Shared-Memory Parallelization of the GROMOS96 Molecular Dynamics Code on a SCI-Coupled NUMA Cluster

Marcus Dormanns
Lehrstuhl für Betriebssysteme, RWTH Aachen
Kopernikusstr. 16, D-52056 Aachen, Germany
Email: contact@lfbs.rwth-aachen.de
URL:http://www.lfbs.rwth-aachen.de

Abstract. *This paper describes the parallelization of a commercial molecular dynamics simulation code, GROMOS96, on a SCI (Scalable Coherent Interface) interconnected PC cluster. The underlying programming model is that of shared data structures, exploiting SCI's capabilities of enabling access to segments of remote memory in an entirely transparent way. Methodologies are elaborated that allow to obtain high performance in presence of the NUMA (Non-Uniform Memory Access) performance characteristic of the cluster platform. It is demonstrated that this type of cluster platform allows a step-by-step parallelization process in distinction to a message-passing parallelization with it's partitioned and distributed data structures. Performance figures of the resulting parallel code are presented and discussed.*

Keywords: *NUMA, shared memory, application parallelization, cluster, molecular dynamics.*

1 Introduction

This paper describes the parallelization of GROMOS96, a molecular dynamics simulation code, on recently emerging parallel cluster architectures [17]. In distinction to dedicated parallel computing systems (like Cray T3E), cluster systems comprise out of commercial off-the-shelf compute nodes (PCs or workstations) that are coupled with some kind of communication network. This paper focuses on clusters, equipped with one of the most noticeable high-speed cluster networks in light of it's functionality, the Scalable Coherent Interface (SCI) [10]. SCI differs from most other networks, which just possess message passing capabilities, in that it implements transparent read and write access to memory of remote compute nodes.
With the aid of the respective device driver services, each process of a parallel application can allocate SCI memory segments. These can be mapped into the allocating process' virtual address space as well as into that of any other (remote) process. Subsequently, each process can transparently access this memory. Accesses to remote memory are mapped to the PCI-based SCI-adapter and serviced by a network transaction with the assistance of the corresponding remote SCI-adapter. Although being entire transparent, remote memory accesses show a latency that is about one order of magnitude higher than local one. Due to this performance characteristic, such systems are commonly classified as NUMA (Non-Uniform Memory Access) parallel systems.
The simulation of the dynamics of molecular systems is one of the central fields of compute-intensive technical/scientific computing. It has therefore been subject to parallel processing already for a long time. GROMOS96 [20] is one of the major well-known codes in this area that has already quite a long history. It is the completely re-designed successor of GROMOS87, regarding it's functionality as well as it's implementation. Due to it's irregular and time-varying data structures that are processed by different kinds of algorithms this code is well-suited to discover a lot of aspects of application parallelization on the cluster platform under consideration. This is virgin territory in that most projects so far, dealing with NUMA cluster platforms, are concerned with lower-level performance issues and programming models but do not yet deal with larger application parallelization. The herein presented effort is part of a larger application parallelization project [16] that is concerned with codes from different areas, e.g. a module from decision a support system [12] and an acoustics simulation code.
The parallelization of GROMOS96 exploits the shared memory capabilities of the cluster platform via a comfortable parallelization library, the Shared Memory Interface (short SMI [5]) which serves as the

basis for an efficient and comfortable parallelization process and the resulting code performance. SMI aims at supporting the programmer regarding the following subjects:

- Providing a richer and more comfort set of functionality than the pure SCI device driver. All common services for shared memory programming are available, e.g. dynamic shared memory allocation, synchronization mechanisms, loop-scheduling, etc.
- Hiding platform peculiarities (regarding performance and functionality) from the programmer.
- Allowing a simple migration from a given sequential code to a parallel one that can be performed in a step-by step fashion.

This is one of the first parallelization efforts that deals with a relevant commercial code and is directly based on SCI-shared memory within a cluster system. The common, more conservative but nevertheless justified approach, is to implement message-passing on top of SCI [23] and to deal with message passing application parallelization (e.g. [9]). But in light of the fact that memory-coupling becomes the more and more dominant architectural principle of parallel systems, developing techniques for the direct exploitation of this feature should be of high interest.
Section 2 provides some more information about GROMOS and it's code structure. The SMI programming library is described in section 3. Based on this, section 4 describes the parallelization of GROMOS96 which has two different aspects. The first one regards the overall principles and software engineering issues, while the second one eventually sketches the parallelization of the kernel algorithms. Succeeding, section 5 discusses the resulting performance and relates it to similar parallelization efforts. Finally, some conclusions are drawn in section 6.

2 The GROMOS Code

2.1 General

The purpose of molecular dynamics simulation is to track the dynamics of a molecular ensemble whose individual atoms interact via several types of forces over time. Depending on the concrete application, results of interest are the particles' trajectories (or time-evolving quantities that can be computed from them) as well as the final configuration.
The whole program package consists of several programs. Besides pre- and post-processing programs, the main component is the actual molecular dynamics simulation program. It consists of 31 source code modules with about 42,000 lines of Fortran 77 code altogether. The difference to the former GROMOS87 becomes obvious, relating these quantities to the 22 modules with just about 9,000 lines of Fortran 77 code of GROMOS87.
GROMOS87 was already subject to several parallelization efforts. UHGROMOS and EulerGromos have been developed at the Texas Center for Advanced Molecular Computation (Univ. of Houston) [3,13,15]. Furthermore, GROMOS87 was parallelized within the framework of the European Community sponsored Europort project [6,14]. Both efforts led to message passing programs. The new GROMOS96 code has only been parallelized once, using threads for shared-memory Silicon Graphics multiprocessors [7].

