its predecessors
- a successors list (SL) containing the subtask IDs of its successors
- user-defined control information which includes the operation of the subtask and the scope of the shared data (physically replicated) that the subtask manipulates
- statistical data about resource use (used for subtask scheduling).

```
        .
        .
        .
i1 = Create_Subtask();
i2 = Create_Subtask();
i4 = Create_Subtask();
        .
        .
        .
Set_Subtask_Dependency(i1,i4);
Set_Subtask_Dependency(i2,i4);
        .
        .
        .
```

Figure 5: Subtasks creation and dependency specification.

Parsley constructs this data structure when a user defines subtasks and specifies their precedence constraints. For example, the system adds subtask IDs to PLs and SLs according to the user-specified subtask dependencies. For applications in which precedence constraints are determined as execution proceeds, the subtask data structure is constructed dynamically at execution time.

2.3 Subtassk Scheduling

2.3.1 Passing the execution result

Two of the schemes that might be used for passing the execution result of each subtask to its immediate successors are the following:

- A slave sends the result back to the master, and then the master sends it to the successors.
- A slave sends the result directly to the successors.

Although the first scheme is simpler in that a slave need not maintain the execution result after the subtask execution, we use the second scheme to reduce the load on the master. In this scheme, when a subtask is completed, the execution result is temporarily stored in the slave. Note that at this time the destination processor, to which the execution result will be passed, is not determined if the successor subtask is not assigned to a processor. In our design, when a slave begins executing a subtask, it tries to get the execution results of its predecessors.

2.3.2 Granularity of subtasks

The granularity of subtasks greatly affects the performance: too small and the master is too busy allocating processor, too large and the effect of load balancing is reduced. It is easy for a programmer to divide a program into subtasks based on the original application algorithm. In some applications, however, such a straightforward approach may be inefficient. Parsley provides a mechanism allocating multiple consecutive subtasks to a processor at run time. When the subtasks are small, they are dynamically combined at run time in a manner that provides the best performance. Although the current implementation is very restricted, we are designing a more general and powerful framework for changing subtask granularities dynamically.

2.3.3 Scheduling policies

The dynamic processor allocation in Parsley can provide dynamic load balancing in a given hardware environment, but the current processor allocation policy is very simple: it allocates any available processors to any ready subtasks in a FIFO order. Performace can be improved by using a processor-allocation policy that considers the calculation and communication times of the subtasks. In addition, dynamic load balancing incurs on the overhead for transferring the execution request to the busy processor. We therefoe use static scheduling when a whole set of subatasks and their dependencies are known in advance. The Parsley system measures the calculation and communication times of subtasks and uses this information for the scheduling for the next run of the same application. We extended the CP/MISF algorithm [3] with distributed-memory multiprocessor systems by combining the communication time with the calculation time of each subtask.

3 PARALLEL MOLECULAR DYNAMICS SIMULATION ON PARSLEY

3.1 Molecular Dynamics Simulation

Molecular dynamics (MD) simulations are widely used for simulating the motion of molecules in order to gain a deeper understanding of the chemical reactions, fluid flow, phase transitions, and other physical phenomena due to molecular interactions. In this simulation of a continuous process is broken down into discrete small timesteps, each which is an iteration has two parts: force calculation (calculating the forces from the evaluated conformational energies) and atom update (calculating new coordinates of the molecules).

3.2 Parallel Programming on Parsley

There are two popular parallel MD simulation algorithms: atom decomposition (AD) [10] and space decomposition (SD) [9]. In AD each processor is assigned a subset N/P of

N atoms (P is the number of processors), and it updates each atom's position throughout the simulation. It distributes atoms among the processors evenly, thus achieving static load balancing, but it requires global communication for exchanging forces and atomic positions. In SD the simulation domain is usually broken into P subdomains (cells), and each processor computes forces on only the atoms in its subdomain. In the large N limit, this algorithm scales optimally as N/P, since each processor needs only information from processors that are assigned neighboring subdomains. The SD algorithm allows atoms to migrate between subdomains in order to maintain the locality of force calculation. Owing to its low communication cost, the SD algorithm usually performs better than the AD algorithm [11] . So we used the SD algorithm in present study. An important problem with the SD algorithm is the load imbalance caused by nonuniform atom density, which is typical of bio-molecules (e.g., proteins) surrounded by solvent water molecules. Figure 6(a) shows a such load imbalance in the SD algorithm. The subtasks for the force calculation and atom update in each timestep must be synchronized with each other.

