On the performance of parallel tasking runtimes for an irregular fast multipole method application

Patrick Atkinson and Simon McIntosh-Smith

Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, United Kingdom
`{p.atkinson,simonm}@bristol.ac.uk`

Abstract. This paper will present our work on optimising and comparing the performance of an irregular algorithm for the increasingly important fast multipole method with the use of tasks. Our aim is to provide insight into how different methods of synchronisation can affect the performance of tree-based particle methods, finding that performance can be improved by 21% on some platforms. We also compare the performance of the chosen application between different OpenMP implementations and to other task-parallel programming models, finding that significant performance differences can be observed on both NUMA and Many Integrated Core architectures.

Keywords: OpenMP, tasks, mini-apps, locks, atomics

1 Introduction

Introduced in 2007, OpenMP tasks have allowed for the simplification of expressing parallel execution of irregular problems, such as divide and conquer algorithms. The mapping of tasks to threads is non-deterministic and the scheduling efficiency is highly dependant on the underlying runtime. The availability of different OpenMP implementations and other similar task-parallel programming models, such as OmpSs [1], Intel Threading Building Blocks [2], and Cilk [3], has given application developers many options to choose from, whilst differences in scheduling techniques and the features provided has lead to differences in performance.

As the tasks constructs in the OpenMP standard have been expanded and matured in the past 10 years, the level of parallelism in current architectures has increased dramatically. Non-unified Memory Access (NUMA) architectures are now commonplace in high performance systems, with current generation Intel architectures comprising of as many as 22 cores per socket. In addition to NUMA architectures, the introduction of many integrated core architectures, such as the 72 core Intel Knights Landing chip, has demonstrated the need for low-overhead and scalable parallel runtimes.

In addition to simplifying the expression of irregular problems, task-parallelism has the potential to increase performance on systems with large numbers of cores.

Using fine-grained parallelism by way of task dependencies, tasks are only executed when the specific data they operate on is available. This is in contrast to conventional OpenMP programs that make use of fork-join constructs whereby steps of an algorithm are executed in a series of parallel for-loops, whilst a task-based approach would allow for each step to overlap and correctness assured through the programmer's use of task-dependencies.

OpenMP has become the de facto standard for thread-parallelism in HPC, with a large range of different implementations from groups such as Intel, GNU, and Cray; it has become a simple and powerful way to parallelise both existing and new applications. It is not the only parallel programming model that offers task-parallelism however. Cilk [3], TBB [2], StarPU [4], OmpSs [1], Kokkos [5], and even the C++11 standard all now offer task constructs, giving an application developer a wide range of options. However, there is also great uncertainty in which models provide both the richest and the most convenient APIs, while also offering the greatest performance on modern, highly parallel architectures.

This paper will present a comparison of a range of different programming models and OpenMP implementations using a representative application, known as a 'mini-app'. Mini-apps are scaled-down applications that capture the performance characteristics of real scientific codes; they are commonly used to rapidly compare and test both programming models and architectures [6]. Currently however, few mini-apps exist that can make good use of tasks and can be used to assess current tasking programming models. Hence, the comparison will be performed using a new Fast Multipole Method mini-app, MiniFMM [1], developed at the University of Bristol. The method works primarily around a tree traversal algorithm and can exhibit high load imbalance, thus providing an interesting real application to compare and test tasking performance.

The outline of the paper is as follows: Section 3 briefly describes the FMM and details of the particular variant used, Section 4 gives details of the OpenMP implementation and discusses the challenges faced using the tasking model with FMM, Section 5 gives an overview of differences in OpenMP implementations and similar programming models, Section 6 provides a comparison and discussion of the different programming models, and Section 7 concludes the paper describing how the results can be generalised and applied to other task-based methods to improve performance.

2 Background and related work

Previous work has shown the significant design and performance differences between OpenMP implementations. Most work has focused on comparing the performance and runtime execution characteristics of micro-benchmarks, such as computing Fibonacci numbers and sorting arrays. Olivier et al. [7] had previously compared the parallel performance of OpenMP tasks using the BOTS benchmark suite [8], finding that the overhead costs and idle thread times varied

[1] `https://github.com/uob-hpc/minifmm`

greatly between benchmarks. This work was then extended by Virouleau et al. who looked at the KASTORS benchmark suite from BSC [9], which examined the performance of tasks with data dependencies, finding that the performance of some of the benchmarks could be impacted by the OpenMP runtime used. Whilst these benchmarks have provided key insights, the aim of this paper is to examine how the performance of a representative application is affected by both parallel overheads and runtime decisions.

