AUTOMATIC PARALLELIZATION OF GROMOS96™ MOLECULAR DYNAMICS SIMULATION PACKAGE FOR DISTRIBUTED MEMORY ENVIRONMENTS

Martin Gerber[1] and Thomas Gössi[2]

[1]Computer Engineering and Networks Lab, gerber@tik.ee.ethz.ch

[2]Electronics Lab, goessi@ife.ee.ethz.ch

Swiss Federal Institute of Technology (ETH), CH-8092 Zürich, Switzerland

Keywords: computational chemistry, parallel algorithms, program generator, load balancing, algorithm specification

ABSTRACT

In this paper, we present a control/data flow specification model for replicated data parallel algorithms. Combined with an architecture graph and a mapping function, our domain-specific, semi-automatic approach takes on the job of generating source code of static data initialization and the schedule according to the specification graph, including all interprocessor communications. The remaining manual work is composed of two tasks: firstly the parametrization of the functions (parallelization), secondly the specification of the algorithm. MPI (Message Passing Interface) is used for interprocessor communication, appropriate function calls are inserted automatically.

For the Molecular Dynamics application GROMOS™ (GROningen MOlecular Simulation Software) we discuss load balancing issues as well as scalability and communication overhead problems. With the new environment it is still possible to generate sequential code with an overhead lower than 10% compared with the original hand-coded sequential program. Several platforms have been tested with the parallel code, e.g. a comparison of a workstation cluster interconnected with FastEthernet with an IBM RS/6000 SP distributed memory multiprocessor is presented.

1 INTRODUCTION

GROMOS™ is a general-purpose Molecular Dynamics computer simulation package for the study of biomolecular systems. Its purpose is threefold: (1) Simulation of arbitrary molecules in solution or crystalline state by the method of molecular dynamics (MD), stochastic dynamics (SD) or the path-integral method; (2) energy minimisation of arbitrary molecules; (3) analysis of conformations obtained by experiment or by computer simulation.

We consider molecular systems with an overall complexity of about 50,000 to 100,000 atoms. Such systems typically consist of one or more solute molecules surrounded by a liquid. In classical MD simulation, every time step Newton‘s equations of motion are solved by numerical integration of the intra- and intermolecular atomic forces. Despite the use of a pairlist (the pairlist contains all particle pairs interacting with each other, not including pairs with distances exceeding a certain cutoff radius) combined with reaction field correction techniques (van Gunsteren 1996) one time step takes at least 30 seconds on a today's high performance workstation (Gerber 1998). Then, a ten nanosecond simulation would take about two days, whereas for a breakthrough in MD research, simulations over microseconds are required.

1.1 Parallel Molecular Dynamics So Far

General acceleration techniques have been developed in recent time: efficient algorithms, parallel simulations on general purpose computers and dedicated hardware accelerators. The gain of more efficient algorithms is obvious and indisputable, and the choice of an algorithm is a compromise between accuracy and calculation speed. One example of an algorithmic improvement is the usage of an efficient neighbour searching method (van Gunsteren and Berendsen 1983) to build the pairlist. Most recently published parallel software implementations are either aimed at multi-million particle systems (Lomdahl and Beazley 1995) or massively parallel machines. Dedicated parallel hardware architectures are often developed for astrophysical systems with millions of particles (Makino 1997). Corresponding software for MPPs (massively parallel processor) is optimized for thousands of processors, parallel I/O and irregular particle distribution (Taylor, Stevens and Arnold 1997; Banicescu and Lu 1998). Highly specialized hardware for molecular systems with ASIC's for the force calculation do not support special MD functions like energy summation or the virial calculation, never mind Newton‘s Third Law (Fukushige and Taiji 1996). In addition, parallel MD systems often are targeted on reaching highest performance and do not provide full production code functionality (Müller-Plathe, Scott and van Gunsteren 1994).

