Science & Technology Facilities Council

Technical Report
DL-TR-2015-002

Exploiting multi-core processors for scientific applications using hybrid MPI-OpenMP

L Anton, M Ashworth, X Guo, S Pickles, A Porter, A Sunderland

January 2015

Enquiries concerning this report should be addressed to:

Chadwick Library
STFC Daresbury Laboratory
Sci-Tech Daresbury
Keckwick Lane
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1362-0207

Exploiting Multi-core Processors for Scientific Applications Using Hybrid MPI-OpenMP

Lucian Anton, Mike Ashworth, Xiaohu Guo, Stephen Pickles, Andrew Porter and Andrew Sunderland

Application Performance Engineering Group,
STFC Scientific Computing Department

STFC Daresbury Laboratory,
Sci-Tech Daresbury,
Warrington WA4 4AD
United Kingdom

Abstract

Most current and emerging high-performance systems consist of large numbers of processors set within an architecture with 'fat' shared memory nodes supporting tens of threads per node. There are good reasons to adopt a hybrid MPI-OpenMP programming model for large-scale applications on such architectures, but this adds complexity to the parallel program and demands scalability at two levels: MPI across nodes and OpenMP within a node.

We present performance and scaling studies for four applications (Fluidity-ICOM, NEMO, PRMAT and a 3D Red-Black Smoother) that use the hybrid MPI-OpenMP programming model. We show that for computations that use a large number of cores the hybrid approach provides a significant improvement to the performance provided that algorithms with minimal synchronisation and suitable libraries are used.

Introduction

The trend for HPC architectures is towards large number of lower frequency cores with a decreasing memory to core ratio. This is imposing a strong evolutionary pressure on numerical algorithms and software to utilise efficiently the available memory and network bandwidth. Using modern multi-core processors presents new challenges for scientific software applications, due to the new node architectures: multiple processors each with multiple cores, sharing caches at different levels, multiple memory controllers with affinities to a subset of the cores, as well as non-uniform main memory access times.

For modern multi-core architecture supercomputers, hybrid MPI/OpenMP also offers new possibilities for optimisation of numerical algorithms beyond pure distributed memory parallelism. For example, scaling of algebraic multi-grid methods is hampered when the number of subdomains is increased due to difficulties coarsening across domain boundaries. The scaling of mesh adaptivity methods is also adversely affected by the need to adapt across domain boundaries.

Portability across different systems is highly critical for application software packages, and the directive-based approach available in OpenMP expresses parallelism in a portable manner. It offers potential capabilities to use the same code base to explore accelerated and non-accelerator enabled systems as OpenMP version 4[1] expands the scope of the API to embedded systems and accelerators.

Because of this, there is a growing interest in hybrid parallel approaches where threaded parallelism is exploited at the node level, while MPI is used for inter-process communications. Significant benefits can be expected from implementing such mixed-mode parallelism. First of all, this approach decreases the memory footprint of the application as compared with a pure MPI approach. Secondly, the memory footprint is further decreased through the removal of the halo regions which would be otherwise required within the node. For example, the total size of the mesh halo increases with number of partitions (i.e. number of processes). It can be shown empirically that the size of the vertex halo in a linear tetrahedral mesh grows as $O(P^{1.5})$, where P is the number of partitions. Reducing the number of MPI ranks on a node reduces contention for network adaptors and reducing the total number of MPI ranks across the system reduces the MPI communications time, keeping the communications within a more scalable regime. This can be particularly important if collective MPI operations are being employed. Finally, only one process per node will be involved in I/O (in contrast to the pure MPI case where potentially 32 processes per compute node could be performing I/O), which will significantly reduce the number of meta data operations on the file system at large process counts for those applications based on files-per-processes I/O strategy. Therefore, the use of hybrid OpenMP/MPI will decrease the total memory footprint per compute node, the number of communications across the network fabric, the total volume of data to write to disk, and the total number of metadata operations given the applications' files-per-process I/O strategy.

In this report, we highlight our progress in implementing hybrid MPI/OpenMP versions of several software packages, namely the Fluidity-ICOM unstructured-grid ocean model, the NEMO structured-grid ocean model, the DL_MG multigrid solver, and RAD, one of a suite of programs based on the 'R-

[1] OpenMP Application Program Interface version 4.0 http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

matrix' ab initio approach to variational solution of the many-electron Schrodinger equation. We demonstrate that utilising non-blocking algorithms and libraries are critical to mixed-mode applications so that they can achieve better parallel performance than the pure MPI versions.

CASE 1: Developing hybrid MPI/OpenMP parallelism for next generation ocean modelling software: Fluidity-ICOM

Fluidity-ICOM is built on top of Fluidity[2], an adaptive unstructured finite element code for computational fluid dynamics. It consists of a three-dimensional non-hydrostatic parallel multiscale ocean model, which implements various finite element and finite volume discretisation methods on unstructured anisotropic adaptive meshes so that a very wide range of coupled solution structures may be accurately and efficiently represented in a single numerical simulation without the need for nested grids. It is used in a number of different scientific areas including geophysical fluid dynamics, computational fluid dynamics, ocean modelling and mantle convection. Fluidity-ICOM uses state-of-the-art and standardised 3rd party software components whenever possible. For example, PETSc [3] is used for solving sparse linear systems while Zoltan[4] is used for many critical parallel data-management services both of which have compatible open source licenses. Python is widely used within Fluidity-ICOM at run time for user-defined functions and for diagnostic tools and problem setup. It requires in total about 17 other third party software packages and use three languages (Fortran, C++, Python). Fluidity-ICOM is coupled to a mesh optimisation library allowing for dynamic mesh adaptivity.

Previous performance analysis [1] has already shown that the two dominant simulation costs are sparse matrix assembly 30%-40% of total computation, and solving the sparse linear systems defined by these equations. Therefore, the sparse matrix assembly kernels and the spare linear solvers are the most important components to be parallelised using OpenMP.

The non-blocking finite element matrix assembly approach

Non-blocking finite element matrix assembly can be realised through well-established graph colouring techniques. This is implemented by first forming a graph, where the nodes of the graph correspond to mesh elements, and the edges of the graph define data dependencies arising from the matrix assembly between elements. Each colour then defines an independent set of elements whose term can be added to the global matrix concurrently. This approach removes data contention, so called critical sections in OpenMP, allowing very efficient parallelisation.

To parallelise matrix assembly using graph colouring techniques [2], a loop over colours is first added around the main assembly loop. The main assembly loop over elements will be parallelised using the OpenMP parallel directives with a static schedule. This will divide the loop into chunks of size ceiling (number_of_elements/number_of_threads) and assign a thread to each separate chunk. Within this loop an element is only assembled into the matrix if it has the same colour as the colour iteration.

[2] http://amcg.ese.ic.ac.uk/index.php?title=Fluidity
[3] http://www.mcs.anl.gov/petsc
[4] http://www.cs.sandia.gov/Zoltan

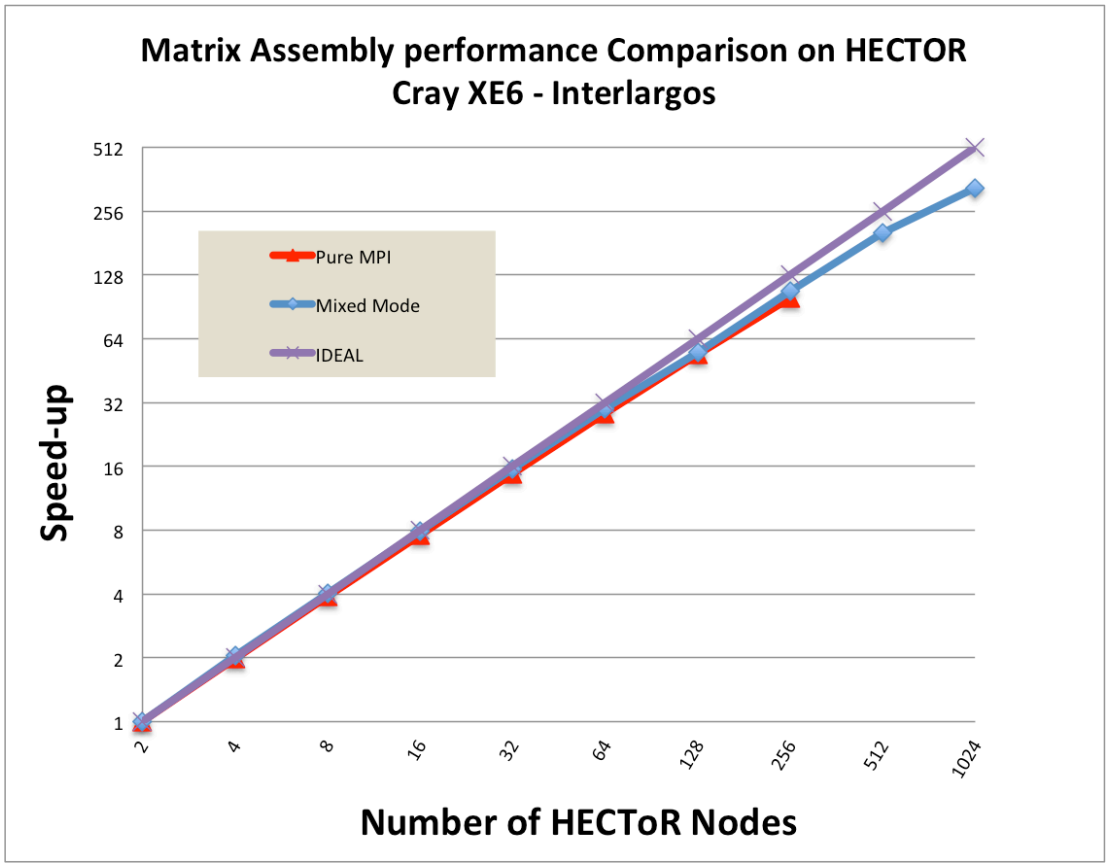


Figure 1: Matrix Assembly Performance Comparison on HECToR XE6-Interlagos. All hybrid modes use 4 MPI ranks per node and 8 threads per rank.

