Improving the Performance of COSMO-CLM

Matthew Cordery *and* **Will Sawyer**, *CSCS-Swiss National Supercomputing Centre, and* **Ulrich Schättler**, *Deutscher Wetterdienst*

ABSTRACT: *The COSMO-CLM model, originally developed by Deutscher Wetterdienst, is a non-hydrostatic regional atmospheric model which can be used for numerical weather prediction and climate simulations and is now in use by a number of weather services for operational forecasting (e.g. MeteoSwiss). One current software engineering goal is to improve its scaling characteristics on multicore architectures by making it a hybrid MPI-OpenMP code. We will present hybridization strategies for different components of the model, show some first performance results, and discuss the impact on further development of the model.*

KEYWORDS: COSMO, weather, climate, hybrid programming

1. Introduction

The COSMO-CLM (COsortium for Small-scale Modelling-CLimate Mode) model is a non-hydrostatic regional atmospheric model, originally designed for numerical weather prediction (COSMO) on vector architectures that has been extended for use by the climate modelling community. While operational numerical weather prediction (NWP) models must reliably produce results for three-day forecasts on the order of half an hour of CPU time, extending the same performance to a 100-year simulation of climate on a grid with similar resolution would require several months of CPU time. If a researcher requires numerous climate runs in order to extract useful science, then the need to improve the performance of climate models like COSMO-CLM becomes self-evident.

Improving the performance of a mature and complex scientific application is, in general, no easy task. This is especially true if one desires to improve the performance by an order of magnitude. Often, one attempts to either (a) exploit as much parallelism as possible, (b) reduce the memory footprint and access of data structures in order to improve cache performance, (c) improve communications protocols, (d) re-factor data structures and loops to exploit next generation hardware (e.g. GPGPUs), new algorithms, or some combination of all these.

The goal of this work is to examine the performance characteristics of COSMO-CLM and to investigate the feasibility means of reducing the total run time of COSMO-CLM through exploiting additional parallelism. The code currently exploits coarse-grained parallelism through the use of MPI tasking. Our goal here is to exploit additional parallelism at the loop level through OpenMP directives.

The code itself, like many atmospheric models, is designed around both a dynamical core, which solves for the motion of air and water in the atmosphere, and a physical core that approximates a number of physical processes, such as heating and cooling by radiation, precipitation, moist convection, and so forth that contribute significantly to heat and mass transfer in the dynamical core. These cores solve their respective equations on three-dimensional structured mesh, represented internally by Fortran arrays whose indices reflect rotated latitude (x), rotated longitude (y), and height (z), respectively. Information is shared between these arrays via simple packed halo exchanges with the surrounding MPI tasks.

The main computational region of COSMO-CLM is centred on a time-stepping loop. Within this are the subroutine call controlling the updates to the dynamics

and to the physics. Ideally, at this stage, the data structures would be designed such that there would be loops over one of the indices noted above surrounding each of the subroutine calls. This would make exploiting parallelism quite easy as one could simply parallelize each loop without having to determine if many of the variables are private or shared or if any complex data dependencies exist. Rather, the main computational workload resides in the subroutines themselves, in numerous triply nested, and some doubly nested, loops. For triply nested loops, the outermost loop is over the number of height levels (typically 60 to 90) and for doubly nested loops, it is over the number of longitudes bands. As an initial attempt, then, our hybridization strategy is to insert OpenMP directives before computationally intensive loops within the subroutines and keeping any software design changes to a minimum. This will give us some indication of the performance gains that might be expected and provide useful information for any future efforts towards refactoring the code for cache-based multi-processor architectures.

2. Performance Description

The COSMO code and its related data sets have already been successfully ported and verified on Cray XT systems. The tests for this study were performed on the Cray XT5 *Monte Rosa* at the Swiss National Supercomputing Center. The system was upgraded in 2009 and currently houses 3688 AMD hex-core Opteron processors running at 2.4 Ghz and has a total of 28.8 TB of DDR2 RAM and a 9.6 GB/s interconnect bandwidth.

The code was compiled and linked with the Portland Group (PGI) Fortran compiler version 10.3 and linked to version 4 of the Cray MPT library. None of the benchmarking or profiling runs were launched on a dedicated system.

Key kernel performance.

We performed a number of scaling benchmarks on a grid with a 1-km resolution (1142x765x90) using decompositions with different aspect ratios. The 1.5x1 aspect ratio runs most closely match the global aspect ratio of the entire computational grid. The results show that, at this resolution, the code scales well out to over 6000 cores and that sub-grids assigned to MPI tasks perform better if they closely match the aspect ratio of the global computation region. Moreover, sub-grids with an aspect ratio that is longer in the x-direction perform better than those in that are longer in the y-direction, indicating that the code is likely benefitting from longer vectors. There is a limit to this, however, as changing the aspect ratio from approximately 1.5x1 increases the inter-task communications.

