ELSEVIER

Computer Physics Communications 147 (2002) 711–715

Computer Physics Communications

www.elsevier.com/locate/cpc

# On the parallelization of molecular dynamics codes [☆]

G.P. Trabado [*], O. Plata, E.L. Zapata

*Department Computer Architecture, University of Málaga, E-29071 Málaga, Spain*

**Abstract**

Molecular dynamics (MD) codes present a high degree of spatial data locality and a significant amount of independent computations. However, most of the parallelization strategies are usually based on the manual transformation of sequential programs either by completely rewriting the code with message passing routines or using specific libraries intended for writing new MD programs. In this paper we propose a new library-based approach (DDLY) which supports parallelization of existing short-range MD sequential codes. The novelty of this approach is that it can directly handle the distribution of common data structures used in MD codes to represent data (arrays, Verlet lists, link cells), using domain decomposition. Thus, the insertion of run-time support for distribution and communication in a MD program does not imply significant changes to its structure. The method is simple, efficient and portable. It may be also used to extend existing parallel programming languages, such as HPF. © 2002 Elsevier Science B.V. All rights reserved.

*PACS:* 02.70.Ns; 07.05.Tp; 31.15.Qg

*Keywords:* Molecular dynamics; Short range; Parallel computers; Runtime system; Hierarchical domain decomposition

## 1. Introduction

Codes simulating large particle systems, specially those using short-range molecular dynamics (MD) techniques, present a high degree of spatial data locality and a significant amount of independent computations. The parallelism of these codes is mainly in the computationally intensive force evaluation, with a cost in the order $\mathcal{O}(N^2)$ ($N$ is the number of particles in the system), assuming short-range particle interactions. However, the most efficient parallel MD codes described in the literature [2,6] are based on manual restructuring of sequential programs and the introduction of communications specific for the problem structure. The main reason lies in the high computational complexity of this problem, which makes it very difficult for a compiler to automatically exploit the inherent parallelism or for a parallel language to express it to its full extent. This complexity is mainly due to several factors: the use of the cell index and the Verlet neighbor list book-keeping optimization techniques [1] to speed up the search for interacting particles; the possible non-uniform distribution of particles in the domain (this may introduce workload balancing problems); and the dynamic evolution of the system, with the corresponding migration of particles across the domain.

With all these difficulties MD has turned into a very interesting problem for the parallel computing

[☆] This work was supported by the Ministry of Education and Culture (CICYT), Spain, through grant TIC2000-1658.

[*] Corresponding author.
*E-mail addresses:* guille@ac.uma.es (G.P. Trabado), oscar@ac.uma.es (O. Plata), ezapata@ac.uma.es (E.L. Zapata).

0010-4655/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0010-4655(02)00381-8

community. Different parallelization alternatives have been devised during the last years. We may classify them into four categories:

- *Manual*: The complexity of MD programs has caused that many of the existing high-quality parallel codes have been developed manually (explicit problem partitioning with message passing primitives, mainly). However, manual code restructuring usually requires tedious and complex analysis and full rewriting of the program (some data structures may be cleanly partitioned but others not). This fact turns manual parallelization into a very hard programming exercise.
- *Automatic*: This is an alternative to avoid the tedious task of manual restructuring. In the case of irregular codes current technology produces partially parallelized codes based on the inspector-executor model [5]. The main drawback, however, is that an automatic parallelizer can only exploit program parallelism, that is, the parallelism available at the programming language level. However, important properties, such as the short-range nature of particle interactions, are not usually achievable at the above level (using conventional languages). This property, however, is very important in order to obtain a high-quality parallel code.
- *Language-based*: Language-based parallelization consists in extending a conventional language with new parallel syntactic/semantic constructs. The aim is to allow the programmer to express problem properties at the language level. This way the compiler has the opportunity to generate a better parallel code. Important progress has been made in, for instance, HPF [3,8]. However, this approach is currently at the research stage, due in part to severe difficulties in compiler analysis and implementation.
- *Library-based*: Another approach consists in the design of specific libraries intended for solving data distribution and communication. These two issues usually correspond to the most tedious and error-prone tasks while writing parallel MD programs. Some solutions have been proposed in the literature [4,7]. However, they require major recoding work when used to parallelize existing MD codes, very often because the library only supports some particular format for program data intended for quick development of a new code.

In this paper we propose a library-based approach, known as *Data Distribution Layer* (DDLY), which reduces manual intervention in the parallelization of existing short-range MD sequential codes. Our approach takes advantage of the knowledge about the problem nature to reduce the parallelization effort by minimizing changes to the original sequential program [9]. The novelty of this approach is that the distribution and communication support can handle "*as-is*" common data structures used in MD codes to represent data (particle arrays, Verlet lists, link cells).

The method requires only a quick analysis of the sequential MD code, introduces minor rewrites to obtain the parallel version, generates efficient parallel codes (performance similar to using fully manual restructuring), is portable and existing parallel languages can be extended to use it (extensions to HPF have been proposed [8]).