Figure 1 shows that matrix assembly scales well up to 32768 cores. The speedup of the mixed-mode implementations is 107.1 compared with 99.3 for pure MPI when using 256 nodes (8192 cores). This is due to the use of local assembly which makes this part of the code essentially a local process. The hybrid mode performs slightly better than pure MPI, which can scale well up to 32768 cores. The details can be seen in the references [3, 4].

The sparse linear systems defined by various equations are solved by using threaded PETSc and HYPRE is utilised as a threaded preconditioner through the PETSc interface. Since unstructured finite element codes are well known to be memory bound, particular attention has to be paid to ccNUMA architectures where data locality is particularly important to achieve good intra-node scaling characteristics. Figure 2 shows the total Fluidity-ICOM run-time and parallel efficiency. Clearly pure MPI runs faster up to 2048 cores. However, due to the halo size increasing exponentially with number of MPI tasks, the cost of MPI communication becomes dominant from 4096 cores onwards, where the mixed mode begins to outperform the pure MPI version.

With a full implementation of mixed mode MPI/OpenMP and previous several years efforts[1][2], Fluidity-ICOM can now run well above 32K cores job, which offers Fluidity-ICOM the capability to solve the "grand-challenge" problems.

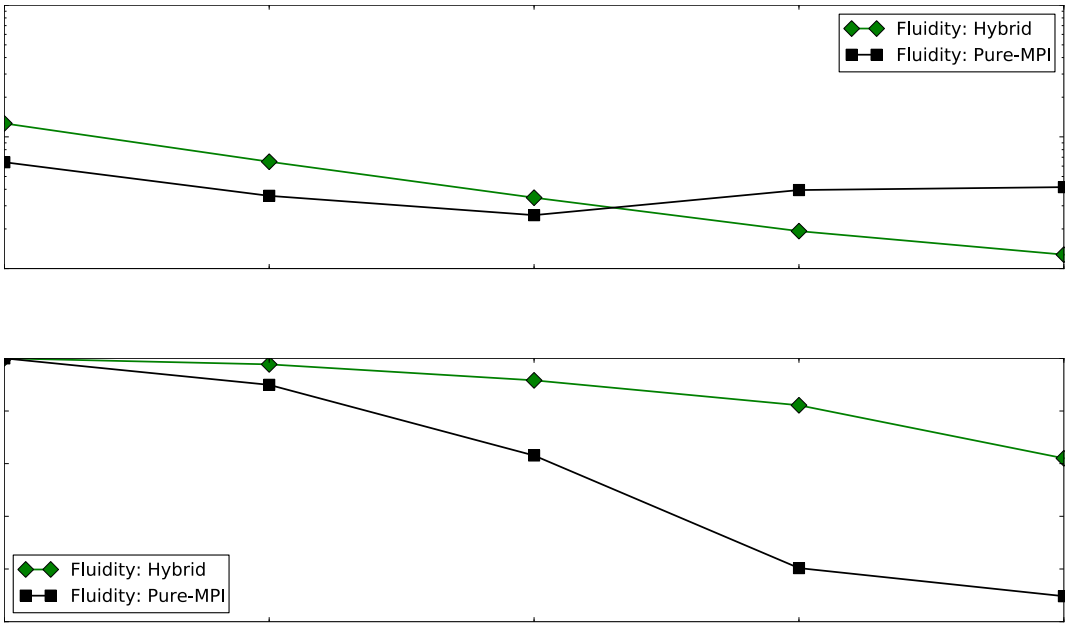


Figure 2: Strong scaling results for the whole Fluidity-ICOM up to 256 XE6 nodes (8192 cores). All hybrid modes use 4 MPI ranks per node and 8 threads per rank.

CASE 2: Developing hybrid MPI/OpenMP parallelism for NEMO

In conjunction with work under the PRACE 2IP project [5], we have investigated various strategies for using OpenMP within NEMO[5] (Nucleus for European Modelling of the Ocean). NEMO uses a finite-difference scheme on a simply-connected grid in latitude/longitude to solve the fundamental equations governing ocean behaviour [6]. It is parallelised for distributed-memory machines using the Message Passing Interface (MPI).

In common with many oceanographic and atmospheric modelling codes, NEMO is memory-bandwidth limited. It also has a very flat profile with no single routine accounting for more than about 5% of runtime. We have therefore chosen to experiment with two routines; *tra_ldf_iso* and *tra_adv_tvd.* The former has a simple structure and deals with lateral diffusion of tracers (temperature and salinity) while the latter advects the tracers forwards in time using the TVD (Total Variation Diminishing) scheme. For the global, 2-degree resolution model incorporating sea ice, this routine and its child (which is called twice) accounted for 7.6% of run-time on 12 cores of a Cray XT. Unless stated otherwise, all results here are for the ORCA2 grid (182×149×31).

As with many NEMO routines, *tra_adv_tvd* consists of many triply-nested loops interspersed with calls to the MPI library to exchange halo information. Arrays in NEMO are constructed with the longitude/*x* dimension first, the latitude/*y* dimension second and the depth/*z* index third. Therefore, for a typical nested loop, the innermost loop is over *x* and the outermost over *z*. The *tra_ldf_iso* routine is simpler and does not contain any halo swaps. Therefore implementing an OpenMP version of it enables us to investigate the effect of the thread synchronisation required by the halo exchanges in *tra_adv_tvd.*

[5] http://www.nemo-ocean.eu/

We have experimented with two different OpenMP implementations which we term 'loop-level' and 'tiled.' In both cases, the whole body of the routine including its child is enclosed within a PARALLEL region to save costs associated with thread-team creation and destruction. The loop-level implementation is the simplest and consists of enclosing each loop within OpenMP DO...ENDDO directives. For triply-nested loops, best performance is obtained by using the COLLAPSE directive to parallelise over all the iterations of the outer two loops while leaving the innermost loop to be auto-vectorised by the compiler. The latter makes use of the SIMD parallelism available in hardware.

For the tiled implementation, the 2D domain belonging to the process is decomposed into a regular grid of tiles. To reduce inter-tile dependencies, tiles are overlapped in exactly the same way as the MPI sub-domains. The number of tiles is configurable at run time and may be greater than or equal to the number of OpenMP threads available. Use of a greater number of smaller tiles can bring benefits in cache usage which is very valuable in a memory-bandwidth-limited code. It can also be used to improve load balance across threads when combined with the use of dynamic OpenMP scheduling.

In addition to the two OpenMP approaches, we have also developed a version of NEMO with array indices re-ordered such that the vertical *z*-index is first or fastest-varying. Nested loops are then re-ordered such that z is innermost and y outermost. This has the advantage that the trip-count of the vectorisable inner-most loop is now independent of the number of OpenMP threads (and indeed MPI processes).

We have tested the performance of these various approaches on a selection of hardware hosted at STFC's Daresbury Laboratory; dual-socket Intel Westmere and Sandybridge systems, the Intel Xeon Phi (Knights Corner) and the IBM Power 7 CPU. For the simpler *tra_ldf_iso* kernel the Intel Phi exceeds the performance of the two-socket, 16 (physical) cores, 2.6 GHz Sandy Bridge host system. The effect of saturating the memory bandwidth of a single Sandy Bridge socket can be seen in the plateau of the red line at 8 threads in Figure 3.

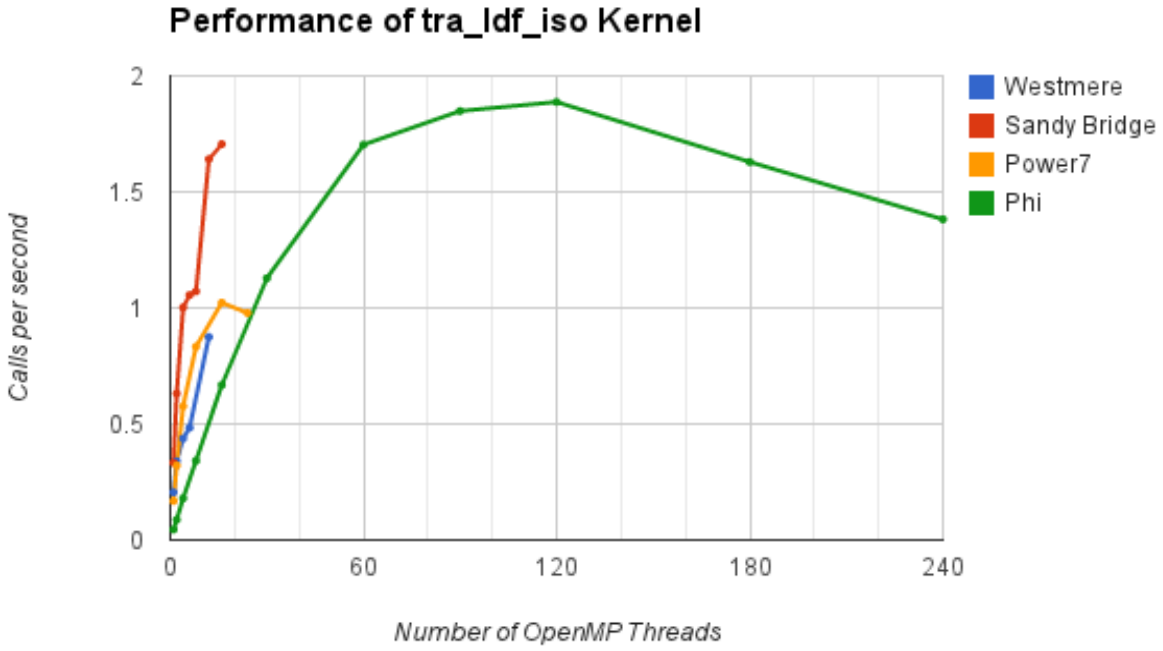


Figure 3: Performance of the tra_ldf_iso kernel on a variety of processor hardware.