Previous efforts to parallelise the fast multipole method (FMM) have shown that task-based approaches provide large performance benefits. The tree-traversal algorithm designed by Yakota et al. [10] was initially implemented with tasking features from Intel TBB and large performance improvements were gained over similar methods. Following on from this, Pericas et al [11] extended this work by implementing the tree-traversal step using tasks with data-dependencies in OpenMP, finding only minor performance improvements could be gained. Making use of extra data-dependency constructs available in StarPU, Agullo et. al [12], found that performance could be improved over OpenMP.

3 Method overview

The FMM has many uses in the fields of physics and computational mathematics, including calculating gravitational/electrostatic forces, fluid dynamics, plasma simulation, and acoustics [13]. Fundamentally, the algorithm provides a linear time approximation to $O(n^2)$ problems and allows for tunable precision of results. It works by grouping particles via a space partitioning tree (such as an octree), where groups of particles are located at each tree node. As in the N-body problem, each particle will need to calculate the force due to all other particles in the system. The difference using the FMM is that particles are compared group to group; each target group of particles is compared to all other nodes in the tree, resulting in two outcomes:

1. If the two nodes are far enough away, the force contribution for a source node can be approximated for the target node.
2. Else the forces for each particle will be calculated directly.

If the force contribution can be approximated, then the target doesn't need to consider any tree node below that source node. This has the very important property of the application's control flow not being known until runtime; the control flow is data-dependant. It is also of note that the application is compute-bound due to the high FLOP/byte ratio of directly computing the forces of particles in nodes that are close together.

The Dual Tree Traversal method for FMM, devised by Yakota and Dehnen [10], has been shown to be an efficient tree traversal method that also allows user control over the distance required to approximate node interactions, hence greater control over the precision of the final results. It is worth noting that other FMM implementations exist that do not allow for control over the distance at which approximations are made; this affects the implementation when using tasks and is outlined further in Section 4.1. Pseudo-code for the tree traversal is shown in Listing 1.1.

```
dtt(node source, node target)
{
  // calculate distance between source and target
  ...

  if (source and target well separated)
    approximate_force(source, target)

  else if (is_leaf(source) && is_leaf(target)
    direct_force(source, target)

  else
  {
    if (source.radius > target.radius)
      for (child in target)
        dtt(child, source)

    else
      for (child in source)
        dtt(target, child)
  }
}
```

Listing 1.1: Dual Tree Traversal

All of the results collected in this paper are run with an input of $O(10^6)$ particles uniformly distributed inside a box. At the finest level of the tree structure, the maximum number of particles per node is set to 300. These input parameters were selected to match those seen in previous work [10][11]. Unless stated, all tests are performed using double precision values.

4 Implementation overview

As the method evaluates all pairs of nodes in the tree, it is possible for two threads to be calculating the force contribution for the same target node. In OpenMP, task dependencies, atomics, and locks can all be used to ensure correctness. This section will detail efforts to increase performance of synchronisation in a task-based application using the architectures listed in Table 1 and using the Intel C Compiler (17.0).

4.1 Task dependencies

Using task dependencies introduced in the OpenMP 4.0 standard, we can avoid memory read/write conflicts. However, optimal performance won't be achieved for two reasons. Firstly, task dependencies are resolved in the order in which tasks are created, hence an unnecessary ordering on tasks is enforced; the fast multipole method permits updating particle values in any order. Secondly, the work of finding the distance between nodes and deciding whether to approximate or calculate the force directly and creating a task to do so, is too great for a single thread to perform whilst issuing enough tasks to saturate the other threads with work, hence the entire computation is stalled by the thread issuing tasks. With large numbers of threads this can cause a severe bottleneck; running on 256 threads of a KNL and using data-dependencies in this way results in performance that is ∼22x slower than alternatives and as such a parallel traversal is required. However, using a parallel traversal with data-dependencies has the issue of task dependencies only being enforced for the immediate child tasks of the current task, hence data conflicts would not be enforced across threads.

4.2 Atomics

An alternative to task dependencies would be to make the accumulations within a task be atomic operations. Hence, for both the direct and approximate calculations, the force updates are applied atomically for each particle. As the force calculations are over all particles in a node, this can mean many atomic operations are required. In addition, the method requires complex numbers (added to C standard in ISO C99) and built-in complex data-type atomic operations are not supported within OpenMP, hence separate arrays of real and imaginary types are required instead.