2.2 Structure of the code

The code structure is highly pre-determined by it´s underlying physics and it´s numerical solution method (see e.g. [1,20]). Denoting the spatial position of an atom i with mass m_i at a specific point in time t by $r_i(t)$, a set of coupled (nonlinear) differential equations is solved in the time domain that determines the atoms' trajectories due to Newton's equations of motion considering an interaction potential E:

(2.1) $$\frac{\partial^2}{\partial t^2} r_i(t) = -\frac{\nabla E(r_i(t))}{m_i} \qquad i = 1, 2, 3, \ldots$$

This is done in a time-step fashion. Starting from a given initial configuration, for each point in the discretized time the forces acting on each particle are computed and accumulated. From these, Newton's equations of motion allow to determine position and velocity of all particles at the following time step by integration.
Relevant forces can be divided into two types:

- Non-bonded interactions that act between any pair of atoms. These can further be divided into long-range and short-range interactions, depending on their decay rate with increasing distance.
- Bonded interactions result from chemical bonds between atoms. They capture e.g. bon-lengths, -angles and -dihedral angles.

Short-range interactions are typically neglected for all atom pairs beyond a certain cut-off radius. To allow an efficient evaluation of all relevant forces, all interacting atom pairs are kept in a so-called pair-list. Due to the dynamic's smoothness, it is sufficient to update the pair-list just every t_{pl} time steps (e.g. t_{pl}=10). Analogously, long-range interactions are evaluated only from time to time and assumed to be constant in between. In GROMOS96, long-range interaction evaluation is paired with pair-list construction.
Additional to these mechanisms, it is often required to restrain some degrees of freedom of the molecular ensemble, e.g. bond-lengths, that would undergo forbidden modifications within the simulation process. This is achieved with an iterative procedure, commonly called SHAKE. SHAKE adjusts the ensemble iteratively according to the (partially contrary) restrictions. The code structure is summarized in figure 1. For an exemplary molecular ensemble, a protein molecule thrombin (3,078 atoms) together with 5,427 solvent water molecules (altogether 19,359 atoms), calling frequencies and computation time contributions of the major modules are given.

3 The Parallelization Library - SMI

The parallelization of GROMOS96 is based on the Shared Memory Interface (short: SMI) parallelization library. This section provides a short overview of it's principles and functionalities. More details can be found in [5]. The library has been implemented on SCI-coupled clusters of workstations and PC and is available for Windows NT, Solaris and Linux.

3.1 Principles of SMI

An application that is parallelized with SMI is executed by a couple of concurrent complete processes in a SPMD style. Initially, all processes possess entirely private address spaces. Using the respective SMI functions, *regions* of shared memory can be established among them. To allow data-locality optimizations, each region can comprise out of several *segments*. Each one can be physically located on a different compute node, it's *home*. All accesses from processes on the home node are fast local accesses, all others are expensive remote accesses.

3.2 SMI functionality

Functions provided by SMI can be divided into five categories:

- Initialization of the run-time environment and information gathering about it.
  Several functions exist to request the number of participating processes and machines, ranks of processes and machines, and information about where each process is executing.
- Establishment and management of shared memory regions.
  Shared memory regions can be allocated with several policies that guide the way they are com-

				Count	Time (in %)
main					
	load configuration data			1	~0
	perform T time-steps of simulation				
		all t_{pl} time-steps: assemble pair-list and compute long-range interactions			
			solute-solute and -solvent	T/t_{pl}	13.8
			solvent-solvent	T/t_{pl}	33.3
		compute short-range interactions			
			solute-solute and -solvent	T	10.4
			solvent-solvent	T	36.1
		SHAKE		4+2T	2.6
		integrate to next time-step		T	~0
		all t_{out} time-steps: write data fo file		T/t_{out}	~0
	write final configuration data to file			1	~0

Figure 1: GROMOS96 code structure together with the modules' computational complexity and frequency for a thrombin molecule (3,078 atoms) with 5,427 solvent water molecules.

posed out of individual segments. Such a region can be used to store a single flat data structure in it (e.g. an array) or for dynamic memory allocation within it (with the aid of a memory manager).

- Synchronization primitives.
  They cover mutex and barrier synchronization as well as what is called *progress counters*. A progress counter is an integer variable for each process that can be incremented to signal other processes the current state of computation progress of a process. Several functions exist to ensure that a process waits until the counter variables of the other processes signal that the computation process has reached at least a specific state. Furthermore, memory consistency is ensured at synchronization points. An explicit handling of this issue is necessary due to load- and store-buffers on the SCI adapters.

- Loop-scheduling.
  Advanced loop-scheduling and -splitting facilities exist that can be used to parallelize a loop [19]. They are scalable and account for minimal overhead. They allow e.g. dynamic load balancing with respect to data locality maximization.

- Services that enable a step-by-step parallelization.
  A set of functions is provided that allow to temporary switch to replication of a shared memory region in each process. Later on, it is possible to switch back to sharing, providing various ways to combine the replications to a single consistent global shared region, which may have evolved differently in the meantime.

4 Parallelization

The parallelization of GROMOS96 has been performed in accordance to SMI´s philosophy of enabling a step-by-step parallelization. This methodology has several advantages:

- Reduced complexity. Especially for the parallelization of given sequential codes (in distinction to the development of a parallel application from scratch) it is critical to deal with all data structures and sub-algorithms at once.
- Scalability of the parallelization process. I.e. the parallel performance of the code should scale with the amount of work spent on the parallelization process. A sophisticated analysis of where the most time is spend in the sequential code may allow to achieve huge performance improvements while just parallelizing small code sections. This is especially desirable in projects with limited (financial) resources.