## 2. Parallelization of MD codes with DDLY

The principal goal of our method is to preserve as much as possible the original code of MD algorithms. The analysis of different data representations used in a representative subset of MD codes ([1,6] and the *Perfect Benchmarks*) shows that there are some facts common to all of them:

- Positions and other properties are stored in one multicolumn or several single-column arrays.
- Specific indices are not relevant for MD calculation (except for some special cases in which particle tracking is needed).
- Short-range MD is usually optimized using Verlet lists (nearest neighbor lists). Usual representation is an array of integer indices.
- Link cells are usually introduced to speed up nearest neighbor calculation.

From this remarks we observed the convenience of designing a run-time support which accepted to manage those sets of arrays previously identified as playing a specific role in the program. The resulting parallel code should use the same data structures. Due to ir-

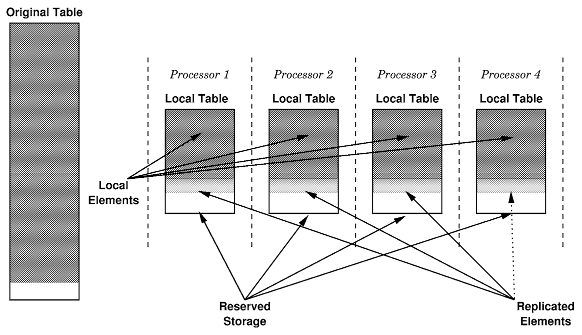

Fig. 1. Memory model. Nodes only store elements assigned by domain decomposition. Elements received from neighbor subdomains are copied after local ones.

relevance of indices in computations, the use of local indices is possible without a global-to-local translation mechanism. Another advantage of code structure is that communication can be hidden by mixing local and received elements (Fig. 1) in the same data structures so that processing of non-local elements is homogeneous.

On the other hand, domain decomposition has proved to be an efficient technique for the parallelization of short-range MD problems. Each partition contains data which is most likely to interact. However, some well-known difficulties exist:

- Computation of subdomain boundaries and overlaps should be efficient.
- The representation of the decomposition should be compact and allow irregular partitions for load-balancing purposes.
- Interaction across subdomain boundaries requires communication of particles on the surrounds of a partition (called *overlay area*).

The use of link cells is fundamental to solve such problems. Fig. 2 shows how the link cell is used to compute a domain decomposition with a balanced number of particles. The resulting decomposition can be expressed in terms of the starting indices of each piece of the $T$, $T_1$ and $T_2$ vectors ($\{1,4\}$ for the first dimension and $\{1,4\},\{1,3\}$ for the second). Those indices describe the hierarchical decomposition in terms of indices in the link cell. $T=\{1,4\}$ means that the $y$ dimension is split in two subdomains containing cell rows 1 to 3 and 4 to 6.

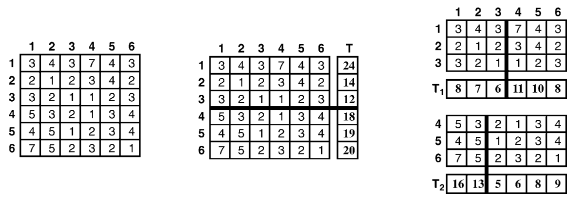

Fig. 2. 2D Hierarchical domain decomposition using a *link cell*. Left: The array shows the particle count for each cell. Center: Cells are added by rows into vector $T$, which is split keeping the number of particles balanced. Right: Each subdomain is split again by the next dimension.

The link-cell structure is also very useful to compute the communication schedules for overlay areas, and to detect when a particle enters a new subdomain and its ownership must be changed.

## 2.1. The DDLY support

The only object that the DDLY library introduces is the *DDLY descriptor*, which is a data structure which internally stores control data needed by the run-time support. A subset of the DDLY support is *declarative functions*, which are intended to describe the role of the main data structures of the MD code. The usual way of doing this is creating a descriptor for each data structure and then using a DDLY function to describe attributes of the data structure such as size. The descriptor is now *associated* to the data structure. After this point, the DDLY library is responsible for distribution and communication of the contents of the structure between the different processes.

With this method, the programmer may identify particle lists (coordinates arrays) and link cells from the MD code. Any other values associated with particles can be distributed through *alignment* operations, which are also declarative functions of DDLY.

DDLY has also a set of distribution and communication functions which can be invoked on the descriptors. For example, the domain decomposition shown in Fig. 2 can be performed by inserting a call to the *DDLY_DISTRIBUTE* function on the descriptor associated to the link cell.

