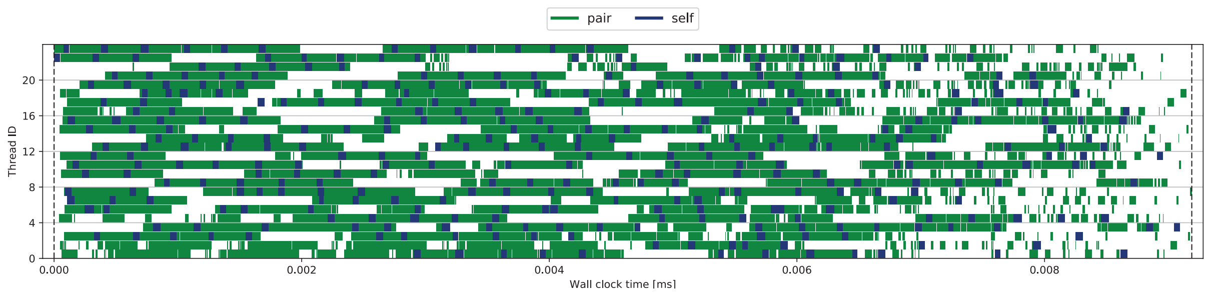
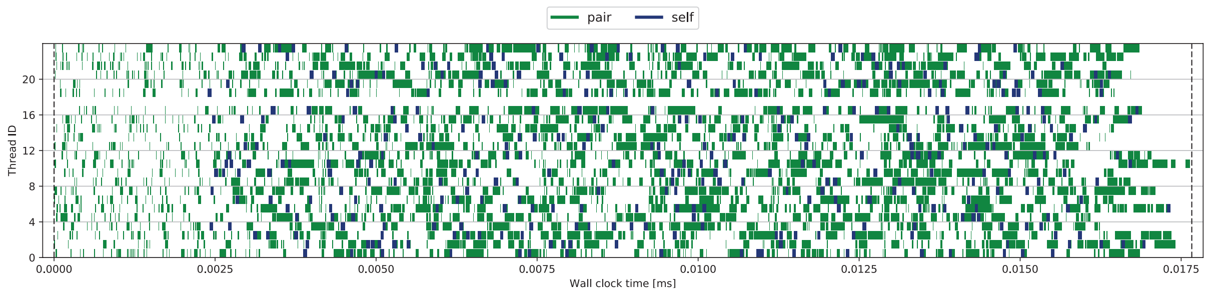
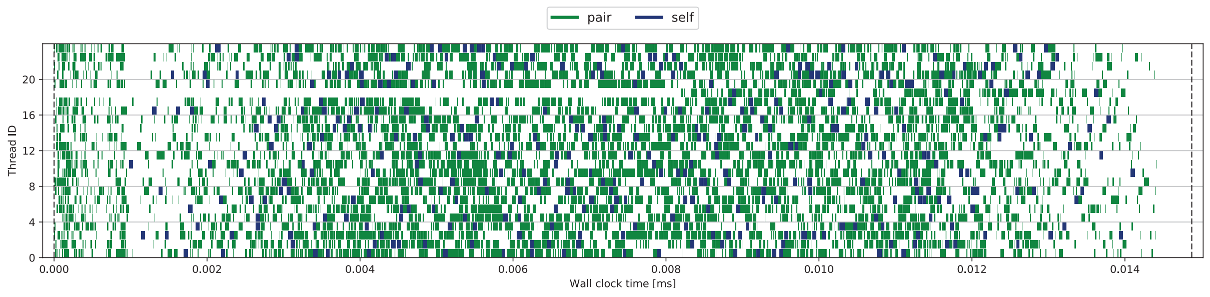
Strategic Theme 1

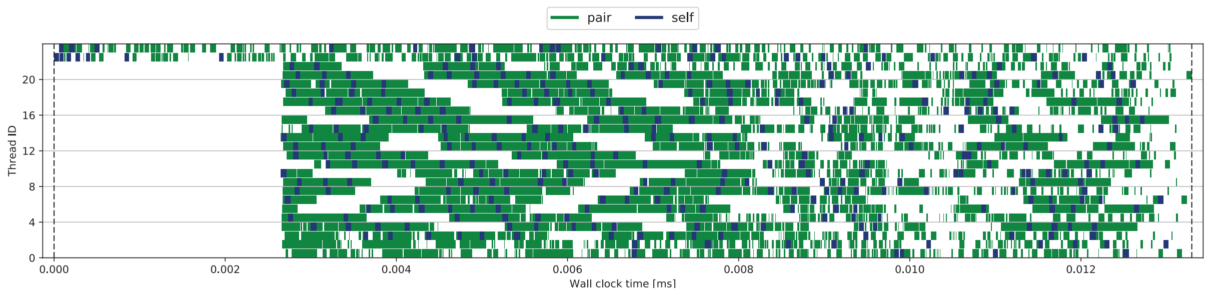
SCD will build and strengthen our existing capability of providing enabling computational expertise and e-infrastructure to support STFC in fulfilling its strategic goals: “world class research, world class innovation, and world class skills.”