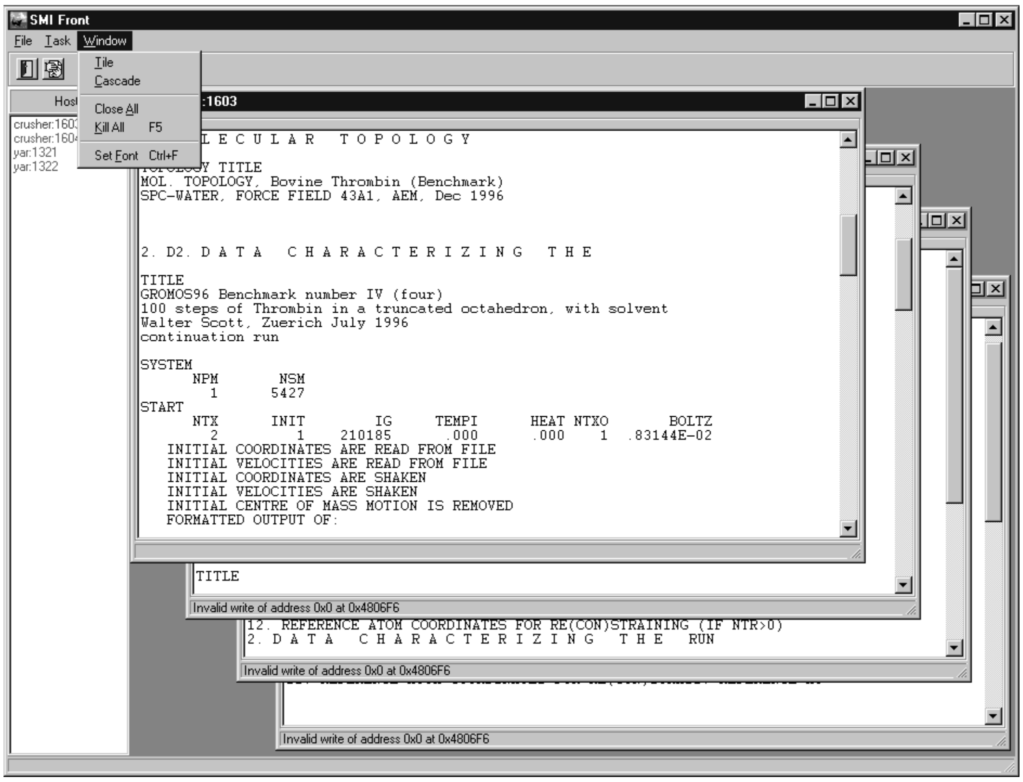
Shared memory is an essential requirement for both issues. Being forced to a message-passing parallelization that comes along with partitioned and distributed data structures, it is impossible to proceed in such a step-by-step process. The possibility to start with the parallelization of just a few code parts is an enormous advantage especially for a code like GROMOS96, which consists of a couple of different algorithms. Although contributing not considerably to the computational complexity, some of them are quite complex to parallelize which can be omitted in the first steps. Therefore, SCI-based NUMA shared-memory cluster show much more advantages than just performance considerations would suggest.
The individual steps undertaken in the GROMOS96 parallelization effort are described in the following subsections.

4.1 Starting with parallelism and coordinating I/O

The very first step is just to ensure a start-up mechanism for the application that establishes a desired couple of parallel processes and coordinates their I/O. Although sounding quite trivial, it turned out to be of considerable importance and afflicted with more problems than one might imagine on a Windows NT operated cluster (in distinction to e.g. Unix).
The parallel SMI-based version of GROMOS96 is started with the aid of a shell script. It contains some variables that have to be set from the user:

- name and path of the executable
- the names of the input files (e.g. configuration data, simulation parameters, etc.)
- machines to run the application on (which defines the degree of parallelism at the same time)

This information is sufficient to start the parallel application with an appropriate mechanism. Doing so, a window-based front-end is invoked on the machine, the user issued the job from. It provides an individual console for each process' console output (standard output and error) and gives control over the application. There exists functionality e.g. to terminate all processes of the application. Such a front-end is essential for a cluster that is operated under Windows NT. This operating system lacks support for I/O redirection to/from other machines and multi-user support which disables to run a process on a remote machine (on which the user is not logged on) within the entire user´s environment. Figure 2 shows a snapshot of this font-end.
The parallel execution environment within each process is set-up by initializing SMI and requesting several parameters from it, e.g. the total number of processes, each process´ rank etc. What remains is coordination of output. GROMOS96 uses several streams of output. Some go to certain files (e.g. trajectory data of the atoms over time), one goes to the console (simulation parameters, error messages, ensemble-averaged quantities for individual time-steps as well as averaged over the entire simulation run). For the purpose of parallelization, the standard error stream is used (e.g. for error messages and performance evaluation). Using SMI's capabilities, standard error is redirected to the front-end. Standard output is redirected to files, ensuring a different file-name for each process automatically. File-out-



```
 L E C U L A R   T O P O L O G Y

    Y TITLE
MOL. TOPOLOGY, Bovine Thrombin (Benchmark)
SPC-WATER, FORCE FIELD 43A1, AEM, Dec 1996



2. D2. D A T A   C H A R A C T E R I Z I N G   T H E

TITLE
GROMOS96 Benchmark number IV (four)
100 steps of Thrombin in a truncated octahedron, with solvent
Walter Scott, Zuerich July 1996
continuation run

SYSTEM
      NPM       NSM
        1      5427
START
      NTX      INIT        IG     TEMPI      HEAT NTXO       BOLTZ
        2         1    210185      .000      .000    1  .83144E-02
   INITIAL COORDINATES ARE READ FROM FILE
   INITIAL VELOCITIES ARE READ FROM FILE
   INITIAL COORDINATES ARE SHAKEN
   INITIAL VELOCITIES ARE SHAKEN
   INITIAL CENTRE OF MASS MOTION IS REMOVED
   FORMATTED OUTPUT OF:
```

```
TITLE
```

```
Invalid write of address 0x0 at 0x4806F6
```

```
12. REFERENCE ATOM COORDINATES FOR RE(CON)STRAINING (IF NTR>0)
2. D A T A   C H A R A C T E R I Z I N G   T H E   RUN
```

```
Invalid write of address 0x0 at 0x4806F6
```

```
Invalid write of address 0x0 at 0x4806F6
```

Figure 2: Screen-shot of SMI´s window-based front-end. The output of a GROMOS96 run with 4 processes is redirected to the front-end on the user´s machine.

put is performed by a single process. This is ensured using the known rank of each process to skip the corresponding code fragments for each but the process with rank zero. This can also remain in the later parallelized version due to the fact that within the shared data programming model, each process has access to all data. Clearly, this approach in not scalable in terms of Amdahl’s law. But the major focus of this parallelization effort is to study the later algorithm parallelization.
The result is a code that states the basis for the following parallelization of some of the most time consuming modules, one after another.

