Developing Hybrid OpenMP-MPI Parallelism for Fluidity - Next Generation Geophysical Fluid Modelling Technology

Xiaohu Guo*, Gerard Gorman†, Andrew Sunderland*and Mike Ashworth*
* *Science and Technology Facilities Council*
Daresbury Laboratory, Daresbury Science and Innovation Campus
Warrington, Cheshire WA4 4AD, UK.
Email: xiaohu.guo@stfc.ac.uk
†*Department of Earth Science and Engineering*
Imperial College London
London SW7 2AZ, UK.
Email: g.gorman@imperial.ac.uk

***Abstract*—Most modern high performance computing platforms can be described as clusters of multi-core compute nodes. The trend for compute nodes is towards greater numbers of lower power cores, with a decreasing memory to core ratio. This is imposing a strong evolutionary pressure on numerical algorithms and software to efficiently utilise the available memory and network bandwidth.**

Unstructured finite elements codes have been effectively parallelised with domain decomposition methods by using libraries such as the Message Passing Interface for a long time. However, there are many algorithmic and implementation optimisation opportunities when threading is used for intra-node parallelisation for the latest multi-core/many-core platforms. For example, reduced memory requirements, cache sharing, reduced number of partitions and less MPI communication. While OpenMP is promoted as being easy to use and allows incremental parallelisation of codes, naive implementations frequently yield poor performance. In practice, as with MPI, equal care and attention should be exercised over algorithm and hardware details when programming with OpenMP.

In this paper, we report progress implementing hybrid OpenMP-MPI for finite element matrix assembly within the unstructured finite element application software named Fluidity. The OpenMP parallel algorithm uses graph colouring to identify independent sets of elements that can be assembled simultaneously with no race conditions. Unstructured finite element codes are well known to be memory bound, therefore, particular attention is paid to ccNUMA architectures where data locality is particularly important to achieve good intra-node scaling characteristics. The profiling and the benchmark results on the latest CRAY platforms show that the best performance can be achieved by pure OpenMP within a node.

***Keywords*-Fluidity; FEM; OpenMP; MPI; ccNUMA; Graph Colouring;**

I. Introduction

Fluidity[1] is an open source, general purpose, multi-phase CFD application capable of solving numerically the Navier-Stokes and accompanying field equations on arbitrary unstructured finite element meshes in one, two and three dimensions. It uses a moving finite element/control volume method which allows arbitrary movement of the mesh with time dependent problems. It offers a wide range of finite element/control volume element choices together with mixed formulations. It is being used in a diverse range of geophysical fluid flow applications. Fluidity uses three languages (FORTRAN, C++, Python), is fully parallelised using MPI and uses state-of-the-art and standardised 3rd party software components whenever possible. Fluidity is coupled to a mesh optimisation library allowing for dynamic mesh adaptivity.

The change of shifting from the clock speed race to using multi-core/many-core processors is as disruptive to scientific software as the shift from vector to distributed memory supercomputers decades ago. The shift to multi-core/many-core systems is driving applications to exploit much higher levels of fine-grained parallelism and overcome significant reductions in the bandwidth and volume of memory available to each CPU. This scalability challenge driven by the exponential increase in the amount of parallelism in the system affects all aspects of the use of high performance computing.

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). Finally, only one process per node will be involved in off-node communications (in contrast to the pure MPI case where potentially 32 processes per node could be communicating on Phase 3 of HECToR). Depending on the I/O strategy there could also be a significant reduction in the number of meta data operations on the file system at large process counts.

[1]http://amcg.ese.ic.ac.uk/Fluidity

Therefore, the use of hybrid OpenMP-MPI will decrease the total memory footprint per compute node, decrease the load on the inter-processor communications network and decrease the total number of meta data operations given Fluidity's files-per-process I/O strategy.

For modern multi-core architecture supercomputers, hybrid OpenMP-MPI also offers new possibilities for optimisation of numerical algorithms beyond pure distributed memory parallelism. For example, scaling of algebraic multigrid 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 critical for application software development, and directive-based approaches are an excellent way to express parallelism in a portable manner. They offer potential capabilities for using the same code base to explore accelerated and non-accelerator enabled systems because OpenMP is expanding its scope to embedded systems and accelerators. Therefore, there is strong motivation to further develop OpenMP parallelism in Fluidity to exploit current and future architectures.