Changing now to the more complex *tra_adv_tvd* kernel with its need for repeated thread synchronisation, we see in Figure 4 that the Phi only attains 79% of the performance of a *single* Sandy Bridge socket (eight threads).

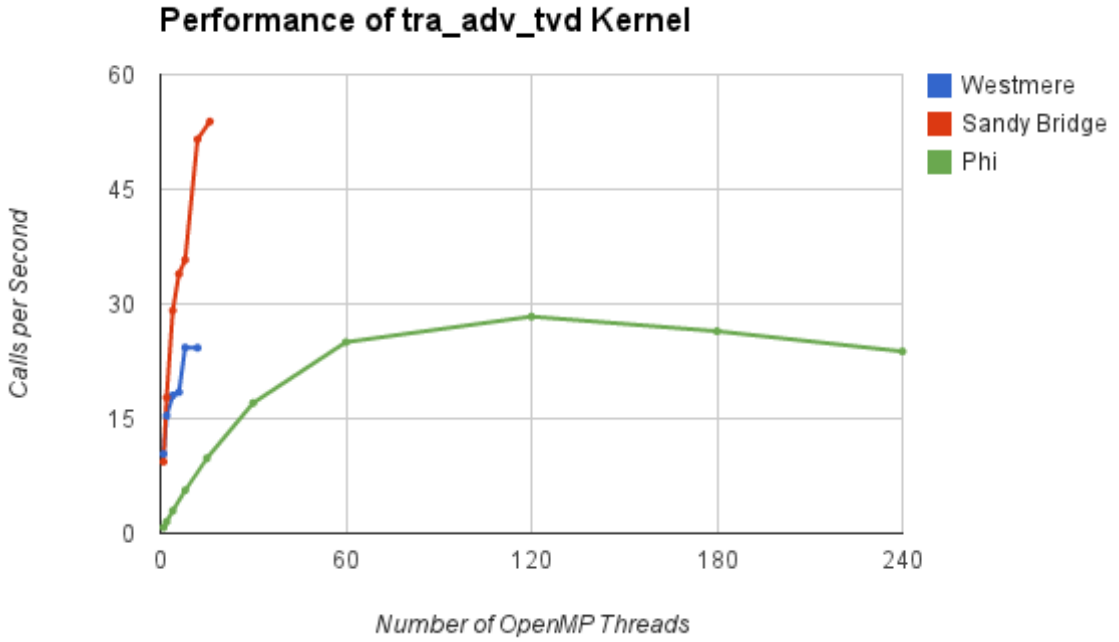


Figure 4: Performance of the tra_adv_tvd kernel on Intel Westmere Sandy Bridge and Xeon Phi processors. Loop-level OpenMP is used for parallelisation.

For the tiled implementation of the kernel, we searched for the optimum number of tiles at each thread count. This version is a clear improvement over the loop-level version on the Sandy Bridge (compare Figure 5 with Figure 4). However, this is not the case on the Xeon Phi where the kernel now runs about 30% more slowly.

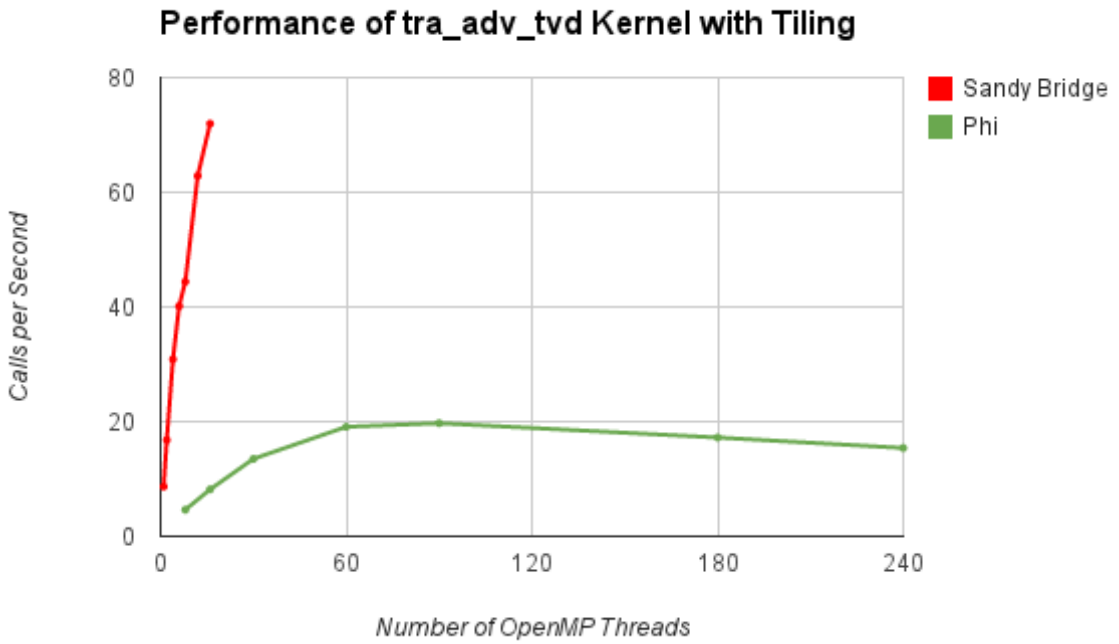


Figure 5: Performance of the tra_adv_tvd kernel when work is broken into tiles and shared amongst OpenMP threads.

Since this is in the z-last ordering (where it is the *x*-dimension that is vectorised) and with the relatively small ORCA2 grid, using large numbers of threads (and hence tiles) inevitably has a dramatic effect on the length of the vectors involved in the calculation, i.e. lots of tiles with small extent in the *x* dimension will not make best use of the Phi's support for 512-bit vectors. This is supported by the performance improvement seen (Figure 6) on the Phi when the array-index ordering is changed such that it is the *z*-dimension that is vectorised (z-first).

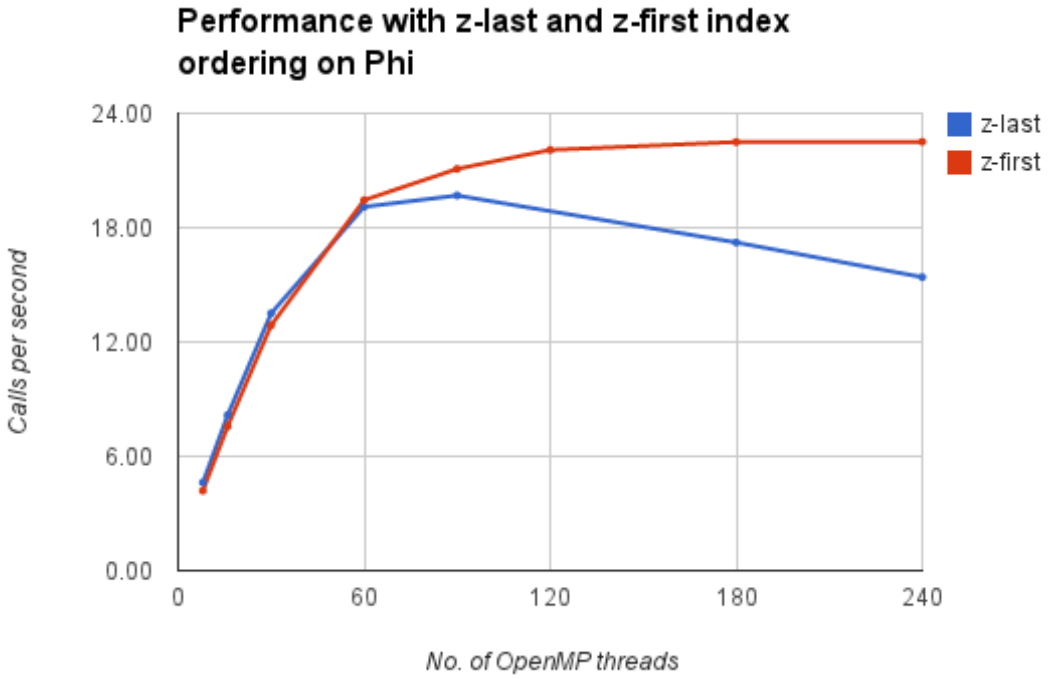


Figure 6: Effect of array-index ordering on kernel performance on the Intel Xeon Phi.

Despite this improvement the performance of the z-first version with tiling is still only 79% of the loop-level version of the kernel. The reasons for this are still under investigation.

Figure 7 emphasises the performance improvement obtained from tiling when running on the Sandy Bridge. That better use is made of cache and available memory bandwidth can be seen by the great reduction in the plateau at eight threads when the first socket becomes full.