Figure 6(b) shows an example of dependence-driven scheduling in Parsley for the same set of subtasks as in Figure 6(a). No barrier synchronization is required. A subtask can start its execution when all the predecessors complete their executions. High concurrency is obtained because the precedence constraints in the SD simulation are limited to subtasks of neighboring subdomains. Thus Parsley MD simulation proceeds asynchronously.

Figure 7 shows a part of subtask dependencies in the SD algorithm. Force calculation subtasks of timestep t send the results only to their successors (atom updating subtasks of timestep t). In the SD algorithm each subtask's successors are limited to subtasks of neighboring subdomains. The atom updating subtasks of timestep t can start execution without synchronization when they receive the results of their predecessors.

3.3 Results

We ran the simulation for a system consisting of one BPTI (bovine pancreatic trypsin inhibitor) protein surrounded by water molecules, and comprising a total of 16,375 atoms. Our MD simulation used the SD algorithm, and in the BPTI's simulation space was divided into 125 $5 \times 5 \times 5$ cells. Thus, force calculation and atom update were decomposed to 125 subtasks per timestep. Figure 8 is a log-log plot of the computational time (seconds) for 10 timesteps. The solid line shows the ideal (perfectly linear) speedup. We mean by the ideal speedup that the computation time for P processors is $1/P$ of that for a single processor.

As can be seen in the Figure 8, for from 1 to 16 processors the SD simulation is more efficient than the Parsley simulation. As the number of processors increases, however, the Parsley simulation becomes more efficient than the SD simulation. For 125 processors, the Parsley simulation is 3.49 times faster than the SD simulation and is 30.4 times faster than the simulation ran on a single processor. The performance degradation of the SD simulation for 16 - 125 processors is explained as follows. Efficient load balancing requires that number of simultaneously executable subtasks be sufficiently greater than number of processors. Load balancing becomes less effective, when the number of processors is close to the number of decompositions.

Figure 9 shows the details of the execution time. "Calculation" time in the figure includes time for calculation of

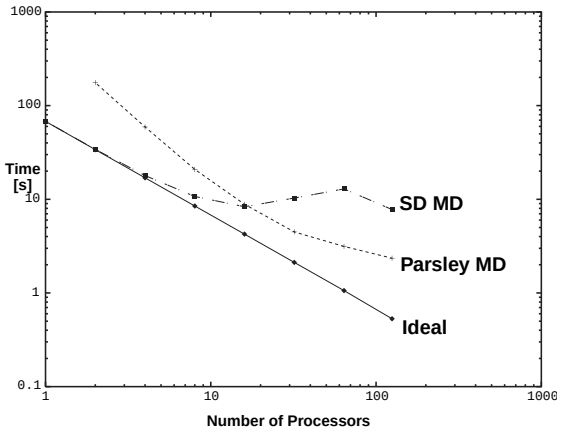

Figure 8: Performance of Parsley MD simulation and SD MD simulation (BPTI + water).

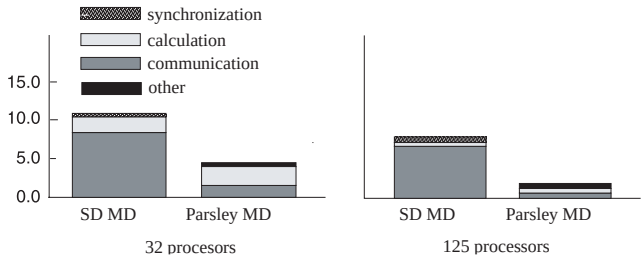

Figure 9: Details of execution times (BPTI + water).

the bonded forces and the nonbonded forces. "Other" includes the time for system processes such as memory allocation. As can be seen in this figure, for 125 processors, the SD algorithm's communication time is greater than that of the Parsley MD simulation. This difference is due to the load imbalance caused by nonuniform atomic densities. Parsley does not need barrier synchronization and can execute subtasks (force calculation and atom update) asynchronously, so load balancing is effective even when the number of processors is close to the number of decompositions.