However, writing a truly efficient OpenMP-MPI scalable program is entirely non-trivial, despite the apparent simplicity of the incremental parallelisation approach. This paper will demonstrate how we tackle the race conditions and performance pitfalls during OpenMP parallelisation.

The remaining part of this paper is organised as follows. In the next section we describe in detail the finite element matrix assembly and greedy colouring method. Section III will address thread safety issues during OpenMP parallelisation and performance gained by solving these issues. Section IV discusses how we optimise memory bandwidth which is particularly important for OpenMP performance. The section V contains summary and conclusions. There is a discussion of further work in the last section.

II. Finite element matrix assembly

Previous performance analysis [1] has already shown that the two dominant costs in Fluidity are the sparse matrix assembly (30%-40% of total computation), and the solution of the sparse linear systems defined by these equations. The HYPRE[2] library's hybrid sparse linear system solvers/preconditioners, which can be used by Fluidity through the PETSc interface, are competitive with the pure MPI implementation. Therefore, in order to run a complete simulation using OpenMP parallelism, the sparse matrix assembly kernel is now the most important component remaining to be parallelised using OpenMP. The finite element matrix assembly kernel is expensive for a number of reasons including: significant loop nesting, where the innermost loop increases in size with increasing quadrature; many matrices have to be assembled, *e.g.* coupled momentum, pressure, free-surface and one for each advected quantity; indirect addressing; and cache re-use (a particularly severe challenge for unstructured mesh methods). The cost of matrix assembly also increases with higher order, and Discontinuous Galerkin (DG) discretisations are used.

For a given simulation, a number of different matrices need to be assembled, *e.g.* continuous and discontinuous finite element formulations for velocity, pressure and tracer fields for the Navier-Stokes equations and Stokes flow. Each of these have to be individually parallelised using OpenMP. The global matrix to be solved is formed by looping over all the elements of the mesh (or subdomain if this is using a domain decomposition method) and adding the contributions from that element into the global matrix. Sparse matrices are stored in PETSc's (Compressed Sparse Row) CSR containers (these include block-CSR for use with velocity vectors for example and DG). The element contributions are added into a sparse matrix which is stored in CSR format. A simple illustration of this loop is given in algorithm 1.

Algorithm 1 Generic matrix assembly loop.

$global\_matrix \leftarrow 0$
for $e = 1 \rightarrow number\_of\_elements$ **do**
  $local\_matrix = assemble\_element(e)$
  $global\_matrix+ = local\_matrix$
end for

A. Greedy colouring method

In order to thread the assembly loop illustrated in algorithm 1, it is clear that both the operation to assemble an element into a local matrix, and the addition of that local matrix into the global matrix must be thread safe.

This can be realised through well-established graph colouring techniques[2]-[7], and 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 terms can be added to the global matrix concurrently. This approach removes data contention, and therefore removes the need for OpenMP critical or atomic statements, allowing efficient parallelisation.

Generally, we try to colour as many vertices as possible with the first colour, then as many as possible of the uncoloured vertices with the second colour, and so on. To colour vertices with a new colour, we perform the following steps.

1) Select some uncoloured vertex and colour it with the new colour.
2) Scan the list of uncoloured vertices. For each uncoloured vertex, determine whether it has an edge to

[2]http://acts.nersc.gov/hypre/

any vertex already coloured with the new colour. If there is no such edge, colour the present vertex with the new colour.

This approach is called "greedy" because it colours a vertex whenever it can, without considering the potential drawbacks inherent in making such a move. There are situations where we could colour more vertices with one colour if we were less "greedy" and skipped some vertex we could legally colour.

To parallelise the matrix assembly algorithm (1) using colouring, 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 static scheduling. 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.

The threaded assembly loop is summarised in algorithm 2.

Algorithm 2 Threaded matrix assembly loop.

$graph \leftarrow create\_graph(mesh, discretisation)$
$colour \leftarrow calculate\_colouring(graph)$
$k\_colouring = \max(colour)$
$global\_matrix \leftarrow 0$
!$OMP PARALLEL
for $k = 1 \rightarrow k\_colouring$ **do**
  $independent\_elements = \{e | colour[e] \equiv k\}$
  !$OMP DO SCHEDULE(STATIC)
  for all $e \in independent\_elements$ **do**
    $local\_matrix = assemble\_element(e)$
    $global\_matrix+ = local\_matrix$
  end for
  !$OMP END DO
end for
!$OMP END PARALLEL

Generally, using the above colouring method, the number of elements is not balanced between each colour group. For OpenMP, this is not a problem as long as each thread has enough work load. The performance is not sensitive to the total number of colour groups.

