Parallelisation of Nonequilibrium Molecular Dynamics Code for Polymer Melts Using OpenMP

Zhongwu Zhou[1], B.D. Todd[1], and Peter J. Daivis[2]

[1] Centre for Molecular Simulation, Swinburne University of Technology
PO Box 218, Hawthorn, Vic. 3122, Australia
[2] Dept. of Applied Physics, RMIT University
GPO Box 2476V, Melbourne, Vic. 3001, Australia

Abstract. The parallelisation of a sequential nonequilibrium molecular dynamics (NEMD) code for simulating polymer melts is presented. The issues impacting the efficiency of the parallel executable are probed. Various techniques, such as loop interchange, loop fusion and code restructure, have been applied to the incremental OpenMP parallelisation. Significant performance improvement and speed up are achieved for large sized systems when the parallelized code is compared to the existing sequential code. The parallelised code has successfully been applied to simulate the shear rheology of a polymer melt system.

1 Introduction

Nonequilibrium molecular dynamics (NEMD) has proven to be a useful tool in investigateing transport properties of materials. However, NEMD simulations of polymeric systems are often crippled by the excessively high computational effort required. The complexity arises mostly from the large number of atoms involved and the longer relaxation time of the materials. NEMD code for the simulation of polymer melts under shear and planar elongational flows at realistic flow conditions, recently developed in Fortran 90 by Matin, Daivis and Todd [1–3], was specifically optimised for high efficiency on vector architecture processors. However, it performs poorly on cluster supercomputers. The purpose of this work is to parallelise the NEMD code using OpenMP parallelism and explore the issues that impact the efficiencies of the parallelised code. Within the paper we also report the application of this parallelised code in successfully simulating the shear rheology of a polymer system.

OpenMP [4] is a portable programming model for shared memory architecture based on threads. It offers a small but efficient set of language constructs that support both fine- and coarse-grained parallelism paradigms. The fine-grained paradigm parallelises most of the loops in a code, which is simple and only requires a quick analysis of the loop in question. However, sometimes the number of loops is so large and the computation task in a loop is so small that the fine-grained scheme is probably not appropriate. The coarse-grained paradigm requires a parallelisation strategy similar to a MPI strategy and explicit synchronisation is required. The strength of the OpenMP approach lies in the possibility to proceed incrementally. It is easier to use OpenMP to convert an existing code. Although do-loop splitting with OpenMP is less efficient and scalable than domain decomposition (DD) using MPI approaches, its fast implementation involves significantly less programming effort.

2 The NEMD Code

The NEMD code was developed for simulating linear chain polymers under shear and planar elongation [1–3, 5]. The polymer chain is treated as a freely joined chain (FJC) where a chain can be characterized by the number of beads (or sites) and the bond length between two adjacent sites. For performance gains, an efficient cell-code algorithm for constructing neighbour lists was implemented to calculate the forces [2]. The appropriate equations of motion for the positions and momenta are based on the SLLOD equations of motion for molecular fluids [6,7]. A fifth order Gear Predictor-Corrector Scheme was used for integrating the equations of motion [8]. Bond lengths between adjacent sites on the same molecule are held constant by bond-constraint forces, determined by using Gauss's principle of least constrain [9].

Normally, the effort exerted to generate the parallel code must be weighted against the speed-up reachable. Extensive performance analysis on the code and profiling tests reveal that various subroutines contribute cooperatively towards the total execution time, such as, the subroutines for advancing particles, calculating forces, and applying constraints. The amount of computational efforts involved in each subroutine can vary considerably depending on simulation parameters. For example, the force procedure becomes dominant in time consumption for systems of high density and short chains, while the time spent in computing bond constraints significantly increases as the number of sites per chain increases. The information from the code analysis and profiling tests suggests that most major subroutines have to be parallelised.