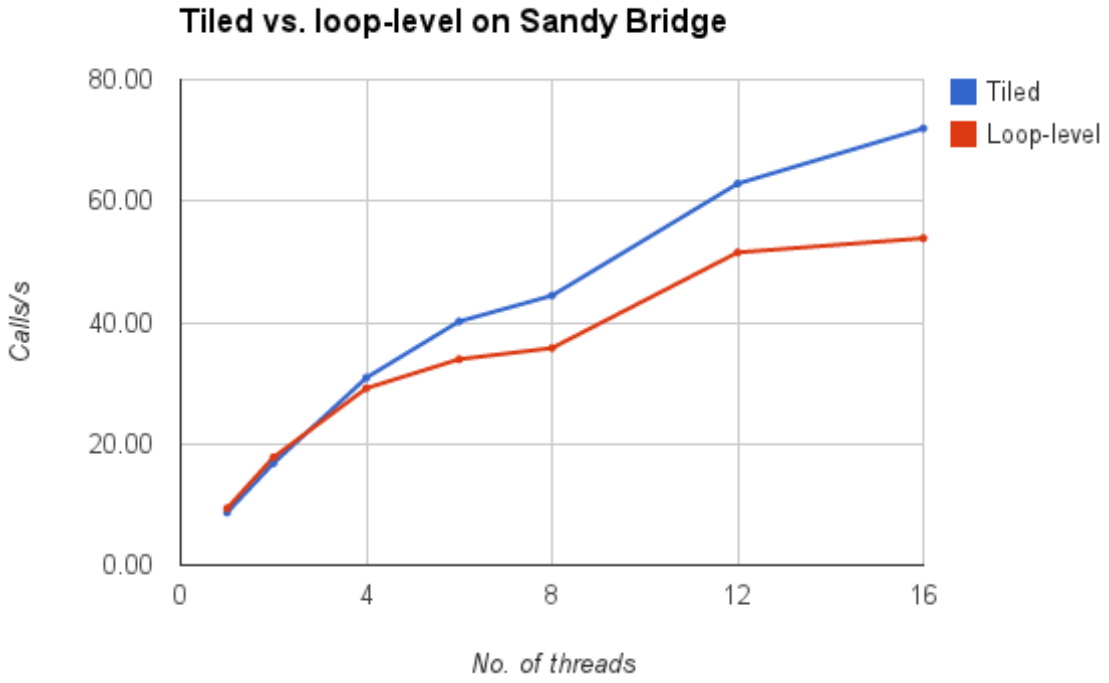


Figure 7: Comparison of performance obtained using tiling and loop-level parallelism for the tra_adv_tvd kernel running on a dual-socket Intel Sandy Bridge system.

Now we consider the results when running with a larger data set in order to investigate weak scaling. The ORCA1 grid used here has dimension 362×292×46.We can see from Figure 8 that the Intel Phi now performs more strongly relative to Sandy Bridge - it achieves 51% of the Sandy Bridge Performance. Presumably, performance on the Phi is aided by the greater extent of the vectorisable *z* dimension on this grid since it has 46 levels rather than the 31 of ORCA2.

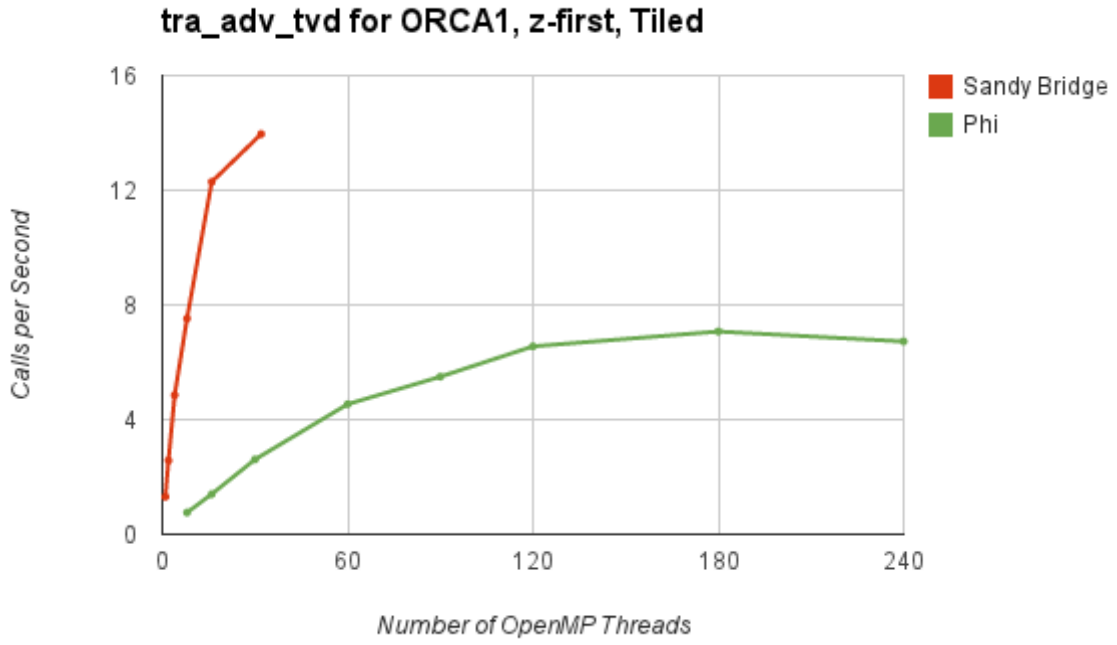


Figure 8:Performance of the tra_adv_tvd kernel with the larger ORCA1 mesh on Intel Sandy Bridge and Intel Xeon Phi.

Notes on Tools and Environment

When we began this work Amplifier was not available for the Knights Ferry platform and the Intel team at Versailles recommended that we use the TSC register for fine-grained timing. We therefore profiled by manually inserting timing calls in the code. This approach was perfectly adequate given the small size of the kernels. We have tried Amplifier on the Knights Corner platform and had some success once we'd worked through the difficulty of instructing it to run the code on the Phi rather than the host. However, in the absence of detailed hardware counter information (e.g. for cache usage and vector instructions) it is actually no more useful than our own timing code. Hopefully the functionality of Amplifier on the Phi will be extended in the future.

CASE 3: Mixed Mode Scaling For 3D Red-Black Smoother

Multigrid techniques are very useful for fast solution of large scale linear and non-linear system of equations derived by discretisation of from several classes of partial derivatives equations [7]. The algorithms used by multigrid have multiple level of parallelism, e.g. : the grid over which the solution is sought can be partition to MPI tasks which exchange only halos with the topological neighbours, the update operation for smoothing, restriction and prolongation can be parallelised with OpenMP threads.

Amongst the equation of interest that can be solved with multigrid techniques, Poisson Equation (PE) and Poisson-Boltzmann Equation (PBE) have a central position. In classic and quantum mechanical simulation the solution of PE or PBE are needed to model systems with electrostatic interaction. For this kind of problems speed and scalability are fundamental requirement for a suitable multigrid solver as the electrostatic potential needs to be computed repeatedly during MD simulation or quantum mechanical iterative solution steps.

This is a preliminary study for the scaling behaviour of two versions of a parallel mixed mode (MPI+OpenMP) red-black smoother which is part of a multigrid solver currently being developed in a Hector dCSE project for the quantum density functional code ONETEP.

Overlapping and non-overlapping mixed mode

The current high performance computing systems are based on multicore nodes linked by high bandwidth and low latency networks. As all cores on a node share the memory it is appealing to replace MPI with OpenMP inside the node in order to save communication time. However in practice it was found that mixed mode implementation needs a detail analysis in order to find the optimal algorithm. In general this depends on hardware characteristics (intra and inter node bandwidth, CPU speed and cache sizes) and algorithm characteristics, most importantly being the computational intensity [8].

In order understand better and quantify the interplay of MPI and OpenMP this study presents the scaling performance for two patterns of mixed mode parallelism for a 3D red-black smoother: a) non-overlapping: this algorithm has distinct phase for MPI communication and computation separated by a OpenMP barrier, b) overlapping: communication is handled by the master thread while the local domain values are updated by other threads. The main steps of algorithm are presented in Figure 9.

```
!$OMP SINGLE
!copy data into MPI buffers
…
!$OMP END SINGLE NOWAIT

!$OMP MASTER
! do MPI isend irecv and wait
…
!$OMP END MASTER
!$OMP BARRIER

!$OMP DO SCHEDULE(STATIC)
! iterate over the inside of
! the local domain
…
!$OMP END DO NOWAIT

!$OMP SINGLE
! compute boundary values with data
! received for MPI neighbours
…
!$OMP END SINGLE NOWAIT
```

```
!$OMP SINGLE
!copy data into MPI buffers
…
!$OMP END SINGLE NOWAIT

!$OMP MASTER
! do MPI isend, irecv and wait
…
!$OMP END MASTER

! distribute work to other threads
thid = omp_get_thread()
chunk = loop_trip/nthreads

loop_start = (thid-1) * chunk +1
loop_end = thid * chunk
do jk = loops_start, loop_end
! iterate over the inside of
! the local domain
…
enddo
!$OMP BARRIER
! we need to wait here to ensure that
! halo data has arrived

!$OMP SINGLE
! compute boundary values with data
! received from MPI neighbours
…
!$OMP END SINGLE NOWAIT
```

Figure 9. The main steps of the non-overlapping (left) and overlapping (right) algorithms in a generic parallel grid iterator with halo exchange.

Numerical experiment details

The grid used is 3D cube containing 500^3 points which is representative for the grids used currently by ONETEP[9], the timings where collected for 10 red-black iteration steps.

The global grid was distributed to the MPI ranks according to the following rules:

$2^{n1} \times 2^{n2} \times 2^{n3}$ = Ncores/Nthreads,

with $n_1 \le n_2 \le n_3,\ 0 \le n_j - n_i \le 1,\ j > i.$

For Blue Gene/Q two threads per core were used in certain cases in order to test if multithreading is beneficial for the studied smoother. We mention also that the test code used local grids without halo regions, because ONETEP grids don't use halos regions. Hence one swap update consists of two phases: 1) Update of the inner domain and 2) update of the surface layer using separated buffers for the halo data.