Task Based Parallelism with OpenMP: A Case Study with DL_POLY_4









Our scientists are developing ways to speed-up codes and produce faster and more efficient processing. One of these is Task-Based Parallelism, a method for passing data between processes in which a program is split into a series of tasks, which are then assigned to multiple cores (processing units) all working in parallel until the calculations are completed. This method reduces the amount of computing time needed, and increases the time scientists can spend on research.

Introduction

When performing computations where load balancing is complex, dynamic load balancing is becoming increasingly necessary. In this paper we examine one of these methods, Task-Based Parallelism. Many libraries implement Task-Based Parallelism, however in this paper we examine the OpenMP standard and implementations, and apply it to the Classical Molecular Dynamics code, DL_POLY_4, focusing on the two body force calculations that make up a large percentage of the compute in many simulation runs. Our results show reasonable performance using Open MP tasks, however some of the extensions available in other libraries such as OmpSs or StarPU may help with performance for problems similar to Molecular Dynamics, where avoiding race conditions between tasks can have a substantial scheduling overhead.

Task-Based Parallelism is a method for shared memory parallel programming in which a program is split into a series of tasks, which are picked up and executed by a set of cores in parallel until the computation is completed. To avoid concurrency issues, the *dependencies* between tasks need to be considered, i.e. if there are 2 tasks, A and B, where task A produces a result needed to compute task B, we say B is dependent on A, or A *unlocks* B. The tasks and their dependencies form a Directed Acyclic Graph (DAG). Task-Based Parallelism has two major advantages over traditional parallel processing techniques.
Firstly, since the tasks are assigned to the cores dynamically, the work is automatically load balanced. Secondly, the task dependencies avoid the need for explicit synchronisation between the cores. In the OpenMP 3.0, 4.0 and 4.5 specifications, Task-Based Paralellism has been added and extended, allowing a standardised way to use task-based parallelism.

DL_POLY_4 [1] is a classical molecular dynamics code developed at Daresbury Laboratory, containing a large number of time integrators and statistical ensembles. The code is MPI parallelised, and uses a simple domain decomposition to distribute work between the processors. In this paper we show how we used OpenMP tasks inside DL_POLY_4, and show the performance of the resulting code.

OpenMP added explicit tasks in the OpenMP 3.0 specification, and expanded upon it in the 4.0 and 4.5 specifications. As with many OpenMP directives, additional options should be added to the task directive to control the data sharing between threads. Tasks can be executed by any thread from the same (innermost) parallel region as the thread that encounters the task directive (known as the *spawning thread*). Each task's execution is usually (but not always) deferred, but the standard specifies a variety of points at which tasks may be executed, known as task scheduling points. The most important of these for this work are: i) Immediately after generation of an explicit task; ii) Immediately after completion of a task region; iii) When encountering a taskyield directive; iv) Inside any implicit or explicit barrier. When a thread encounters a taskyield directive while inside a task region, the current task may be suspended and another task be executed instead. The depend clause allows the programmer to specify the data requirements of tasks, as either *in, out,* or *inout*. The priority clause allows the programmer to give each task an integer priority, and allows the scheduler to prioritise tasks of high priority.

OpenMP tasks in DL_POLY_4

To use Task-Based Parallelism in a scientific code, the code needs to be able to be broken down into chunks of work which can be executed in parallel. To help achieve this in DL_POLY_4, we rewrote the algorithm that performs the pairwise force loops. DL_POLY_4 uses the linked-cell method [2] [3] to build Verlet lists to compute pairwise interactions. While this method has been used with tasks in [4] it is not expected to give the best performance, as global Verlet lists have been shown to perform poorly on modern architectures [5]. Instead, we use the sorted cell-list algorithm [6], with the aim of improving to the pseudo-Verlet algorithm if successful [7].