III. Performance improvement and analysis

The benchmark tests were carried out on the HECToR Cray XE6-Magny Cours and Cray XE6-Interlagos systems. On the system of CRAY XE6-Magny Cours[3], each node consists of 24 cores sharing a total of 32 GB of memory accessible through a NUMA design. The 24 cores are packaged as 2 AMD Opteron 6172 2.1GHz processors, code-named "Magny-Cours". The total number of processor cores is 44,544. HECToR CRAY XE6-Interlagos system[4] offers a total of 2816 XE6 compute nodes. Each compute node contains two AMD 2.3 GHz 16-core processors giving a total of 90,112 cores; offering a theoretical peak performance of over 800 Tflops. There is presently 32 GB of main memory available per node, which is shared between its thirty-two cores. Both systems use CRAY Gemini communication networks.

The benchmark test case used here is wind-driven baroclinic gyre. The mesh used in the baroclinic gyre benchmark test case has up to 10 million vertices; resulting in 200 million degrees of freedom for velocity due to the use of DG. The basic configuration is set-up to run for 4 time steps and not to adapt. It hence considers primarily the matrix assembly and linear solver stages of a model run. The details of solving equations and configuration can be found in reference [1].

The momentum equation assembly kernel using Discontinuous Galerkin (DG) and Continuous Galerkin (CG) methods has been parallelised with the above-mentioned procedures. Several thread safe issues have been solved resulting in performance benefits.

A. Local assembly versus non-local assembly

For mesh elements that are shared as part of the halo, it is clear that more than one process can contribute to the same rows of the global matrix. PETSc offers two different approaches for handling this. The first approach is called local assembly. This assumes that all processes which have a copy of an element, will assemble that element and add the entries into the PETSc sparse matrix. Internally, PETSc will then quietly drop any values corresponding to rows not owned by the local process as this entry will be handled by the process that does own that row. For this the PETSc parameter *MAT_IGNORE_OFF_PROC_ENTRIES* must be set. The second approach, global assembly, assumes that across all MPI processes each element will only be assembled once. Any entries to the matrix rows which are not owned by the local process are then *stashed*, and communicated to the row owners when *MatAssemblyEnd* is called. Similarly for vectors.

Figures 1 and 2 show the benchmark results comparing local and non-local assembly for the oceanic gyre test case which use DG for momentum and CG for advection/diffusion. For low core counts the difference is negligible. This highlights the fact that the redundant calculations are not significantly impacting performance when local assembly is used. However, at higher core counts, the scaling is significantly better. With 768 cores, the local assembly code is 40% faster, effectively increasing the scaling regime by a factor of two.

[3] www.hector.ac.uk/cse/documentation/Phase2b

[4] http://www.hector.ac.uk/support/documentation/userguide/hardware.php

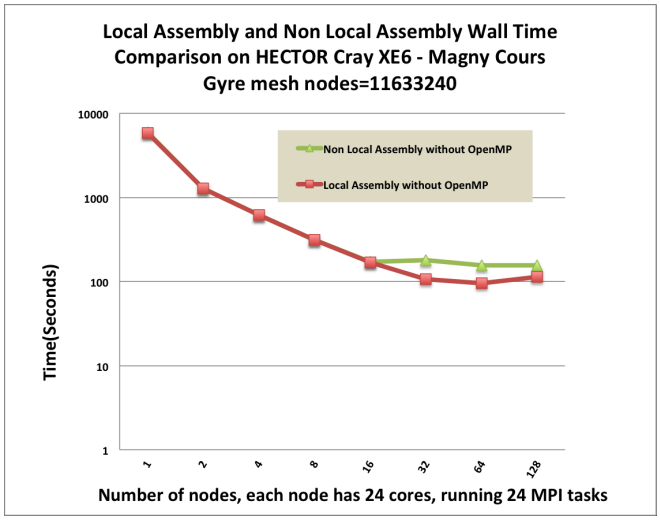

Figure 1. Wall-time for non-local and local assembly are compared. Compute nodes are 24 core AMD Opterons (HECToR Cray XE6-Magny Cours, therefore multiply by 24 to get the number of cores).