The main goal of our research is to speed-up classical MD simulations of liquids (Allen and Tildesley 1987) by a factor of ten with GROMOS™ as simulation software. To find the optimal hardware architecture for our needs, taking into account the software issues, extensive design space exploration has been done. We could show that a general purpose distributed memory multiprocessor or a workstation cluster with a high performance network can fulfil our requirements, whereas dedicated hardware or special purpose processors are not well suited (Gerber and Gössi 1998).

Therefore, we concentrated on the parallelization of the GROMOS™ MD algorithm. Already existing third party parallel versions of GROMOS™ suffer from a reduced functionality, e.g. parallel simulation of a truncated octahedron is not possible. Examples are EULERGROMOS (Clark *et al.* 1994) and UHGROMOS (Kufrin and Subramaniam 1994). The standard GROMOS96 distribution is delivered with parallel code, such as vectorized or parallel non-bonded forces calculation functions. Due to the fact that firstly the maintenance cost to support production code for so many different versions is very high, secondly performance and scalability are not fulfilling our demands, the development of GROMOS96P was inevitable.

1.2 The Gromos96P Approach

A number of great demands were made to this new release:

- to improve: scalability, maintainability, extendability, parallelism, ease of use for parallel implementations
- to retain: neglectable overhead for single-processor implementation, platform independency, compatibility, portability; the accuracy of MD simulations, the „GROMOS96 look and feel“, the acceptance within the community

Two alternatives seem evident: (1) The expansion of the already existing parallel MD functions library with a bunch of new parallel MD functions and versions. This would be the ‘status quo‘ made worse. (2) writing one parallel program containing all functions. This ‘big‘ program specializes itself at run-time by choosing functions on the basis of parameters. The second solution would result in unreadable code, slow executables and would require a high knowledge of MD algorithms, GROMOS™, parallel programming, MPI, etc. from all users.

Our approach suggests an intermediate way:

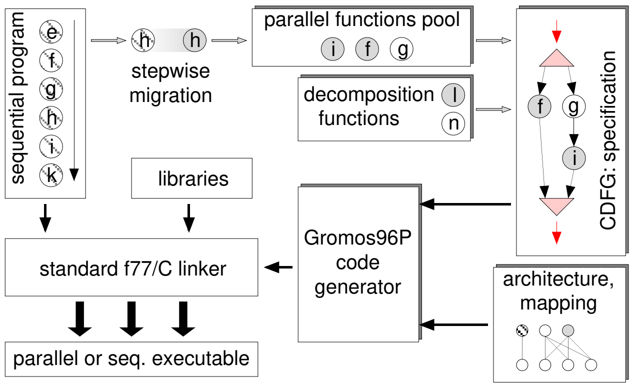

Figure 1. Programming overview

Coming from a sequential program, functions suspected of internal parallelism will be parametrized by following strict guidelines (stepwise migration). Parametrized functions, either parallel or sequential, comply the rules of the chosen decomposition and appear in the *parallel functions pool*. A „domain expert“ adds special *decomposition functions* and describes the algorithm with a control/data flow graph (CDFG), the architecture and mapping appropriate to the target. CDFG and mapping is specified with a domain specific language, for which a code generator has been implemented. A standard fortran or C linker is used to produce either sequential or parallel executable programs for different platforms.

As a prerequisite, MPI (Pachero 1997) must be supported on the selected platforms. As a restriction to employ our method, the algorithm must be parallelizable with the replicated data method. In return, our approach offers a number of additional assets:

- specification of the language with EBNF (Extended Backus-Naur Form) results in well documented and standardized interfaces. Additional semantic checks reduce the susceptibility of errors
- automatically produced source code is deadlock free, has no unnecessary code resulting in fast executables
- design decisions are made based on the CAD supported specification graph, e.g. design space exploration (Gerber and Gössi 1998), testing new algorithms/algorithm changes, generating different implementations according to the target architecture, replacement of functions
- parallel versions and sequential versions may be derived from the same source code: maintenance cost is reduced drastically, with an impact on the sequential performance
- the existing program is parallelized stepwise, by starting with the most time consuming parts; most of the existing code is left untouched

The paper is organized as follows: In section 2, we describe the specification model, section 3 explains how our code generator works, section 4 outlines our parallelization strategy of the MD algorithm and discusses load balancing issues, section 5 summarizes performance results, section 6 concludes the paper.

