# Stitching Threads into the Unified Model

M. J. Glover, A. J. Malcolm, M. Guidolin and P. M. Selwood
Met Office, FitzRoy Road, Exeter, UK. EX1 3PB

***Abstract*—The Met Office Unified Model (UM) uses a hybrid parallelization strategy: MPI and OpenMP. Being legacy code, OpenMP has been retrofitted in a piecemeal fashion over recent years.**

**As OpenMP coverage expands, we are able to perform operational runs on fewer MPI tasks for a given machine resource, achieving the aim of reduced communication overheads. Operationally, we are running with 2 threads today; but we are on the cusp of 4 threads becoming more efficient for some model configurations. Outside of operational resource windows, we have been able to scale the UM to 88920 cores, which would not have been possible with MPI alone.**

**We have experimented with a resource-stealing strategy to mitigate load imbalance between co-located MPI tasks. The method is working from a technical point of view and showing large performance benefits in the relevant code, but there are significant overheads which require attention.**

## I. Introduction

The Unified Model (UM) is developed and used by the Met Office to simulate the Earth's atmosphere, in order to produce operational weather forecasts as well as seasonal and climate-scale simulations. The model is gridpoint-based and has historically been parallelised with domain decompostion and MPI. There are limits to the effectiveness of this: fewer gridpoints per MPI task limits the ratio of computation to communications. Moreover, the underlying latitude-longitude grid causes particular issues at the poles, such as a need for numerical consistency along the polar rows.

Hybrid — or mixed-mode — parallelisation has been with us since the dawn of OpenMP. Threading is used to allow an application, as much as possible, to use some given amount of machine resource with fewer MPI tasks. Fewer tasks means fewer communications overheads, which translates to better performance and scaling. The work required to achieve this in a legacy application with a substantial amount of rapidly changing code does not happen overnight. In this paper, we aim to describe our hybrid parallelisation effort over recent years, and the benefit we derive from that effort today. We further discuss characteristics of the code which offer a clean implementation of high-level OpenMP, and the scope for resource-stealing between MPI tasks as a means of reducing load imbalance.

Section II gives a brief description of the Unified Model (UM). Section III describes the procedure we follow to highlight code which would benefit from OpenMP directives. Threading performance is described in Section IV. Section V describes an experimental method to improve load balancing between MPI tasks which share the same node, and Section VI brings the paper to a close.

## II. The Unified Model

### A. Overview

The Met Office Unified Model (UM) is our atmospheric simulation package. It has been in operational service for approximately three decades, and is used today for short-term weather forecasting, through medium-term seasonal and decadal forecasting, to long-term climate timescales. At its core is a CFD solver — ENDGame [1] — which attempts to solve the fluid flow on a rotating sphere, but with numerous sub-gridscale processes (cloud, precipitation, turbulance, surface exchange, ...) being approximated to grid points.

### B. Parallelisation

The parallelisation strategy employed is primarily one of domain decomposition. At scale, domain decomposition hurts and helps: helped by natural cache-blocking that comes with smaller domain sizes, but also hurt by attendant reductions in vector lengths, in addition to communications costs.

Aside from communications, load imbalance represents a major cost. The effect manifests primarily in the time taken for halo exchanges: a weakly synchronising operation. Significant imbalance arises because the workload for different parts of the globe is inherently non-uniform. For example, at any one time, half the globe is in darkness, representing an uneven distribution of sunlight even before the effects of the non-homogeneous nature of cloud are factored in. Similarly, there is a great deal of convection in the tropics, but very little at the poles, as well as similar effects in many of the other sub-gridscale processes.

OpenMP offers us a way to reduce communications costs by limiting the number of MPI tasks needed for a given machine resource. This may seem like old news: OpenMP has been with us for more than a decade, and developers have long-since adopted hybrid parallelisation strategies. We aim to show here the current state of our years of hybridisation work, and the benefits realised in the shape of enhanced scalability and parallel efficiency in our real-world and highly complex application.

### C. OpenMP Coverage

Amdahl's law dictates that parallel efficiency is dependent upon the proportion of time spent executing parallelised code. At over a million lines of code, with no particular performance hotspots, OpenMP work on the UM is slow and highly incremental. Early OpenMP work began with the arrival of our IBM Power6 in 2009. That machine used in-order execution, which meant that simultaneous multi-threading (SMT) gave a useful performance benefit. We could employ two

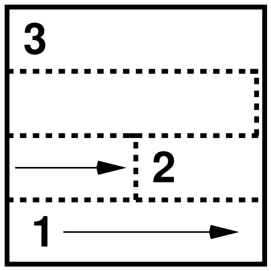

Fig. 1. Segmentation. The square represents a single model domain. Numbering and arrows are faithful to the layout of data: working West-East and South-North.

threads per physical core without risk of cores standing idle in OpenMP-serial code sections: SMT gave us benefit, even though OpenMP coverage was limited.