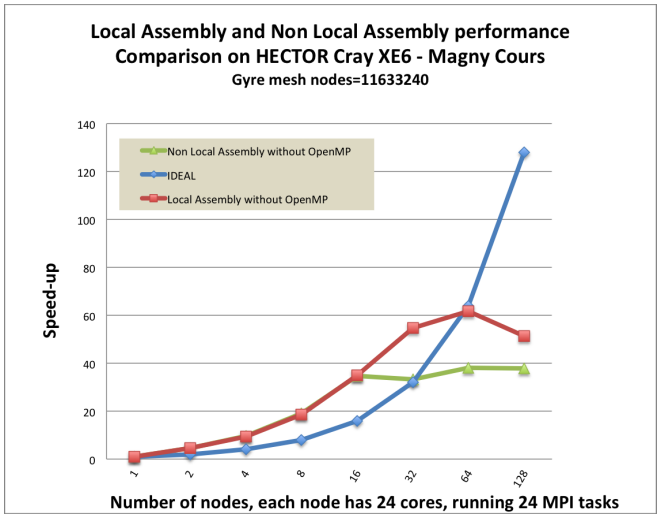

Figure 2. Speedup comparison between matrix local assembly and non-local assembly.

This makes matrix assembly an inherently local process, therefore we can focus on optimising intra-node performance.

B. Thread safety of memory reference counting

For any defined type objects in Fluidity being allocated or deallocated, the reference count will be plus one or minus one. If the objects counter equals zero, the objects should then be deallocated. In general, the element-wise physical quantities should not perform allocation or deallocation in the element loop, but this is not the case in the kernels. The solution could be to either add critical directives around reference counter or move allocation or deallocation outside of element loop. We have implemented both solutions and

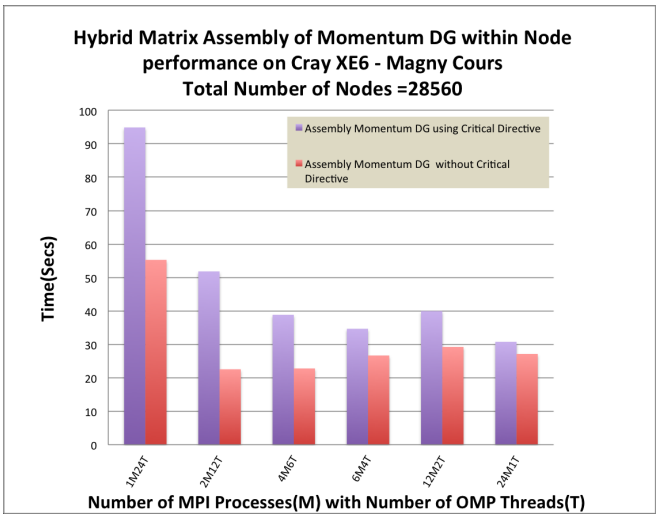

Figure 3. Comparison between using critical directive and without critical directive.

performance comparison has been made in Figure 3 with different OpenMP and MPI combinations within a compute node. Using critical directives, pure MPI outperformed all the other combinations. Without using critical directives, the performance have been improved by more than 50% for 12 and 24 threads. Therefore, the mutual synchronisation directives (*e.g.* critical) should be avoided. Moving allocation or deallocation outside of element loop has also improved the pure MPI versions performance (see 24M1T in Figure 3).

IV. Optimisation of memory bandwidth

Many multi-core machines are typically cache coherent Non-Uniform Memory Architectures (ccNUMA). This means that memory latency (access time) depends on the physical memory location relative to a processor. Within a ccNUMA system, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors, Therefore, it is important that memory should be allocated from mapped local memory nodes, so that software running on the machine may take advantage of ccNUMA locality and therefore have a optimal utilisation of memory bandwidth. To achieve this, the following methods have been employed to ensure good performance:

- First-touch initialisation ensures that page faults are satisfied by the memory bank directly connected to the CPU that raises the page fault.
- Thread pinning to ensure that individual threads are bound to the same core throughout the computation.

The Linux kernel's memory is partitioned by memory node. By default, page faults are satisfied by memory attached to the page-faulting CPU. Because the first CPU to touch the page will be the CPU that faults the page in, this default policy is called *first touch*.