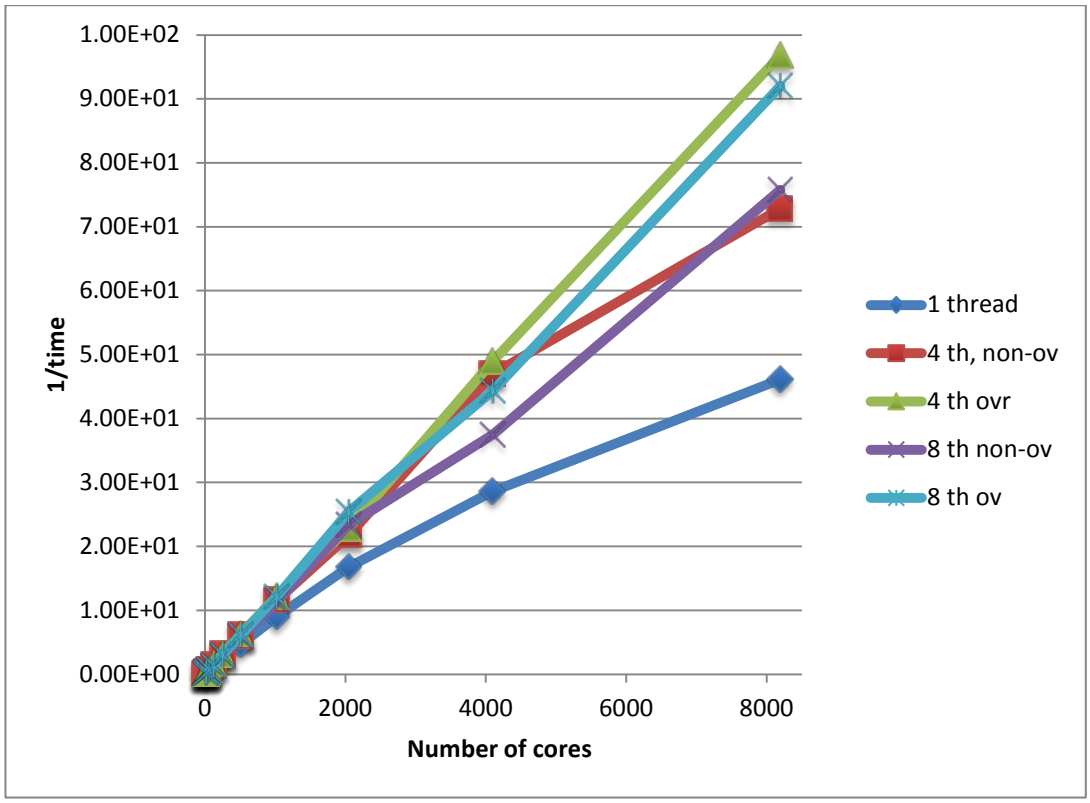


Figure 10 Scaling data for HECToR (Cray-XE6), overlapping and non-overlapping algorithms with 4 and 8 OpenMP threads.

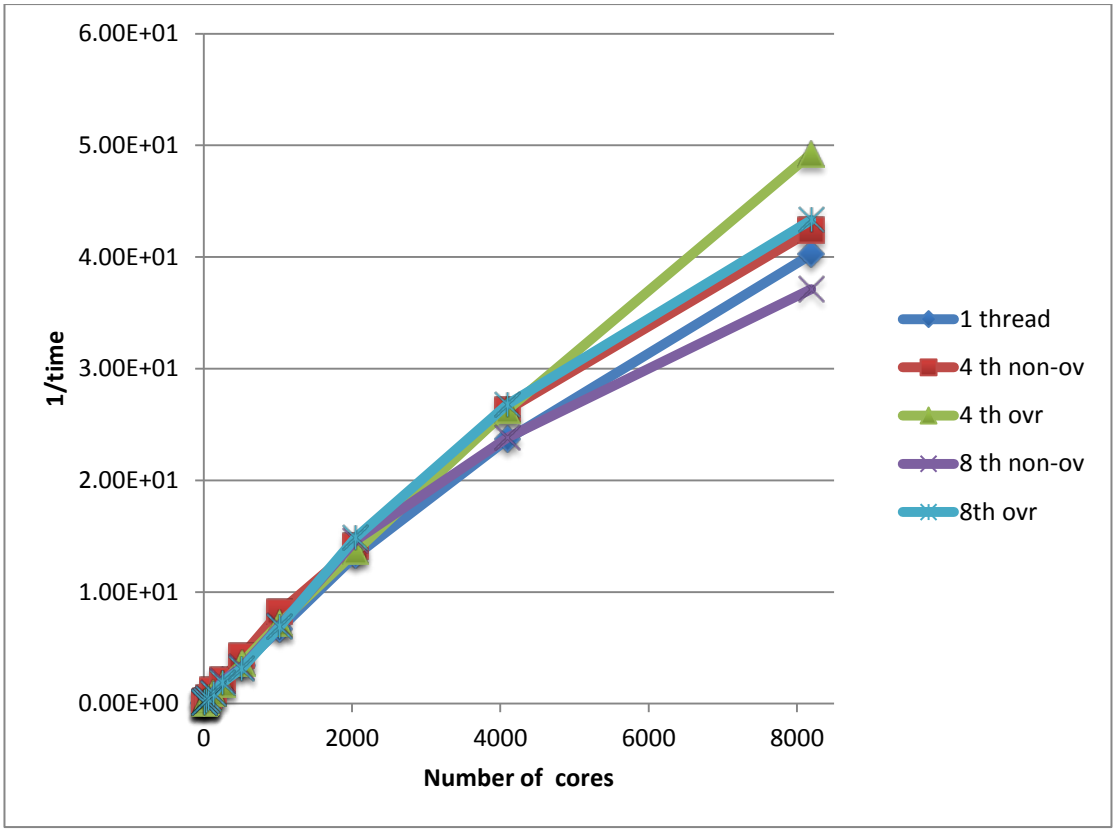


Figure 11 Scaling data for Blue Joule, (Blue Gene/Q system), overlapping and non-overlapping algorithms with 4 and 8 OpenMP threads.

Results discussion

Timing data were collected on HECToR (Cray XE6) and Blue Joule (Blue Gene/Q) systems.

Figures 10 and 11 show that the mixed mode algorithm performs better at large core counts, as one would expect from a simple estimate of computation communication ratios. The best scaling overlapping algorithm is notably faster on the Cray XE6 by a factor around 2 for large core counts. The Blue Gene/Q system shows less variance of performance over the tested configurations; however the fastest algorithm is also the overlapping version by a factor around 1.25.

All collected data are presented in Annex, Tables 1-8; the communication timings are presented separated in Tables 2, 4, 6 and 8.

CASE 4: Developing hybrid MPI/OpenMP parallelism for PRMAT

The application and its major characteristics

The accurate computation of much of the data required in astrophysics, plasma and optical physics presents huge computational challenges, even on the latest generation of high-performance computer architectures. A suite of programs based on the 'R-matrix' ab initio approach to variational solution of the many-electron Schrodinger equation has been developed and has enabled much accurate scattering data to be produced [9]. However, current and future calculations will require substantial increases in both the numbers of channels and scattering energies involved. The scattering energy independent 'inner region' in which intricate configuration interaction Hamiltonian

and multipole matrices for the many-electron system are constructed (and diagonalized) has been the focus of code optimisation projects. Inner-region radial integrals and angular couplings are calculated separately then combined together.

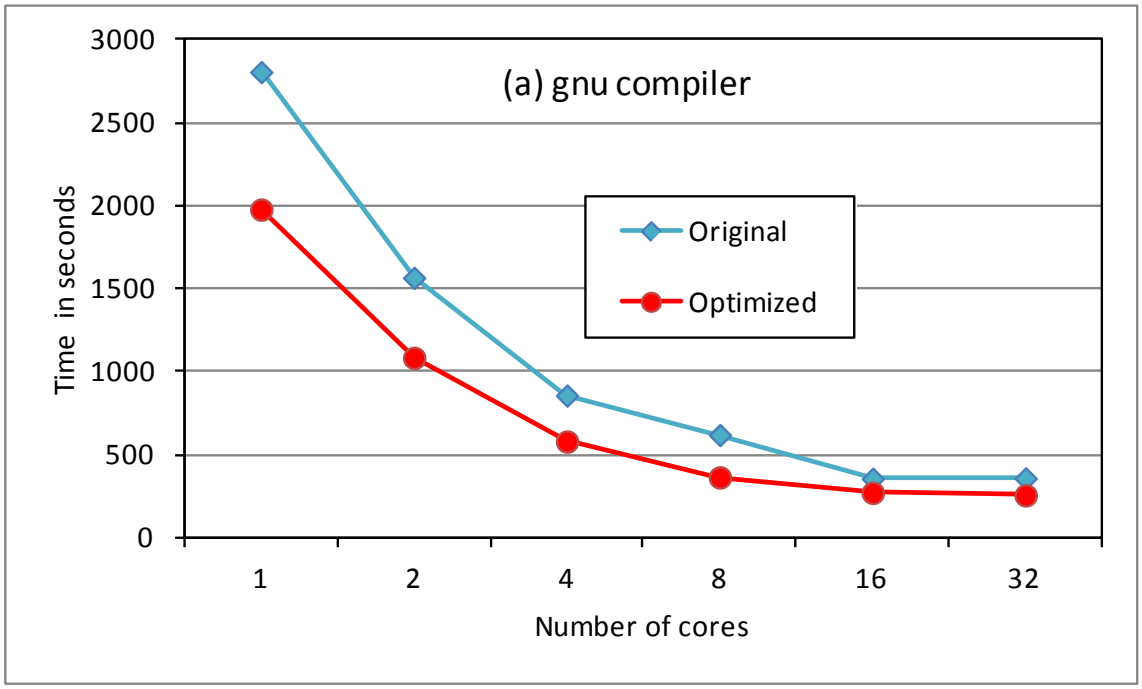
OpenMP-MPI and Parallel I/O Implementation