With the advent of our Cray XC40, any potential benefits of hyperthreading are less clear. The benefit of OpenMP largely falls back to the original intent of the exercise: to reduce the communications overheads.

### D. Deployment

The majority of OpenMP directives are confined to individual loops, or a relatively small number of loops in a common parallel region. For legacy code that was not designed to be thread-parallel, this is a sensible approach. The impact of fork-join overheads is no doubt mitigated by thread pooling strategies employed by compilers: threads are not created and destroyed each time, but a pool of threads remain open to be called upon at the start of each parallel region.

Some code sections are amenable to directives at a higher-level, particularly some of the sub-gridscale parametrizations such as radiation and microphysics. These sections of code operate on vertical (skyward) columns of data points which do not need to interact with their horizontal neighbours. Hence there is scope for bunches of columns to be processed concurrently without fear of race conditions. The benefits of this are two-fold. The column-independence exposes parallelism, but further subdivision into smaller bunches — or *segments* — helps to exploit cache and reduce the impact of memory bandwidth. Segmentation is illustrated in Fig. 1. It is similar to the `NPROMA` technique employed by ECMWF [2].

## III. Targets for OpenMP

### A. Proprietary tools

There are platform-specific tools available to study performance, such as CrayPat/CrayPat-lite, and other open-source tools such as Score-P/Scalasca. But there are difficulties surrounding them, particularly compared with DrHook as described in III-B. They normally rely on some auto-instrumentation of the code and different tools present their results in different ways. We find it greatly helpful to have available a tool which is permanently coded into the UM, is threadsafe, and produces the same output on different platforms. That tool is DrHook.

### B. DrHook

DrHook was written at ECMWF, and originally conceived as a debugging tool [3]. The idea was to leave the DrHook compiled in, so that it could be activated to obtain a code traceback without the need to recompile. But the presence of callipers in the application code also allows it to extract profiling information.

DrHook offers advantages over other solutions discussed above. It produces a common format of output across multiple architectures, and the output is plain text; post-analysis scripting is simple to automate. Being already present in the code base, there is no need for the tool to perform automatic instrumentation, and the developer can see clearly where timing regions begin and end; even add new regions on an *ad hoc* basis to study a particular code snippet. DrHook is threadsafe, via an intuitive mechanism.

Moreover, in principle it would be a simple matter to run DrHook across a model containing a coupler and several different executables, so long as each component model was instrumented with DrHook. Our seasonal and climate models, which couple our atmosphere model with ocean and sea ice models, are an example of this. Today however, only the atmosphere model is instrumented.

When profiling is not taking place, a dummy DrHook library is used. All calls to DrHook are protected by the test of a logical value, which in this case is hardwired as a compile-time parameter to `.false.`. An optimising compiler is able to eliminate the calls at the compilation stage, removing any potential performance degradation in production work.

### C. Scaling score

In optimisation work, the challenge is to identify those areas of code which will benefit the most. To find routines which do not scale well in thread-space, we run a baseline with one thread, and the same run on more threads, with the consequent increase in machine resource.

Once timings have been obtained from DrHook, we compute a scaling score using the output from the baseline and multi-threaded runs. Subroutines are listed according to their score value, smallest first. The scaling score is given by Eqn. (1).

$$S = \left(\frac{p_1 t_1 T_1}{p_2 t_2 T_2} - 1\right) / \left(\frac{p_1 t_1}{p_2 t_2} - 1\right) + 1 \qquad (1)$$

where $p$ is the number of MPI tasks, $t$ is the number of threads per MPI task and $T$ is the elapsed time. The index 1 refer to reference (baseline) time and resources; index 2 is the test time and resources.

Values of the score and their meaning are listed in Table I. The score is not sufficient by itself to identify performance hotspots: a small routine may scale horrendously but be insignificant with regard to the overall wallclock time. Cross-analysis with an ordinary wallclock time profile indentifies those costly routines that do not exhibit good scaling.

Of course, this technique is not limited to threading; it is used to study scaling characteristics in MPI-space, too.