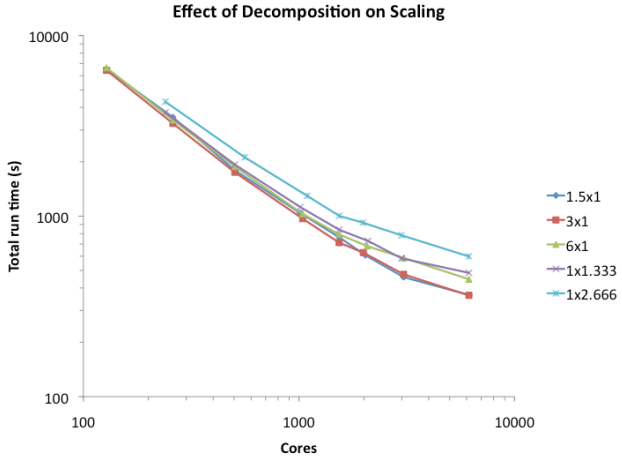

Figure 1. COSMO-CLM 1-hr run time with 1-km grid spacing and various decomposition aspect ratios.

An examination of the performance and scaling of the physical and dynamical cores (Figure 2) shows that that both scale quite well for the grid and that the dynamical core takes about three times (3x) as much compute time as the physical core, regardless of core count.

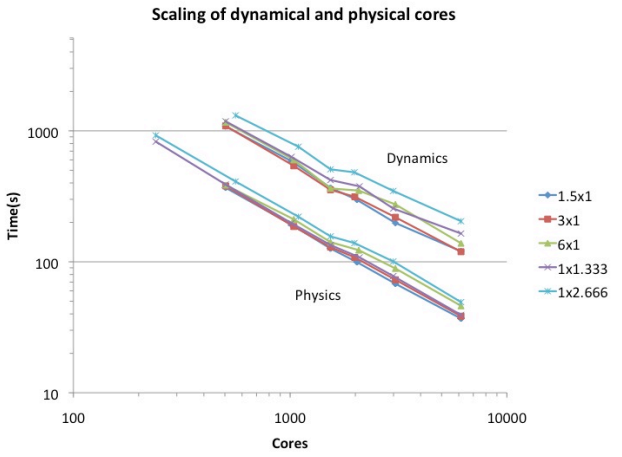

Figure 2. Breakdown of run time for physical and dynamical cores at various aspect ratios.

Within this range, profiling indicates that few of the top computational routines are load-imbalanced and that the most of the top routines have a computational intensity < 1 and high cache hit/miss ratios (>95%), with the most expensive routine, fast_waves_runge_kutta (which controls sound and gravity-wave propagation), having a computational intensity of 0.2, regardless of core count. These features, along with the strong scaling,

indicates the computational part of the code is largely network-bandwidth limited rather than network-bandwidth or cache limited. The percentage peak for these runs is in the range 3.5-4.0% For the remainder of this study, we no longer consider any aspect ratios other than the default for a particular grid.

3. Hybridization Strategy and Results

3.1 1-km grid

In this study, we concentrated on performing loop-level OpenMP threading within subroutines, without performing any large-scale restructuring of data structures or significant rewriting of code to allow for blocking. Thus, we simply inserted numerous OpenMP directives based on results from the Craypat profiling tool. We concentrated our efforts on physics and dynamics routines called from the main time-stepping loop, ignoring routines involved in file I/O as they are not significant factors of the total run time. The current version involves over 600 OpenMP parallel directives in 11 module files.

As part of our initial assessment of the performance assessment in this study, our first implementation of the OpenMP directives included the use of the new COLLAPSE directive. The graph in Figure 3 shows the results for both the hybrid code and the MPI only code.

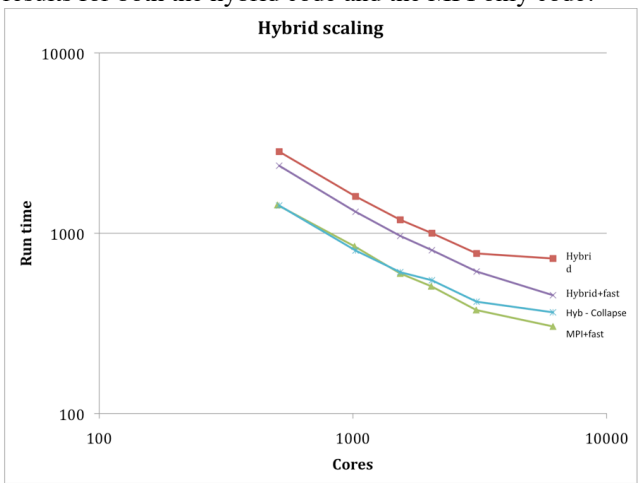

Figure 3. Scaling results for the hybrid code. Hybrid results begin at 512 MPI tasks and 1 OpenMP thread and increase to 2, 3, 4, 6, and 12 threads. Hybrid code with collapse directives (red squares). Hybrid code with collapse directives and compiled with SSE instructions enabled (purple crosses). Hybrid code without collapse directives and compiled with SSE instructions enabled (light blue curve). Non-hybrid code compiled with SSE instructions enabled (green triangles).