The sorted cell-list algorithm involves organising the particles into cells of size equal to or greater than the cutoff radius (r_c) in each dimension. Since DL_POLY_4 divides the domain into equal sized domains with its domain decomposition, we split these domains into equal sized cells. The cells size must be bigger than r_c in each direction but not too large as this will affect performance. The cells are then sorted in each of the axes between the centres of pairs of adjacent cells (in 3 dimensions there are 26 axes). Gonnet et al. in [6] recommends sorting in 13 axes and then reversing the indices for the other 13 axes.

In DL_POLY_4 however, halo cells may have different dimensions to the domain cells, so boundary cells (those adjacent to halo cells) need to be sorted in all 26 axes. The cells can be sorted in parallel, though we did not use OpenMP tasks to parallelise this section of the code (due to issues with dependencies). The particles themselves are not sorted, but the indices of the sorted particle arrays are stored for each direction. Figure 1 shows how the particles are sorted on the cell pair axis.

The traditional neighbour list method can be intensive with respect to memory usage, as the neighbour access may not be cache-friendly. The addition of the link-cell does not help with this in DL_POLY_4, as the cells are just stored using a linked-list, as are the neighbour lists. The sorted-cell lists however involve a reordering of some of each particle's data, such as the position and forces upon the particle, resulting in this information being located adjacent in memory, which may improve cache performance. Additionally, since particles in the same cell are likely to interact with similar particles from an adjacent cell, performing interactions by cell can improve cache reuse.

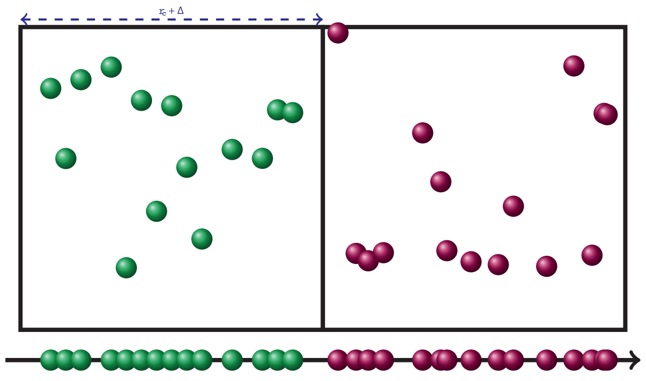


Figure 1: The figure shows two cells and the particles contained within, and the particles shown sorted along the axis between the cells. rc is the cutoff radius between the cells and Δ is a positive constant to enforce that each cell is at least r_c.

As the cells are at least as large as r_c in each dimension, we know all the neighbours of any particle in a cell are contained either in the same cell or an adjacent cell. In DL_POLY_4 this means we have 3 types of tasks: Cellself tasks. These tasks compute the pairwise interactions between all particles within a single cell; Local cell-pair tasks. These tasks compute the pairwise interactions between all particles within a pair of non-halo cells, and update the force on the particles in both cells; Nonlocal cell-pair tasks. These tasks compute the pairwise interactions between all particles within a non-halo cell and a halo cell, and update the force on the particles in the non-halo cell. In addition to specifying the tasks, we need to ensure that we avoid race conditions between tasks. Each task writes to the particle data associated with either one or two cells, so we need to avoid multiple tasks that write to the same cells being executed simultaneously. In OpenMP tasks, this can be done either manually, or automatically using the depend keyword. Avoiding race conditions using the depend keyword is straightforward. We add a depend (inout:cells), where cells is a list of the cell objects written to during each task. For a cell-self or non-local cell-pair task, this will be a single cell, whereas for the local cell-pair tasks it will contain both the cells required for the task. When using dependencies with OpenMP, any tasks that depend must be spawned by the same thread. The depend keyword adds dependencies between any tasks that share a variable (or array section) contained in any dependency clause (subject to certain rules). In our system this means dependencies between any tasks that write to the same cell. The first task to be spawned must be executed before any dependent tasks can be executed. This can lead to serialisation of the work (and thus poor parallel performance) for particle methods, as shown in [8].