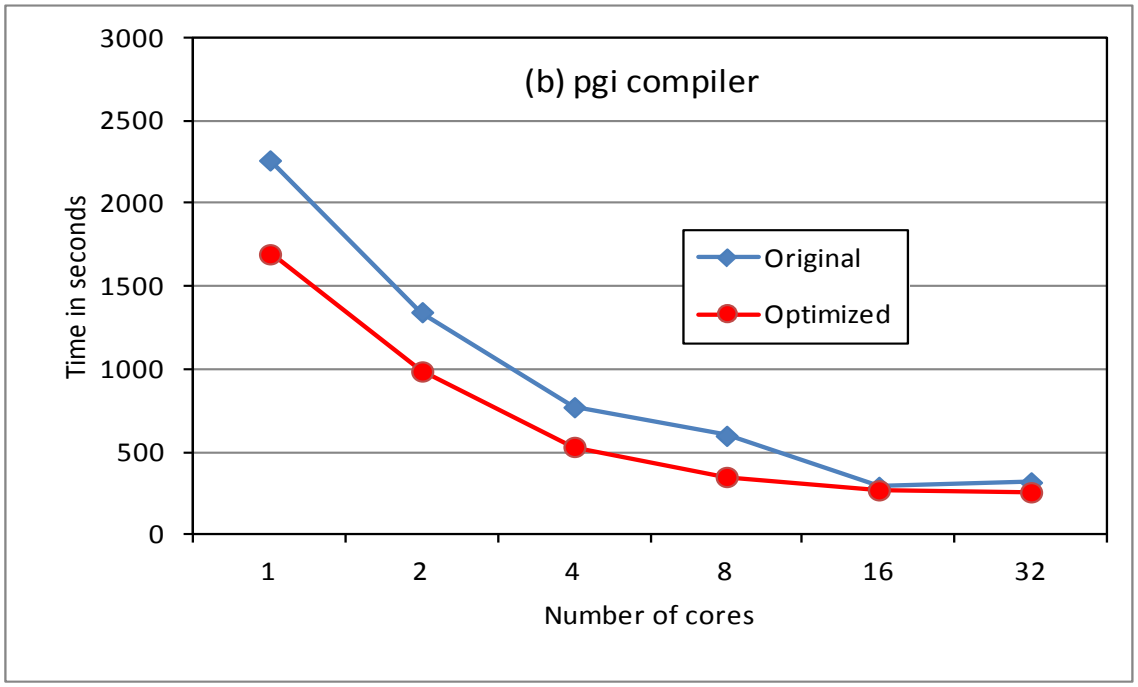
Serial, and in certain cases OpenMP-parallelized codes in the inner-region suite, have been extended to full many-node parallelism using a mixture of (a) mixed-mode MPI plus OpenMP techniques, and (b) pure MPI with intra-node shared memory segments and object-oriented Fortran 2003 [10]. Both methods take full advantage of the multicore nature of modern architectures. The three inner-region 'construction' codes RAD, ANG and HAM now scale across multiple Shared-memory processor nodes. We have also developed two utility packages in object-oriented Fortran 2003: a shared memory segment package (with associated semaphores, intra- and inter- (virtual) node communicators etc.) and a parallel I/O package adapted using asynchronous MPI-IO from an existing serial double-buffered direct access package: this allows independent parallel reading and writing combined with straightforward access to non-contiguous selections from large amounts of stored data. These utility packages are of course particularly suited to the R-matrix codes but are also of general interest.

We will focus here on describing briefly the OpenMP version of RAD. Our main modification was to re-order loops in the second half of the code that calculate 2-electron exchange integrals over four orbitals and an inter-electronic potential term $\{r_<^k/r_>^{(k+1)}\}$ ($r_<$ is the lesser or r_1 and r_2). The algorithmic set-up of the code is such that where possible the inner integral is stored for a particular combination of orbitals to avoid recalculation, however the orbital index ordering for the exchange integrals ran counter to this and much unnecessary recalculation was being done. The loop reordering was performed while maintaining the 'correct' expected ordering for the writing to the output files. In addition, the integrals were in many cases written individually to the serial filehand output buffers: while the buffering mechanism in filehand compensated to a reasonable extent, this has been rearranged so that filehand routines can be called less frequently for arrays of integrals. We also collapsed some inner loops and put in an option to move the OpenMP loop up one level, hopefully to reduce OpenMP overheads.

OpenMP Implementation Performance Results

Results are shown in Figure 12 for an oxygen test case using 180 B-spline functions per angular momentum on the Hector Cray XE6 national UK facility. We compare original and revised code execution time for up to 32 threads using gnu, pgi and cray compilers (at –O3 level, -fast for pgi).


(a) gnu compiler
Time in seconds
3000
2500
2000
1500
1000
500
0
Original
Optimized
1
2
4
8
16
32
Number of cores


(b) pgi compiler
Time in seconds
3000
2500
2000
1500
1000
500
0
Original
Optimized
1
2
4
8
16
32
Number of cores

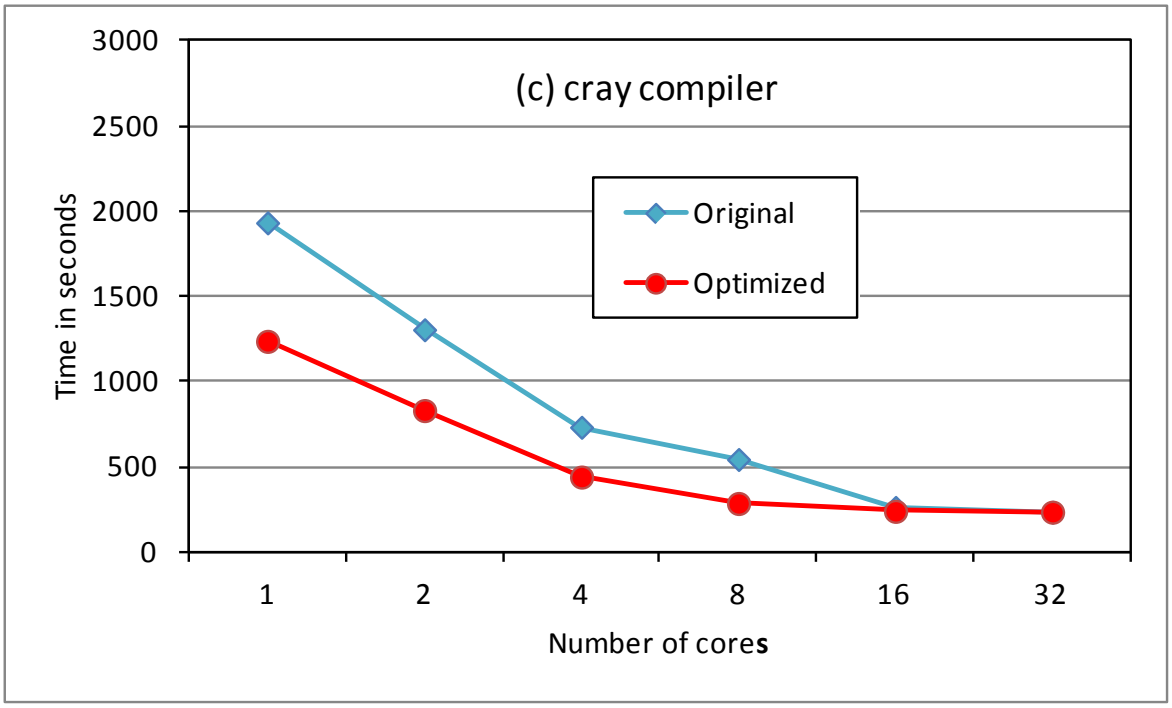


Figure 12. RAD OpenMP tests with the GNU (a), PGI (b) and Cray (c) compilers on the Cray XE6.

The loop reordering provides substantial serial optimization. Above 8 cores, the storage option slows down the scaling. We are investigating returning the OpenMP loop to its previous position for 16 and 32 cores (i.e. so that less data needs to be held in memory). The Cray compiler is much better suited to the Cray XE6 architecture for this code, though this may not be the case on other platforms. Note that all tests were performed on a full node (i.e. the few-thread runs had access to all the memory on the node if required, etc.).

The results above include the writing (and re-reading) of sets of orbitals to temporary files. This approach has traditionally been used in the serial code in order to minimize memory overheads on former memory-limited architectures. However, the latest HPC architectures, such as the Cray XE6, provide relatively generous memory provisions (32GB) per node, and therefore this out-of-core storage approach has been replaced with on-core (memory) storage in the new parallel code.

Hybrid MPI/OpenMP Implementation Performance Results

We now consider results (for the optimized code) using the Cray compiler following the introduction of the MPI communications. The main code developments for implementing this new level of parallelization in RAD involve:

i. distributing input configurations to all MPI Tasks at the start of the run,
ii. the replacement of temporary files with memory-based storage and
iii. a distribution of outer loop angular momentum l values amongst MPI tasks for both basis function and radial integral calculations.

Note that the last option not only allows for use of more than one node, but also can improve overall performance by determining optimal MPI task/ OpenMP thread combinations within a node.