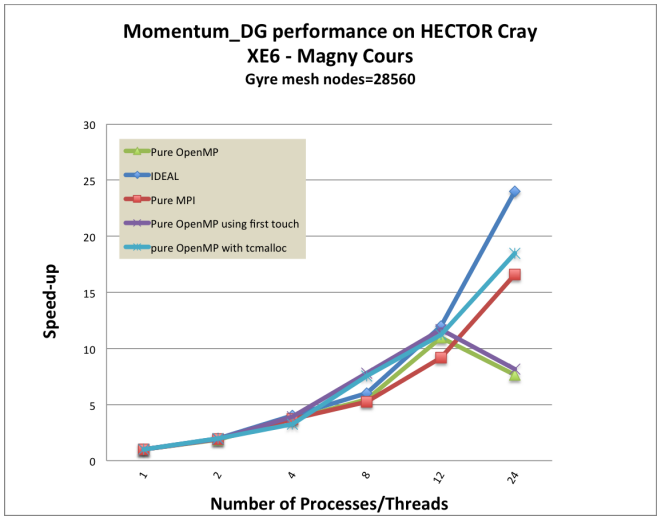

Figure 5. Momentum DG Performance Comparison on HECToR XE6-Magny Cours.

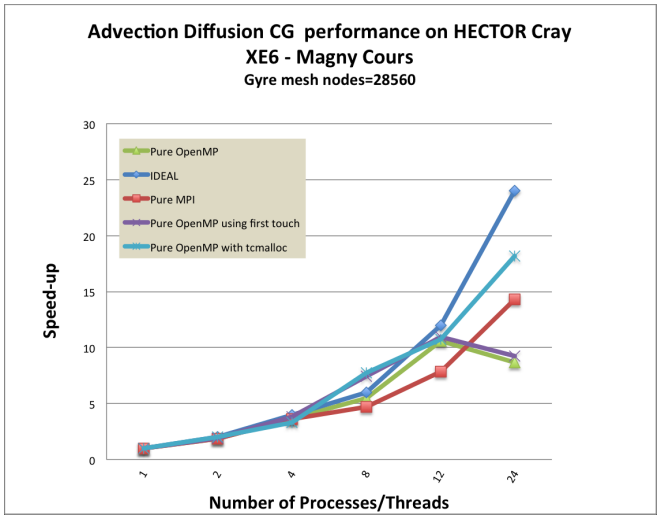

Figure 6. Advection Diffusion CG Performance Comparison on HECToR XE6-Magny Cours.

Thread pinning has been used through Cray *aprun* with all benchmark tests. After applying the first touch policy, the wall time has been reduced from 45.127 seconds to 38.303 seconds using 12 threads on Cray XE6-Magny Cours. From Figure 5 and Figure 6, the speedup has been improved up to 12 threads compared between with and without first touch. But even after applying the first-touch policy, there is still a sharp performance drop from 12 threads to 24 threads.

This problem has been investigated by profiling with CrayPAT. From Figure 4, we can see that the highest costs in *Momentum_DG* are dominated by memory allocation. As we have moved all explicit memory allocation outside of element loop, the culprit appears to be the use of FORTRAN automatic arrays in the *Momentum_DG* assembly kernel for

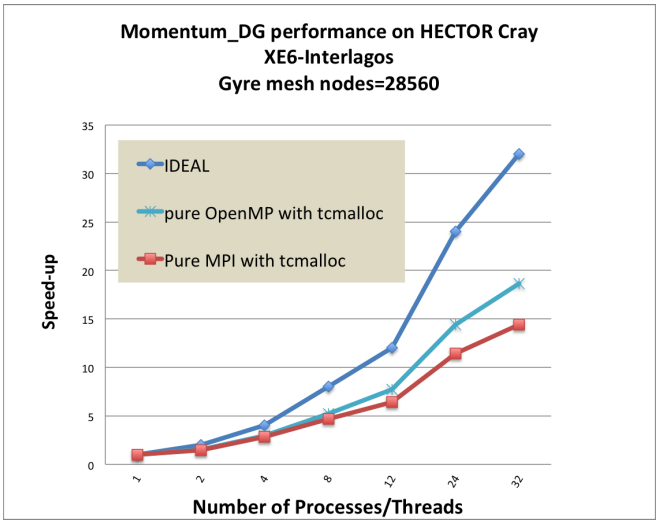

Figure 7. Momentum DG Performance Comparison on HECToR XE6-Interlagos.

support of p-adaptivity. There are a lot of such arrays in the kernel. Since the compiler cannot predict their length, it allocates the automatic arrays on the heap.

