Performance and Profiling of the LAMMPS Code on HPCx

Fiona J. L. Reid and Lorna A. Smith

May 12, 2005

Abstract

In this paper we investigate the performance of the LAMMPS molecular dynamics code on two systems; an IBM p690+ cluster (HPCx) and an SGI Altix 3700 (CSAR). The performance of LAMMPS is measured using a number of user and application supplied benchmarks. We have investigated the speed up of the code on up to 1024 processors on the HPCx system and up to 240 processors on the CSAR system. The code is demonstrated to scale well to 1024 processors with a drop in speed up beyond 256 processors. To understand this drop off we have profiled the code on the HPCx system. Profiling the code has enabled us to determine the computationally intense components and to understand the communication patterns and overheads.
## Contents

1 Introduction 1

2 Computer systems 1
   2.1 IBM p690+ Cluster - HPCx ............................................. 1
   2.2 SGI Altix 3700 - CSAR .................................................. 1

3 LAMMPS code - introduction 2

4 LAMMPS performance - including description of the benchmarks 2

5 LAMMPS performance - discussion 4

6 Reproducibility 6

7 Profiling the LAMMPS code on the HPCx system 6
   7.1 Timings output from the code ....................................... 7
   7.2 Communications via MPITrace ...................................... 9
   7.3 Communications via Vampir ........................................ 12

8 Conclusions 14

9 Appendix 16
   9.1 Installing LAMMPS on HPCx .......................................... 16
   9.2 Compilers and flags on HPCx ...................................... 17
   9.3 Batch system settings on HPCx ................................... 17
   9.4 Installing LAMMPS on CSAR ....................................... 17
   9.5 Compilers and flags on CSAR .................................... 18
   9.6 Fixing the deadlock problem .................................... 18
1 Introduction

Molecular dynamics (MD) methods are an established mechanism for determining the equilibrium and transport properties of a classical many-body system. Today, molecular dynamics techniques are being used to simulate ever larger and more complex biological systems, in areas such as disease, vaccine and drug research. This places increasing demands on high performance computing (HPC) resources. A vast range of molecular dynamics codes have been developed to meet these goals. One such code is LAMMPS ([3], [5]).

In this paper we investigate the performance of the LAMMPS code on two systems; an IBM p690+ cluster and an SGI Altix 3700. The performance of the code is measured via a number of benchmarks - some were supplied with the code, others were supplied by user groups. The scaling of the LAMMPS code is then investigated using a variety of profiling tools.

2 Computer systems

For this investigation we have used the two main national HPC systems currently available to British academia. A description of each system now follows:

2.1 IBM p690+ Cluster - HPCx

The IBM p690+ system (subsequently referred to as HPCx) consists of 50 IBM pSeries p690+ Regatta nodes, containing 32 1.7 GHz POWER4+ processors (a total of 1600 processors). Each frame operates as a 32-way SMP node, with a total main memory of 32 GB shared between all 32 processors. The frames of the HPCx system are connected via IBM’s High Performance Switch (HPS). The system has a zero message size latency of 6 $\mu$s. The bandwidth between frames is 4.5 GB/s and the intra-frame bandwidth is 8 GB/s. Each frame runs its own copy of the AIX operating system. HPCx is currently running AIX 5.2.

2.2 SGI Altix 3700 - CSAR

The SGI Altix 3700 system (subsequently referred to as CSAR) consists of 512 Itanium 2 processors, 384 of which have a clock speed of 1.3 GHz and 128 of which have a clock speed of 1.5 GHz. The 1.3 GHz and 1.5 GHz processors have a peak performances of 5.2 GigaFlops and 6 GigaFlops respectively. The system has a theoretical peak performance of 2.7 TeraFlops (based on 384 processors x 5.2 GigaFlops + 128 processors x 6 GigaFlops). At the time of writing the system is configured as two separate partitions, each consisting of 256 processors and 512 GB of memory. This partitioning means that the maximum job size is presently 250 processors. The machine uses SGI’s NUMAlink interconnect. The zero message latency is reported to be 3-5 $\mu$s and the aggregate bandwidth is reported to be 6.4 GB/s per brick (4 CPU’s). Each partition runs its own copy of the Linux kernel with SGI Propack extensions (based on Redhat 7.2).
3 LAMMPS code - introduction

The Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) is a freely available classical molecular dynamics code. LAMMPS can be used to model an ensemble of particles in a liquid, solid or gaseous state. It can model atomic, polymeric, biological, metallic or granular systems. The code has been designed to run well on parallel machines but also runs on single-processor machines.