In performing the hybrid code runs, we chose the number of threads for each test case so that a node would be fully packed if there were enough tasks available. Thus, we only ran test cases where the number of threads times the number of MPI tasks on a node equalled twelve. A cursory examination of the results shows that this version of the hybrid code runs significantly slower than the non-hybrid version. Removing the collapse directives, as well as making minor adjustments to some of the OpenMP directives to allow the global use of SSE instructions, results in a hybrid code that, in most cases at this resolution, runs as fast as the MPI-only version.

If we examine the scaling of the physical and dynamical cores (Figure 4), we see that neither of the hybrid cores scales as well as the non-hybrid version at higher core counts. In fact, the physical core appears to be significantly more impacted by remaining load imbalances or lack of parallelism.

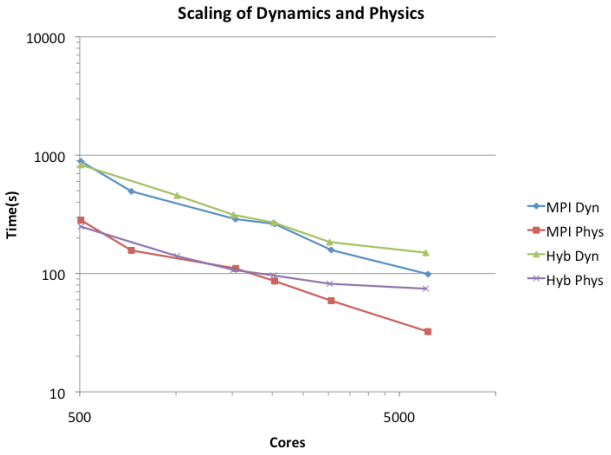

Figure 4. Scaling of they dynamical and physical cores. Hyb = hybrid code, Dyn=dynamical core, Phys=physical core.

We also examined the performance of the code using different OpenMP schedulers (static and dynamic) with different chunk sizes and found that variations other than the default did provide any significant performance benefit, if any.

3.2 Small 'Climate' grid

As mentioned above, of particular interest to us is the possibility of improving the performance of COSMO-CLM when applied to the problem of performing climate simulations that require significant wall-clock and CPU time. To this end, we defined a 102x102 grid which, when run with around 50 cores, would approximate the per-core size of problem expected on future cloud-resolving climate simulations.

In Figure 5, we show the results for running a 24-hour simulation. The pure MPI results scale linearly to approximately 100 cores and then the scaling begins to decline becoming flat after 576 cores. After 1152 cores, the performance actually decreases slightly.

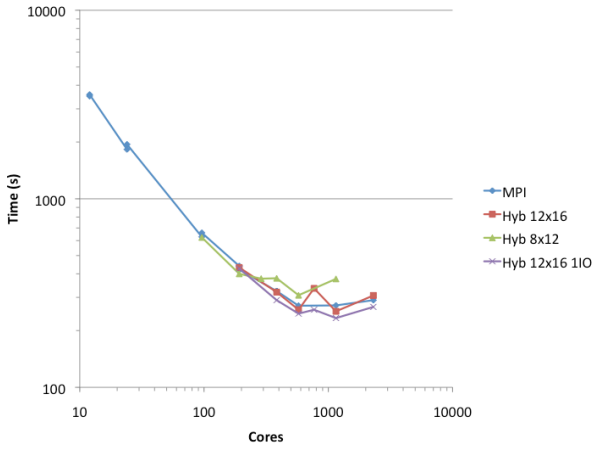

Figure 5. Scaling results for a small climate test grid. Shown are the results for a pure MPI test case, two hybrid code test cases using 8x12 and 12x16 decompositions and 1, 2, 3, 4, 6, and 12 threads, respectively. Also shown are the results for the 12x16 hybrid test case when using only 1 I/O task for the run, rather than 4.