2 SPECIFICATION MODEL

Considering only the white parts of fig. 2, the execution scheme of a typical sequential simulation algorithm is shown, corresponding to the left side of fig. 1. For such an existing program the most time consuming parts of the simulation are parametrized and specified with our new model. The code generated from this specification appear in fig. 2 as grey parts. Most of the manual work is the parametrization (migration to parallel functions) and to provide *decomposition functions*. This task is not part of the specification model and requires the knowledge of the existing program and our specification tool.

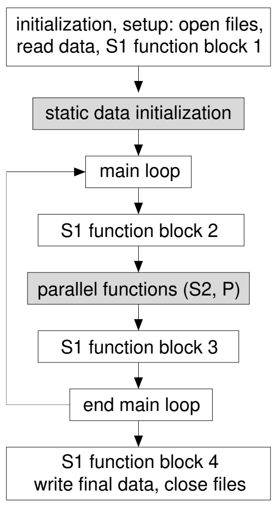

Figure 2. Typical simulation execution sequence

The software development environment consists of three parts: *specification*, *verification* and *code generation*. The unity CDFG (fig. 3) and hardware mapping (fig. 4) is called *specification*. The *verification* step ensures the correctness of the specification and the availability of the resources, e.g. a communication library. The *code generator* finally generates C code automatically depending on the specification and the mapping on a specific target.

2.1 Specification

From within the main loop, the parallel functions block is called (fig. 2). This block is specified with the CDFG as in fig. 3. The CDFG may be understood as a *schedule* frame. *Schedule* is not executable code but the starting point for the code generator. Obviously, only one CDFG with one start and end node may exist in a system. The CDFG is used to describe generic parallelism with different node types representing parallel or sequential functions. Edges between nodes describing data dependencies and control flow are specified graphically in the CDFG.

The model distinguishes four kinds of functions: (1) Type 1 sequential functions *S1* which must be executed on the host; (2) type 2 sequential functions *S2*; (3) parametrized or parallel functions *P* (grey) and (4) *special functions* which are generated automatically. *S1* functions (striped in fig. 1 and fig. 4) typically are the original functions before parallelizing anything and have restrictions in no way. *S2* and *P* functions must meet the function interface standard an other constraints and requirements of our model:

- function prototype ('i' and 'o' are reserved variables):
  `void function(struct functioninstr *i,`
  `struct functionoutstr *o);`
- function interface declaration: the structures *function*instr and *function*outstr are declared in the file *function*.h and contain all input and output data of *function*. Access to external data from within *function* not using these structs (e.g. using other include files) is forbidden
- include files: C standard libraries are allowed. Additionally, only *function*.h and c.h are included. Other libraries (e.g. MPI libraries) are not allowed
- c.h provide the possibility to select conditional error checks, define general constants, and determines if a sequential or parallel version is to be compiled
- *function*.h must provide a comment text where input struct members are declared either static (unchanged during program execution) or dynamic (communicated each time the function is called)
- sub-functions (called from within *function*) must be declared static and implemented in *function*.c or handled as an own *S2* or *P* function
- forbidden are: pointer to functions, anything else as single data types in the interface declaration (arrays are allowed, a pointer is always treated as an array), structs within structs, etc.

Only *S2* and *P* function may appear in the CDFG. *P* functions often have additional input parameters denoting the sub-domain of the data or space decomposition. These parameters are provided by the *decomposition functions* (fig. 1) which may be of type *S2* or *P*. It is the programmer's responsibilty to ensure that the parallelization strategy of functions from the *parallel functions pool* coincide well with the *decomposition functions*. The algorithm decomposition should be chosen such that parallel functions have neglectable overhead when used in a single-processor application.