In molecular dynamics the order in which the tasks are executed does not matter, at least within the pairwise interactions, but one wants to avoid race conditions. As such, the dependencies currently available in OpenMP are too constraining. Instead, we want to use conflicts, as described in [9], also known as commutative dependencies in OmpSs [10] and StarPU [11]. It is possible to implement something similar using OpenMP 4.5 using taskyield and locks. We extend the cell type to contain an OpenMP lock. When executing a task, we attempt to lock any cells that are written to by the task. If successful, then the task executes as normal. If one of the locks can't be obtained, we unlock any locks obtained and yield the task using the taskyield OpenMP directive:

```
do while(.not. locked)
 if(omp_test_lock(cell_i%cell_lock)) then
  if(omp_test_lock(cell_j%cell_lock)) then
   locked = .true.
  else
   Call omp_unset_lock(cell_i%cell_lock))
  end if
 end if
 if(.not. locked) then
  !$omp taskyield
 end if
end do
```

If we just lock cell_i then cell_j, it is possible for deadlock to occur, where three threads are attemping to lock cells k, l, m and each locking a different cell first. This problem is known as the Dining Philosophers problem. Since each cell is assigned a unique integer ID, we attempt to lock the lower numbered cell first, which is the most straightforward solution to the problem. Removing dependencies also allows us to spawn the tasks in parallel, which may lead to a performance improvement.

Results

We implemented 4 variants of the code for DL_POLY_4: i) A method using inout dependencies to control race conditions, with a single thread spawning all of the tasks; ii) A method using locks plus taskyield, with a single thread spawning all of the tasks; iii) A method using locks plus taskyield, using all threads to spawn tasks; iv) A method using locks plus taskyield with task priorities, using all threads to spawn tasks. We ran these variants on Intel Xeon Ivy Bridge (E5-2697 v2) and Intel Xeon Phi Knight's Landing (KNL 7210), with the Intel 17.2.050 and gcc 6.3 runtimes. We used a simple testcase involving only van der Waals forces, though tests with additional short-range electrostatic forces showed similar results.

The scaling and parallel efficiency of the parallel region containing the tasks is shown in Figure 2 with the Intel runtimes on both processors. The parallel region achieves roughly 80% parallel efficiency when using the entirety of both processors, meaning we lose a significant amount of performance due to additional task overheads when compared to the serial version. On the KNL, we compare the performance with different numbers of threads per core, relative to a single core with a single thread. At low core counts, more threads perform better, however when using the entire KNL best performance 2 is achieved with two threads per core, and one thread per core performs better than using three or four.

To be able to see the performance of the parallel region in more detail, we manually instrumented the code. At the start of each task (or after the locks are obtained if using taskyield) the code records the system time using omp_get_wtime(), and records it again at the end of the task region. This allows us to create a timeline showing how the tasks are executed. These are shown on the title page, the upper three graphs for taskyield with parallel spawning, taskyield with serial spawning and dependencies with the Intel runtime.

When using the OpenMP inout dependencies, the amount of possible parallelism is reduced. Firstly, the tasks have to be spawned by a single thread, however most task frameworks struggle with parallel task creation. Secondly, the dependencies enforce an ordering on the tasks that need to be executed. This serialises certain sections of the work. Its possible that by changing the order in which tasks are created that the amount of parallelism will increase. However, we spent minimal time investigating this as the taskyield method performed so much better, and the commutative dependencies suggested in other runtimes seemed more ideal.