TABLE I
MEANINGS BEHIND VALUES OF THE SCALING SCORE IN EQN. (1).

| Score value | Meaning |
|---|---|
| $S < 0$ | Routine is anti-scaling. |
| $S = 0$ | Routine shows no scaling. |
| $0 < S < 1$ | Routine shows sub-linear scaling. |
| $S = 1$ | Routine shows ideal scaling. |
| $S > 1$ | Routine shows super-linear scaling. |

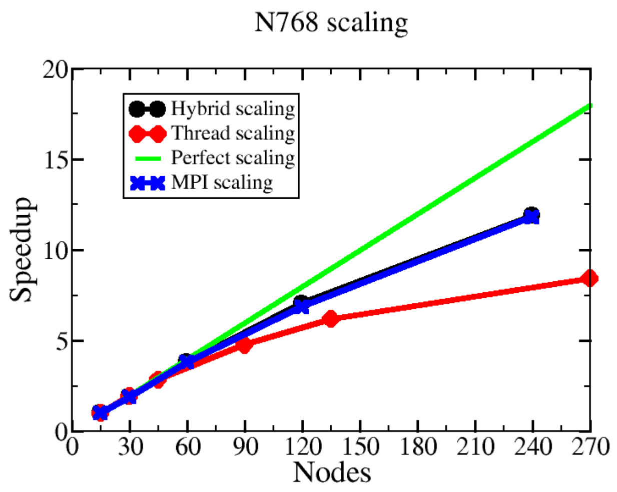

Fig. 2. Scaling of the N768 model at UM version 10.3* (see text).

## IV. THREADING PERFORMANCE

We now turn to performance of the UM with respect to threading, in particular comparing today's performance with earlier UM versions. Asterisks (*) denote that the code used fractionally predated official release of the accompanying UM version, and may have some differences.

Fig. 2 allows comparison of scaling behaviour with various combinations of MPI and threading. The MPI curve uses one thread only. The hybrid scaling curve represents the best combination found of the two parallelisation strategies for a given amount of machine resource. The thread scaling curve uses a fixed MPI decomposition baseline, with the scaling coming purely from OpenMP coverage. Recall that much of the OpenMP is loop-level; the thread scaling essentially serves to demonstrate the degree of coverage.

Fig. 3 shows percentage parallel efficiencies at different UM versions and processor generations. Here, the domain decomposition is held constant as the thread count increases. The plot shows clear improvements with each subsequent model version. For example, with 8 threads, UM10.2 gave a parallel efficiency of about 40%, which has increased to over 75% at UM10.4*. To add some chronology: UM10.2 dates from mid-2015, and UM10.4* from early 2016. The increased parallel efficiency is not purely a function of model version, however: the Broadwell (E5-2695v4) results are demonstrably better than the equivalent Haswell (E5-2698v3) results, as seen in the two UM10.3* curves. We speculate that the underlying reason relates to the increased shared L3 cache

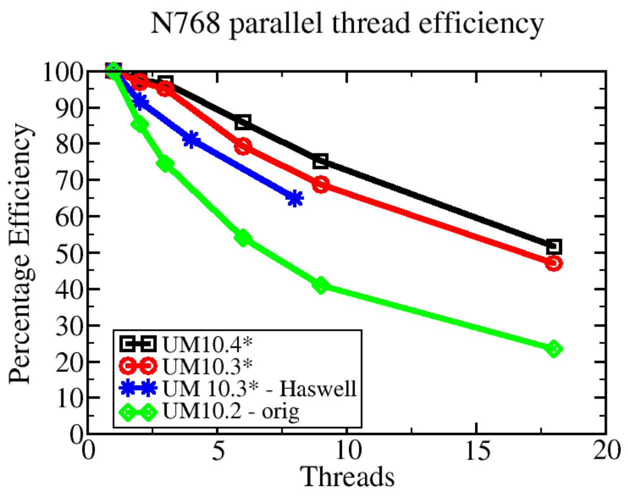

Fig. 3. Parallel efficiency as a function of thread count at N768 resolution.

on the Broadwell processors.

### A. Operational resources

Here we constrain ourselves to model resolutions and node counts comparable to those used today to produce operational weather forecasts.

Over recent years, the best performance with a given operational resource window has usually been achieved using a low number of threads. Utilising more than 2 threads with a corresponding reduction in the number of MPI tasks has invariably not given any benefit. However, OpenMP coverage is now becoming sufficiently extensive that the situation is changing. Table II shows optimal combinations of MPI decomposition and threading for an N768 model over a range of node counts. Notice that on 120 nodes and above, three threads on Broadwell processors become more efficient than 2 threads. (With 36 cores per node, 6 threads would be the next highest thread count which maintains full subscription.) Hence the OpenMP is giving noticeable benefit for operational Model configurations.