3 Parallelisation of the NEMD Code

When one writes parallel programs, one expects a linear speed up in performance. However, there are some hurdles to overcome in the efficient parallelisation of loops. For example, loops may have data dependencies among iterations caused by shared variables which result in some un-parallelisable code. There may be no sufficient work in a loop body and the performance suffers from the high parallel start-up costs. There may be too many references to shared variables and low cache affinity. Various techniques, such as loop interchange and loop fusion, were applied in the loop transformations to improve the parallel efficiency. In order to optimise data locality and efficient memory utilisation some procedures were restructured. To achieve this, it is necessary to maintain a global perspective of the program so that changes in one procedure have no side effects on others. The following sections describe the incremental parallelisation techniques used in detail and discuss the performance effects of the various techniques.

3.1 Data Dependencies

Data dependencies prevent relevant sections of code to be parallelised. The serial parts limit the performance of parallel code and pose an upper limit on the efficiency. In order for a loop to parallelise, access to the shared data must be mutually exclusive. Data dependency can be valid if all the iterations in a loop can be executed in any order and give the same result at the end of the execution. One example is the creation of neighbour lists, nlist[][]. The code fragment is shown as:

```
j = 1
do i = 1, n
  if (nmask(i))  then
    nlist(j, 1) = iindex(i)
    nlist(j, 2) = jindex(i)
    j = j + 1
  endif
enddo
```

nmask is a logical array that identifies the particle pair within a cut-off distance, iindex and jindex are arrays of indices of the paired particles. Obviously, the value of *j* in one iteration step depends on the results of previous iteration, therefore it can only be executed sequentially. Data dependency may also occur if a shared variable is written in one iteration and possibly read in another one. A typical example is the main force computing loop for accumulating forces as given below in pseudo code:

```
loop i <- 1 ... n
   compute k from i
   compute fijx[i]
   fx[k] <= fx[k] + fijx[i]
end loop i
```

fijx[i] is the force between *i* and *j* particles in the x direction and fx[k] is the accumulated force on the particle k in the x direction due to all other particles. From a parallelism perspective, when pair-wise interactions are treated using a neighbour list, complexities arise when atom *k* interacts with atom *i*, and simultaneously with atom *j*. In this case, the force exerted on *k*, fx[k], is updated from the contributions due to atoms *i* and *j*, which cannot be done concurrently on different threads. We had implemented synchronised code by using OpenMP ‘atomic’ updates or ‘critical’ [4] directives. However, testing revealed that the cost of the synchronisation is expensive due to significant synchronisation overhead. Thereafter, this data dependency problem was solved through moving the dependency part into a separate loop, and then only parallelising the main loop without the dependency. The distribution of loop iterations is often based on loop index, therefore, a clean relationship between the loop index variable and array indices is of fundamental importance. The data dependencies in the existing code were introduced from array linearisation in implementing the particular algorithm for the neighbour list. Array linearisation is a common efficient way to facilitate vectorisation of nested loops. However, it can obfuscate the relationship between array and loop indices, thereby foiling parallelisation efforts [10].

3.2 Loop Fusion and Interchange: Performance Improvement by Increasing Parallel Loop Granularity

The number of loops incrementally transformed in a code is sometimes so large that the parallelised code is too fine-grained. For example, the initial transformations of the force procedure produced about 30 parallel do loops. Rather than performance gains, the parallelised code ran slower on 4 processors. This is because many small loops have no sufficient work in their loop bodies, resulting in high parallel overhead when entering and exiting the loops. In later modifications, two techniques, loop fusion and loop interchange, were applied to increase the parallel loop granularity.

