Evaluating Performance of OpenMP Tasks in a Seismic Stencil Application

Eric Raut
Stony Brook University, eric.raut@stonybrook.edu

Jie Meng
Total EP R&T

Mauricio Araya-Polo
Total EP R&T

Barbara Chapman
Stony Brook University, barbara.chapman@stonybrook.edu

Evaluating Performance of OpenMP Tasks in a Seismic Stencil Application

Eric Raut[1], Jie Meng[2], Mauricio Araya-Polo[2], and Barbara Chapman[1]

[1] Stony Brook University, Stony Brook NY 11794, USA
{eric.raut,barbara.chapman}@stonybrook.edu
[2] Total EP R&T, Houston TX 77002, USA

Abstract. Simulations based on stencil computations (widely used in geosciences) have been dominated by the MPI+OpenMP programming model paradigm. Little effort has been devoted to experimenting with task-based parallelism in this context. We address this by introducing OpenMP task parallelism into the kernel of an industrial seismic modeling code, Minimod. We observe that even for these highly regular stencil computations, taskified kernels are competitive with traditional OpenMP-augmented loops, and in some experiments tasks even outperform loop parallelism.
This promising result sets the stage for more complex computational patterns. Simulations involve more than just the stencil calculation: a collection of kernels is often needed to accomplish the scientific objective (e.g., I/O, boundary conditions). These kernels can often be computed simultaneously; however, implementing this simultaneous computation with traditional programming models is not trivial. The presented approach will be extended to cover simultaneous execution of several kernels, where we expect to fully exploit the benefits of task-based programming.

Keywords: OpenMP · task parallelism · stencil computation · loop scheduling

1 Introduction

Many industrial and scientific applications use stencil computation for solving PDEs discretized with Finite Difference (FD) or Finite Volume (FV) methods. These can range from geophysics to weather forecasting models [32]. Improving performance is of utmost interest since this facilitates faster decision making as well as more opportunities to explore further scientific questions. Optimization of stencil computation has been addressed in the past aplenty (see Section 2) from many different angles, e.g. low-level optimization, parallelism at different levels, and DSLs.

In this work, we create OpenMP task-based versions of an industrial stencil-based seismic modeling code and compare performance of the task-based versions to traditional loop-parallelized versions of the code. The motivation of this work is to explore how task-based programming models and task parallelism can support the stencil computaion pattern in practice.

OpenMP [26] is the de-facto standard programming model for shared-memory parallelism. OpenMP introduced tasks in version 3.0. OpenMP 4.0 added automatic dependency analysis to tasks, such that the compiler can automatically determine the order of task execution based on user-supplied data dependences.

In task-based OpenMP programming, an application is written as a set of units of work called *tasks*. Each task is executed sequentially, but multiple tasks can be run simultaneously subject to the availability of resources and dependencies between the tasks. The set of tasks and dependencies between them can be represented as a directed acyclic graph (DAG).

Our main contributions are the following: (1) we introduce task parallelism to a stencil code in a proxy for an industrial application; (2) we test our task-based stencil code on several architectures and compilers; and (3) we analyze its behavior and compare results of the task-based stencil with several variants written using parallel loops.

The paper is organized as follows: Section 2 describes relevant literature works and contributions. Section 3 describes the target application. Section 4 detailes the application code structure and how it was ported to task parallelism. In Section 5, the experimental environment and results are presented. Section 6 and 7 provide discussion and conclusions.

2 Related Work

A great amount of research effort has been devoted to optimizing stencil computations to achieve higher performance. For example, Nguyen et al. [24] introduced higher dimension cache optimizations, and de la Cruz et al. proposed the semi-stencil algorithm [8] which offers an improved memory access pattern and efficiently reuses accessed data by dividing the computation into several updates. In 2012, Ghosh et al. [13] analyzed the performance and programmability of three high-level directive-based GPU programming models (PGI, CAPS, and OpenACC) on an NVIDIA GPU against isotropic and tilted transversely isotropic finite difference kernels in reverse time migration (RTM), which is a widely used method in exploration geophysics. In 2017, Qawasmeh et al. [28] implemented an MPI + OpenACC approach for seismic modeling and RTM. Also, from a programming language perspective, domain-specific languages (DSLs) for stencils have been proposed (e.g., [19]). Even performance models have been developed for this computing pattern (see [9]).

In recent years, task-based parallel programming has been recognized as a promising approach to improve performance in scientific applications such as stencil-based algorithms. For example, in [22], Moustafa et al. illustrated the design and implementation of a FD method-based seismic wave propagation simulator using PaRSEC.

Researchers have been working on exploring the advantages of tasking in OpenMP since tasks were introduced in version 3.0. Right after its release, Virouleau et al. [34] evaluated OpenMP tasks and dependencies with the KASTORS benchmark suite. Duran et al. [12] evaluated different OpenMP task scheduling strategies with several applications. Rico et al. [30] provided insights on the benefits of tasking over the work-sharing loop model by introducing tasking to an adaptive mesh refinement proxy application. Atkinson et al. [2] optimized the performance of an irregular algorithm for the fast multipole method with the use of tasks in OpenMP. Vidal et al. [33] evaluated the task features of OpenMP 4.0 extensions with the OmpSs programming model.