3.4 Effect of Improving the Scheduling Policy

Parsley measures the execution time of subtasks and uses the results to improve the scheduling policy for the next execution of the program, and the consequent effect of the improvement for the MD simulation is shown by the values listed in Table 1. The initial policy is one that assumes all the subtasks have the same execution time. The improved policy (which uses the information about the previous execution times) performs better, but the improvement is rather small. This is because the basic scheduling policy does not consider the communication cost, which varies according to the number of processors allocated to the subtasks.

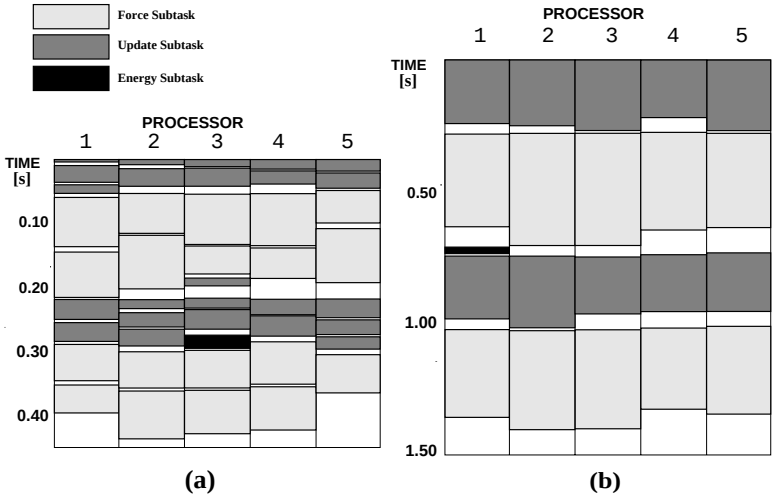

Figure 6: Processor utilization in (a) Parsley MD and (b) conventional parallel MD.

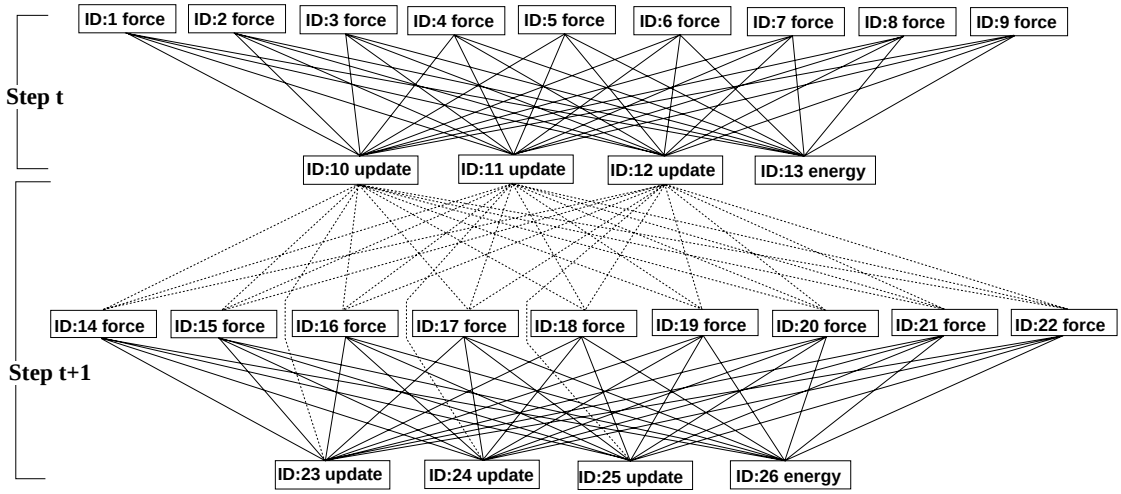

Figure 7: Subtask dependencies in a molecular dynamics simulation.