Three different versions of the LAMMPS code have been tested: LAMMPS 2001; LAMMPS 2004; LAMMPS 2005. LAMMPS 2001 is a Fortran 90 code. LAMMPS 2004 is a C++ code. LAMMPS 2005 is an update to LAMMPS 2004. A number of new features have been added to LAMMPS 2005, please see the 17 Jan 2005 entry at www.cs.sandia.gov/sjplimp/lammps/bug.html for full details. The most significant addition to the 2005 version is the inclusion of pre-compute tables for the pairwise portion of long-range coulombic solvers. These pre-compute tables can reduce the computational cost by up to a factor of 2 (relative to the 2004 version). LAMMPS 2004 and 2005 have many similar features to earlier versions of the code, however the names and syntax of many of the commands have changed meaning the input files are not backwards compatible. All versions of the code considered by this paper are parallelised using a domain decomposition approach.

Instructions for installing and compiling LAMMPS on HPCx and CSAR can be found in the Appendix. Environment settings used with the batch system on HPCx can also be found in the Appendix.

4 LAMMPS performance - including description of the benchmarks

The performance of LAMMPS has been measured using a number of benchmarks. The benchmarks consist both of user supplied (i.e. real research problems) and application supplied (i.e. those supplied with the LAMMPS codes) systems. The main difference between the user and application benchmarks is that the user benchmarks include disk I/O. These molecular dynamics simulations are typically intended to run for many thousands of time steps. Therefore the system will checkpoint every few hundred steps in order to avoid losing valuable work should the system become unstable or a hardware problem develop. Checkpointing typically involves writing out atom positions, energies and restart files such that the simulation can be restarted if necessary.

The user supplied benchmarks consist of four clay-polymer nanocomposite systems ranging from 31760 to 1016320 atoms [1]. To date, the clay-polymer systems have only been run on LAMMPS 2001, however it is our intention to run these benchmarks on LAMMPS 2005 when the input files become available. The application supplied benchmarks consist of a 32000 and a 128000 atom rhodopsin system. The system consists of a rhodopsin protein solvated in a lipid bilayer with a CHARMM [2] force field applied. The 128000 atom rhodopsin system is generated by scaling up (replicating) the 32000 atom system. The replication factors used to generate the 128000 atom system are 2, 2, 1 in the X, Y and Z directions respectively. Unfortunately we do not have directly comparable systems for LAMMPS 2001 and 2004/05 making comparing different versions of the code problematic. Table 1 gives a summary of the benchmark systems tested.
Table 1: Summary of systems used to benchmark the LAMMPS codes.

<table>
<thead>
<tr>
<th>Benchmark system</th>
<th>LAMMPS version</th>
<th>Number of atoms</th>
<th>I/O</th>
<th>HPCx</th>
<th>CSAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clay-polymer 32K</td>
<td>2001</td>
<td>31760</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Clay-polymer 125K</td>
<td>2001</td>
<td>127040</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Clay-polymer 250K</td>
<td>2001</td>
<td>254080</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Clay-polymer 1000K</td>
<td>2001</td>
<td>1016320</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Rhodopsin</td>
<td>2004</td>
<td>32000</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Rhodopsin</td>
<td>2004</td>
<td>128000</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Rhodopsin</td>
<td>2005</td>
<td>32000</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Rhodopsin</td>
<td>2005</td>
<td>128000</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 1 shows the performance of LAMMPS 2001 measured on the HPCx and CSAR systems for the clay-polymer systems. Results are shown for all four benchmarks on both HPCx and CSAR. The performance is measured in steps per second to allow different system sizes to be compared on the same plot. The timings are taken from the “loop time” reported by the code which represents the total time spent in the main MD loop. To ensure that any start up costs or instabilities were removed from the timings, two runs of each benchmark were performed; one for 250 time steps and one for 500 time steps and the resulting times subtracted.

Figure 1: Performance of LAMMPS 2001 on the HPCx and CSAR systems for the clay-polymer nanocomposite systems. System sizes range from 32K atoms to 1000K atoms.

Figure 2 shows the performance of LAMMPS 2004 and 2005 measured on HPCx and CSAR for the rhodopsin systems. Results for LAMMPS 2004 are only available for the
HPCx system. Results for LAMMPS 2005 are available for both HPCx and CSAR. As with the clay-polymer systems the performance is measured in steps per second. The timings are taken from the “loop time” reported by the code and the benchmarks were run for 50 and 100 steps to remove any start up costs. It turns out that the start up costs associated with the rhodopsin benchmarks are are almost negligible as input data file is very small (~6MB). This means that the cost per steps for a 100 step run is identical to the cost per step for a 100-50 step run.

![Performance Graph](image)

Figure 2: Performance of LAMMPS 2004/05 on the HPCx and CSAR systems for application supplied rhodopsin benchmark. The 128000 atom system has been created by replicating the 32000 atom system 2x2x1 times.