The heap memory manager must keep track of which parts of memory have been allocated and which parts of memory are free. In a multi-threaded environment, this task is further complicated by multiple threads requesting allocation or deallocation of memory from the heap memory manager. In order to keep memory allocation thread safe, the typical solution to this is to apply mutual synchronisation methods, *e.g.* a single lock. In a multi-threaded environment, memory allocation by all threads will be effectively serialised by waiting at the same lock.

Thread-Caching malloc(TCMalloc)[5] resolves this problem by using a lock-free approach. It allocates and deallocates memory (at least in some cases) without using locks for synchronization. This delivers a significant performance boost for the pure OpenMP version which is now better than the pure MPI version within compute node. Figure 5 shows that the speedup of the *Momentum_DG* kernel using 24 threads on Cray XE6-Magny Cours is 18.46 compared with 1 thread. On the Cray XE6-Interlagos (Figure 7), the pure OpenMP still performs better than the pure MPI, though the speed up for 24 threads on Cray XE6-Interlagos drop to 14.42 due to the Interlagos's memory bandwidth per core being smaller than the Magny Cours.

We have also compared the different combination of number of MPI tasks and OpenMP threads within Cray XE6-Interlagos compute node. From Figure 8, we can see that 1 MPI task with 32 OpenMP threads is competitive with 2 MPI tasks / 16 threads and 4 MPI tasks / 8 OpenMP threads.

[5]http://goog-perftools.sourceforge.net/doc/tcmalloc.html

Figure 4. CrayPAT Sample Profiling Statistic of *Momentum_DG* with 24 threads.

```
Samp%  | Samp   | Imb.   |  Imb.   |Group
       |        | Samp   | Samp%   | Function
       |        |        |         |  PE=HIDE

 100.0% | 75471 |    -- |     -- |Total
|-------------------------------------------------------------------------
|  95.8% | 72324 |    -- |     -- |ETC
||------------------------------------------------------------------------
||  14.6% | 11002 |  0.00 |   0.0% |_int_malloc
||  13.8% | 10417 |  0.00 |   0.0% |__lll_unlock_wake_private
||   9.7% |  7284 |  0.00 |   0.0% |free
||   9.5% |  7172 |  0.00 |   0.0% |__lll_lock_wait_private
||   6.4% |  4862 |  0.00 |   0.0% |malloc
||   6.2% |  4674 |  0.00 |   0.0% |__momentum_dg_MOD_construct_momentum_element_dg
||   4.0% |  3046 |  0.00 |   0.0% |_int_free
||   3.2% |  2439 |  0.00 |   0.0% |__momentum_dg_MOD_construct_momentum_interface_dg
||   3.0% |  2272 |  0.00 |   0.0% |_gfortran_matmul_r8
||   3.0% |  2251 |  0.00 |   0.0% |__sparse_tools_MOD_block_csr_blocks_addto
||   2.8% |  2090 |  0.00 |   0.0% |malloc_consolidate
||   2.1% |  1574 |  0.00 |   0.0% |__fetools_MOD_shape_shape
```

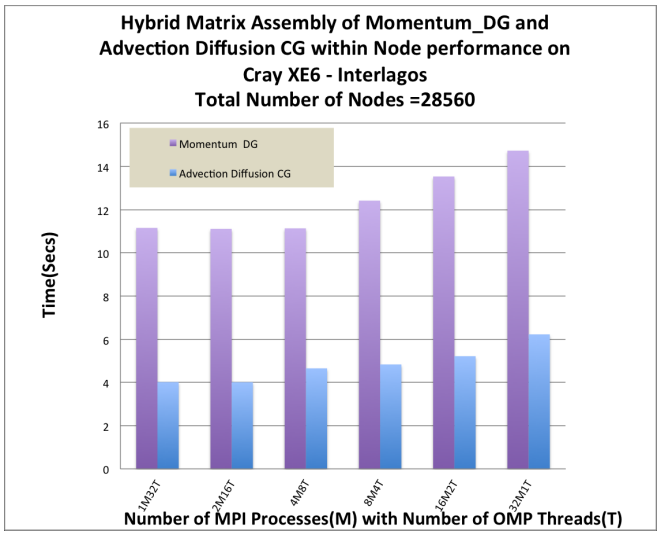

Figure 8. Node Performance Comparison on HECToR XE6-Interlagos.