Digging a little deeper, Table III lists timings for individual code sections within the UM. The leftmost column represents some baseline timings, and the rightmost two columns represent the same resource-usage, but composed in different ways: more domain decomposition with fewer threads, and the decomposition held constant but using a larger thread count. Some code sections demonstrate the benefit of hybrid programming: notably the solver. This section contains a lot of communications, and the OpenMP mitigates against these overheads by allowing execution on a smaller number of MPI tasks. The OpenMP is serving its intended purpose.

But the same is not true for all code sections: physics sections `Atmos_phys1` and `Atmos_phys2` demonstrate this. These code sections influence the plain fluid dynamics, modelling many processes such as radiation coming direct from the sun (shortwave) and reflected from cloud (longwave), and vertical

TABLE II
HYBRID SCALING FOR N768

| Nodes | Cores | Threads | Time(s) | Perfect Scaling | Hybrid Scaling |
|---|---|---|---|---|---|
| 15 | 540 | 2 | 1137 | 1 | 1 |
| 30 | 1080 | 2 | 1137 | 2 | 1.89 |
| 60 | 2160 | 2 | 564 | 4 | 3.81 |
| 120 | 4320 | 3 | 305 | 8 | 7.05 |
| 240 | 8640 | 3 | 181 | 16 | 11.88 |
| 480 | 17280 | 3 | 111 | 32 | 19.37 |

TABLE III
TIMINGS FOR INDIVIDUAL CODE SECTIONS WITHIN THE UM. ALL TIMES ARE IN SECONDS. HERE, NXM REPRESENTS THE DOMAIN DECOMPOSITION (MPI), AND _6 DENOTES SIX THREADS.

| Code Section | 36x60_2 Baseline | 72x90_2 MPI | 36x60_6 Threads |
|---|---|---|---|
| `U_MODEL4A` | 1792 | 829 | 825 |
| `ATM_STEP_4A` | 1571 | 624 | 599 |
| `AS SOLVER` | 510 | 176 | 158 |
| `AS S-L Advect` | 356 | 183 | 153 |
| `AS Atmos_phys2` | 344 | 121 | 135 |
| `AS Atmos_phys1` | 283 | 105 | 114 |
| `INITIAL` | 209 | 197 | 218 |

transport of air (convection). These sections tend to have fewer communications than the model dynamics, so the OpenMP for these sections will require a higher degree of coverage in order to match the performance of MPI domain-decomposition. There are also load balancing issues to consider. The longwave radiation and convection sections are both segmented, yet the segment size does not correlate so directly with the amount of work: depending on the path through the code, gridpoints may have different amounts of computation. Whilst it can help to code up some forced load balancing of gridpoints between threads, that strategy is only effective if the amount of work per gridpoint remains constant. When it does not, the segment size may need to be reduced in order to allow an OpenMP `DYNAMIC` schedule to assist: smaller segments permit better load balancing between threads.

We have not yet addressed the use of hyperthreading in this discussion. Our experience thus far shows a tendency towards the existence of some performance gain with the combination of 2 threads and a passive wait policy. Introducing more threads, experience suggests, has a harmful effect on performance, although this depends strongly on model configuration.

### B. Further scaling

Having looked at the N768 model on relatively small node counts, we now push the node count far higher. The scaling characteristics are shown in Fig. 4. The vertical dotted lines indicate the number of threads with which the underlying timings were obtained. The line labelled "> 2 threads" is significant: that is the line beyond which, under test, MPI ceases to give any more benefit. Any further scaling results from OpenMP coverage. On 9 threads, the model model scales up to at least 80,000 cores, or well over 2000 nodes. This would not be possible without the hybrid parallelisation: scaling would otherwise be limited to well under 10,000 cores.

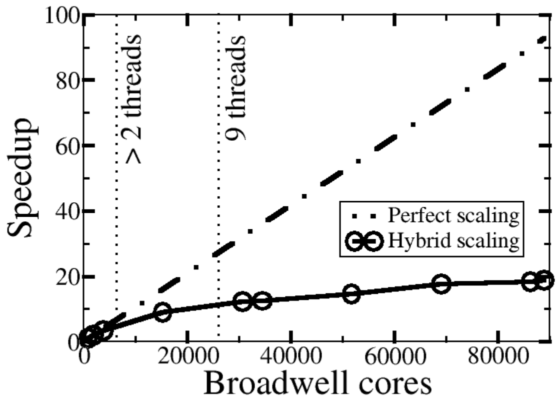

Fig. 4. N768 scaling extended to a larger number of cores. Vertical dotted lines show the number of threads with which the timings were taken.

### C. Limited Area Model (LAM)

All model configurations so far have been global: simulating the atmosphere around the entire globe. But some operational products only simulate the atmosphere over a local region of interest, such as over the UK. The model is configured as a Limited Area Model (or LAM). Boundary conditions — data around the region's perimeter — comes from the global model.