Figures 3 and 4 show the speed up of the clay-polymer and rhodopsin systems respectively. For the clay-polymer systems (figure 3) it was not possible to run the largest benchmarks on less than 32 processors due to the large runtimes involved, therefore, the speed ups are computed relative to the 32 processor loop times. For ease of comparison the speed ups for the rhodopsin systems are also given relative to the 32 processor times.

5 LAMMPS performance - discussion

From figures 1 and 2 we can see that with the exception of the 32000 atom rhodopsin benchmark the LAMMPS execution time continues to reduce up to 1024 processors on HPCx and 240 on CSAR. For the 32000 atom rhodopsin system the loop time increases from 512 to 1024 processors. This is likely to be due to the small problem size and lack of I/O. As expected, as the number of atoms increases the number of steps computed per second decreases. For both the clay-polymer and rhodopsin systems the performance of LAMMPS 2005 on the HPCx system is greater than that of the CSAR system. The
Figure 3: Speed up of LAMMPS 2001 (relative to 32 processors) on HPCx and CSAR for clay-polymer nanocomposite systems. Ideal scaling (relative to 32 processors) is denoted by the dashed line.

Figure 4: Speed up (relative to 32 processors) of LAMMPS 2004/05 on HPCx and CSAR for application supplied rhodopsin benchmark. Ideal scaling (relative to 32 processors) is denoted by the dashed line.
ratio in clock speeds is between HPCx and CSAR is 1.3 however the performance ratio is as much as 1.5.

From figure 3 we can see that the speed up of LAMMPS generally increases with the number of atoms. The speed ups obtained on the CSAR system tend to be slightly higher than those obtained on HPCx. For the largest benchmark (1000K system) the CSAR system exhibits super-linear scaling for 32 and 64 processors. We believe this to be a cache related effect.

From figure 4 we can see that for more than 256 processors the speed up of LAMMPS 2004 is higher than LAMMPS 2005 for the same benchmark (compare open and solid symbols). The runtimes obtained from LAMMPS 2004 were up to twice as slow as for LAMMPS 2005 due to the addition of the pre-compute tables in the 2005 version. Comparing the speed ups obtained on HPCx and CSAR (solid and square shaded symbols) we see that the CSAR system shows slightly poorer speed ups than HPCx, particularly for more than 128 processors.

6 Reproducibility

All the results presented in this report have been run on a production system, i.e. one in which other user codes are running concurrently. On the HPCx system each job has exclusive access to the nodes which it runs on and does not share memory or processors with anyone else. However, each node has its own copy of the operating system which may interfere with the performance of the code. All I/O (read and write) on the system occurs between two separate I/O nodes. These I/O nodes are shared between all users of the system and therefore can influence the performance of a code.

On the CSAR system each job has access to an exclusive number of processors and their associated memory (~2 GB memory is available locally per processor). As the CSAR system comprises two 256 processor partitions each running their own copy of the operating system we cannot guarantee that other users actions do not interfere with our runs. As with HPCx file I/O is shared between all users of the system.

The benchmark results presented here have been run multiple times, with the lowest times reported. For low processor counts (≤256) the timings were generally very stable. On HPCx, for 512 and 1024 processor runs some variability was observed between the runtimes. It is thought that this variability may be I/O related as LAMMPS regularly writes out large (up to 600 MB) position and restart files several times per run. If another user is also writing/reading large files then it is possible that an I/O bottleneck could result.

7 Profiling the LAMMPS code on the HPCx system

Benchmarking of the 128000 atom rhodopsin system on LAMMPS 2005 has resulted in the code being awarded a Bronze Capability Incentive on HPCx [4]. The scaling of LAMMPS (all versions) is generally good up to 256 processors but drops off beyond 256 processors. By profiling the code we aim to understand the reasons for the poor scaling beyond 256 processors and to suggest ways of improving the scaling.

Our profiling will concentrate on the 2001 version of the code with results from the 2004/05 versions included where relevant. The reason for concentrating on the 2001
version is that we have a realistic user benchmark to work with rather than an application supplied benchmark. We plan to run the clay-polymer systems through the 2005 code when the input files become available.

To investigate the performance of LAMMPS 2001 several different methods and tools have been utilised: timings output directly by the code, MPITrace [6] and Vampir [7].

### 7.1 Timings output from the code

The LAMMPS code outputs timings from the main routines. Using these timings we can plot a stacked bar chart showing how the time spent in each section of the code varies with processor count. Figure 5 shows such a chart for the 125K atom clay-polymer benchmark run on LAMMPS 2001. The time spent in each section of the code is given as a percentage of the total loop time. The timings reported by LAMMPS include estimates of the time spent in for example, pairwise calculations, bond calculations, I/O and communications. Table 2 gives a summary of the timings reported by LAMMPS 2001 and what they represent.