4.2 Parallelization of the interaction calculation kernel

The two most time consuming modules are:

- pair-list construction (including long-range interaction evaluation) and
- short-range interaction evaluation.

The pair-list is constructed not on the basis of individual atoms but on the basis of small clusters of atoms that together possess a (near-)neutral charge, the so-called charge-groups. A charge-group is for example an entire solvent water molecule. Besides physical advantages, this also reduces the problem-size for pair-list construction by a factor of about three.
The pair-list construction that is coupled with long-range force evaluation (if this feature has been enabled) is a simple $O(N^2)$ algorithm that tests the distance of all pairs of charge-groups. If it is only small enough, a long-range force contribution is evaluated and accumulated in a long-range force array for the considered atoms. If it is even smaller than the short-range interaction cut-off radius, the respective charge-group pair is added to the pair list.
The short-range interaction evaluation module consists of a loop over all entries in the pair-list. For

each charge-group pair, all interactions between all constituting atoms are evaluated and accumulated in a corresponding short-range force array.
Three properties of GROMOS96 are worth noting because they influence the parallelization considerably:

- Due to Newton´s law of actio and reactio, forces between atoms are anti-symmetric, i.e. identical in magnitude but contrary in direction. Once a force has been evaluated, it is crucial to accumulate it to both atoms under consideration to gain performance. Therefore, it is impossible to parallelize the application in a way that no two processes perform accumulations to the same entries in the force arrays.
- The GROMOS96 implementation extensively relies on a very specific mapping of atoms to indices in the force arrays. One major assumption is that all solute atoms precede the solvent atoms. Furthermore, all atoms of each individual solute molecule (if there are several) are consecutive in the force array, again in a specific common order. Departing from this would result in major implementation changes although it might be very reasonable considering parallelization solely.
- The code to compute a single solute-solute or solute-solvent differs considerable from pure solvent-solvent interactions (this applies to long- as well as short-range interactions). Due to this, the computation of the two sets of interactions is separated in the code. To retain it’s structure, a parallelized code should do this as well.

4.2.1 Pair-list construction and long-range force evaluation

The major data structures, affected by a parallelization of this module are:

- the array of all atoms’ forces that is subject to accumulation operations and
- the pair-list.

The basic principle is to split the outer-most loop of the N^2-algorithm among the processes. This means that contributions to an atom’s long-range force may come from all processes. To allow this, the long-range force array is placed in a globally accessible SCI shared memory region. The region is allocated with the `BLOCKED` directive, i.e. it is assembled from equally-sized segments on each machine to balance the number of remote memory accesses among the processes. Then, an essential requirement for correctness is to perform accumulations from different processes to identical atoms under mutual exclusion.
The straight-forward way to ensure this would be to guard each accumulation operation by a mutex. Although SCI allows to implement efficient synchronization mechanism, their cost (on the order of 50 µs) is much to high to allow the guarding of each single accumulation operation. The implemented solution for this problem is to block the execution of the (spited) loop (see e.g. [11]). The resulting temporal locality is advantageous and useful in several ways:

- The cache usage is improved.
- For each atom that is processed within this algorithm, quite an amount of data that influences the computation has to be looked-up in several different data structures. For a small and deterministic number of atoms it is possible to do this just once during the processing of the entire loop-block and to keep it in a suitable data structure during that phase. This saves look-up overhead.
- If already some kind of software-caching is done it can be expanded to also capture the atoms’ (partial) forces. An accumulation to the global force array is then performed for all atoms at the end of the processing of an entire loop-block, capturing all partial forces at once that result from all inter-block interactions. This compensates the necessary synchronization operation and also the overhead due to remote memory access.

```
SMI_Switch_to_sharing(long_range_force_array)
SMI_Loop_schedule_init(global_min = 1, global_max = N)
While work Do                                                  /* outer loop */
  SMI_Get_index_range(oblock_idx_min, oblock_idx_max)
  Load data of atoms oblock_idx_min...oblock_idx_max
  For iblock = 1 To Nblocks Do                                 /* inner loop */
    iblock_idx_min = iblock*BlockSZ;
    iblock_idx_max = min(N, (iblock+1)*BlockSZ - 1)
    Load data of atoms iblock_idx_min...iblock_idx_max
    For i = oblock_idx_min To oblock_idx_max Do
      For j = oblock_idx_min To oblock_idx_max Do
        Process atom pair (i,j)
          - compute distance
          - compute long-range force if distance small enough
          - insert into pair-list if distance small enough
      End For
    End For
    SMI_Mutex_lock()
    Accumulate partial forces of atoms iblock_idx_min...iblock_idx_max
    SMI_Mutex_unlock()
  End For
  SMI_Mutex_lock()
  Accumulate partial forces of atoms oblock_idx_min...oblock_idx_max
  SMI_Mutex_unlock()
End While
SMI_Switch_to_replication(long_range_force_array)
```

Figure 3: Outline of the parallelized pair-list construction and long-range force evaluation algorithm.

The scheduling of loop-blocks to processes is performed by SMI's advanced loop-scheduling facilities [19] that minimize load imbalance while introducing only a minimum of scheduling overhead. The resulting code fragment is sketched in figure 3.
Besides long-range forces, the results are per-process pair-lists corresponding to the considered atom-pairs within each process. This implicit partitioning of the overall list of atom pairs is already the basis for the parallelization of the short-range interactions. So far, load balance has been optimized for pair-list computation itself, but not for the resulting pair-lists. This will eventually result in load imbalance during the short-range force computation phase. This problem will be addressed below, going along with that what is understood under the term "scalable parallelization" in the context of this paper.
To hide the parallelization of this module from the rest of the application, SMI's capabilities to switch between sharing and replication of memory regions are used (see figure 3). Although not mentioned in particular, this parallelization scheme is applied to solute-solute or solute-solvent as well as pure solvent-solvent interactions in the same way.