4.3 Locks

Another option would be to create a lock for every node in the tree, then lock and unlock a node to update the entire group of particles inside a node.

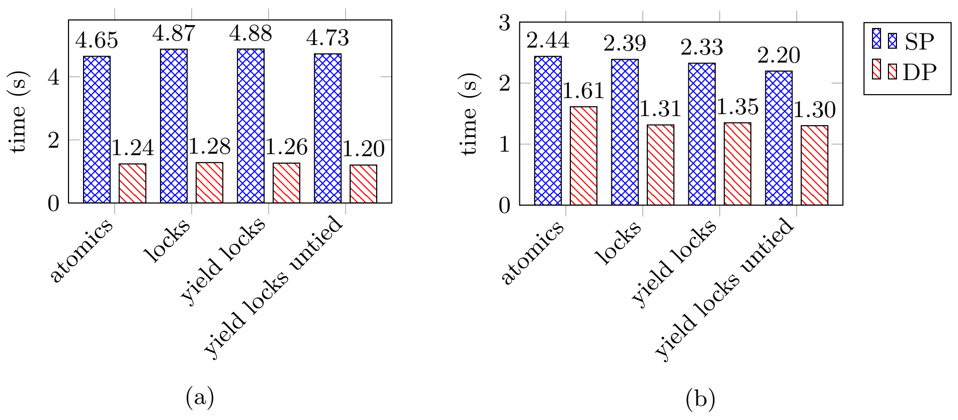


Fig. 1: Comparison of synchronisation methods on a) two sockets of 22-core Broadwell b) 64-core Xeon Phi Knights Landing

Which of these two options (locks or atomics) performs better depends entirely on the execution of the method. Atomically updating the forces for each particle introduces a fixed overhead compared to locks, however, high lock contention will cause large amounts of idle thread time. As can be seen in Figure

1a, using atomics results in superior performance on Broadwell for both single and double precision data. However, on Intel Xeon Phi Knights Landing (KNL), Figure 1b shows that double precision performance is roughly equivalent, whilst locks outperform atomics for single precision.

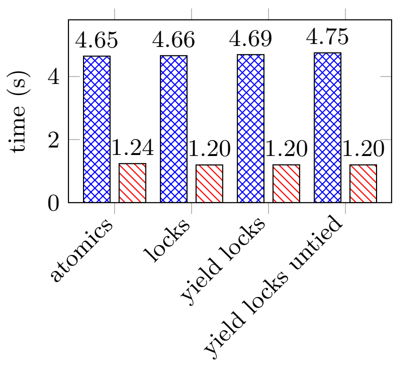
It is also possible to improve the performance of locks when combined with task constructs in OpenMP. Introduced by Chalk [14], the use of `taskyield` when a task cannot acquire a lock, shown in Figure 1.2, as opposed to using `omp_set_lock`, can dramatically improve performance. Essentially, a thread executing a task tries to acquire a lock and if it is unsuccessful, a task scheduling point is reached, allowing for the runtime to suspend the execution of the current task. This allows the executing thread to do other work in the hope that when the task execution is resumed, the lock can now be acquired. In Figures 1a and 1b, this method is referred to as 'yield lock' and, as can be seen, this alone has little effect when compared to `omp_set_lock`. However, when combined with untied tasks, i.e. `pragma omp task untied`, the performance difference is noticeably improved. The use of the `untied` keyword allows for any thread to resume the execution of a suspended task. Whilst it was measured to have no performance impact when combined with atomics or `omp_set_lock`, using untied tasks in conjunction with `taskyield` and locks leads to a performance increase ('yield locks untied' in Figures 1a and 1b). This is due to threads being able to resume tasks that were suspended by another thread when a lock could not be acquired; overall this leads to better load balance of tasks.

```
int locked = 0;
while (!locked)
{
    locked = omp_test_lock(&target->lock);
    if (!locked)
    {
        #pragma omp taskyield
    }
}
```