V. Summary and Conclusions

We have focused on matrix assembly in the Fluidity CFD code. Performance results indicate that node optimisation can be done mostly using OpenMP with an efficient colouring method, which can avoid the use of mutual synchronization directives, *e.g.* critical.

Regarding matrix stashing, it does not have any redundant calculations. However, it does incur the cost of maintaining and communicating stashed rows, and this overhead will increase for higher MPI process counts. A further complication of non-local assembly is that the stashing code within PETSc is not thread safe.

Local assembly has the advantage of not requiring any MPI communications as everything is performed locally, and the benchmark results also highlight the fact that the redundant calculations are not significantly impacting performance when local assembly is used. Furthermore, the scaling of local assembly is significantly better than non-local assembly at higher core counts. This makes assembly an inherently local process. The focus is then on optimising local (to the compute node) performance.

The current OpenMP standard (3.0), which has been implemented by most popular compilers, does not cover page placement. For memory bandwidth bound applications, like Fluidity, it is very important to make sure that memory gets mapped into the local domains of cores that actually access them, minimising ccNUMA traffic across the network. Using thread pinning is critically important to guarantee that those cores which had initially mapped their memory regions maintain locality of access. Our implementation of first touch policy also further improves data locality access.

For the Fluidity matrix assembly kernels, the performance bottle neck becomes memory allocation for automatic arrays. Using a ccNUMA aware heap managers, TCMalloc, can greatly help the pure OpenMP version outperform the pure MPI version within a single compute node.

VI. Future work

More effort is required on solvers, including investigation of threaded HYPRE that can be called through PETSc. We will also be benchmarking using the PETSc development

branch that supports OpenMP. These projects offer multiple avenues for fully parallelising Fluidity with OpenMP.

After this, we will investigate the fully OpenMP parallelised Fluidity on Intel MIC and Cray XK6, which will further guide us the future development.

Work is also ongoing to integrate Fluidity with a newly-developed adaptive mesh library which also supports hybrid OpenMP-MPI parallelisation[6].

ACKNOWLEDGEMENTS

The authors would like to acknowledge the support of a HECToR *distributed Computational Science and Engineering* award. The authors thank the HECToR and NAG support teams for their help throughout this work. The authors would also like to thank Dr. Lawrence Mitchell and Dr. Michele Weiland for their valuable contributions, Dr. Stephan Kramer for sharing his pearls of wisdom with us during code development and Dr. Stephen Pickles, Dr. Andrew Porter and Dr. Michael Seaton for their valuable suggestions and discussions.

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)*, Edingburgh, UK, 24th-27th May 2010

[2] Welsh, D. J. A.; Powell, M. B., *An upper bound for the chromatic number of a graph and its application to timetabling problems*, *The Computer Journal*, 10(1):8586, 1967 doi:10.1093/comjnl/10.1.85

[3] P. Berger, P. Brouaye, J.C. Syre, *A mesh coloring method for efficient MIMD processing in finite element problems*, in: *Proceedings of the International Conference on Parallel Processing*, ICPP'82, August 24-27, 1982, Bellaire, Michigan, USA, IEEE Computer Society, 1982, pp. 41-46.

[4] T.J.R. Hughes, R.M. Ferencz, J.O. Hallquist, *Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients*, *Comput. Methods Appl. Mech. Engrg*. 61(2), (1987a), 215-248.

[5] C. Farhat, L. Crivelli, *A general approach to nonlinear finite-element computations on shared-memory multiprocessors*, *Comput. Methods Appl. Mech. Engry*. 72(2), (1989), 153-171.

[6] D. Komatitsch, D. Michaa, G. Erlebacher, *Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA*, *J. Parallel Distrib. Comput*. 69, (2009), 451-460.

[7] C. Cecka, A.J. Lew, and E. Darve, *Assembly of Finite Element Methods on Graphics Processors*, *Int. J. Numer. Meth. Engng* 2000, 1-6.

[6]https://code.launchpad.net/pragmatic