The major scaling issues in global models surround complications at the poles. LAMs do not have any poles, and so in theory ought to demonstrate better scaling characteristics. But Fig. 5 shows otherwise. Note that the LAM baseline decomposition has been adjusted to give each MPI task broadly the same number of gridpoints as for the global model (within 10%). Whilst choice of an optimal hybrid scaling curve outperforms purer MPI or thread scaling, the hybrid curve still lags the global model equivalent. The reason behind this is not currently known.

## V. MALLEABLE THREAD COUNTS

### A. Overview and motivation

As already discussed, the UM suffers from a large degree of load imbalance, particularly introduced by the uneven physical processes around the globe. Shortwave radiation is a particularly notable example: the effect of solar radiation coming directly from the sun. Aside from the large imbalance from the diurnal cycle, clouds cast shadows which change the radiation pattern and hence the computational load.

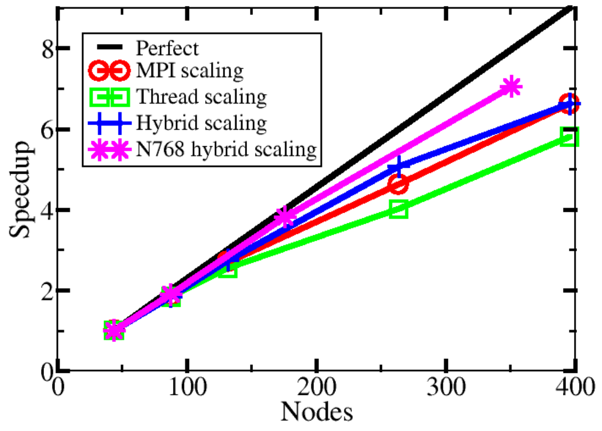

Fig. 5. Scaling of the Limited Area Model (LAM), and comparison against global model scaling.

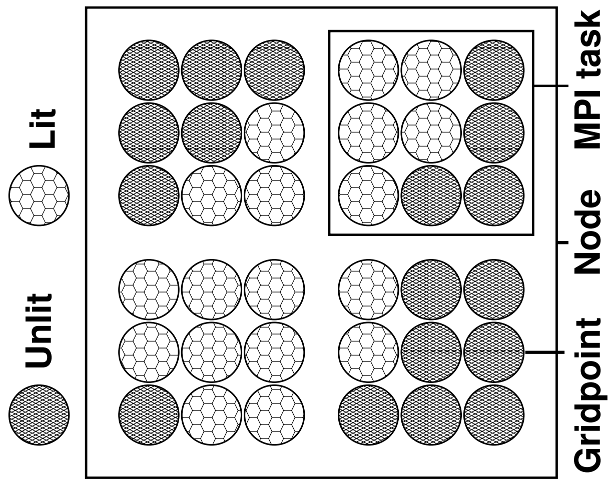

Fig. 6. Illustration of on-node load imbalance. Only lit points require computation.

The coverage of OpenMP in the UM is not yet sufficient to run with one MPI task per node, or even one MPI task per socket. Hence there are multiple MPI tasks present, computing different sub-domains with varying amounts of work. This is illustrated in Fig. 6. It seems sensible to assign the available cores to MPI tasks in proportion to their workload, relative to other MPI tasks on the same node.

## *B. Implementation*

The shortware radiation code is segmented: cache-friendly bunches of independent columns — but contiguous-in-memory — are operated on by different threads. Not all gridpoints have any computation associated with them; we refer to gridpoints with and without work as *lit* and *unlit*, respectively.

The UM forcibly distributes all lit points among threads within a given MPI task, imposing good load balancing between threads. For all results presented below, the segment size was fixed at 80 gridpoints. Tests with a smaller size showed no significant difference.

Shortwave (SW) radiation was chosen to investigate this approach because there is a large degree of variation between MPI tasks, and the UM knows up-front how many gridpoints are lit and how many are unlit which the redistribution algorithm can leverage.

For MPI tasks to perform some on-node organisation/co-operation between themselves, there must be some level of communication between them: all tasks must know the total amount of work on the node, in order to compute their share of the resource and create/affinitize their threads accordingly.

There is an API in MPI-3 which helps greatly with this kind of book-keeping: in particular, handling shared memory segments. The API allows one to create a communicator based on whether or not tasks are on the same shared-memory node. Two shared memory arrays are needed:

- The number of gridpoints requiring computation on each MPI task.
- The number of threads to be spawned on each MPI task.