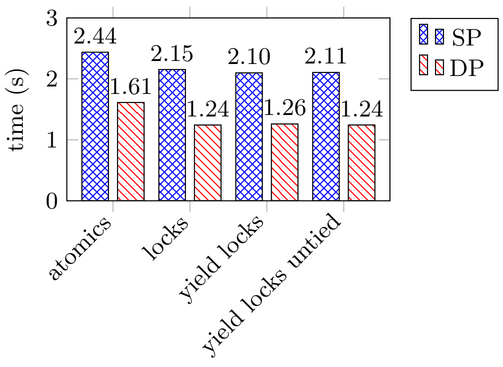
Listing 1.2: Locking with taskyield

Instead of using a single lock per tree node, two locks could also be used. One to prevent a race condition on the approximate force accumulation and one to prevent the race condition on the direct force accumulation. Using two locks, the same synchronisation methods were tested and the results are displayed in Figures 2a and 2b. Overall, this results in the best performance as lock contention was reduced, however some interesting effects were observed. Firstly, there is little benefit gained from the use of `#taskyield` lock variant on Broadwell. This is because as lock contention is lower, the `#taskyield` is less likely to be encountered. Whilst on KNL, higher thread counts result in high enough lock contention that the use of `#taskyield` is still marginally beneficial when using double-precision values. The use of `untied` tasks generally results in worse performance when using two locks.

From these results it was concluded that if you have highly contended locks, as in the case where a single lock was used per tree node, then there are performance benefits from using ‘yield locks’ and `untied` tasks. In contrast, with locks that have lower contention, these keywords aren’t needed and can result in the same or even worse performance.



(a)



(b)

Fig. 2: Comparison of synchronisation methods using two locks per tree node on a) two sockets of 22-core Broadwell b) 64-core Xeon Phi Knights Landing

Another attempt to optimise lock performance was to specify the lock implementation via the `omp_init_lock_with_hint` function added in OpenMP 4.5. This allows a user to request a lock optimised for high contention (`uncontended` / `contended`) and/or speculative locks. It was found that in all cases `uncontended` locks performed worse than `contended`, and whilst speculative locks are supported on Intel Xeon CPUs (but not Xeon Phi), the use of the hint had no effect. The ability to specify the lock implementation was only available in the Intel OpenMP implementation, whilst the Cray compiler and GCC lacked this feature.

Whilst this alternative to task dependencies, referred to as ‘conflicts’ in Chalk [14], could be added to the OpenMP standard as a task clause, it can be seen that in this application, there is not a definitive method to implementing ‘conflicts’; as seen in Figure 1a, atomics still outperform the alternatives when using double precision values in the mini-app, whilst locking with `taskyield` and untied tasks perform better in other cases.

4.4 Extensions to task dependencies

Programming models such as OmpSs and StarPU have the ability to declare commutative task dependencies. This feature allows for the specified data locations to be updated in any order, regardless of the order the tasks were issued. This is in contrast to data dependencies in OpenMP, where for a data dependency, the order in which tasks are created is the order in which the tasks have to be executed. Hence, commutative task dependencies provide benefits to applications such as fast multipole; however, due to having to perform a parallel tree traversal, the data dependencies won't be enforced for all threads (as in Section 4.1).

4.5 Comparison baseline

In contrast to a task-based approach, the algorithm can also be implemented in a thread-parallel fashion. This is done by recursing down the tree and instead of issuing tasks, we record whether to perform the direct or approximate force calculation for the current node. Then, each node can be iterated over in a `parallel for` loop, performing the required operations. A dynamic schedule was found to be optimal due to the high load imbalance between the number of operations each node needed to perform. This has the advantage of avoiding the race condition in that no two threads will write to the same target node. However, using tasks still has a number of advantages. Firstly, there's an overhead cost of initially building the list of nodes needing to be operated on per thread; a small cost in performance, which can dramatically increase memory usage; in the worst case each node will store a list of all other nodes in the tree. When implemented, this thread-parallel version tripled the number of lines of code compared to the task-based approach of the tree traversal. Therefore, whilst it is possible that the task-based approach may not offer a significant performance increase over this approach, a runtime that is able to match the performance of a thread-parallel implementation will be deemed a success, demonstrating tasking can reduce code size without impacting performance. However, due to the overhead of initially finding the lists of interactions, it was hoped that tasking implementations could be slightly faster than the thread-parallel equivalent. This thread-parallel implementation of the algorithm is referred to as the 'loop' implementation of the algorithm for the remainder of the paper.