Loop fusion increases the work in a loop body by combining several loops. Fusion promotes software pipelining and reduces the frequency of branches, synchronisation and scheduling overhead. Loop fusion can be inhibited by statements between loops which may have dependencies with data accessed by the loops. To promote fusion, it is often necessary to reorder the code to get loops which are not separated by statements creating data dependencies. One example is the calculation of forces and potential energy. As the existing serial code needs to handle different flow types (e.g., shear flow, elongational flow, bulk compression), the force and potential calculations were scheduled into several loops. A segment of the pseudo code is given below:

```
loop i <- 1 ... n
   calculate rijx[], rijy[], rijz[]
end i loop
if not do_elongation
   loop j <- 1 ... n
      calculate PBC's rijx[], rijy[], rijz[]
      calculate rijsq[]
   end j loop
else
   loop j <- 1 ... nab
      transform rijx[]... to rijx_trans[] ...
      calculate PBC's rijx_trans[]...
      transform rijx_trans[] ... back to rijx[]...
      calculate rijsq[]
   end j loop
end if
loop i <- 1 ... nab
   calculate rijxc[],rijyc[],rijzc[],fmask[]
end i loop
loop j <- 1 ... nab
   if rijsq[j] ≤ rcutsq
      calculate uij[],fijx[],fijy[],fijz[]
   end if
end j loop
```

```
sum forces fx[],fy[],fz[]
sum potential uintra[],uinter[]
```

The first loop calculates distances over the whole neighbour list. Next, if the flow is not do_elongation the loop simply applies standard periodic boundary conditions (PBCs) and computes the square of distances, otherwise it first needs to transform distances, rijx, to the appropriate elongational flow PBC frame, rijx_trans [11-13]. The next loop computes centre of mass distances, rijxc, rijyc and rijzc. They are used later for computing the molecular pressure tensor. fmask is a logical array that identifies those pairs within the cut-off distance. After fmask is determined, the loop calculates forces and potential, uij, over all pair interactions within the cut-off distance. The accumulation of forces is conducted in three loops. To sum potential energy over all the particles, several additional loops are used first to separate intra and intermolecular interactions (uintra and uinter) and then to sum them separately.

The reorder and fusion made to the code combined most of the loops into a main loop. First, the *if* condition statement was moved before the computation of distances. This gives a higher level branch and allows distance computation, PBC and force calculations to be done in one main loop. The force accumulation may be combined into the main loop as well. However, as discussed in section 3.1, this can result in data dependency problems, therefore the three loops were modified into one nested loop. The several loops for computing potentials were simplified and then fused into the main loop, with only several lines of code. The loop for the computation of centre of mass distances was moved into other parallel sections. Now the force and potential calculations become two main loops for parallelization as shown in the following:

```
if not do_elongation
   loop i <- 1 ... nab
      calculate rijx[], rijy[], rijz[]
      calculate PBC's rijx[], rijy[], rijz[]
      calculate rijsq[], fmask[]
      if rijsq[j] ≤ rcutsq
         calculate uij[],fijx[]... uintra[],uinter[]
      end if
   end loop i
else
   loop j <- 1 ... nab
      calculate rijx[], rijy[], rijz[]
      transform rijx[]... to rijx_trans[] ...
      calculate PBC's rijx_trans[] ...
      transform rijx_trans[] ... back to rijx[]...
      calculate rijsq[], fmask[]
      if rijsq[j] ≤ rcutsq
         calculate uij[],fijx[]... uintra[],uinter[]
      end if
   end loop j
end if
sum forces fx[], fy[], fz[]
```

In the case of nested loops, once an array dimension and its corresponding loop have been selected for parallelisation, performance can be obtained by moving this loop to the outmost position. This loop interchange method reduces the frequency of entering and exiting a parallel loop and hence the parallel overhead. The loop interchange method was extensively used in parallelising the procedures involving the calculation of constraint forces, proportional feedback and linear equation solvers. Given below is an example of the code to calculate the symmetric dot product.

```
      do i = 1, n
        do j = i, n
          a(i,j,:)=(x12(i,:)*x12(j,:)+y12(i,:)*y12(j,:) &
          &        + z12(i,:)*z12(j,:))*ka(i,j,:)
          a(j,i,:) = a(i,j,:)
        enddo
      enddo
```

The innermost loop is an implicit one expressed as an array operation. The parallelisation of this nested loop includes the conversion of the implicit loop to an explicit one and loop interchange. The parallelised loop is shown below:

```
!$omp  parallel do default(shared) private(i, j, k)
    do k = 1, nm
      do i = 1, n
        do j = i, n
          a(i,j,k)=(x12(i,k)*x12(j,k)+y12(i,k)*y12(j,k) &
           &            + z12(i,k)*z12(j,k))*ka(i,j,k)
          a(j,i,k) = a(i,j,k)
        enddo
      enddo
    enddo
```