![Figure 5: Timing breakdown for clay-polymer nanocomposite benchmark on LAMMPS 2001.](image)

The I/O time as reported by LAMMPS gives the total time associated with writing out any of the following files: restart, atom positions, atom velocities and atom forces. The time reported includes both the actual file I/O time (i.e. time spent writing to disk) and the time associated with gathering the relevant data back from all processors (i.e. `MPI_Allreduce`, and send and receive calls).

From figure 5 we can see that as processor count increases the percentage of time spent in I/O, Fcomm, Comm, Long and Other begins to dominate. As Fcomm, Comm, Long
and I/O all involve \texttt{MPI\_allreduce} and \texttt{MPI\_Irecv/MPI\_Send} calls this is not unexpected. Figure 6 shows the absolute timings (in seconds) output from LAMMPS 2001 plotted against processor count. It is clear that the time spent in I/O, and Comm remains constant regardless of the processor count. The time spent in Long decreases as the number of processors rises to 256, but increases above 512 processors. The increase after 512 processors could be explained by the eager limit - at this point the eager limit is reduced and so messages above 32K have to be sent synchronously. The time spent in Fcomm is also interesting.

We have also looked at the timings output by the 2004 and 2005 versions of LAMMPS. Table 3 gives a summary of the timings reported by the 2004/05 codes and what they represent. Figure 7 shows the absolute timings (in seconds) output by LAMMPS 2004 and 2005 plotted against processor count for the 128000 atom rhodopsin benchmark.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Total time spent in minimize</td>
</tr>
<tr>
<td>Nbond</td>
<td>Time spent computing forces due to long range interactions</td>
</tr>
<tr>
<td>Long</td>
<td>Time spent in either Ewald or PPPM computations</td>
</tr>
<tr>
<td>Bond</td>
<td>Time spent computing forces due to covalent bonds</td>
</tr>
<tr>
<td>Angle</td>
<td>Time spent computing forces due to angles</td>
</tr>
<tr>
<td>Dihed</td>
<td>Time spent computing forces due to dihedral angles</td>
</tr>
<tr>
<td>Impro</td>
<td>Time spent computing forces due to improper angles</td>
</tr>
<tr>
<td>Nay-1</td>
<td>Time spent computing new neighbour lists</td>
</tr>
<tr>
<td>Nay-2</td>
<td>Time spent verifying new neighbour lists</td>
</tr>
<tr>
<td>Exch</td>
<td>Communication time for shifting atoms to new processors</td>
</tr>
<tr>
<td>Comm</td>
<td>Communication time for updating atom positions</td>
</tr>
<tr>
<td>Fcomm</td>
<td>Communication time for updating force calculations</td>
</tr>
<tr>
<td>I/O</td>
<td>Time to output restart, atom position, velocity and force files</td>
</tr>
<tr>
<td>Other</td>
<td>Difference between loop time and all other times listed in table</td>
</tr>
</tbody>
</table>

Table 2: Description of timings output directly by LAMMPS 2001 code.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Total time spent in minimize</td>
</tr>
<tr>
<td>Pair</td>
<td>Time spent computing pairwise interactions; includes VDW, LJ and short-range coulombic interactions</td>
</tr>
<tr>
<td>Bond</td>
<td>Time spent computing forces due to covalent bonds</td>
</tr>
<tr>
<td>Kspace</td>
<td>Time spent computing long-range coulombic interactions e.g. Ewald, PPPM</td>
</tr>
<tr>
<td>Neigh</td>
<td>Time spent computing new neighbour lists</td>
</tr>
<tr>
<td>Comm</td>
<td>Time spent in communications</td>
</tr>
<tr>
<td>Outpt</td>
<td>Time to output restart, atom position, velocity and force files</td>
</tr>
<tr>
<td>Other</td>
<td>Difference between loop time and all other times listed in table</td>
</tr>
</tbody>
</table>

Table 3: Description of timings output directly by LAMMPS 2004 and 2005 codes.
Figure 6: Timing breakdown for 125K atom clay-polymer benchmark on LAMMPS 2001.

Solid lines correspond to timings output by LAMMPS 2004 whereas dashed lines correspond to timings output by LAMMPS 2005. Examining figure 7 it is clear that the most significant difference between the two versions of the code is the amount of time spent in Pair time, which is up to a factor of two less for the 2005 code. The reason for the reduction in Pair time is the inclusion of pre-compute tables for pairwise interactions in the 2005 version of the code. The inclusion of the pre-compute table translates directly into reducing the amount of time the LAMMPS code spends computing pairwise interactions. The time spent in all other sections of the code should be largely unaffected and figure 7 confirms that the time spent in Bond, Kspace, Neigh, Comm, Outpt and Other are nearly identical for both LAMMPS 2004 and 2005.