As an example, a small part of the GROMOS96P specification is shown in fig. 3, namely the pairlist algorithm:

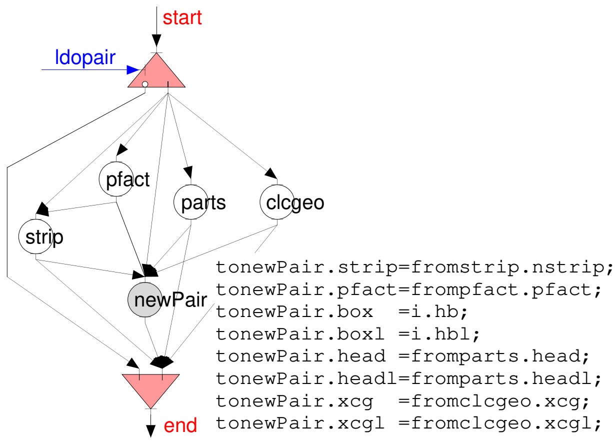

Figure 3. CDFG of the pairlist specification

Start and end points of the CDFG represent the interface of *schedule* and obey the same interface standard as *S2* and *P* functions. Dynamic data access is via the 'i.'-and 'o.'-struct as for *S2* or *P* function bodies.

The triangles are control flow elements representing an if/endif statement: if *ldopair* is true, the right part of the graph is executed, if not, nothing is executed. Other control structures like for loops, while statements, etc. are not supported. Therefore, the main loop of the program must reside outside the specification (fig. 2) and the CDFG is a directed graph without loops. Control flow variables like *ldopair* may not be output variables of predecessor functions but must be *schedule* input ('i.') variables. Although it might be straightforward, hierarchy is not supported.

An edge between two nodes indicates data dependency and is a directed arrow which may carry one token without data type. The token indicates that data associated to this edge is valid. A function is ready to be executed if data on all input edges is valid. Input data for a particular node may come from all connected predecessor nodes, including *schedule* 'i.' variables. Individual interface struct members are assigned to input or output data items of other functions through the use of a textual description in so-called map tables. One table provides the unique input data assignment of one function. In fig. 3, input data of the function *newPair* is derived from output structs of predecessor nodes and *schedule* input ('i.').

2.2 Architecture and Mapping

An architecture is a set of processing elements PE which are interconnected with each other. PE's must provide MPI, the underlying network topology is arbitrary. We differentiate two types of PE's, the host machine and coprocessors. Our specification environment treats all coprocessors identical, although heterogeneous systems are possible. The architecture graph reduces to the number of nodes, the connection between the processors are not shown. Fig. 4 outlines three mapping examples with four processing nodes each, the Host (H), and three coprocessors (C1..3). The mapping determines which functions are calculated on host and which are distributed on the coprocessors. Our current environment support only a restricted mapping. The restriction forces all *P* functions to be mapped equally. The same is true for *S2*, but independent of the *P* mapping. *S1* functions are always on host. Therefore, each function type appear in the hardware map only once.

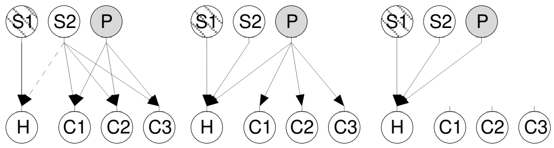

Figure 4. Architecture and hardware mapping

P functions are either distributed on all nodes (H+C1..3, fig. 4, middle), on coprocessors only (fig. 4, left), or on host only (fig. 4, right). *S2* functions are either on host only or have the same mapping like *P* functions, this is indicated in the left mapping with the dashed line. These restrictions allow an efficient but easy load balancing and simplify code generation considerably. For example, the same schedule function is used on all coprocessors.