3.3 Parallel Regions: Performance Improvement by Reducing fork/joins

There are some cases in which loop fusion is not suitable. One case is when several loops have to be executed sequentially. Another case is when various loops have a different loop index or step. We can parallelise each individual do loop. However, this possibly produces parallel start-up overhead because of many thread forks and joins involved. OpenMP supports parallel regions. Several do loops can be put inside the parallel region. New threads are forked when entering the region and then joined when exiting the region. Parallel loops inside the parallel region are executed in a work-sharing fashion. The execution of associated statements is distributed among existing threads without new threads created when entering the next loop from the current one inside the parallel region. In the parallel NEMD code, performance improved when a series of parallel loops were enclosed within a parallel region. This replaced multiple fork/joins with a single fork/join. Another parallel region construction is parallel sections, which define a sequence of contiguous blocks. The beginning of each block is marked with a "section" directive and one block is assigned to one thread. This method was applied for computing the molecular pressure tensor.

3.4 Efficient Use of Memory

Effective parallelisation and efficient memory utilisation are tightly coupled. This requires maximising cache reuse and minimising cache misses. The way to efficiently use local memory (caches) is to use a memory stride of one. This means array elements are accessed in the same order they are stored in memory. Fortran uses "Column-major" order for storing array elements. When possible, nested loops in the code were interchanged to make the leftmost index of a multi-dimensional array in the inner loop to achieve the preferred order. This allows the inner loop to correspond to access in consecutive data elements and promotes the reuse of caches. The second example presented in Section 3.2 can also illustrate this point. In order to increase the data locality, most array variables used for holding intermediate data in the code were replaced with scalar variables. The parallel performance was improved by declaring these variables as private to threads. This also reduces the memory usage, particularly in the case of using the so-called 'brute force' method [5].

3.5 Parallelised Code Performance

In order to probe the efficiency of the parallelised NEMD code, a series of tests were performed. Most tests were conducted on systems at equilibrium and at a reduced density of 0.8, reduced molecular temperature of 1.0, and reduced time-step of 0.001. The data in Table 1 compare the performances of the existing sequential code and parallelised code running on 4 processors where the cell-code implementation of neighbour list construction is used [2]. It can be clearly seen that significant performance improvement has been achieved for big systems with large numbers of sites per chain. It should be noted that the restructure and optimization of the code during the parallelisation implementation also contribute to the high speed-up for the long chain polymers which is much higher than the processors used. The parallelised code also reduces the memory requirement for running the program. In the case of using the brute force method, nearly 80% memory reduction was achieved for large sized systems. More than 30% memory reduction can be achieved when the cell method is used.

Table 1. Execution time (in CPU hh:mm:ss) of the sequential and parallel codes for systems of fixed particle number (np = 40000) and various numbers of molecules (nm) and sites (ns). Note that nm = np/ns

Ns	1	4	10	20	50	100
Seq. code	00:17:19	00:19:12	00:35:11	00:53:15	05:25:46	22:07:52
Paral. code	00:06:02	00:06:16	00:06:42	00:09:09	00:20:45	01:09:41

For long chain polymers with large number of sites, ns, most of the execution time is consumed in the procedures to solve liner equations and calculate proportional feedback as a result of using Gaussian constraint forces to constrain bond lengths. These procedures contain few nested loops which are very intensive when the site number, ns, is large. The parallel strategies and optimisation techniques discussed in Section 3 are particularly efficient for these procedures, where linear and sometimes super-linear speed-up has been achieved in the case of large site and particle numbers. The super linear speed up may stem from an optimal usage of memory cache and hence the speed up obtained can be higher than the ratio of processors used.