4.2.2 Short-range force evaluation

The parallelization, i.e. the work partitioning, of the short-range force evaluation module is already predetermined by the existence of a partial pair-list within each process. Analogously to the pair-list construction, the central data structure that is processed in parallel is the short-range force array for all the

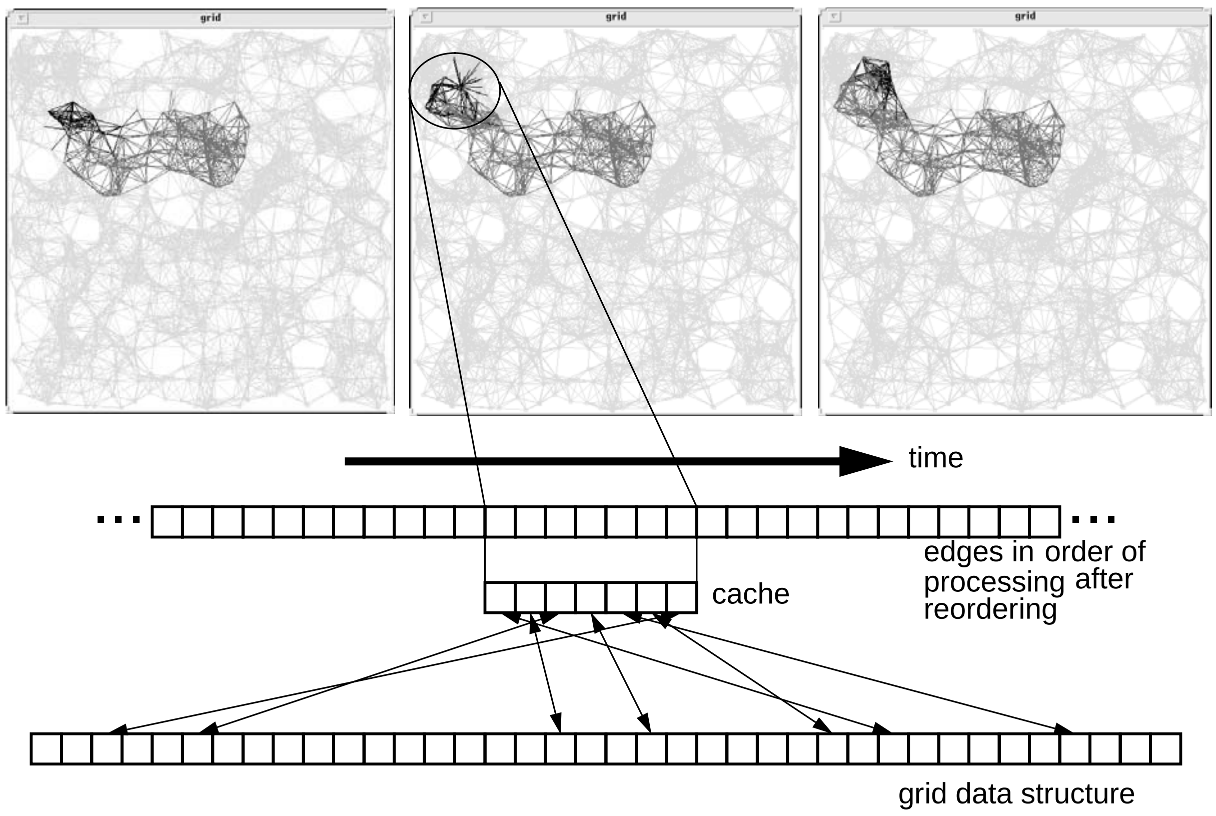


Figure 4: Three consecutive snapshots of temporal locality optimized grid traversing. Black edges denote the latest visited edges (a cache-size corresponding to 64 nodes was assumed).

atoms. Also, the parallelization of this module is faced with similar problems:

- Minimize the number of force-array accesses because a fraction of them are expensive accesses of remote memory.
- Minimize the number of lock/unlock operations that are necessary to guarantee atomicity of accumulation operations from different processes.

Blocking, as done to parallelize pair-list construction is not straight-forward, because the atom-pairs that are to be processed are determined by the pair-list that states an irregular grid structures (assuming each charge-group as a node and drawing an edge between all with an Euclidean distance smaller than the cut-off radius). The original sequential mode of processing was a node-oriented procedure that processed interactions in order of the first node. Denoting the number of charge-groups by G and the pair-list length by P, this requires G+2P atom data load phases with a proportional amount of cache misses in the sequential case and additionally a proportional amount of remote memory accesses in the parallelized case. P is on the order of one magnitude higher than N, i.e. each node possesses some tenths of interaction neighbors (60 is a realistic quantity). Furthermore, there is no obvious way of reducing the number of lock/unlock operations.
For this purpose, blocking is done at the grid level by re-ordering the pair-list in a pre-processing step. Processing the pair-list after that step corresponds to traversing the grid in a way such that edges connecting nodes that already participated in an interaction calculation a short time before are considered as next edges in the schedule. This ensures temporal locality (see figure 4). Furthermore, the ratio

no. charge-groups	13,824
pair-list length	442,584
orig. no. of charge-group data purge/load operations	13,824 + 442,484 = 456,308
software cache-size	256
no. of locks	16
optimized no. of purge/load operations of charge-group data	49,154
unoptimized no. lock/unlock operations	49,154
optimized no. lock/unlock operations	2,870

Table 1: Characteristic quantities for a 13,824 water molecule ensemble.