To compare to other FMM implementations we profiled the task-parallel method, observing that 96% of the runtime was spent calculating the forces directly. Counting the number of interactions between particles and knowing the number of FLOPs per interaction tells us the compute performance achieved in the direct force calculation, which was measured to be approximately 882 DP GFLOPs on the dual socket Broadwell CPU. Comparing this to previous work [10] (and to the peak FLOPs) would indicate that the mini-app was both representative of larger FMM applications and achieved reasonable performance.

5 Programming Models

This section briefly introduces each of the programming models used in the comparison and discusses key features identified in each.

OmpSs - OmpSs provides a testing ground for new OpenMP features, and has previously motivated changes to the OpenMP standard, such as task data-dependencies. The OmpSs programming model is syntactically similar to OpenMP and provides both a compiler that allows for additional task extensions as well as a runtime system. For our tests we are using the Intel compiler backend (17.0) for OmpSs [1].

BOLT - BOLT stands for 'BOLT is OpenMP over Lightweight Threads'. From Argonne National Laboratory, the BOLT project aims to provide a light-weight threading runtime based on the LLVM OpenMP runtime [15]. In contrast to current OpenMP implementations based on OS-level threads, BOLT aims to use light-weight threads, provided by Argobots [15], to improve performance.

Intel Cilk Plus - Built as an extension to Cilk++, Cilk plus provides a simple interface of three keywords that enable task and data parallelism. The scheduling policy has been shown to provide load balance close to optimal [3].

Intel TBB - An object-oriented C++ runtime library, Intel TBB maintains a double-ended queue per thread, retrieving new tasks from the back of its queue to exploit temporal locality. If a thread has finished its work, it steals from the front of another thread's queue [2].

OpenMP - Previous work has highlighted some of the implementation decisions of each of the OpenMP runtimes finding that, depending on the architecture, significant performance differences can be observed. For example, the Intel implementation maintains a task queue per thread as opposed to a single task queue for all threads (as in GNU OpenMP). This has the effect of improving data locality by allowing threads to enqueue tasks on each thread's own queue first, in the hope that data can be reused from recently executed tasks.

6 Results

	Broadwell	**KNL**
Processor	Xeon E5-2699 v4	Xeon Phi 7210
Sockets	2	-
Total Cores	22	64
Total Threads	44	256
Total TFLOPS	1.54	2.66

Table 1: Target machines

The performance evaluation was conducted on two of the most recently released architectures. This was done to both reflect current devices in some of the largest supercomputers and to examine the performance characteristics of different task-parallel runtimes with both high numbers of threads and NUMA architectures. The details of the target machines appear in Table 1. The results were obtained with both Hyper-Threading turned on and off for Broadwell, whilst on KNL three different configurations were tested with 1, 2, or 4 threads per core.

For the results, the Intel Compiler (17.0) was used for OpenMP, TBB, Cilk, and the OpenMP parallel loop version of the algorithm. GCC 6.3 and Cray CCE 8.5.8 were used for the OpenMP GNU and Cray results respectively. The OmpSs version used was 16.06.3.

6.1 Broadwell

	OpenMP							
	Intel	GNU	Cray	BOLT	Loop	OmpSs	Cilk	TBB
Serial (s)	156.057	157.365	154.120	156.170	156.721	157.855	156.100	156.825
Parallel (s)	4.654	4.843	4.871	4.656	4.654	4.719	4.632	4.745

Table 2: Serial and fastest runtimes achieved using Broadwell

Figures 3 and 4 show the parallel speedup when increasing the number of cores and, as can be seen, the different programming models and runtimes exhibit similar scaling performance. The GNU and Cray OpenMP implementations exhibit the poorest parallel times, whilst the serial times do not differ from the other frameworks. The Intel OpenMP runtime performs well however and was consistently measured, along with Cilk, to give the best performance.