However, the performance improvements for the systems of short chains or small number of particles is less significant. Various factors affect the parallel efficiency. One is the high parallel start-up cost due to the lack of sufficient work in a parallel loop, particularly when the system size is small. This may be attributed to the nature of the 'fork-join parallelism' of OpenMP and the structure of the code where some loops have less computational workload, so that the 'forking' and 'joining' processes are expensive. The serial parts in the procedures involving the formation of neighbour lists and implementation of the cell-code algorithm limit the performance of the parallel code as well. This limitation can become more significant if the neighbour lists need to be updated frequently. Another major factor is the less efficient use of memory caches. For example, the main force computation loop contains intensive computation tasks but with many references to various array variables, and memory access to the elements of some arrays is in an irregular manner. This results in low cache affinity and more cache missing. Despite the less significant performance improvement of the parallel code for small sized systems, the great improvement for large systems is of particular importance for our work, as future simulations will be concentrated on long chain polymers.

4 Simulation Results

In addition to the performance tests discussed in Section 3.5, we conducted simulations on a 50-site chain system of 400 molecules. Equilibrium and shear properties were examined. The simulations were performed at a site density of 0.84 and a molecular temperature of 1.0 with a time-step of 0.001. Due to the longer relaxation time of this system, each simulation typically requires over 1 million steps depending on the value of shear rate applied. This involves around one month of simulation time for a single task if the sequential code is used. The execution time for a simulation is now reduced to several days when the parallelized code runs on 4 processors.

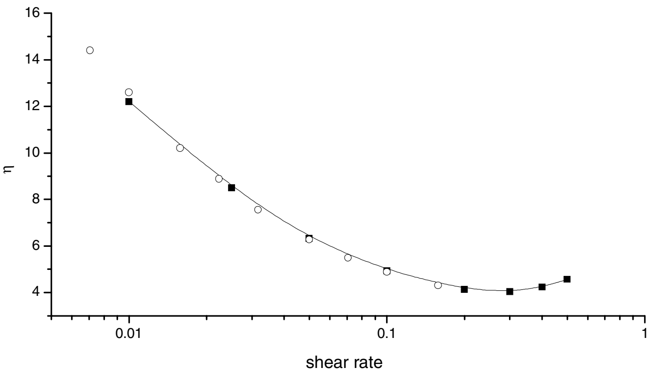

Fig. 1. Non-Newtonian shear viscosity as a function of the shear rate. The open circle data are taken from Matin [5], while our current parallelised simulation data are represented by solid squares

The shear viscosity versus the shear rate is plotted in Fig. 1 for constant volume simulations. A shear-thinning region is evident. The lower Newtonian regime was not reached because the examined shear rates were still not low enough. A slight shear-thickening can be observed from Fig. 1, due to performing the simulations at constant volume [14]. The data plotted in Fig. 1 also include the results from a system of 256 molecules, taken from Matin [5]. Very good agreement is obtained, which validates the accuracy and correctness of the parallel implementation of the code.