4 CONCLUSIONS

Application-specific dynamic parallelism is a key to high-efficiency parallel programs. Since the programmer is aware of the application-specific parallelism, it seems to be very effective for the programmer to pass such information to the run-time scheduling system. Unlike the conventional run-time scheduling systems that are coupled with compilers, our system allows a user to decompose an application program into subtasks and specify the precedence constraints between subtasks. It provides a more general framework for resource allocation and supports a much wider range of parallel applications. The Parsley system is very portable because it is implemented in C language and on MPI libraries, and it is now available on the HITACHI SR2201, the IBM SP/2, the Sun Ultra Enterprise 10000 (STAR FIRE), and the NEC Cenju. The dynamic load balancing and automatic improvement of scheduling policies can adopt the load balancing to the hardware environments.

We have developed, in addition to the MD simulation described in this paper, several computational chemistry applications such as the fast multipole method [15] and molecular mechanics internal coordinates. We found it is easy to describe the subtasks and their precedence constraints in Parsley. In this paper we have shown that our system is efficient for large-scale MD simulation and has good scalability. We are going to apply our system to more complicated applications such as multi-timestep MD and to the MD with the solvent-effect calculation. We think that our system will be even more effective for these applications because they include many different kinds of subtasks of various sizes.

References

[1] B. C. Neuman and S. Rao, The Prospero Resource Manager: A scalable framework for processor allocation in distributed systems, *Concurrency: Practice and Experience* **6**, 1994, 339-355.

[2] H. Topcuglu, S. Hariri, W. Furmanski, J. Valente, I.

Table 1: Execution times (seconds) for various of scheduling policies.

number of processors	dynamic allocation	static allocation (initial policy)	static allocation (improved policy)
16	6.38	6.17	5.77
32	2.73	2.99	2.58

Ra, D. Kim, Y. Kim, X. Bing, and B. Ye, The software architecture of a virtual distributed computing environment, *Proc. High-Performance Distributed Computing Conf.*, 1997, 40-49.

[3] B. Shirazi, M. Wang, and G. Pathak, Analysis and Evaluation of Heuristic Methods for Static Task Scheduling, *J. Parallel and Distributed Comput.* **10**, 1990, 222-223.

[4] D. L. Eager and J. Zahorjan, Chores: Enhanced run-time support for shared-memory parallel computing, *ACM Transactions on Computer Systems* **11**, 1993, 1-32.

[5] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Mancheck and V. S. Sunderam, Graphical development tools for network-based concurrent supercomputing, *Proc. Supercomputing 91*, 1991, 435-444.

[6] D. Grunwald and S. Vajracharya, The DUDE runtime system: An object-oriented macro-dataflow approach to integrated task and object parallelism, http://www.cs.colorado.edu/šuvas/paper/paper.html.

[7] J. Pruyne and M. Livny, Providing resource management services to parallel applications, *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, 1994.

[8] J. Pruyne and M. Livny, Parallel processing on dynamic resources with CARMI, *Job scheduling strategies for parallel processing, LNCS 949*, Springer-Verlag, 1995, 259-278.

[9] W. Smith, Molecular dynamics on hypercube parallel computers, *Comp. Phys. Comm.* **62**, 1990, 229-248.

[10] D. Fincham, Parallel computers and molecular simulation, *Molecular Simulation*, **1**, 1990, 1-45.

[11] S. Plimpton and B. Hendrickson, A new parallel method for molecular dynamics simulation of macromolecular systems, *J. Comp. Chem.*, **17**, 1996, 326-337.

[12] S. Zhou, X. Zheng, J. Wand, and P. Delisle, Utopia: A load sharing facility for large, heterogeneous distributed computer systems, *Software – Practice and Experiences*, **23**, 1993.

[13] J. Gehring and A. Reinefeld, MARS – a framework for minimization the job execution time in a metacomputing environment, *Future Generation Computer Systems*, **12**, 1996, 87-99.

[14] J. E. Moreira, V. K. Naik and R. B. Konuru, A programming environment for dynamic resource allocation and data distribution, *Proceedings of 9th International Workshop on Languages and Compilers for Parallel Computing, LNCS*, 1996.

[15] C. White and M. Head-Gordon, Derivation and efficient implementation of the fast multipole method, *J. Chem. Phys.*, **101**, 1994, 6593-6605.