between a breath-first to a depth-first search can be adjusted to enforce a working-set size that corresponds best to the size of a 'software cache'. Introducing such a software cache to keep node data temporal, it is possible to reduce the number of data-load phases for charge-groups for a factor of about 20 for a 'software cache-size' of 256.
To decrease the number of required lock/unlock operations, elements are purged from and loaded into the software cache not solely but in groups. This already helps a lot, but it turned out that it is not yet sufficient. Ensuring mutual exclusion with a single lock results in considerable contention for that lock. Fortunately, the locality-enforcing pair-list re-ordering can also be exploited to reduce lock contention. The idea is to assign different locks to nearby nodes in the grid (those that are geometrically adjacent or connected by short paths in the grid). Processing interactions in a temporal local way means at the same time that also the groups of nodes that are purged from the cache show some degree of locality and therefore correspond to the same lock or at least just some few. Using e.g. 16 locks, which is a quantity that allows a good scaling behavior in terms of lock contention for reasonable degrees of parallelism, it was possible to reduce the number of lock/unlock operations to just one for about every 10 nodes that are together purged from cache as a block. (see table 1 for a summary).
Analogously to the pair-list construction module, the short-range force array is switch to a sharing mode when this module is entered and switched back to a replicating mode at it's end to allow to keep other modules unchanged.

4.2.3 Load balancing

It has already been mentioned that the parallelization as described so far results in load imbalance within the short-range interaction module. The reason for this is that the outcome of the pair-list construction module per process directly defines the work-load of the short-range interaction evaluation module. The work-load of the pair-list construction module has been scheduled for load-balance but computational load of the pair-list construction process is not necessarily proportional to the amount of generated load for the succeeding short-range interaction evaluation module.
To eliminate load imbalance in the short-range interaction module, a redistribution of the process-local pair-lists is performed. Besides enforcing load-balance (in terms of an equal number of short-range interactions per process) this allows even more optimizations. So far, the pair-list grid has been partitioned implicitly by the loop-scheduling within the pair-list construction module among the processes. Clearly, this results in a distribution in which each process' pair-list contains edges from allover the grid. The pair-list reordering step is able to deliver the more temporal locality the more geometrically adjacent grid-regions are concentrated within single processes. Such a distribution is enforced at the same time the pair-lists are re-distributed for load-balancing reasons. This is done with a simple geo-

	thrombin in water	water (large data set)
no. solute molecules	1	0
no. solute atoms	3,078	0
no. solute charge-groups	1,285	0
no. solvent molecules (H_2O; = no. of solvent charge-groups)	5,427	13,824
no. solvent atoms	16,281	41,472
total no. of atoms	19,359	41,472
pair-list update / long-range force evaluation rate (t_{pl})	5	5
no. short-range solute-solute and -solvent charge-groups interactions	57,735	0
no short-range solvent-solvent charge-groups interactions	175,326	442,584

Table 2: Some characteristic quantities of the benchmark data sets.

metrical partitioning of the three-dimensional solution domain (see e.g. [8,22] for grid-partitioning).

4.3 Possibilities to expand the parallelization

The module that is time-critical in third place is the SHAKE module (see figure 1). The next step in the described step-by-step parallelization process would be to parallelize this one. Although SHAKE contributes only minor to the sequential execution time, it limits the scalability considerably for higher degrees of freedom according to Amdahl's law. Parallelizing SHAKE would also allow to eliminate most of the calls to replicate the force shared memory regions (before SHAKE) and to switch back to sharing again later on (after SHAKE), together with the respective overhead.
However, this has not been done yet.

5 Performance

5.1 Evaluation platform

The cluster platform used for evaluation purposes comprises out of dual-processor Intel Pentium Pro multiprocessors (200 MHz; 256 KByte L2-cache) that run under Windows NT 4.0. These, together with a file-server, are interconnected with a Fast Ethernet e.g. for access to a common, cluster-wide file system. For parallelization purposes, the machines are memory-coupled with Dolphin´s first generation PCI-SCI adapters [4,18]. Based solely on commodity components, such a cluster shows a great price/ performance ratio (currently, each dual-processor compute node equipped with a SCI adapter cost about $5000).

5.2 Results

The performance of the parallelized code is evaluated using two benchmark problems from Biomos itself (the distributor of GROMOS) [7]: a thrombin protein molecule in water and a large water ensemble (for some parameters of the data sets, see table 2). The thrombin data set is of interest because it is the only one of relevant size that considers also long-range interactions. The water data set was chosen because it's size is comparable to those problems that are of real interest.

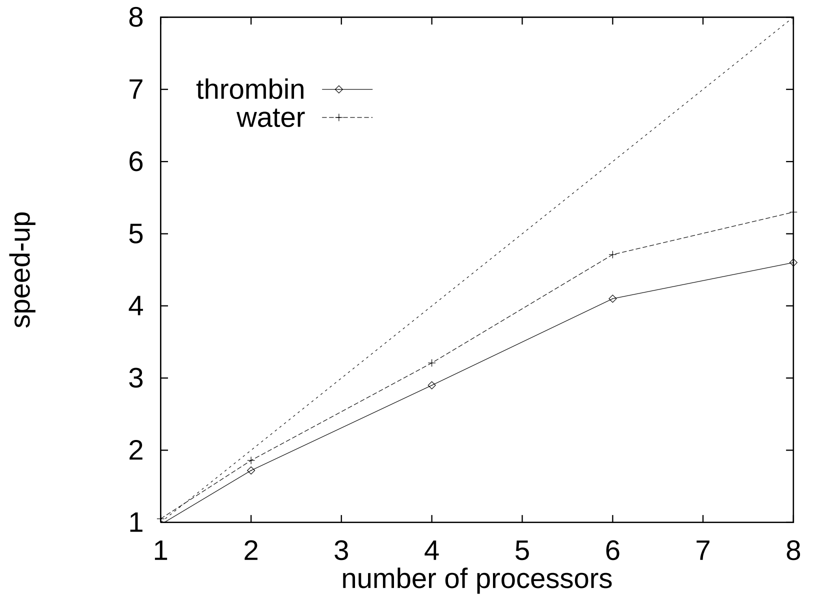


Figure 5: Speed-up figures of the parallelized code on the SCI cluster.