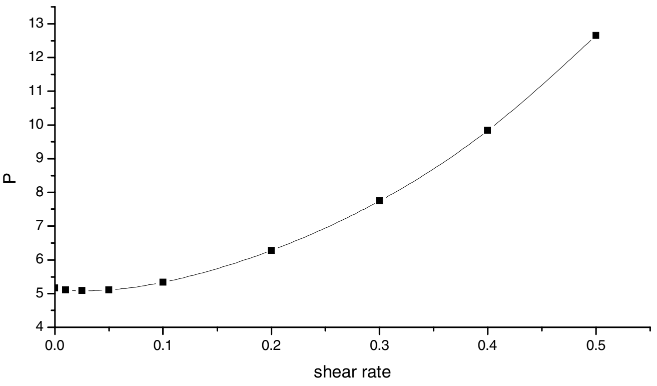

Fig. 2. Pressure versus shear rate

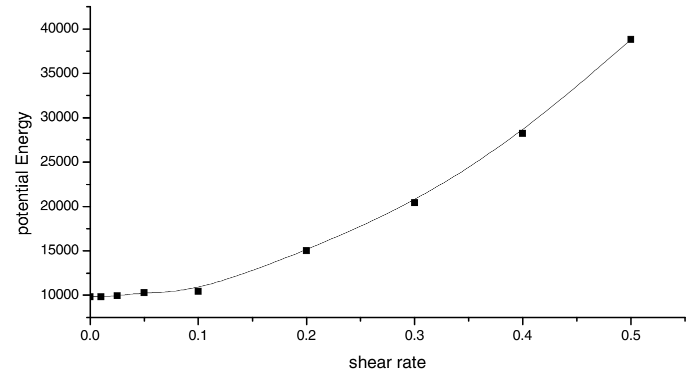

Fig. 3. System potential energy versus shear rate

The results for the molecular pressure and the potential energy are plotted against the shear rate respectively on Figs. 2 and 3. Both the molecular pressure and the potential energy change little when the shear rate is less than 0.1. After this value, they increase rapidly as the shear rate increases. For simple fluids, NEMD simulations have demonstrated that the pressure or energy as a function of shear rate under planar shear flow follows a power law [15, 16]. The results from this study seem unlikely to give such a simple relationship for long chain systems.

5 Summary

A sequential NEMD code has been successfully converted into an efficient parallel version using OpenMP directives. Major techniques, such as loop interchange, loop fusion, and code restructure, have been applied to the incremental OpenMP parallelisation. Performance studies of the parallel code are made. The speed up achieved in the present work is significant for large sized systems with large numbers of sites per chain. This is particularly important for the simulation of long-chain polymer melt systems. Linear or super-linear speed-up is achieved on some parallelised procedures in the case of large sized systems, while the speed up of other parallelised procedures is lower than the theoretical speed up value. The computational workload that can be parallelised varies, depending on the frequency of updating neighbour lists, and hence depends on parametric conditions. The parallelised code has successfully been applied to simulate the shear rheology of a polymer system and produces results well consistent with previously validated serial and vector code.

References

1. M.L. Matin, P.J. Daivis and B.D. Todd, *J. Chem. Phys.* **113** (2000), 9122. [Erratum: *J. Chem. Phys.*, **115** (2001), 5338]
2. M. L. Matin, P.J. Daivis and B.D. Todd, *Computer Physics Communications* **151** (2003), 35.
3. P.J. Daivis, M.L. Matin and B.D. Todd, *Nonlinear shear and elongational rheology of model polymer melts by non-equilibrium molecular dynamics*. Accepted, *J. Non-Newtonian Fluid Mechanics*.
4. `http://www.openmp.org`
5. M.L. Matin, *Molecular Simulation of Polymer Rheology*. Ph.D. thesis, RMIT University (Oct. 2001).
6. R. Edberg, D.J. Evans and G.P. Morriss, *J. Chem. Phys.* **84** (1986), 6933.
7. R. Edberg, G.P. Morriss and D.J. Evans, *J. Chem. Phys.* **86** (1987), 4555.
8. M.P. Allen and D.J. Tildesley, *Computer Simulation of Liquids*, Clarendon Press, Oxford, (1987).
9. D.J. Evans and G.P. Morriss, *Statistical Mechanics of Nonequilibrium Liquids*, Academic Press, New York (1990).
10. T.W.Clark, R. von Hanxleden, etc., '*Programming Issues for Molecular Dynamics*', Computational Biomedicine Symposium, Houston, Texas, Dec. 1997.
11. B.D. Todd and P.J. Daivis. *Phys. Rev. Lett.* **81** (5) (1998) 1118.
12. B.D. Todd and P.J. Daivis. *Computer Physics Communications* **117** (3) (1999) 191.
13. B.D. Todd and P.J. Daivis. *J. Chem. Phys.* **112** (1) (200) 40.
14. P.J. Daivis and D.J. Evans, *J. Chem. Phys*. **100** (1994), 541.
15. G. Marcelli, B.D. Todd and R.J. Sadus, *Phys. Rev. E* **63** (2001), 02 12041.
16. J. Ge, G. Marcelli, B.D. Todd and R.J. Sadus, *Phys. Rev. E* **64** (2001), 02 12011.