7.2 Communications via MPITrace

We have investigated the amount of time that LAMMPS 2001 spends in communications using MPITrace. MPITrace is an IBM tool which provides details of the amount of time spent in communication routines. It reports the total time spent in MPI routines by each process, the number of times a routine is called and the average message size for that routine. Figure 8 shows the average amount of time spent in communications against processor count for the 125K atom clay-polymer system run on LAMMPS 2001 for 500 times steps. From figure 8 it is clear that communication plays a significant role in the performance of LAMMPS 2001 especially for large processor counts. The majority of the communication time is spent within MPI_Bcast, MPI_Wait, MPI_Barrier, MPI_Send and MPI_Send_ext. For more than 512 processors the time spent in MPI_Allreduce also becomes significant. Calls to MPI_Send and MPI_Send_ext represent MPI_Send calls.
Figure 7: Timing breakdown for rhodopsin benchmark (128,000 atoms) on LAMMPS 2004 (solid lines) and 2005 (dashed lines).
taking place within and outwith a 32 processor frame respectively.

To investigate the effects of initialisation costs we have also run MPITrace for 500 and 250 time steps and subtracted the resulting times. Table 4 gives the average amount of time spent in each MPI call for 500 and 250 steps along with the difference. From the last column of table 4 we can see that calls to MPI_Wait, MPI_Send_ext, MPI_Barrier and MPI_Send dominate. Interestingly, the time spent in both MPI_Bcast and MPI_Allreduce seems to be attributed to either initialisation or file. I/O. Examination of the source code confirms that calls to MPI_Bcast are made by the following source codes: input.f, initialise.f, read_data.f and read_restart.f all of which are associated with initialisation. Calls to MPI_Allreduce are scattered throughout the code with the vast majority relating to the collection of data from all processes at the end of each time step or prior to outputting data to file.

Table 4 suggests that the majority of the communication time (that which cannot be associated with initialisation) is spent within the following MPI calls: MPI_Wait, MPI_Send_ext, MPI_Barrier and MPI_Send. Calls to MPI_Wait and MPI_Send are associated with file I/O and communications between processes at the end of each time step. The source files which contain calls to MPI_Wait and MPI_Send are: communicate.f, dump_atom.f, dump_force.f, dump_vel.f, pppm_coeffs.f, pppm_remap.f, shake.f, special.f and write_restart.f. Calls to MPI_Send_ext are associated with MPI_Send and correspond to communications taking place outside a 32 processor frame. Calls to MPI_Barrier are made from a large number of source files including I/O, communication and computation routines.

A large number of subroutines begin with a call to MPI_Barrier or contain calls

Figure 8: Average time spent in MPI communications for 125K atom clay-polymer nanocomposite benchmark on LAMMPS 2001.
<table>
<thead>
<tr>
<th>MPI call</th>
<th>Time (seconds)</th>
<th>500 steps</th>
<th>250 steps</th>
<th>500-250 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allgather</td>
<td>0.0050</td>
<td>0.0110</td>
<td>-0.0060</td>
<td></td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>2.7692</td>
<td>2.4588</td>
<td>0.3104</td>
<td></td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>3.9343</td>
<td>3.1233</td>
<td>0.8110</td>
<td></td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>8.9995</td>
<td>9.0089</td>
<td>-0.0094</td>
<td></td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>0.2662</td>
<td>0.1402</td>
<td>0.1260</td>
<td></td>
</tr>
<tr>
<td>MPI_Irecv_ext</td>
<td>0.0610</td>
<td>0.0315</td>
<td>0.0295</td>
<td></td>
</tr>
<tr>
<td>MPI_Send</td>
<td>1.1943</td>
<td>0.6306</td>
<td>0.5636</td>
<td></td>
</tr>
<tr>
<td>MPI_Send_ext</td>
<td>6.4568</td>
<td>4.1834</td>
<td>2.2733</td>
<td></td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>6.2594</td>
<td>3.8258</td>
<td>2.4336</td>
<td></td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>0.0349</td>
<td>0.0242</td>
<td>0.0107</td>
<td></td>
</tr>
<tr>
<td>MPI_Waitany</td>
<td>1.4577</td>
<td>0.9437</td>
<td>0.5139</td>
<td></td>
</tr>
<tr>
<td>Total communication time</td>
<td>31.4381</td>
<td>24.3814</td>
<td>7.0567</td>
<td></td>
</tr>
<tr>
<td>Total elapsed time</td>
<td>45.4614</td>
<td>31.7866</td>
<td>13.6749</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Average timings output by MPITrace for a 512 processor run of the 125K clay-polymer benchmark on LAMMPS 2001.

to MPI_Barrier throughout which we believe to be unnecessary. Calls to MPI_Barrier shouldn’t really be required in an MPI code as the MPI should handle the necessary synchronisation. In an attempt to reduce the amount of time spent in MPI communications all 48 calls to MPI_Barrier were commented out of the source code. It was found the code ran successfully without the barriers producing identical scientific results. Unfortunately, the effect on the total runtime or time spent in communications was minimal. Removing the MPI_Barrier calls simply resulted in the time being spent in MPI_Allreduce instead. For example, for a 500 step run performed on 512 processors the code spent 2.77 seconds in MPI_Allreduce and 3.93 seconds in MPI_Barrier. The corresponding times for the code with the barriers removed were 6.97 seconds for MPI_Allreduce and 0.0 seconds for MPI_Barrier.