Also shown are the results for three different hybrid test cases. For the 8x12 decomposition, which begins where the linear scaling of the MPI runs ends, the hybrid test case runs about 10% better than the MPI-only run when two threads are used but is slower than the latter after that. The model runs best with six threads and two MPI tasks per node implying there are still some scaling issues to be resolved. There is also a slight decrease in performance going from three to four threads that is also seen in the other hybrid runs in the graph. This performance degradation is, perhaps, not surprising since when four threads are used per MPI task, the threads of one of the tasks must span the two cores on the node and implies there is memory bandwidth contention in this case. The 12x16 hybrid case scales further than the 8x12 hybrid case but still shows the same performance degradation past 576 cores as the non-hybrid case. The performance degradation going from 3 to 4 threads is also most pronounced in this case. Compared to the non-hybrid case, this model is, at best 7% faster using six threads.

As an experiment, we also show the results for the same 12x16 hybrid model using only 1 MPI task for the I/O rather than 4 tasks. In COSMO-CLM, the user may specify a number of MPI tasks that will be reserved for writing results to disk. Previous benchmarking results on the non-hybrid code indicated that the optimal number of I/O tasks was about four. In this case, however, we achieved better performance when using only a single task for I/O. The reason for this performance improvement is not obvious at this time. However, using only a single task for I/O in the hybrid code makes better use of system resources, as fewer cores will be idle (e.g. using four I/O tasks and six threads uses only four cores in total with 20 cores doing nothing).

If we examine a breakdown of the scaling for various sections of the code in the 12x16 case (Figure 6), we see that, regardless of thread count, the total time spent in the

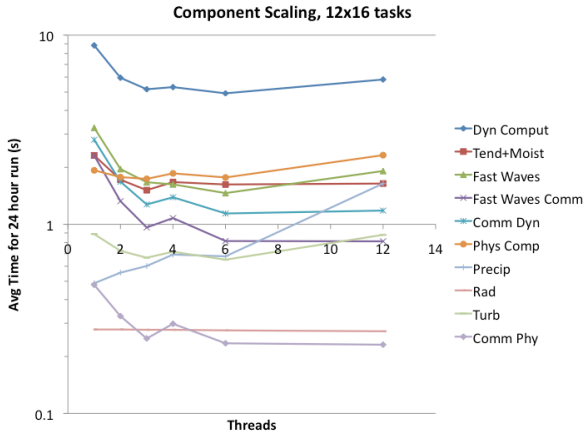

Figure 6. Breakdown of multi-thread scaling of important components in the dynamical and physical cores for the case of 192 MPI tasks (12x16 decomposition).

dynamics per time step is still approximately three times that spent in the physics. Furthermore, All of the important sub-sections of the dynamical core show the same scaling characteristics with thread count, with minima at six threads per task. The physical core, and its sub-sections, show a much flatter profile, in general with the section covering precipitation showing a marked steady increase in computational time with thread count and the radiation showing a completely flat profile. While fixing these problems with the physical core need to be addressed, doing so would not alter that fact that the dynamical core ceases to scale significantly after three threads.

3. Conclusions and Future Directions

While some performance improvements have been realized by this study, it is clear that significant improvements will likely require a fundamental reorganization of the code and/or better algorithms. Even

though much loop-level parallelism has been exploited, the fact remains that the computational intensity is low in many subroutines implying low data reuse. Additional performance improvements in the existing version may be possible through custom placements of tasks, thread-core affinity, and blocking. Furthermore, there are enough OpenMP directives in the code to make blocking both tedious and error-prone to implement and a significant software engineering problem from the point of view of maintenance and addition of new physics.

Another option is to reorganize the code and data structures to enable hoisting one of the loop indices out of the subroutines to a higher level. This would significantly reduce the amount of work necessary to implement loop-level parallelism. This might also ameliorate the task of improving data reuse and give us an opportunity to reduce the memory footprint of each task/thread thereby taking pressure off the memory subsystem.

Acknowledgments

The authors wish to thank Oliver Fuhrer for input, advice, and for providing useful test cases.

References

COSMO Model Documentation

About the Authors

Matthew Cordery (Email: cordery@cscs.ch) is an HPC Applications Analyst in the National Supercomputing Service (NSS) section of the Swiss National Supercomputing Centre (CSCS). William Sawyer (Email: wsawyer@cscs.ch) is an HPC Applications Analyst in the Scientific Computing Research Section of the Swiss National Supercomputing Centre (CSCS). They can be reached at CSCS, Galleria 2 - Via Cantonale, CH-6928 Manno, Switzerland. Ulrich Schättler (Email: Ulrich.Schaettler@dwd.de) is scientist in the Business Unit for Research and Development of the Deutscher Wetterdienst (German Weather Service). He is one of the primary developers of the COSMO code. He can be reached at Deutscher Wetterdienst, Frankfurter Straße 135, 63067 Offenbach.