Figure 5 shows the obtained speed-up quantities, running the program on up to four machines, corresponding to a total of eight processors. It should be noted that these quantities have been derived by relating the measured run-times to that of the original sequential code, not to the parallelized code running with a single process. However, both perform very similar.
The decay of the speed-up increase with an increasing number of processors has three major reasons:

- The fraction of each global shared memory region that is local to each process decreases. Because of this and because the data-layout in the regions has not been adapted according to locality and work partitioning, the fraction of remote accesses grows proportional to the number of machines.
- The relative overhead of switching between sharing and replication of the shared memory regions grows.
- Modules that remained sequential limit scalability for higher degrees of freedom according to Amdahl's law.

All three issues result from the strategy to limit code modifications for the purpose of parallelization to just a few functions and to parallelize just the most time-consuming modules, not the whole code. This approach has been chosen with consciousness and payed off. There are no principle reasons that would prohibit to remove a large fraction of this overhead.

5.3 Comparison to other parallelization efforts

It is interesting to compare the achieved performance figures to other parallelization efforts of GROMOS. First of all, Biomos itself presents some performance figures of a thread-based shared memory parallelization of GROMOS96 [7] on a SGI Power Challenge. Figure 6 relates the speed-up figures of the Power Challenge to those of the SCI-cluster.
Within the Europort parallelization project [6,14], different data-sets have been used, but of comparable size (ranging from ~2,000 to ~30,000 atoms: proteins with water). The evaluation of the resulting message-passing code has been done on various parallel machines, e.g. IBM SP1/2, Intel i860 Hypercube and SGI Power Challenge. For comparison purposes, the speed-up quantities for 8 processors are of

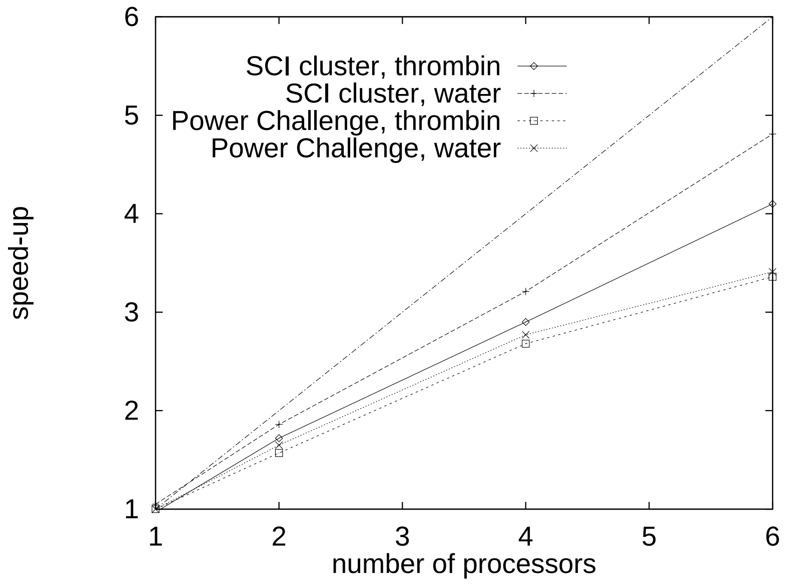


Figure 6: Comparisson of speed-up figures for the thrombin and the water benchmark data set for the Biomos parallelizationa on a SGI Power Challenge and the herein presented parallelization on a SCI-cluster.

interest. Values between 4.4 and 5.6 are reported in [6] and between 4.8 and 6.7 in [14]. The first one refer to a not entirely parallelized code, comparable to that described herein, while the later refer to an entirely parallelized code.

The National Center for Supercomputing Application provides performance figures for UHGROMOS [15] on various platforms (HP/Convex SPP1200 and SPP2000, SGI Challenge and Origin). The reported speed-ups for comparable data sets of 10,000 to 15,000 atoms (protein in water) employing 8 processors are between 4.3 and 5.4. It should furthermore be denoted that the thereby employed basis for speed-up calculation is the computation time of the parallel version of this code, running on a single processor. But this code is already about 15% slower than the original one.

All those benchmarks were run on dedicated (and therefore expensive) parallel machines. Some of them are pure message-passing machines (IBM SP1/2, Intel i860 Hypercube). Some are UMA (Uniform Memory Access) shared memory machines (SGI Challenge and Power Challenge, HP/Convex SPP1200 and SPP2000 for this number of processors) with a highly sophisticated memory system and memory interface (a high-performance bus or a cross-bar) and the SGI Origin is a highly sophisticated CC-NUMA machine. But anyway, the achieved scaling behavior is not significantly better than that, reported for the herein presented parallelization on the SCI-cluster system.

6 Conclusions

From a technological point of view as well as market observations, it seems that NUMA shared memory becomes the predominant architectural principle of parallel systems. This regards entire systems, like those from Sequent, Data General, HP/Convex, SGI, etc., as well as clusters which are assembled from commodity off-the-shelf components.

The described work is part of a larger effort that aims at application parallelization on especially the NUMA shared memory cluster systems. In this paper, methodologies are sketched that allow to exploit the given architecture for shared memory parallelization. At first glance, one might get the impression that the parallelization effort is quite high. But most of the work spend was to improve temporal data locality. But as the performance of the processors grows much more rapidly than that of memory, this is

highly demanded also for a pure sequential program. Also other studies report the necessity to improve per-process data locality as a precondition for a scalable parallelization [2].
All of the advantages of shared memory parallelization could be kept also on the present cluster platform:

- the possibility of a “scalable” step-by-step parallelization process and
- the common view of all of the data for all processes which makes parallel programming much simpler than dealing with partitioned and distributed data structures as usually within a message passing programming model.

Concretely, for the described parallelization of GROMOS96 just 7 source code modules had to be touched. 5 of them saw only minor modifications:

- 3 for parallelism- and shared-memory regions initialization purposes (~ 6,500 lines of code) and
- 2 for I/O adoption (~ 2,700 lines of source code).

Just 2 have been modified more extensively, that’s where the actual parallelization took place:

- the pair-list construction with long-range force evaluation (~ 2,200 lines of code) and
- the short-range force evaluation (~ 2,300 lines of code).