7.3 Communications via Vampir

We can use the Vampir tool to gain a better understanding of the communication patterns taking place inside LAMMPS 2001. Vampir is a commercial post-mortem trace visualisation tool that can handle both MPI and non MPI (user code) calls. Only MPI calls are handled in any detail. Figure 9 shows a graphical representation of the main communications occurring within LAMMPS on 16 processors for a 10 step run. Lines between processes represent MPI communications between processes. The lines have been coloured according to the type of communication taking place. Yellow lines correspond to MPI_Bcast, orange lines to MPI_Allreduce, blue lines to MPI_Barrier, purple lines to all other collective communications and black lines to all point to point communications. Red blocks represent time spent in MPI calls, whereas green blocks represent time spent in computation (labelled User_Code). The code can be broken down into several
sections:

1. 0-11.5 seconds: initialisation, data read in from file, data distributed to all processors

2. 9-11.5 seconds: time spent in `MPI_Send`, `MPI_Barrier` and `MPI_Allreduce`.

3. 11.5-22.5 seconds: Main molecular dynamics computation. This trace is for 10 times steps with two blocks of MPI communications per time step. Communication between processors is required at every time step to update energies, forces, atom positions etc. Normally this stage would also include file I/O.

4. 22.5-28.0 seconds: Gather all results, timings then output to file/screen etc.

![Graphical representation of the communication patterns in LAMMPS 2001 for a 10 step run of the 125K clay-polymer benchmark on 16 processors.](image)

Figure 9: Graphical representation of the communication patterns in LAMMPS 2001 for a 10 step run of the 125K clay-polymer benchmark on 16 processors.

We should point out that for a typical MD simulation the time spent in the initialisation and finalise stages (steps 1. and 4. above) would be a small part of the total runtime - the example shown here has been chosen to illustrate the communication pattern. From figure 9 we can see that the first 8.5 seconds are spent broadcasting data from the master to all other processors. During this initial 8.5 seconds the input data are read in on process 0 and therefore all other processes have to wait until process 0 has read in the data before the broadcast operation can complete. This confirms that all the `MPI_Bcast` calls are associated with the initial distribution of data to processors.

We now concentrate on the communications taking place at each time step of the MD simulation, i.e. those occurring between 11.5-22.5 seconds on figure 9. As the number
of processors increases the amount of time each time step spends within \texttt{MPI\_Send} and \texttt{MPI\_Wait} starts to dominate. For a 16 processor run 3\% of the total runtime is spent in point to point communications (\texttt{MPI\_Send, MPI\_Irecv, MPI\_Wait}). This increases to 16\% and 31\% for 128 and 512 processors respectively. The numbers quoted here are taken from a 500 step run of the 125K atom clay-polymer benchmark. Now consider figure 10 which shows the communications taking place for a single time step of a 500 step run performed on 128 processors. Only data from the first 47 processors are shown however the pattern is similar for the remaining processors. It seems that a significant amount of time is spent in either \texttt{MPI\_Send} or \texttt{MPI\_Wait} at each time step while each process waits for the incoming data to arrive. A truly huge number of messages are sent between 12.286-12.92 seconds but these do not seem to impact on the total runtime.

![Figure 10](image)

Figure 10: Graphical representation of the communication patterns in LAMMPS 2001 for a 500 step run of the 125K clay-polymer benchmark on 128 processors. Plot shows the communications taking place for a single time step only.

8 Conclusions

We have presented a benchmarking and profiling study of the LAMMPS molecular dynamics code. Even accounting for the difference in clock speeds the performance (measured in number of steps computed per second) of LAMMPS (all versions) is found to be higher on the HPCx system than the CSAR system for comparable processor counts. The LAMMPS code scales well to 1024 processors with a drop off in speed up occurring beyond 256 processors. We have profiled the 2001 version of code to understand the poor speed up for large processor counts. The main reason for the poor speed up is found to
be the amount of time spent in point to point communications, particularly \texttt{MPI\_Send}
and \texttt{MPI\_Wait}. As the code needs to communicate between all processors at every time
step is it difficult to envisage a means of reducing this bottleneck. In reality the actual
time spent in point to point communications seems reasonable. It could simply be that
the problem size has reduced such that the amount of computation each process has to
do before communicating is too small.