3 CODE GENERATION

With a complete specification as described in chapter 2 we are able to generate all remaining source code for a sequential or parallel implementation. Before code is generated, the specification is verified and include files for *special functions* are prepared. S*pecial functions* are static data initialization before the main loop (function initHostStatic running on host, and initPStatic for all coprocessors) and the *schedule* for the dynamic part (functions scheduleHost and scheduleP). *Special functions* running on host get their input data from existing code (white parts in fig. 2), following the ‘i.‘ and ‘o.‘-struct rules, whereas *special functions* on the coprocessors have an empty parameter list. Separate functions for host and coprocessors are generated to reduce the number of ‘if (my_node==..)‘-like statements in generated code.

Automatic generated code has the great advantage that it is free of any alternatives except those caused by the algorithm (CDFG), containing only code for the selected hardware mapping and target architecture.

3.1 Verification

An EBNF (Extended Backus-Naur Form) has been defined for all files constituting a specification, plus the *S2* and *P* functions and the respective include files (*function*.h and c.h). The information of the specification graph and the data mapping tables are merged into a single file. The first step of the verification process is to check the complete specification and the allocated functions against the reduced syntax defined in the EBNF‘s.

Additional semantic checks are performed: Interface checks ensure that all input data items of a function are either declared static or dynamic, arrays have a length assigned, etc. The CDFG specification is checked against loops, dangling control structures and consistency with the allocated functions. No tests are implemented to prevent compile errors.

Include file generation for *special functions* is called data preparation. These files contain the interface struct definitions of all allocated *S2* and *P* functions. After the verification step, completeness and correctness of a specification is guaranteed.

3.2 Static data Initialization

The CDFG specification is not used to generate static data initialization routines. To create initHostStatic, the interface descriptions of all involved functions are examined to find static data. Then, the ‘i.‘-struct of initHostStatic is created containing all static data. An empty ‘o.‘-struct and mapping tables for all functions as in the specification (see fig. 3) are created. The mapping tables are filled with the appropriate static data, dynamic input and output data fields are initialized to zero. In addition, code for static data distribution to coprocessors is inserted. Creation of InitPStatic is simpler, code for receiving the data and mapping table assignments is generated similar to initHostStatic.

3.3 Scheduling

Code for the function scheduleHost is assembled as follows: The code generator first inserts code for dynamic control flow data distribution (*ldopair*, refer to fig. 3) to the coprocessors, then, the parser takes the first function of the CDFG (e.g. *clcgeo*) and checks the token on the input edge. This token is set, and code for calling *clcgeo* is generated: First it is determined if some data have to be communicated. Nothing must be communicated assuming that *S2* functions are mapped on host only. Dynamic input data is locally available, assigned to *clcgeo*‘s input, and the function call is inserted. Now, the output edges of *clcgeo* are marked with a token and the code generator proceeds to the next function, e.g. *strip*. *Strip* may not be planned because not all tokens on the input edges are present. The code generator proceeds with *pfact* and so on. If code for all *S2* functions is generated, all input edges of *newPair* have a token. Assuming that *P* functions are mapped on all nodes, all data associated to input edges of *newPair* must be communicated to the coprocessors. Appropriate MPI calls are inserted before the *newPair* mapping table and the *newPair* function call.

Distribution of data is simply done with broadcast MPI calls or with point to point communication. The gathering of results is more sophisticated. A parallel function provides the same output data on all processors, but from where the code generator should know how to assemble the data? Due to the replicated data paradigm, output data may be reduced using a binary operation like addition, multiplication, logical or, etc. Such reduction operations may be applied to *P* functions only which is part of the interface declaration. If the code generator is missing a reduction operation, it stops with an error. MPI provides a set of such reduction functions (Pachero 1997). On the basis of the hardware mapping of *S2/P* functions and the CDFG the code generator determines if reduction code must be inserted.