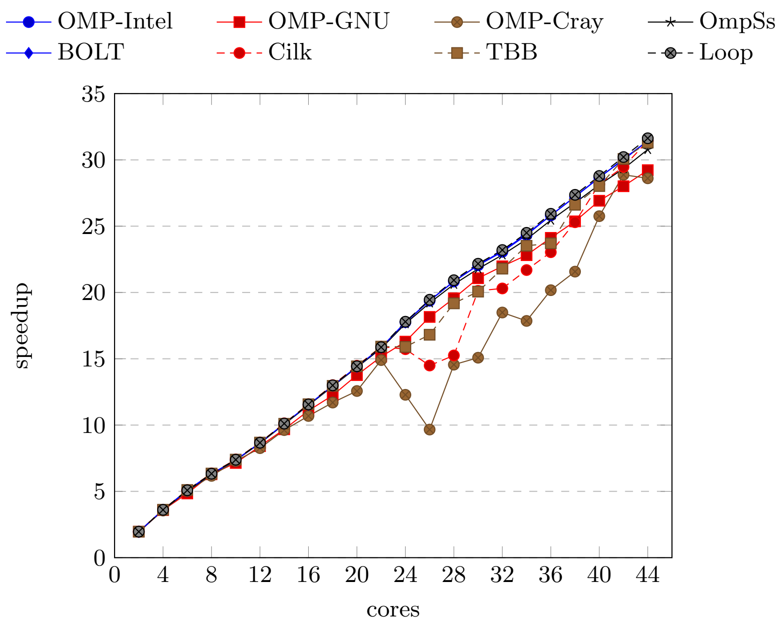


Fig. 3: Parallel speedup on Broadwell with 1 thread per core

The BOLT OpenMP implementation exhibits very similar performance to the Intel OpenMP implementation. This could be due to the Intel OpenMP runtime being open-sourced and used as the OpenMP backend for LLVM, on which BOLT is based. Cilk exhibits good performance on both Broadwell and KNL, being the fastest on both architectures - this is impressive because of its

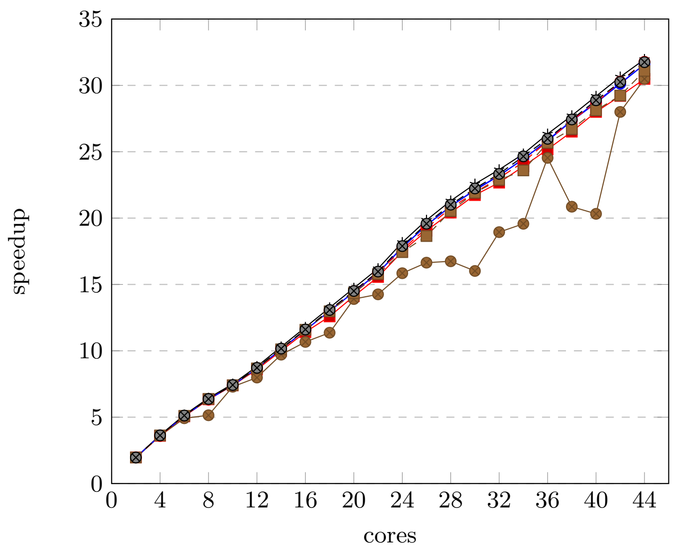


Fig. 4: Parallel speedup on Broadwell with 2 threads per core

relatively small feature-set. Intel TBB achieves reasonable performance on both targets, but slightly lags behind other Intel runtime implementations.

The majority of the runtimes compete with the baseline parallel loop implementation (as described in Section 4.5) when using tasking, hence for this CPU architecture, tasks provide a scalable and efficient way to parallelise the mini-app whilst reducing the amount of code.

6.2 Knights Landing

	OpenMP							
	Intel	GNU	Cray	BOLT	Loop	OmpSs	Cilk	TBB
Serial (s)	181.385	199.271	185.728	181.401	175.975	190.622	181.272	181.371
Parallel (s)	2.059	3.508	3.224	2.054	1.949	2.192	2.010	2.533

Table 3: Serial and fastest runtimes achieved using KNL

On KNL the performance of tasks in all frameworks were slightly worse than the parallel loop implementation. The Intel and Bolt OpenMP runtimes performed the best when running the task parallel approach, yet the parallel loop method was 1.05x faster. Most runtimes achieved similar performance when running with a single thread per core, however, running 2 and 4 Hyper-Threads per core highlighted the weakness in some of the other runtimes. The Intel implementation of OpenMP and Cilk both exhibited good scaling with high numbers of threads whilst TBB lagged slightly behind. However, the Cray OpenMP implementation exhibited poor scaling with all three thread configurations and gave poorer performance as the number of threads per core increased.

The GNU OpenMP runtime actually results in a degradation in performance as more threads are added. Whilst initially showing good performance in Figure 5, it can be seen that performance starts to degrade when using two Hyper-Threads per core (Figure 6). Then finally with 4 Hyper-Threads per core (Figure 7), the runtime of the application becomes severely limited when the number of threads used increases.