Several programming systems supporting tasks have been proposed, some of which (e.g., OpenMP) focus on shared-memory systems. Cilk [6] is an early programming API supporting tasks using `spawn` keyword. Intel Thread Building Blocks [29] also supports shared-memory task parallelism. StarSs [27] is a task-based framework for multi/many-core systems using a pragma syntax. OmpSs [11] is an attempt to extend OpenMP with tasking features using StarSs runtime.

Distributed-memory task-based systems have been explored as well, in which the runtime automatically schedules tasks among the available nodes and takes care of communication and data transfer. Charm++ [1] is a C++ framework supporting distributed task parallelism. Legion [4], and its DSL, Regent [31], are data-centric task-based programming systems developed at Stanford. PaRSEC [7] enables an application to be expressed as a “parameterized task graph” which is problem-size-independent and therefore highly scalable. HPX [15] is a task-based framework which uses a global address space to distibute computations across nodes. XcalableMP [18] is a PGAS language with elementary support for task parallelism. YML [10,14] allows the user to specify a computation as a graph of large-scale tasks; it can be combined with XcalableMP. StarPU [3] supports OpenMP-style pragmas and provides a runtime for distributed execution. Klinkenberg et al. [16] propose a framework for distributing tasks across MPI ranks in MPI+OpenMP hybrid applications.

3 Minimod Description

Minimod is a proxy application that simulates the propagation of waves through the Earth models, by solving a Finite Difference (FD) discretized form of the wave equation. It is designed and developed by Total Exploration and Production Research and Technologies [21]. Minimod is self-contained and designed to be portable across multiple compilers. The application suite provides both non-optimized and optimized versions of computational kernels for targeted platforms. The main purpose is benchmarking of emerging new hardware and programming technologies. Non-optimized versions are provided to allow analysis of pure compiler-based optimizations. Minimod is currently not publicly available; however, the plan is to eventually make it available to the community as open-source software.

In this work, we study one of the kernels contained in Minimod, the isotropic propagator in a constant-density domain [28]. For this propagator, the wave equation PDE has the following form:

$$\frac{1}{\mathbf{V}^2}\frac{\partial^2 \mathbf{u}}{\partial t^2} - \nabla^2 \mathbf{u} = \mathbf{f}, \tag{1}$$

where $\mathbf{u} = \mathbf{u}(x, y, z)$ is the wavefield, $\mathbf{V}$ is the Earth model (with velocity as rock property), and $\mathbf{f}$ is the source perturbation. The equation is discretized in time using a second-order centered stencil, resulting in the semi-discritized equation:

$$\mathbf{u}^{n+1} - \mathbf{Q}\mathbf{u}^n + \mathbf{u}^{n-1} = \left(\Delta t^2\right) \mathbf{V}^2 \mathbf{f}^n, \text{with } \mathbf{Q} = 2 + \Delta t^2 \mathbf{V}^2 \nabla^2. \tag{2}$$

Finally, the equation is discretized in space using a 25-point stencil in 3D space, with four points in each direction as well as the centre point:

$$\begin{aligned}\nabla^2 \mathbf{u}(x, y, z) \approx \sum_{m=0}^{4} & c_{xm} \left[\mathbf{u}(i+m, j, k) + \mathbf{u}(i-m, j, k)\right] && + \\ & c_{ym} \left[\mathbf{u}(i, j+m, k) + \mathbf{u}(i, j-m, k)\right] && + \\ & c_{zm} \left[\mathbf{u}(i, j, k+m) + \mathbf{u}(i, j, k-m)\right] &&\end{aligned}$$

where c_{xm}, c_{ym}, c_{zm} are discretization parameters.

A simulation in Minimod consists of solving the wave equation at each timestep for some number of timesteps. Pseudocode of the algorithm is shown in algorithm 1. We apply a Perfectly Matched Layer (PML) [5] boundary condition to the boundary regions. The resulting domain consists of an "inner" region where Equation 2 is applied, and the outer "boundary" region where a PML calculation is applied, as shown in Figure 1.

```
  Data: $\mathbf{f}$: source
  Result: $\mathbf{u}^n$: wavefield at timestep $n$, for $n \leftarrow 1$ to $T$
1 $\mathbf{u}^0 := 0$;
2 for $n \leftarrow 1$ to $T$ do
3     for each point in wavefield $\mathbf{u}^n$ do
4         Solve Eq. 2 (left hand side) for wavefield $\mathbf{u}^n$;
5     end
6     $\mathbf{u}^n = \mathbf{u}^n + \mathbf{f}^n$ (Eq. 2 right hand side);
7 end
```

Algorithm 1: Minimod high-level description

We note that the stencil does not have a uniform computational intensity across the domain: the PML regions require more calculations than the inner regions. This suggests an inherent load imbalance that may be amenable to improvement with tasks. Furthermore, a full simulation includes additional kernels, such as I/O and compression. These additional kernels are not evaluated in this study but will be added in the future.