\textbf{Acknowledgements}

We would like to acknowledge the following for their support and assistance: Peter
Coveney (UCL), Chris Greenwell (UCL) and Steve Plimpton (Sandia National Labora-
tories).

\textbf{References}

[1] The clay-polymer nanocomposite benchmarks were supplied by Dr H. C. Greenwell at
the Centre for Computational Science, Department of Chemistry, University College
London. \url{http://www.chem.ucl.ac.uk/ccs/}

Field, M.J., Fischer, S., Gao, J., Guo, H., Ha, S., Joseph-McCarthy, D., Kuchnir,
L., Kuczera, K., Lau, F.T.K., Mattos, C., Michnick, S., Ngo, T., Nguyen, D.T.,
Prodhom, B., Reiher, III, W.E., Roux, B., Schlenkrich, M., Smith, J.C., Stote, R.,
\textit{All-atom empirical potential for molecular modeling and dynamics studies of proteins},


[4] \textit{Capability Incentives on HPCx},
\url{http://www.hpcx.ac.uk/services/policies/capability.html}


[6] \textit{MPITrace documentation}
\url{http://www.hpcx.ac.uk/support/documentation/IBMdocuments/mpitrace}

[7] \textit{Visualisation and Analysis of MPI Resources (Vampir)}, Intel GmbH, Software
\url{http://www.pallas.com/e/products/vampir}
9 Appendix

9.1 Installing LAMMPS on HPCx

The LAMMPS 2004 and 2005 codes were installed with minimal alteration to source code and Makefiles. The test examples and benchmarks supplied with the code all ran successfully.

The LAMMPS 2001 code was more problematic to install. A variety of changes were required in order to compile the code:

1. The IBM p690+ system has a Fortran routine called abort() so does LAMMPS 2001. The IBM routine does not accept any arguments so cannot be used in place of the LAMMPS routine. The solution is to rename the abort subroutine in error.f to abortx

2. Change "??" on line 1060 of minimize.f to "QM" as only alpha-numeric values are acceptable on the IBM.

3. Change fftw.h to dfftw.h in fftd3.c.

4. Add system_clock() to mpi_fortran.f if the STUBS library (used for the serial version of LAMMPS) is used as etime() does not exist on HPCx.

These issues have been submitted to the code developers. As a result items 1. and 2. will be changed in subsequent versions of LAMMPS. Items 3. and 4. are specific to the IBM system and therefore will not be changed.

Initially the code appeared to install successfully and many of the test examples supplied with the code ran without error. However, the code crashed with a segmentation violation or malloc error as soon as any attempt to use PPPM (particle-particle particle mesh) was made. Extensive debugging using Totalview revealed that the problem related to the calling of C functions from Fortran and the naming conventions used. Unlike other compilers, the IBM compilers do not give the external C functions underscores. For example, when the fftd3 function is called from the Fortran code rather than calling the fftd3 Fortran wrapper function the fftd3 function is called instead. Although the variable types and sizes are not known the code continued without crashing. The crash would then occur within FFTW - usually because an attempt to malloc an undefined variable is made. As the crash occurred a considerable distance from the original error this made it particularly difficult to detect.

The initial solution to this problem was to rename all relevant functions within the Fortran subroutines so that the extra underscore was included, e.g. rename fftd3 to fftd3, etc. This renaming allowed a fully functional version of the code to be compiled and tested. The problem examples and benchmarks then ran successfully.

Renaming the functions in this way is not a desirable solution as it limits the portability of the code. After extensive communication with the code developers a new version of the fftd3.c has been written. To ensure the correct functions are picked up and to remove the problems caused by having near identical function wrapper and function names the C functions are renamed as follows: fftd3() is replaced with cfftd3(). A combination of #ifdef and #define statements are then used to ensure that the correct wrapper function is called from the Fortran routines. These changes allow LAMMPS
2001 to be compiled and executed on an IBM p690+ with the addition of a \texttt{-DIBM\_KLUDGE} to \texttt{CCFLAGS} in the Makefile. The new version of \texttt{fft\_3d\_c} has been incorporated into the latest release of the 2001 code.