Initially the performance of OmpSs on KNL was extremely limited and with 256 threads was roughly 10x slower than the parallel loop implementation of the method. This is due to the default scheduler being unsuitable for many-integrated core architectures as it maintains a single global ready queue for tasks, which causes high contention on this data-structure when utilising large numbers of threads. Instead, the distributed breadth-first scheduler was used. This scheduler maintains a task queue per thread and work-steals, resulting in performance similar to the other implementations. Like the default scheduler in OmpSs, GNU OpenMP also maintains a single task queue for all threads, thus explaining the poor performance seen in Figures 6 and 7.

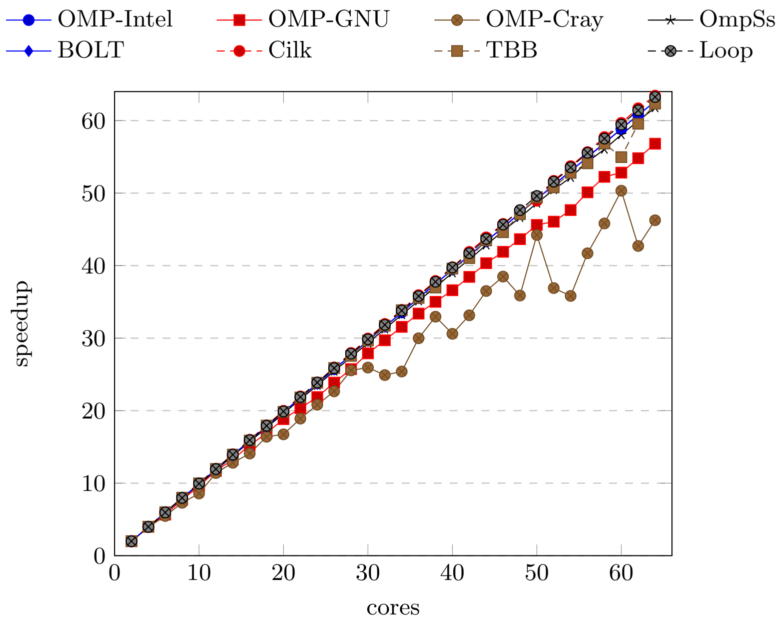


Fig. 5: Parallel speedup on KNL with 1 thread per core

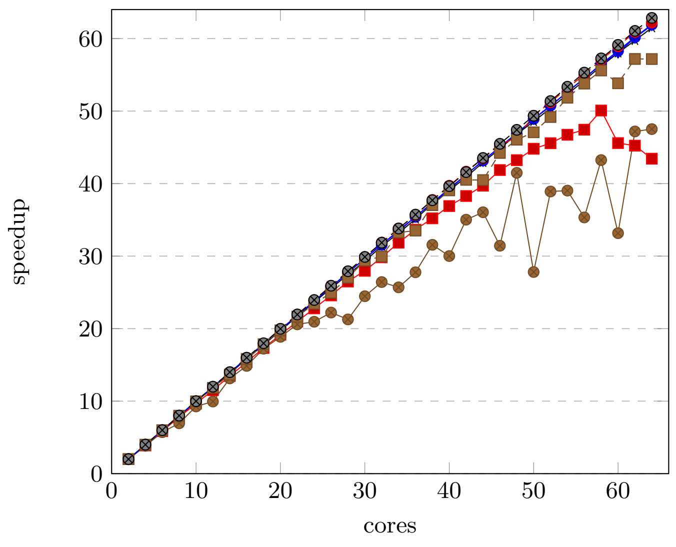


Fig. 6: Parallel speedup on KNL with 2 threads per core

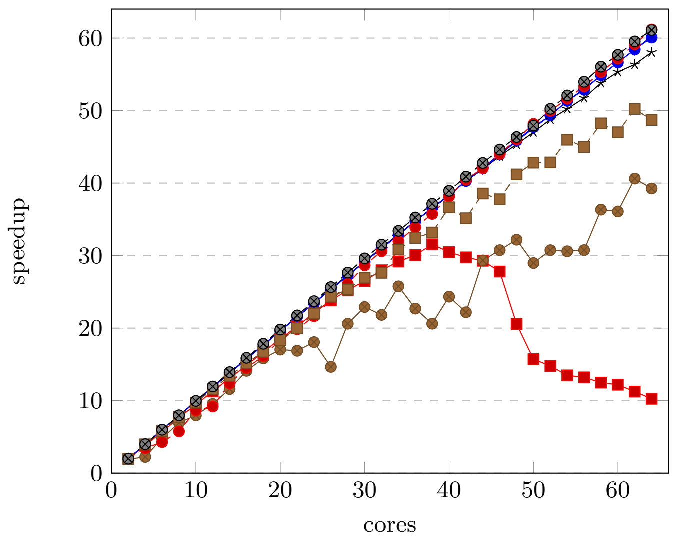


Fig. 7: Parallel speedup on KNL with 4 threads per core

7 Conclusion

The OpenMP tasking constructs were designed to allow users to easily express the parallelism of recursive and irregular algorithms. In terms of productivity, OpenMP task features were a simple and powerful way to parallelise the mini-app, drastically reducing the code required compared to the parallel loop implementation.

Using our FMM mini-app, we have looked at how task synchronisation can be improved for particle methods by comparing atomics and various ways of using locks in OpenMP, finding that performance can be improved by up to 21% on KNL whilst also bringing improvements on Xeon CPUs.

A common pattern in N-body, finite element, and unstructured mesh applications is to have data locations receiving multiple, unordered contributions. Hence, the work done on examining synchronisation features in OpenMP could be generalised and applied to a wide range of applications.

In addition to examining language features, we also compared OpenMP implementations to each other and to other task-parallel programming models. We found that on Broadwell, most programming models and OpenMP implementations performed well, competing with an equivalent parallel loop implementation. However, on KNL we found that a parallel loop implementation outperformed all task implementations of the method. Therefore, future work will focus on understanding this difference and investigating solutions to improving task performance on this platform.

This work builds upon the success of previous mini-app work within the HPC group at the University of Bristol [16], demonstrating that mini-apps are powerful tools to both compare and test different programming models, as well investigate different language features that can lead to increased performance for a more general set of applications.

Acknowledgements