An XC40 node has two NUMA regions; two sockets. One can prevent the reaffinitization process straddling the NUMA boundary by further splitting the shared-memory node communicators into communicators for each socket independently. It is not clear this this would be helpful, since it removes scope for load balancing between sockets, at the expense of increased friction at the NUMA boundary. Indeed, initial experiments suggested that it would be better to allow tasks to bleed over the boundary. At least one core is assigned to each MPI task. The remainder are distributed according to workload, rounding downwards to avoid oversubscription. Hence there is some potential for undercommitting; but to avoid complication, we keep the algorithm simple for the moment. In practical terms, the results presented in Subsection V-G are conservative: there may be room for improvement over those results.

It is important to store the original affinitization pattern — as set initially by `aprun` — to cope with the UM's I/O server, which may occupy one or more cores on any given node. The redistribution process must not attempt to place work on these cores. Hence the importance of a stored pool of cores which are available to the redistribution algorithm. This information is simply collected by asking all compute-threads (not I/O-server threads) to write their affinitization status to a shared memory segment.

## *C. Latent threads*

The Cray compiler appears to leave threads open at the end of a parallel region: threads are not destroyed, in a strategy termed *thread pooling*. The aim is to reduce the cost of fork-join overheads. At the first parallel region, new threads are created. But when subsequent parallel regions begin, threads are acquired from the pool of currently open threads. It is simple enough to verify that thread pooling is taking place: `/proc/<process id>/status` shows the number of threads currently assigned to the specified process number. With a piece of code designed to open a parallel region multiple times with different thread counts, one can verify that the number of threads reported in the file matches the high-water mark, not the value returned by `omp_get_max_threads()`.

The redistribution process causes issues here. At its most effective, a large number of threads may be required by one MPI task, as it uses a greater share of the available resource. Those threads are not destroyed once the parallel region terminates; instead they are left behind, consuming resources as they wait for work.

Ideally, we would utilise an option to temporarily suspend thread pooling: all additional threads would be destroyed at the end of a parallel region, leaving only those threads spawned at the first parallel region. As this is not currently possible, the effects of the pooled threads can be ameliorated with `OMP_WAIT_POLICY=passive`, as distinct from the `active` default. This workaround is not a perfect solution however, because the `active` wait policy is wanted for the remainder of the run. The Cray compiler provides a (vendor-specific) way to switch the policy mid-way through a run: `cray_omp_set_wait_policy()`. However, experiments with this function caused the model to hang at the first timestep.

Some mechanism to share a single common thread pool between MPI tasks would be a solution to the issues in this subsection, but we do not know of any way to make this work at present.

## D. Re-affinitization

We use the default `-cc cpu` affinity settings through the `aprun` launcher. The implementation remembers how many threads there were initially before the redistribution process, and where they were bound. Threads are then returned to their original cores afterwards: the restoration step.

The thread count is changed using the OpenMP API. Re-affinitization of threads is done using the `sched` library: C code behind a Fortran interface.

For reasons covered in Subsection V-C, early attempts at redistribution resulted in a dramatic, prohibitive slowdown; even *after* the restoration step had completed, as compute performance was hindered by latent threads.

A workaround was possible: confine redistributed, malleable-thread code sections to virtual cores, assuming that there is only one thread per physical core (no hyperthreading for the model's work). Once complete, the restoration step affinitizes the original threads back to the physical cores. This method allowed the model to run through to completion.

Later testing with a newer compiler version suggested that the workaround is no longer necessary: we may redistribute work among the physical cores. A specially designed piece of tester code confirmed that the issue had gone with module `cce/8.4.0` and higher. The results below were obtained with `cce/8.4.3`; the workaround was used, but not necessary.

## E. Rank reordering

The default MPI rank-ordering (SMP-style) places neighbouring model domains on the same node. Hence the default rank ordering is likely to place high-workload MPI tasks on the same node, as a consequence of the underlying spatial correlation of light and dark gridpoints.

To break up this spatial correlation — and separate light and dark regions between different nodes — the rank reordering pattern was changed to a simple Round-Robin pattern: `MPICH_RANK_REORDER_METHOD=0`. Whilst useful for the present experimentation, future operational use would require a performance evaluation of the model as a whole against the default SMP-style rank ordering.

## F. Miscellaneous details

Hardware and compiler were as follows:

- XC40 Broadwell nodes (18 cores per socket, 2 sockets)
- Cray compiler: cce/8.4.3
- UM version 10.3

Timings were taken with the UM's internal timer functionality (not DrHook in this case). In all cases, the maximum time is used across MPI tasks, since the model as a whole only progresses as quickly as the slowest task. Timings were measured for both the core code which performs the computation, and the surrounding code which incurs the overheads of segmentation setup and thread redistribution.