A post processor removes unused code from initHostStatic and scheduleHost if a sequential implementation is wanted.

3.4 The Link Step

Recompilation is necessary for different parallel environments, target architectures, *S2/P* mappings, or if a sequential version is desired. A standard C or fortran linker is used (fig. 1).

A lot of potential errors may not be caught by our environment: The data interface between a specification (CDFG) and the existing code is 100% hand-coded and the main source of error. No tools are implemented yet to check e.g. the parameter list when our generated code is called from an existing fortran program. In addition, our specification model does not allow us to perform formal verification on the correctness of mapping tables.

4 PARALLEL MD IN GROMOS96P

Three decomposition methods typically are used with MD simulations. Particle decomposition assigns unique particles to processors; the particles remain with the assigned processor throughout the simulation. Spatial decomposition (Nyland *et al.* 1997) assigns partitions of the geometric space to processors; the particles may migrate to different processors determined by the spatial coordinates. Interaction decomposition (Taylor, Stevens and Arnold 1997) assigns unique particle pairs to the different processors; the pairlist represent the interactions to be calculated during the simulation. These methods are usually implemented with one of the following techniques: Replicated data, systolic loops and/or domain decomposition. In the replicated data method (Müller-Plathe, Scott and van Gunsteren 1994) every processor has a copy of all atomic coordinates, velocities and forces and, in principle, attempts a full simulation of the system of interest, while sharing the workload equally with the other available processors. Systolic loop methods distribute the atomic data equally over all processors and by passing data between them permit the calculation of all possible pair forces. The domain decomposition method is a spatial decomposition without replicating the data. The boundaries between the domains need to be replicated or communicated.

The choice of an algorithm depends on several factors: The replicated data method is good for a few hundred processors and up to about 50,000 atoms; domain decomposition algorithms are very effective for multi-million particles simulation allowing dynamic load balancing for heterogeneous particle density. Other aspects might be: the availability of resources (computing power, money, manpower), the target system, existing reusable code, the desired performance gain, or I/O limitations.

4.1 GROMOS96P Spatial Domain Decomposition

On the grounds that we did not want to rewrite the whole GROMOS™ package, but only the most computation-intensive functions (pairlist generation and the non-bonded forces calculation), the integration of the equations of motions and the bonded forces calculation were kept as *S1* functions. This constraint led us directly to the spatial decomposition method with replicated data.

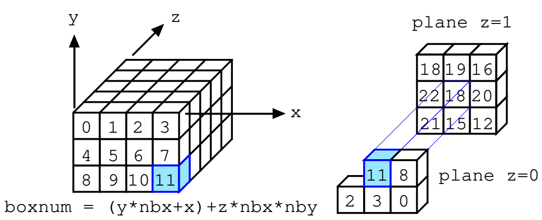

Figure 5. Rectangular periodic boundary conditions

GROMOS™ provides periodic boundary conditions (PBC) to avoid unwanted border distortion effects in simulations with liquids. PBC can be applied to a rectangular or monoclinic (with oblique coordinates) box or a truncated octahedron, see fig. 5. The simulation box is partitioned into small boxes with lengths larger than the cutoff radius. The pairlist algorithm based on a linked-list searches only the neighbouring boxes to find particle pairs. To achieve better performance, the boxes are further divided into sub-boxes (*parts*). Our algorithm reaches the best performance if the total number of parts approximately matches the number of particles *ncg*.

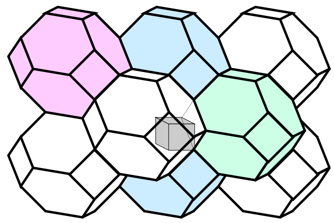

Figure 6. Truncated octahedron

The algorithm to find neighbouring parts is very fast for a rectangular box. Applying Newton‘s Third Law for neighbour searching (we only search neighbours in half of the cutoff sphere in order to count a pair only once), neighbours of particles inside part 11 are searched in the parts as indicated in fig. 5. Assuming *nbz* the number of parts in z-direction, the pairlist construction for the truncated octahedron (fig. 6) is less efficient due to a O(nbz^2) neighbouring parts search algorithm.