The authors would like the thank EPSRC for funding this work, as well as Bristol's Intel Parallel Computing Centre (IPCC) for access to the KNL platform. We would also like to thank GW4 for access to the Isambard supercomputer for Broadwell results.

References

1. A. Duran, E. Ayguad, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626411000151
2. W. Kim and M. Voss, "Multicore desktop programming with intel threading building blocks," *IEEE Software*, vol. 28, no. 1, pp. 23–31, Jan 2011.

3. R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324234
4. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “Starpu: A unified platform for task scheduling on heterogeneous multicore architectures,” *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1631
5. H. C. Edwards and D. Sunderland, “Kokkos Array Performance-portable Manycore Programming Model,” in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM’12)*. ACM, 2012, pp. 1–10.
6. M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, “Improving Performance via Mini-applications,” Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.
7. S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, “Characterizing and mitigating work time inflation in task parallel programs,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 65:1–65:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389085
8. A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, “Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp,” in *2009 International Conference on Parallel Processing*, Sept 2009, pp. 124–131.
9. P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, *Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite*. Cham: Springer International Publishing, 2014, pp. 16–29. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11454-5_2
10. R. Yokota, “An FMM based on dual tree traversal for many-core architectures,” vol. 7, no. 3, pp. 301–324. [Online]. Available: http://journals.sagepub.com/doi/abs/10.1260/1748-3018.7.3.301
11. P. Miquel, A. Abdelhalim, F. Keisuke, M. Naoya, Y. Rio, and M. Satoshi, “Towards a dataflow fmm using the ompss programming model,” , 12, 2012-09-26. [Online]. Available: http://id.nii.ac.jp/0606/00073141
12. E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset, “Bridging the gap between openmp and task-based runtime systems for the fast multipole method,” *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2017.
13. L. F. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems (ACM Distinguished Dissertation)*. The MIT Press, 1988.
14. A. M. E. Aidan Chalk and L. Mason, “Task-based parallelism in dl poly 4,” 2017. [Online]. Available: http://staging.ixpug.org/documents/1491984172IXPUG_Spring_2017_paper_13(1).pdf
15. A. N. Laboratory, “Bolt is openmp over lightweight threads.” [Online]. Available: http://www.bolt-omp.org/
16. M. Martineau, S. McIntosh-Smith, and W. Gaudin, “Evaluating openmp 4.0’s effectiveness as a heterogeneous parallel programming model,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 338–347.