## G. Redistribution timings

Employing an N512 model resolution, Table IV shows the timing of the segmented region, with and without redistribution and with two different rank ordering strategies. These timings exclude the cost of the redistribution process itself, and only represent the cost of the calculation inside. Notice that on 3 threads, the cost of the segmented region comes down by more than 20%, compared with the same calculation without redistribution. There are cost savings on higher thread counts too, though smaller as a percentage. Note in particular the importance of Round-Robin rank ordering to achieve benefits with redistribution.

Table V shows the equivalent total timings from short-wave radiation. These timings are from the parent routine, which itself calls down to the segmented code; they include the overheads of the redistribution process itself. Fig. 7 plots the same information, together with the previous segmented code timings. The parent routine timing shows some benefit with redistribution on 3 threads, but much smaller than the 20% benefit for the segmented region: a modest 2.5%.

Table VI shows the cost of the parent routine, having subtracted away the cost of the segmented region; values provide a measure of the redistribution overheads. Whilst we have demonstrated significant benefit for the segmented code itself with the redistribution technique, future effort must turn towards tackling the large overheads. The most likely culprit is cache invalidation, due to changes in the affinitization pattern.

In the above tables, uncertainty on the values is of the order of a few tenths of a second: any significant differences due to the redistribution procedure are well-resolved.

The results so far were all taken from a 12x24 decomposition. Tests with a 24x36 decomposition demonstrated weaker benefits for the segmented code itself, and a slowdown once

TABLE IV
SEGMENTED REGION ONLY, OMITTING THE SETUP/REDISTRIBUTION OVERHEADS. ALL TIMES IN SECONDS.

| | SMP-style ordering | | | Round-Robin ordering | | |
|---|---|---|---|---|---|---|
| Threads | Original | Redistribute | Change (%) | Original | Redistribute | Change (%) |
| 2 | 35.0 | 42.6 | +21.5 | 32.0 | 33.2 | +3.8 |
| 3 | 23.7 | 27.7 | +17.1 | 22.5 | 18.0 | -20.1 |
| 6 | 12.3 | 13.6 | +10.7 | 11.6 | 9.4 | -18.9 |
| 9 | 8.4 | 9.0 | +7.2 | 7.9 | 6.9 | -12.0 |
| 18 | 4.4 | 4.5 | +2.5 | 4.4 | 4.0 | -8.7 |

TABLE V
SEGMENTED REGION PLUS PARENT TIME WITH 12X24 DECOMPOSITION. CHANGES PROVIDE A MEASURE OF REDISTRIBUTION OVERHEADS. ALL TIMES IN SECONDS.

| Threads | Original | Redistribute | Change (%) |
|---|---|---|---|
| 2 | 38.9 | 46.6 | +19.8 |
| 3 | 27.9 | 27.2 | -2.5 |
| 6 | 15.7 | 15.6 | -0.7 |
| 9 | 11.4 | 12.0 | +5.3 |
| 18 | 7.6 | 8.2 | +7.9 |

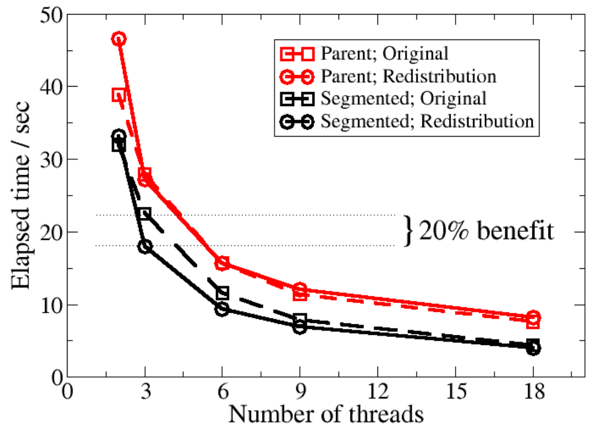

Fig. 7. Elapsed time of the segmented code with and without redistribution and that of the parent routine, inclusive of the segmented code called. The parent includes redistribution overheads.

the redistribution overheads are accounted for. This says that the method preferred the larger domain size for this model resolution. A reason behind this might be argued as follows. The method relies on there being sufficient variation in workload between MPI tasks assigned to the same node, to better enable this resource-stealing strategy to work. Whilst it is desirable to split up a computationally intensive pocket of work by splitting it between different nodes, the Round-Robin rank reordering may also have a homogenising effect. If the globe is divided into smaller domains, and those domains are scattered between nodes, then nodes may acquire a similar spread of relatively loaded and non-loaded domains, such that a highly loaded MPI task cannot claim sufficient resources to significantly improve its performance. This much is supposition.