4.2 Load Balancing

To distribute the workload among *p* processors, the rectangular simulation box is divided up into *p* slices. The numbering scheme determines the direction of the parallelization, e.g. in fig. 5 the simulation box is partitioned into z-planes. Assuming *nbz* being an integer multiple of *p* load balancing will be ideal for homogeneous liquids. If $p^3 > ncg$, the total number of parts *tnbx* will be much greater than *ncg*, and the pairlist algorithm slows down, at least without impacts on the good load balancing.

The truncated octahedron in GROMOS™ is always the most regular one, which means that the smallest rectangular box where the octahedron fits in is a cube. Due to the PBC the neighbours of a particle may be „everywhere“, e.g. neighbours of the central box in fig. 6 might be in all shadowed replicated octahedrons. Especially with large cutoffs the advantage of the linked-list pairlist algorithm is reduced. Using the same load balancing strategy as for the rectangular box, namely dividing the enclosing cube into slices with same widths, would result in a heavily unbalanced workload: a slice in the middle of the simulation box would have a larger volume than one on the border. Therefore, we increase the number of parts such that a border slice consists of more parts than a central one. Then, a well-balanced workload is guaranteed if $p << nbz$. This results in the current implementation which is a compromise between an acceptable load balancing and a fast pairlist algorithm.

Usually, the pairlist is updated every five or ten time steps (compare with fig. 3) in conjunction with particle redistribution and load balancing. The parallel pairlist function *newPair* calculates a pairlist per PE containing only pairs that belong to the associated sub-space. Parallel non-bonded force calculation functions use the local pairlist and work on replicated data structures. The calculated forces are also stored in a replicated data structure which is added up with MPI reduce functions.

5 PERFORMANCE

We tested automatically generated code on several hardware environments (Gerber 1998). As a summary, we present benchmark results of the following hardware environments:

- 'sol': 1-4 single processor SunUltra30 (300MHz, 256MB RAM) workstation cluster, standard Ethernet (office LAN)
- 'a': 1-4 single processor 21164 Alpha (500 MHz, 128MB RAM, 2MB Cache) workstation cluster, dedicated FastEthernet
- 'sp': IBM RS/6000 SP (1-64 160MHz thin nodes with POWER2 Super Chip Uniprocessors, 256MB/4.5GB per node)

machine & benchmark	s	p1	p2	p3	p4	p16	p20
sol-thr2	1.02	1.00	1.79	2.19	2.53	-	-
sol-thrG2	1.11	1.00	1.38	2.15	2.64	-	-
a-thr2	-	1.00	1.71	2.55	3.42	-	-
a-thrG2	-	1.00	1.89	2.75	3.43	-	-
sp-thr2	1.01	1.00	1.93	2.32	3.44	7.79	-
sp-thrG2	0.99	1.00	1.93	2.83	3.68	11.36	13.6

Table 1. GROMOS96P speed-up comparison

The table applies to the benchmark as follows: (thrG2) one Thrombin molecule (3078 atoms) surrounded with 10961 water molecules in a rectangular box; (thr2) the same solute molecule surrounded with 5427 water molecules in the truncated octahedron. The pairlist is updated every 5 steps with a 1.4nm cutoff, the virial is also calculated. Columns 'p1' (parallel version running on one processor) through 'p20' (20 processors) denotes the speed-up compared with column 'p1'. Column 's' is the speed-up of the sequential version compared with column 'p1'.

One of the main goal in parallelizing GROMOS™, not to lose more than 10% of performance compared with the original sequential version, could be achieved. The current implementation distribute data with blocking point to point communication, while data gathering is done with MPI reduce functions.