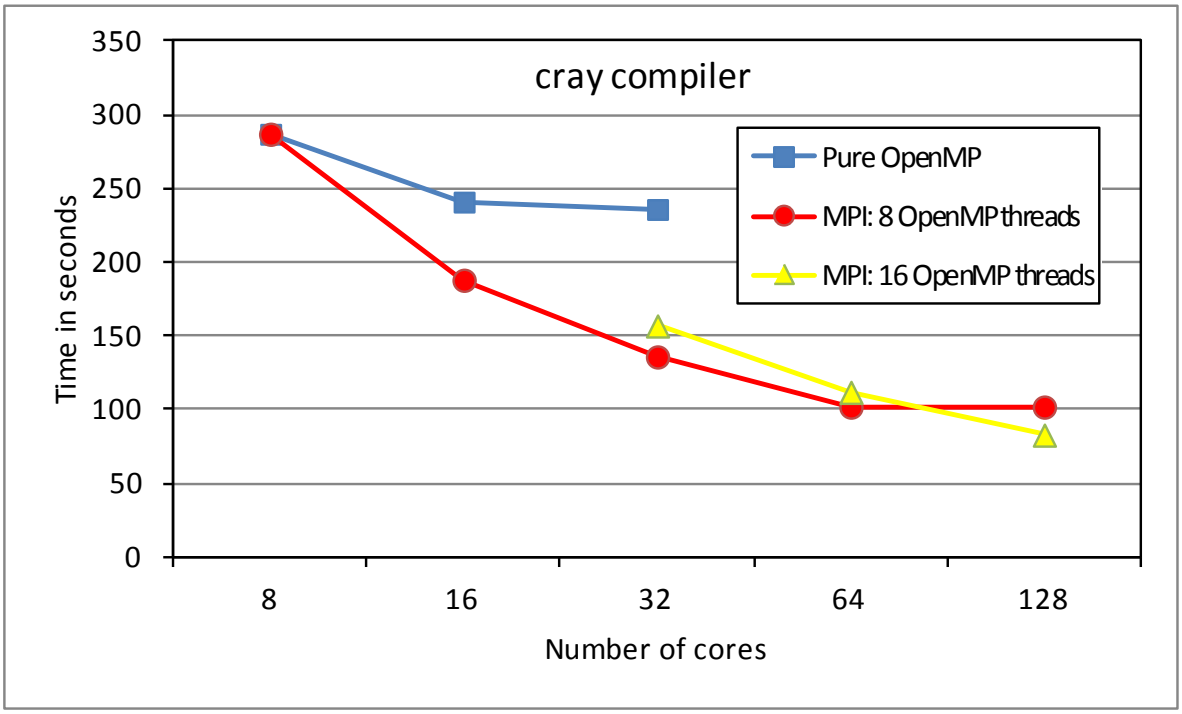


Figure 13. RAD mixed-mode timings using the Oxygen test case.

Figure 13 shows three cases, the pure OpenMP results for {8, 16, 32} threads already shown above, results for 8 OpenMP threads each for {1, 2, 4, 8, 16} MPI tasks (i.e. {1, 1, 1, 2, 4} nodes), and results for 16 OpenMP threads each for {2, 4, 8} MPI tasks (i.e. {1, 2, 4} nodes).

It may be seen that for this Oxygen test case, the intermediate writing to disk did not significantly hold up performance (or its removal has compensated for any (minimal) overhead for one task introduced by the MPI code). The introduction of the MPI tasks, complementary to the OpenMP parallelization, has significantly improved the overall performance within one node, and the performance scales acceptably to two nodes.

The lack of scaling between 2 and 4 nodes for 8 threads per task may be explained as due to the test case: the outer loop parallelized with MPI has 8 independent (after appropriate preliminary data transfer) iterations. Thus we now show multi-node results for an expanded case (more realistic) in Figure 14 with 16 independent outer loop iterations (note the linear horizontal scale for this figure). We have {4, 8, 12, 16} and {2, 4, 6, 8} MPI tasks respectively.

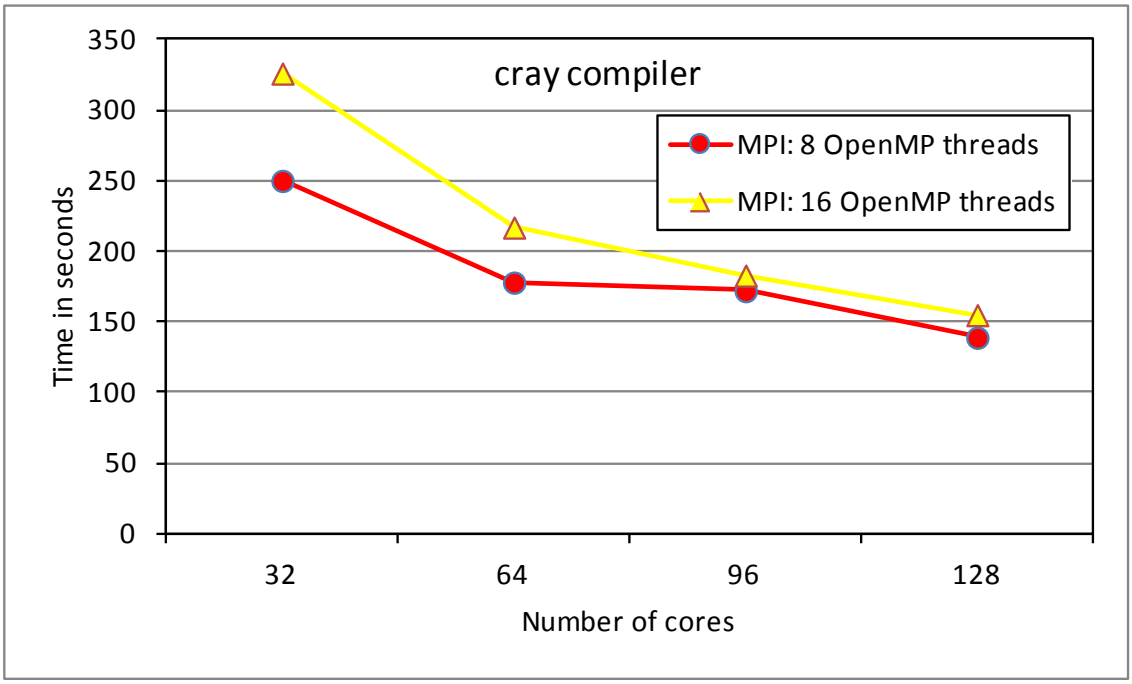


Figure 14. RAD mixed-mode tests (16 loop iterations)

While the 4-node run is now faster than the 2-node run, load-balancing of tasks to iterations is affecting the scaling. The iterations were assigned to tasks in round-robin fashion and while the overall time to completion was stable for repeated testing, the load-balancing between threads became progressively worse: from about 10% difference for 2 tasks, to some tasks taking nearly twice as long as others for 16 tasks. These timing differences reflect the different computational loads associated with different values assigned to the different MPI tasks.

The main objective for extending the OpenMP parallelization in RAD to a mixed mode MPI-OpenMP parallelization was to enable more ambitious calculations to be undertaken, both in terms of memory availability and parallel performance. To this end, the hybridization was a success and was relatively straightforward to implement. One of the main challenges involved dismantling the tightly integrated 'out of core' (highly serial) programming strategies that had been built into the code in order to mitigate the memory limitations of previous HPC systems.

Conclusions

As the current OpenMP standard (3.0), which has been implemented by most popular compilers, does not cover page placement, memory-bound applications, e.g. CASE 1:Fluidity-ICOM and Case 4: RAD, require explicit memory placement. Therefore it is important to ensure that memory fetches are mapped into the locality domains of the individual processors that actually access the data. This was achieved by implementing a first touch policy to minimize NUMA traffic across the network. Thread pinning was then used to guarantee that threads are bound to physical CPUs and maintain locality of data access.

For high core counts simulation, the I/O becomes a major bottleneck for pure MPI versions of these codes. Significant increases in efficiency of I/O, based on a files-per-process strategy has been achieved by using mixed mode parallelism.

From the implementations of the four software packages, we also demonstrate that utilising non-blocking algorithms, e.g. graph colouring in Fluidity-ICOM, array index-reordering in NEMO, and red-black Gauss-Seidel and the use of libraries, e.g. TCMalloc, are critical to mixed-mode applications achieving better parallel performance than the pure MPI version. This can require significant efforts to re-engineering the software packages, and some kernels have had to be rewritten to break data dependencies.

For those applications with requirements of MPI communication within an OpenMP region, overlapping communication with computation proved to be the most efficient strategy.

References

[1] Xiaohu Guo, G. Gorman, M. Ashworth, S. Kramer, M. Piggott, A. Sunderland, “High performance computing driven software development for next-generation modelling of the Worlds oceans”, Cray User Group 2010: Simulation Comes of Age (CUG2010), Edinburgh, UK, 24th-27th May 2010

[2] Xiaohu Guo, G. Gorman, M Ashworth, A. Sunderland, “Developing hybrid OpenMP/MPI parallelism for Fluidity-ICOM - next generation geophysical fluid modelling technology”, Cray User Group 2012: Gre-engineering the Future (CUG2012), Stuttgart, Germany, 29th April - 3rd May 2012

[3] Xiaohu Guo, Gerard Gorman, Michael Lange, Lawrence Mitchell, Michele Weiland, “Exploring the Thread-level Parallelisms for the Next Generation Geophysical Fluid Modelling Framework Fluidity-ICOM”, Procedia Engineering, 2013, vol 6, pp 251-257

[4] Xiaohu Guo, Gerard Gorman, Michael Lange, Lawrence Mitchell, Michele Weiland, “Developing the multi-level parallelisms for Fluidity-ICOM -- Paving the way to exascale for the next generation geophysical fluid modelling technology”, submitted to the Advances in Engineering Software Journal

[5] C. Gheller et al., “D8.1.2: Performance Model of Community Codes”, PRACE 2IP Final Version 1.0, http://www.prace-ri.eu/IMG/pdf/D8-1-2_2IP.pdf

[6] G. Madec, "NEMO ocean engine", Note du Pole de modélisation, Institut Pierre-Simon Laplace (IPSL), France, No 27 ISSN No 1288-1619, 2008

[7] U. Trottenberg, C. W. Oosterlee, A. Shuller, “Multigrid”, Academic Press 2001.