TABLE VI
PARENT ROUTINE TIMINGS. VALUES INCLUDE REDISTRIBUTION OVERHEADS.

| Threads | Original | Redistribute | Change (sec) |
|---|---|---|---|
| 2 | 6.9 | 13.39 | 6.5 |
| 3 | 5.4 | 9.2 | 3.9 |
| 6 | 4.1 | 6.2 | 2.1 |
| 9 | 3.5 | 5.0 | 1.5 |
| 18 | 3.2 | 4.2 | 1.0 |

The implementation and investigations so far are young. There is more to investigate:

- Understand how to mitigate against large redistribution overheads.
- Better algorithm to distribute work between cores inside a node, without undercommitting.
- Behaviour of the different compilers available on the system: Cray, Intel and GNU. In particular, GNU does not appear to perform thread pooling.
- Selective hyperthreading, redistributing across all logical cores.

## VI. SUMMARY & CONCLUSIONS

The scaling potential of the UM with respect to domain decomposition alone is limited. Over recent years, we have added and continuously extended OpenMP coverage in order to circumvent communications overheads; the UM has adopted a hybrid parallelisation strategy. The UM is legacy code, and has many complicating data dependencies which inhibit the level at which OpenMP can be sited: most directives are loop-level, targeting those routines which will give the most immediate benefit at operational model resolutions. The profiling tool DrHook is compiled into our code, and provides a simple and portable way to access thread-scaling information.

The effectiveness of threading is heavily dependent on the degree of OpenMP coverage in the model, and the low-level of the OpenMP directives means that increased coverage means touching a lot of code. But we have gained demonstrable benefits of the work in the Unified Model, both in terms of operational model efficiency and increased scaling potential. For an N768 model, the optimimal strategy today utilises 3 threads, with fewer MPI tasks than would otherwise be necessary. Far outside of operational resource windows, we have demonstrated scaling beyond 80,000 cores: many times more than would have been possible without mixed-mode parallelisation.

We have experimented with novel ways of using threading to further boost the hybrid performance, which does seem to be showing some promise. Threading opens up the possiblity of a resource-stealing strategy to reduce load imbalance: MPI tasks sharing the same node can be assigned a portion of the available cores, proportionate to workload. This in turn relies on a sufficient mixture of lightly and heavily loaded MPI tasks on-node. Fortunately, the shortwave radiation code in the UM meets this criterion, and already computes the number of gridpoints that require computation, so the workload is known up-front and some control code can do the necessary re-affinitization. A crude distribution algorithm gives some demonstrable benefit in the N512 model — around 20% for the segmented region itself — provided that the rank-ordering strategy is Round-Robin; we believe this to be important to break up the underlying spatial correlation of light and dark gridpoints. Surrounding the segmented region itself, there are some significant overheads associated with the redistribution process which need attention.

There is a further complication, related to thread pooling. The Cray compiler does not destroy threads after a parallel region. Instead, the threads remain active, waiting to acquire more work at the next parallel region. Whilst helpful for the remainder of the UM — where fork-join overheads would otherwise be much larger — thread pooling seems to be causing some issues in the redistribution method: it produces a glut of threads which impede subsequent performance. A passive wait policy helps, but that is not appropriate for the remainder of the model, only redistributed sections. We do not have a reliable way of switching the policy mid-run. We would welcome a way to activate/deactivate thread-pooling, such that any additional threads created for the purpose of redistribution are destroyed.

Although it has taken a number of years of incremental improvement to arrive at this point, mixed-mode parallelisation is proving itself to have been a worthwhile investment.

## References

[1] N. Wood, A. Staniforth, A. White, T. Allen, M. Diamantakis, M. Gross., T. Melvin, C. Smith, S. Vosper, M. Zerroukat, and J. Thuburn, "An inherently mass-conserving semi-implicit semi-Lagrangian discretization of the deep-atmosphere global nonhydrostatic equations," *Q.J.R. Meteorol. Soc.*, vol. 140, pp. 1505–1520, 2014, dOI:10.1002/qj.2235.

[2] "IFS documentation. Part VI: Technical and computational procedures," www.ecmwf.int/sites/default/files/IFS_CY40R1_Part6.pdf, ECMWF, [Online. Accessed: April 6, 2016].

[3] S. Saarinen, M. Hamrud, D. Salmond, and J. Hague, "Dr.hook instrumentation tool," https://software.ecmwf.int/wiki/download/attachments/19661682/drhook.pdf, [Online. Accessed: April 6, 2016].