### 9.2 Compilers and flags on HPCx

The 2001 code was compiled using XL Fortran for AIX, Version 8.1.1.7 and C for AIX, version 6.0.0.0. The following flags were used on the HPCx system:

\begin{verbatim}
F90 = mpxlf90_r
F90FLAGS = -q64 -O3 -qarch=pwr4 -qtune=pwr4 -qfixed
CC = mpcc_r
CCFLAGS = -O3 -qarch=pwr4 -qtune=pwr4 -DIBM\_KLUDGE \ 
         -I/usr/local/packages/fftw/include -DFFT\_FFT
LINK = mpxlf90_r
LINKFLAGS = $(F90FLAGS) -O3 -q64 -v -L/usr/local/packages/fftw/lib
USRLIB = -ldfftw
SYSLIB = -lm
\end{verbatim}

The 2004/05 codes were compiled using VisualAge C++ for AIX, Version 6.0.0.0. The following flags were used:

\begin{verbatim}
CC = mpCC_r
CCFLAGS = -q64 -O3 -qarch=pwr4 -qtune=pwr4 \ 
         -I/usr/local/packages/fftw/include -DFFT\_FFT
DEPFLAGS = -M
LINK = mpCC_r
LINKFLAGS = -q64 -O3 -qarch=pwr4 -qtune=pwr4 \ 
          -L/usr/local/packages/fftw/lib
USRLIB = -ldfftw
SYSLIB = -lm
\end{verbatim}

### 9.3 Batch system settings on HPCx

The following environment variables were set within the batch script:

\begin{verbatim}
export MP\_EAGER\_LIMIT=65536
export MP\_SHARED\_MEMORY=yes
export MP\_USE\_BULK\_XFER=yes
export MEMORY\_AFFINITY=MCM
export MP\_STDINMODE=all
\end{verbatim}

### 9.4 Installing LAMMPS on CSAR

As with the HPCx system, LAMMPS 2005 was installed on CSAR with minimal alteration to the original source code. The MPI bug fixes described in section 9.6 were applied to the code.

For the LAMMPS 2001 installation we used the source code from the HPCx system (with changes detailed in section 9.1) with the additional modification:
1. Changed `zomplex` to `complex` on line 172 of `fft_3d.c`

The naming conventions used for cross-calling Fortran and C on the CSAR system did not give rise to any problems.

### 9.5 Compilers and flags on CSAR

The 2001 code was compiled using Intel Fortran Compiler for Linux, Version 8.1.023 and Intel C++ Compiler for Linux, Version 8.1.023 The following flags were used on the CSAR system:

- **F90** = `ifort`
- **F90FLAGS** = `-O3 -v`
- **CC** = `icc -v`
- **CCFLAGS** = `-O3 -DFFT_FFTW -I/usr/local/apps/unsupported/fftw/include`
- **LINK** = `ifort -L/usr/local/apps/unsupported/fftw/lib`
- **LINKFLAGS** = `-O3 -v`
- **USRLIB** = `-lm`
- **SYSLIB** = `-lmpi -lfftw`
- **SIZE** = `size`

The 2005 code was compiled using the Intel C++ Compiler for Linux, Version 8.1.023. The following flags were used:

- **CC** = `icc`
- **CCFLAGS** = `-O2 -DFFT_SCSL -w`
- **DEPFLAGS** = `-M`
- **LINK** = `icc`
- **LINKFLAGS** = `-O2`
- **USRLIB** = `-lm`
- **SYSLIB** = `-lmpi -lscs_mp`
- **SIZE** = `size`

### 9.6 Fixing the deadlock problem

Initial benchmarking of the LAMMPS codes (2001, 2004 and 2005) revealed that the code deadlocked for processor counts greater than or equal to 512. The deadlock occurred regardless of the benchmark system tested. Reducing the eager limit on HPCx to zero resulted in the code deadlocking on any number of processors suggesting that the problem lay in one or more of the send/receive calls.

To investigate the cause of the deadlock the eager limit was set to zero (via the environment variable `MP_EAGER_LIMIT=0`). This ensures all communications are unbuffered (i.e. synchronous). LAMMPS was then executed inside Totalview using 4 processors. Unfortunately running the code inside the debugger destroys the timing so that the deadlock no longer occurs and the code runs successfully. Write (either `write` or `cout`) statements were placed before and after the `MPI_Send` and `MPI_Recv` calls in order to determine the point at which deadlock occurred. To ensure the output appeared in the correct order the I/O buffers were flushed after each write statement (using `call`
flush(6) or cout << flush). This resulted in a number of MPI_Send/MPI_Recv combinations being identified as potential sources of deadlock. Various source files were affected by this problem both in LAMMPS 2001 and LAMMPS 2004/2005.

A solution to the deadlock problem was to replace the original MPI_Send/MPI_Recv combinations with MPI_Irecv/MPI_Send and an additional MPI_Wait. Several of the MPI_Recv/MPI_Rsend combinations were also replaced with MPI_Irecv/MPI_Send and an additional MPI_Wait. The MPI_Recv/MPI_Rsend combinations did not appear to cause deadlock however, MPI_Rsend should generally be avoided as careful synchronisation is needed for it to work correctly. Also, if no corresponding receive exists then the message will be lost.

After making these changes LAMMPS (all versions) ran successfully on any processor count. The changes to the MPI have had no effect on either the results obtained or the runtimes of any versions of the LAMMPS code. These changes have been passed on to the code developers and we are currently awaiting their response.