[8] Georg Hager, Gerhard Wellein , “Introduction to High Performance Computing for Scientists and Engineers”, CRC Press, Taylor & Francis Group, 2011

[9] A.G. Sunderland, C.J. Noble, V.M. Burke, P.G. Burke, "A parallel R-matrix program PRMAT for electron–atom and electron–ion scattering calculations”, Vol 145, Issue 3, Computer Physics Communications, 2002

[10] C.J. Noble, A.G. Sunderland, M. Plummer, “Combined-Multicore Parallelism for the UK electron-atom scattering Inner Region R-matrix codes on HECToR”, dCSE Report, Hector Website, http://www.hector.ac.uk/cse/distributedcse/reports/prmat2/

Appendix A

This section contains timing data collected on HECToR (Cray XE6) and Blue Joule (Blue Gene/Q) for ONETEP.

Compilers used: Cray on Hector with default optimization, IBM on Blue Gene/Q with –O4 –qhot optimisation flags.

Timings are averaged over MPI ranks.

Column 32 in Tables 5, 6, 7, 8 (Blue Joule) presents values for runs that use 2 hyperthreads per core in certain configurations.

Table 1 Total time in seconds for 10 iterations in smoother for non-overlapping algorithm with various distribution of cores over MPI ranks and OpenMP threads on HECToR. Grid size 500^3. Light green values mark the fastest combination of MPI ranks and OpenMP threads for a given number of cores.

Ncores	Number of threads			
	1	2	4	8
8	1.19E+01	1.16E+01	1.17E+01	
16	6.16E+00	6.03E+00	6.04E+00	6.00E+00
32	2.95E+00	3.12E+00	3.07E+00	3.03E+00
64	1.38E+00	1.34E+00	1.55E+00	1.58E+00
128	7.17E-01	6.63E-01	6.63E-01	7.90E-01
256	3.84E-01	3.48E-01	3.20E-01	3.45E-01
512	2.12E-01	1.79E-01	1.66E-01	1.69E-01
1024	1.11E-01	9.80E-02	8.63E-02	8.81E-02
2048	5.95E-02	4.73E-02	4.61E-02	4.28E-02
4096	3.50E-02	2.32E-02	2.13E-02	2.67E-02
8192	2.17E-02	1.37E-02	1.37E-02	1.32E-02

Table 2 Communication times for halo exchange for run described in

Ncores	Number of threads			
	1	2	4	8

8	5.65E-02	3.68E-02	3.82E-02	
16	2.47E-01	1.28E-01	2.57E-01	1.23E-01
32	5.50E-01	2.06E-01	1.33E-01	9.47E-02
64	2.58E-01	1.27E-01	1.18E-01	6.62E-02
128	1.53E-01	9.02E-02	5.08E-02	3.39E-02
256	9.75E-02	7.55E-02	2.90E-02	1.99E-02
512	6.00E-02	4.44E-02	2.28E-02	1.17E-02
1024	4.10E-02	2.61E-02	1.35E-02	9.63E-03
2048	2.61E-02	1.41E-02	1.00E-02	7.23E-03
4096	1.87E-02	7.97E-03	5.32E-03	8.41E-03
8192	1.43E-02	6.43E-03	6.53E-03	5.07E-03

Table 3 Smoother time on HECToR for overlapping algorithm, run parameters as in

Ncores	Number of threads			
	1	2	4	8
8	1.19E+01	1.26E+01	1.18E+01	
16	6.09E+00	5.96E+00	5.91E+00	5.90E+00
32	2.96E+00	3.08E+00	2.87E+00	2.94E+00
64	1.37E+00	1.44E+00	1.44E+00	1.48E+00
128	7.11E-01	7.66E-01	6.29E-01	7.43E-01
256	3.72E-01	3.87E-01	3.16E-01	3.22E-01
512	2.08E-01	1.97E-01	1.59E-01	1.62E-01
1024	1.08E-01	1.10E-01	8.17E-02	8.20E-02
2048	5.85E-02	5.22E-02	4.38E-02	3.93E-02
4096	3.35E-02	2.57E-02	2.05E-02	2.26E-02
8192	2.13E-02	1.42E-02	1.03E-02	1.09E-02

Table 4 Communication times on HECToR for non-overlapping algorithm, run parameters described in

Ncores	Number of threads			
	1	2	4	8
8	5.82E-02	2.13E-01	2.55E-01	
16	2.17E-01	6.11E-01	8.49E-01	2.19E+00
32	5.57E-01	1.87E+00	1.10E+00	4.80E-01
64	2.41E-01	5.87E-01	4.49E-01	2.83E-01
128	1.58E-01	4.54E-01	1.85E-01	1.83E-01
256	8.78E-02	2.36E-01	1.21E-01	7.37E-02
512	5.95E-02	1.21E-01	5.01E-02	5.37E-02
1024	4.02E-02	7.03E-02	2.74E-02	2.66E-02
2048	2.61E-02	2.35E-02	2.06E-02	1.46E-02
4096	1.77E-02	1.41E-02	1.06E-02	1.29E-02
8192	1.42E-02	8.98E-03	5.98E-03	7.14E-03

Table 5 Smoother time on Blue Gene/Q for non-overlapping algorithm.

Ncores	Threads					
	1	2	4	8	16	32
1	8.74E+01					
8	1.27E+01	1.12E+01	1.11E+01	1.22E+01		
16	7.36E+00	7.24E+00	6.78E+00	7.26E+00	8.29E+00	8.28E+00
32	3.38E+00	3.67E+00	3.77E+00	3.63E+00	4.16E+00	
64	1.78E+00	1.67E+00	1.85E+00	2.17E+00	2.09E+00	
128	9.37E-01	8.87E-01	8.47E-01	1.07E+00	1.37E+00	
256	4.71E-01	4.65E-01	4.77E-01	4.82E-01	6.65E-01	

512	2.93E-01	2.36E-01	2.34E-01	3.06E-01	3.01E-01	
1024	1.50E-01	1.43E-01	1.22E-01	1.43E-01	2.03E-01	1.98E-01
2048	7.54E-02	7.47E-02	7.11E-02	6.84E-02	9.26E-02	8.52E-02
4096	4.22E-02	4.10E-02	3.80E-02	4.20E-02	4.69E-02	8.54E-02
8192	2.49E-02	2.75E-02	2.36E-02	2.69E-02	3.00E-02	2.85E-02

Table 6 Communication times on Blue Gene/Q for non-overlapping algorithm.

Ncores	Threads					
	1	2	4	8	16	32
1	1.90E-04					
8	6.42E-02	1.14E-01	5.23E-03	1.82E-04		
16	7.94E-02	6.05E-02	3.21E-02	1.02E-02	1.76E-04	1.82E-04
32	5.22E-02	4.03E-02	5.92E-02	2.26E-02	9.00E-03	
64	5.02E-02	2.06E-02	1.94E-02	1.96E-02	2.09E-02	
128	3.47E-02	2.16E-02	1.07E-02	1.34E-02	1.85E-02	
256	1.69E-02	1.51E-02	1.37E-02	9.76E-03	1.11E-02	
512	2.22E-02	8.61E-03	8.93E-03	1.09E-02	8.63E-03	
1024	1.08E-02	8.37E-03	5.68E-03	6.95E-03	6.88E-03	6.70E-03
2048	5.70E-03	5.23E-03	7.78E-03	4.93E-03	4.58E-03	6.03E-03
4096	2.86E-03	3.92E-03	3.28E-03	5.23E-03	2.83E-03	6.19E-03
8192	2.31E-03	3.18E-03	2.75E-03	3.18E-03	3.58E-03	4.02E-03

Table 7 Smoother time on Blue Gene/Q for overlapping algorithm.

	Threads					
Ncores	1	2	4	8	16	32
1	8.86E+01					
8	1.31E+01	2.15E+01	1.47E+01	1.39E+01		

16	7.38E+00	1.20E+01	7.43E+00	7.27E+00	8.33E+00	8.38E+00
32	3.41E+00	6.04E+00	4.32E+00	3.63E+00	4.18E+00	
64	1.81E+00	3.05E+00	2.14E+00	2.21E+00	2.09E+00	
128	9.46E-01	1.48E+00	1.08E+00	1.08E+00	1.35E+00	
256	4.75E-01	7.53E-01	5.69E-01	5.29E-01	6.55E-01	
512	2.97E-01	3.74E-01	2.73E-01	3.18E-01	3.13E-01	
1024	1.52E-01	2.11E-01	1.39E-01	1.45E-01	2.01E-01	1.93E-01
2048	7.64E-02	1.04E-01	7.33E-02	6.73E-02	9.06E-02	8.30E-02
4096	4.28E-02	5.16E-02	3.80E-02	3.73E-02	4.48E-02	8.31E-02
8192	2.52E-02	3.03E-02	2.03E-02	2.31E-02	2.58E-02	2.30E-02

Table 8 Communication time for halo exchange on Blue Gene/Q, overlapping algorithm.

	Threads					
Ncores	1	2	4	8	16	32
1	1.70E-04					
8	6.62E-02	1.53E-01	1.03E-02	1.88E-04		
16	8.89E-02	1.35E-01	1.64E-01	8.10E-03	3.17E-04	5.80E-04
32	5.19E-02	7.57E-02	1.02E-01	2.33E-02	1.37E-02	
64	5.16E-02	8.28E-02	5.39E-02	1.17E-01	2.02E-02	
128	3.21E-02	1.82E-01	3.71E-02	5.81E-02	1.20E-01	
256	1.70E-02	1.02E-01	8.71E-02	3.58E-02	5.66E-02	
512	2.25E-02	4.35E-02	4.01E-02	4.73E-02	4.45E-02	
1024	1.11E-02	2.57E-02	1.68E-02	2.28E-02	2.84E-02	2.68E-02
2048	5.82E-03	1.35E-02	1.14E-02	1.02E-02	1.42E-02	1.62E-02
4096	2.81E-03	7.56E-03	5.81E-03	8.24E-03	7.35E-03	1.69E-02
8192	2.30E-03	3.59E-03	3.92E-03	3.76E-03	6.54E-03	5.43E-03