6 CONCLUSION AND OUTLOOK

With the presented specification method and code generation technique we are able to generate a parallel version of GROMOS™ Molecular Dynamics Simulation software. Scalability is rather good for less than 20 processors, and about 14 nodes are required to achieve a speed-up factor of ten. The new parallel GROMOS96P support the whole functionality of GROMOS96 and offers a lot of additional assets. With our technique, only one version of source code has to be maintained. The new software is easy to understand and is open for new developments. This is important because registered users all over the world are developing extensions for GROMOS™. Further developments include the improvement of the spatial decomposition algorithm, e.g. decoupling the pairlist construction from load balancing. The communication strategy of the code generator is preliminary and also subject to optimization. Better results are expected through the use of broadcast, asynchronous communication, and by reducing the number of synchronization points.

REFERENCES

Allen, M.P. and D.J. Tildesley. 1987. *Computer Simulation of Liquids*. Oxford University Press

Banicescu, I. and Rong Lu. 1998. „Experiences with Fractiling in N-Body Simulations“. In *High Performance Computing '98, Conference Proceedings*, 121~126

Bekker, H. and H.J.C. Berendsen. 1992. „GROMACS: A parallel computer for molecular dynamics simulation“. In *proceedings of the Physics Computing '92*

Clark, T.W.; R.v. Hanxleden; J.A. McCammon; and L.R. Scott. 1994. „Parallelizing Molecular Dynamics using Spatial Decomposition“. *Proceedings of the Scalable High Performance Computing Conference 1994*, Knoxville, TN

Fukushige, T. and M. Taiji. 1996. „A Highly-parallelized Special-purpose Computer for Many-body Simulations with an Arbitrary Central Force: MD-GRAPE“. *The Astrophysical Journal, no. 468*, 51~61

Gerber, M. 1998. „A new GROMOS™ Benchmark on SunUltra30/60, DEC Alpha21164, IBM RS/6000 SP“. *Internal Paper, http://www.tik.ee.ethz.ch/~md*

Gerber, M. and T. Gössi. 1998. „Parallel Coprocessor Architectures for Molecular Dynamics Simulation: A Case Study in Design Space Exploration“. In proceedings of *The 1998 IEEE International Symposium on Circuits and Systems: ISCAS98*

Kufrin, R. and S. Subramaniam. 1994. „UHGROMOS at the National Centre for Supercomputing Applications“

Lomdahl, P.S. and D.M. Beazley. 1995. „Multi-Million Particle Molecular Dynamics on MPPs“. *Second International Workshop, PARA95*, Lyngby, Denmark

Makino, J. 1997. „Stellar Dynamics on 200 Tflops Special-Purpose Computers: Grape-6“. *International Symposium on Supercomputing*

Müller-Plathe, F.; W. Scott; and W.F. van Gunsteren. 1994. „PARALLACS: a benchmark for parallel molecular dynamics“. *Journal of Computer Physics Communication, no. 384*, 102~114

Nyland, L; J. Prins; R.H. Yun; J. Hermans; H.C. Kum; and L. Wang. 1997. „Achieving Scalable Parallel Molecular Dynamics Using Dynamic Spatial Domain Decomposition Techniques“. *Journal of Parallel and Distributed Computing, no. 47*, 125~138

Pachero, P.S. 1997. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc. San Francisco, CA

Taylor, V.E.; R. L. Stevens; and K. E. Arnold. 1997. „Parallel Molecular Dynamics: Implications for Massively Parallel Machines“. *Journal of Parallel and Distributed Computing, no. 45*, 166~175

van Gunsteren, W.F. and H.J.C Berendson. 1983. „On searching neighbours in computer simulation of macromolecular systems“. *Journal of Computational Chemistry, vol. 5, no. 3*, 272~279

van Gunsteren, W.F. 1996. *Biomolecular Simulation: The GROMOS™96 Manual and User Guide*. Hochschulverlag vdf AG an der ETH Zürich