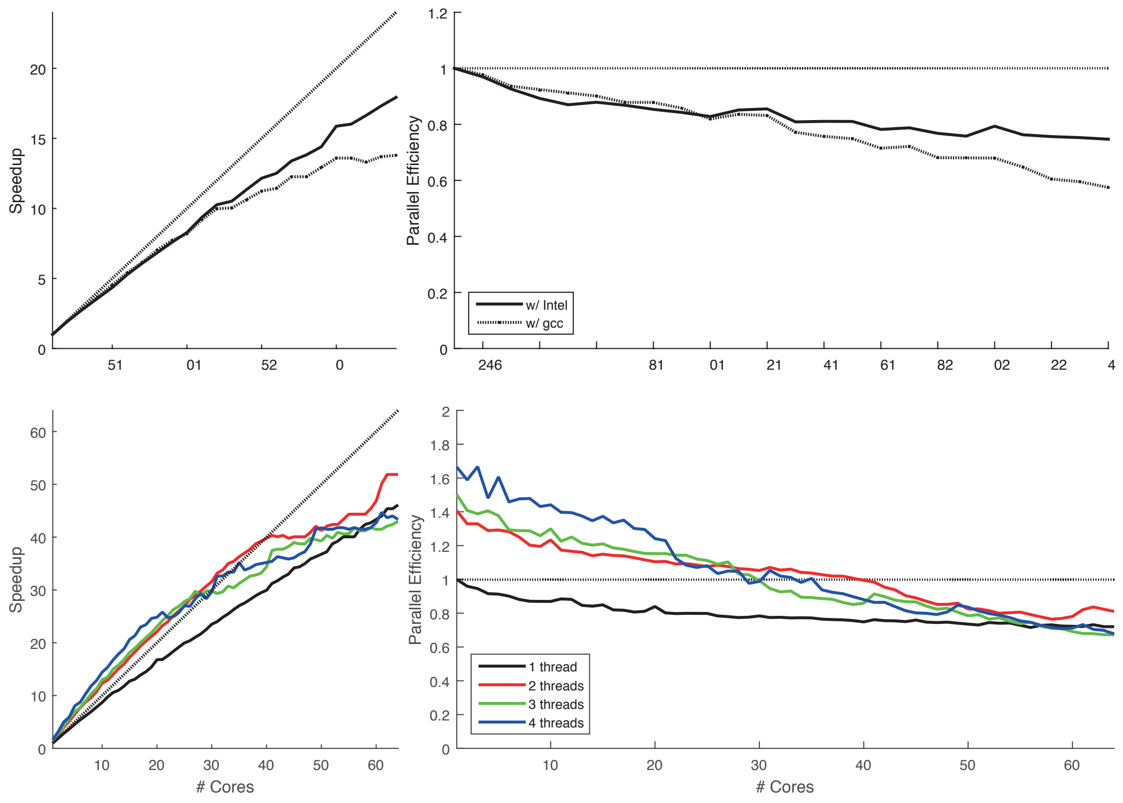


Figure 2: Speedup and Parallel Efficiency of the parallel region on Xeon Ivy Bridge and Knight's Landing. Both lose significant performance due to the overheads associated with the OpenMP task framework. The results with Intel KNL are all shown relative to a single thread on a single core, which means when running with multiple threads per core we can achieve a parallel efficiency of above 1.

With taskyield, the intel runtime significantly outperforms the gcc-6 runtime, shown on the title page last panel. This is due to the way that taskyield is implemented in the two runtimes. In the gcc runtime, the taskyield keyword is ignored, leading to tasks potentially being stuck in a while loop attempting to lock cells, rather than executing other available work. The intel runtime searches for a new task when finding a taskyield, while keeping the yielded task in the executing thread's stack.

In all examples there is significant "white" space throughout the computation, which cannot be entirely due to the methods used to avoid race conditions. When tasks are spawned by a single thread, that thread spends a large amount of time only spawning tasks to be executed. When tasks are spawned in parallel, we can assume that the overall time required to spawn the tasks must be at least as large, which may explain many of the white gaps throughout the computation. It is unclear what causes the threads to spawn tasks vs execute tasks, however this is implementation-defined. From the parallel spawning task plots with Intel and gcc it appears that the Intel runtime spawns relatively few tasks before threads begin executing them, while gcc waits until most of the tasks have been spawned before the majority of threads begin execution. Due to the difference in the taskyield implementation between the two runtimes it is not possible to argue which approach is better from these results, however if the task creation involves memory allocation or other serial operations it may suggest having some threads performing work while others generate tasks may help.

Authors
A. B.G Chalk, A. M.Elena, STFC Daresbury Laboratory

References

[1] Ilian T Todorov, William Smith, Kostya Trachenko, and Martin T Dove. DL_POLY_3: new dimensions in molecular dynamics simulations via massive parallelism. Journal of Materials Chemistry, 16(20):1911–1918, 2006.

[2] B Quentrec and C Brot. New method for searching for neighbors in molecular dynamics computations. Journal of Computational Physics, 13(3):430–432, 1973.

[3] Gary S Grest, Burkhard Dünweg, and Kurt Kremer. Vectorized link cell fortran code for molecular dynamics simulations for a large number of particles. Computer Physics Communications, 55(3):269–285, 1989.

[4] Ralf Meyer. Efficient parallelization of molecular dynamics simulations with short-ranged forces. Journal of Physics: Conference Series, 540(1):012006, 2014.

[5] Pedro Gonnet. Pairwise verlet lists: Combining cell lists and verlet lists to improve memory locality and parallelism. Journal of computational chemistry, 33(1):76–81, 2012.

[6] Pedro Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. Journal of Computational Chemistry, 28(2):570–573, 2007.

[7] Pedro Gonnet. Pseudo-verlet lists: a new, compact neighbour list representation. Molecular Simulation, 39(9):721–727, 2013.

[8] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. Concurrency and Computation: Practice and Experience, 26(11):1935–1946, 2014.

[9] Pedro Gonnet, Matthieu Schaller, Tom Theuns, and Aidan BG Chalk. Swift: Fast algorithms for multi-resolution sph on multi-core architectures. arXiv preprint arXiv:1309.3783, 2013.

[10] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters, 21(02):173–193, 2011.

[11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011.