In summary, the way of parallelizing a MD code with DDLY is the following. The programmer analyzes the code and identifies the data arrays representing particles, link cells, Verlet lists, etc. Calls to DDLY

Table 1
Wall-clock time spent during parallel MD execution for Cray T3E. Left figure: 160 K part. Right figure: 640 K part

| nPE | Simul. | | Dist. | | Update | | Migr. | |
|---|---|---|---|---|---|---|---|---|
| 1 | 637.18 | | 0.17 | | 0.33 | | 0.81 | |
| 2 | 310.71 | | 0.34 | | 2.48 | | 7.45 | |
| 4 | 163.25 | | 0.32 | | 3.27 | | 3.82 | |
| 8 | 85.65 | 319.69 | 0.32 | 1.57 | 3.22 | 4.65 | 2.18 | 7.19 |
| 16 | 46.26 | 166.36 | 0.38 | 1.20 | 2.93 | 6.87 | 1.41 | 3.95 |
| 32 | 26.42 | 87.87 | 0.58 | 1.72 | 2.62 | 3.59 | 1.90 | 2.32 |
| 64 | 15.77 | 47.91 | 1.91 | 2.84 | 3.00 | 2.91 | 0.83 | 1.50 |

are introduced to declare descriptors and to associate each structure to its own descriptor.

Later in the program, calls to functions are introduced to request domain decomposition and array distribution. Other changes to the rest of the MD code which will execute on local data are the introduction of calls to communicate overlays every time step and the substitution of original number of elements in data structures for the number of those assigned to each process (obtained also from call to DDLY).

## 3. Results and conclusions

This section shows the results of tests with a 2D short-range MD code which models atom interactions with a Lennard–Jones potential, truncating force computations at a distance $r_c = 2.5\sigma$. A link cell provides cell decomposition with a cell size of $2.9\sigma$ and periodic boundary conditions. A Verlet list is used with updates every 10 time steps.

Table 1 shows detailed timing results for $N = 160,000$ and $N = 640,000$. Column *Simul* is the wall-clock time taken by the MD computation (including communications). Column *Dist* depicts the time spent by initial domain decomposition and array distribution. Column *Update* is the total time spent on computing communication schedules and exchanging data on overlay areas. *Migr* shows the time spent for migration (inspection of border cells and data exchange). The results show a clear tendency: the synchronization overhead (particle replication and migration) decreases as the number of processors increase. The simulation time scales nearly as $O(N/P)$. Fig. 3 depicts resulting speedups for sets from 40,000 up to 640,000 particles. Speedup gets closer to ideal as predicted for domain decomposition methods. These results can be extrapolated to 3D simulations as the method only differs in the number of cells to be examined and exchanged between processors.

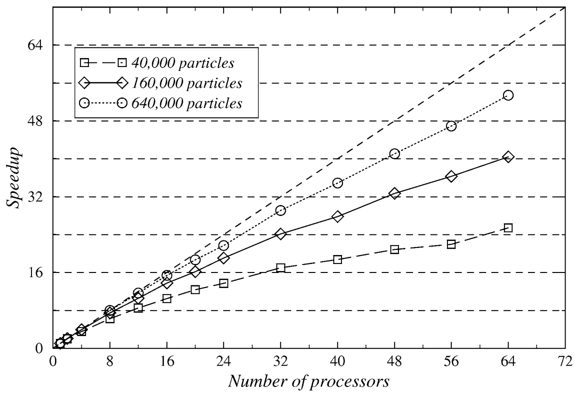

Fig. 3. Speedup of 2D short-range MD simulation tested on the Cray T3E for three different particle sets.

The major motivations of this work are achieving easy parallelization of existing sequential short-range MD codes and yielding efficient execution speedups similar to those of manually parallelized MD codes using domain decomposition. Also, portability is enforced as DDLY is based on PVM and MPI libraries as MP standard. The process of manual parallelization has been greatly simplified, although it is already interesting to design extensions for languages like HPF for automatic parallelization at compilation time.

## References

[1] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Clarendon, Oxford, 1987.

[2] D.M. Beazley, P.S. Lomdahl, N. Grønbech-Jensen, R. Giles, P. Tamayo, World Scientific's Ann. Rev., in: D. Stauffer (Ed.), Computat. Phys., Vol. 3, World Scientific, 1996, p. 119.
[3] P. Mehrotra, J.V. Rosendale, H. Zima, J. Par. Comput. 24 (1998) 325.
[4] R. Ponnusamy, J. Saltz, A manual for the CHAOS runtime library, Tech. Rep., UMIACS, University of Maryland, 1994.
[5] R. Ponnusamy, J. Saltz, A. Choudhary, S. Hwang, G. Foz, IEEE Trans. Par. Distr. Syst. 6 (1995) 815.
[6] S. Plimpton, J. Comp. Phys. 117 (1995) 1.
[7] S.G. Srinivasam, I. Ashok, H. Jônsson, G. Kalonji, J. Zahorjan, Comput. Phys. Commun. 102 (1997) 28.
[8] E.L. Zapata, O. Plata, R. Asenjo, G.P. Trabado, J. Par. Comput. 25 (1999) 1971.
[9] G.P. Trabado, E.L. Zapata, in: Proc. of the LCPC'97, Lecture Notes in Comput. Sci., Vol. 1366, Springer-Verlag, Germany, 1997, p. 218.