The resulting performance of the parallel code is comparable to that of other parallelization efforts that employed expensive dedicated parallel machines. Considering that the expenditure of work for the parallelization (and therefore of money) has been much less than e.g. that of the Europort project and that the hardware platform is assembled of off-the-shelf components that are on an order of magnitude less expensive than a dedicated parallel system, this is really impressive.
The experience gained during this work leeds to the conclusion that NUMA shared-memory cluster platforms are more than just interesting alternative to dedicated parallel systems as well as to LAN-connected cluster systems (e.g. PC clusters of the Beowulf type [21]). Lessons learned during the described work are that there is a considerable demand for suitable programming interfaces and that it is essential for a programmer to be used to the NUMA performance characteristic. The development of programming interfaces is an active area of development, e.g. SMI has been developed for the purpose of shared memory parallelization [5], also the implementation of message passing libraries (MPI and PVM) is on it’s way [23]. The parallelization of applications should receive even more attention in the future, to gain experience with the somewhat special performance characteristic of the platform and to demonstrate the advantages of this type of cluster platform.

References

[1] Allen, M. P.; Tildesley, D. J.: *Computer Simulation of Liquids.* Oxford University Press, 1987.

[2] Bugge, H. O.; Husoy, P. O.: *Efficient SAR processing on the Scali System.* Proc. IPPS, 1997.

[3] Clark, T. W.; v. Hanxleden, R.; McCammon, J. A.; Scott. L. R.: *Parallelizing Molecular Dynamics using Spatial Decomposition.* Proc. Scalable High Perf. Comp. Conf., 1994.

[4] Dolphin Interconnect Solutions: *PCI-SCI Cluster Adapter Specification.* White Paper, January 1996.

[5] Dormanns, M.; Sprangers, W.; Ertl, H.; Bemmerl, T.: *A Programming Interface for NUMA Shared-Memory Clusters.* Proc. High Performance Computing and Networking (HPCN), pp. 608-707, LNCS 1225, Springer, 1997.

[6] Green, D. G.; Meacham, K. E.; van Hoesel, F.: *Parallelization of the molecular dynamics code GROMOS87 for distributed memory parallel architectures.* Proc. High Performance Computing and Networking (HPCN), pp. 875-879, LNCS 919, Springer, 1995.

[7] GROMOS96 benchmark results: http://igc.ethz.ch/gromos/benchmark.html

[8] Heiß, H.-U.; Dormanns, M.: *Partitioning and Mapping of Parallel Programs by Self-Organization.* Concurrency: Practice & Experience, Vol. 8, No. 9, pp. 685-706, Nov. 1996.

[9] Huse, L. P.; Omang, K.: *Large Scientific Calculations on Dedicated Clusters of Workstations.* Int. Conf. on Par. and Distr. Systems, Euro-PDS, 1997.

[10] IEEE: *ANSI/IEEE Std. 1596-1992, Scalable Coherent Interface (SCI).* 1992.

[11] Lam, M. S.; Rothberg, E. E.; Wolf, M. E.: *The Cache Performance of Blocked Algorithms.* Proc. 4th. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), 1991.

[12] Lankes, S.: *Parallelisierung einer Komponente eines Flugplanungs-Codes auf einem speichergekoppeltem PC-Cluster.* Diploma Thesis (in German), Chair for Operating Systems, RWTH Aachen, April 1998.

[13] Lupo, J. A.: *Benchmarking UHGROMOS.* Proc. 28th Int. Conf. on System Sciences, pp. 132-141, 1995.

[14] Meacham, K.; Green, D.: *Parallelization of the GROMOS87 Molecular Dynamics Code: An Update.* Proc. High Performance Computing and Networking (HPCN), pp. 170-176, LNCS 1067, Springer, 1996.

[15] National Center for Supercomputing Applications: *Computational Biology Applications, UHGROMOS and gromos87 Benchmarks.*
http://mithril.ncsa.uiuc.edu/SCD/straka/PerfAnalysis/Apps/cb.html

[16] Paas, S. M.; Dormanns, M.; Bemmerl, T.; Scholtyssik, K.; Lankes, S.: *Computing on a Cluster of PCs: Project Overview and Early Experiences.* 1. Workshop Cluster-Computing, Proceedings published as: Technical Report CSR-97-05, TU Chemnitz, Dept. of Computer Science, 1997.

[17] Pfister, G. F.: *In Search of Clusters.* Prentice Hall, 1995.

[18] Ryan, S. J.; Gjessing, S.; Liaaen, M.: *Cluster communication using a PCI to SCI interface.* Proc. 8th Int. Conf. on Parallel and Distributed Computing and Systems (IASTED), 1996.

[19] Sinnen, O.: *Loop-Scheduling und -Splitting Verfahren auf NUMA Multiprozessoren.* Diploma Thesis (in German), Chair for Operating Systems, RWTH Aachen, 1997.

[20] van Gunsteren, W. F.; Billeter, S. R.; Eising, A. A.; Hünenberger, P. H.; Krüger, P.; Mark, A. E.; Scott, W. R. P.; Tironi, I. G.: *Biomolecular Simulation: The GROMOS96 Manual and User Guide.* BIOMOS b.v., Zürich, Groningen and vdf Hochschulverlag AG an der ETH Zürich, 1996.

[21] Warren, M. S.; Becker, D. J.; Goda, M. P.; Salmon, J. K.; Sterling, T.: *Parallel Supercomputing with Commodity Components.* Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp. 1372-81, 1997.

[22] Wöhler, R.: *Eine Softwareinfrastruktur zur mehrstufigen Graphpartitionierung.* Diploma Thesis (in German), Chair for Operating Systems, RWTH Aachen, 1996.

[23] Zoraja, I.; Hellwagner, H.; Sunderam, V.: *SCIPVM: Parallel Distributed Computing on SCI Workstation Clusters.* To appear in Concurrency: Practice and Experience.