4 Code Structure and Taskification of Minimod

In this section, we describe the code structure of Minimod and explain how it has been ported to a version that makes use of OpenMP tasks. The most computationally expensive component of Minimod (algorithm 1) is the computation

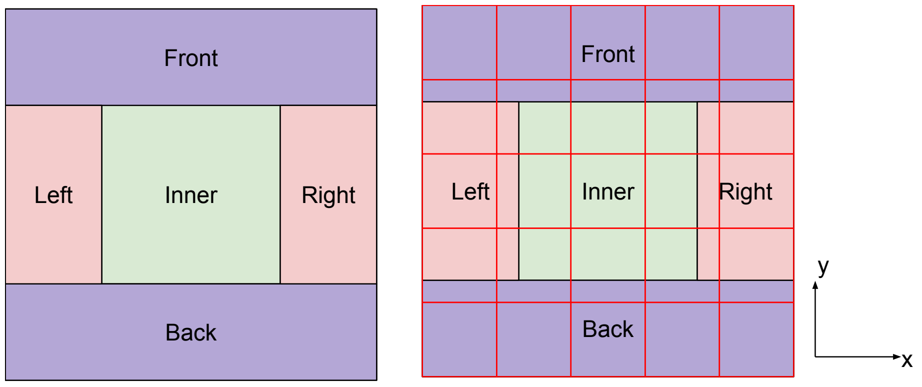

Fig. 1. (left) x-y plane view of domain; (right) xy blocking scheme.

of the wavefield for each point. The (original) serial version of the code has the structure shown in algorithm 2.

```
Data: $u^{n-1}, u^{n-2}$: wavefields at previous two timsteps
Result: $u^n$: wavefield at current timestep
for $i \leftarrow$ xmin to xmax do
    if $i \geq$ x3 and $i \leq$ x4 then
        for $j \leftarrow$ ymin to ymax do
            if $j \geq$ y3 and $j \leq$ y4 then
                // Bottom Damping (i, j, z1...z2)
                // Inner Computation (i, j, z3...z4)
                // Top Damping (i, j, z5...z6)
            else
                // Back and Front Damping (i, j, zmin...zmax)
            end
        end
    else
        // Left and Right Damping (i, ymin...ymax, zmin...zmax)
    end
end
```

Algorithm 2: Wavefield solution step

We evaluate several different configurations for the parallelization of this code, using both OpenMP parallel loops and tasks. In the *x-loop* versions, we simply apply an `omp parallel for` directive to the x-loop on line 1 of algorithm 2. The OpenMP schedule is selected at runtime; we test the static, dynamic, and guided OpenMP schedules in this study.

In addition to simply looping over the x-dimension, we also evaluate the effect of loop blocking in the x-y plane. See Figure 1. In the blocked version, we apply OpenMP loop parallelism to the 2-D loop nest over x-y blocks. Again, we evaluate the static, dynamic, and guided schedules.

In the task-based configurations, we insert a `omp parallel master` region surrounding the entire timestep loop (before line 2 in algorithm 1). Then, in the wavefield solution step we generate tasks representing parallel units of work. The OpenMP `depend` clause is used to manage dependencies between timesteps. In this stencil computation, the computation of each block depends on its neighbors from the previous timestep.

The OpenMP depend clause does not support overlapping array sections as dependencies. The most natural way to express dependencies between the regions is to list, in array section form, the specific array elements that each block depends on. However, this would result in overlapping dependency regions and is therefore not supported. Instead, in our implementation we simply choose one element of each neighboring block to include in the dependency list. This workaround, however, is limited to simple dependence patterns. For example, it is not possible to use more blocks (smaller block size) in the PML regions than in the inner region, because each inner block would depend on multiple PML blocks. OpenMP 5.0 supports using iterators in the depend clause, which provides some additional flexibility; however, iterators are not supported in any compilers we tested.

We evaluate the following configurations in this paper:

- *Loop x static/dynamic/guided*: an OpenMP parallel for loop is applied to the x loop in line 1 of Algorithm 2. A static/dynamic/guided schedule is used.
- *Loop xy static/dynamic/guided*: Uses blocking in the x and y dimensions. A OpenMP parallel for loop is applied to the 2-D loop nest over x-y blocks. (A `collapse(2)` is used to combine the two loops). A static/dynamic/guided schedule is used. Several different block sizes are evaluated.
- *Tasks xy*: Each x-y block is a task. OpenMP's `depend` clause is used to manage dependencies between timesteps.
- *Tasks xy nodep*: Same as above, but OpenMP dependencies are not used. In order to prevent a race condition, an explicit task synchronization point (`taskwait`) is added at the end of the timestep (i.e., before line 7 of Algorithm 1).

An alternative approach, not evaluated here, would be to appy a `taskloop` construct to the loops, generating one task for each chunk of iterations (with configurable size). Currently, the `taskloop` construct does not support dependencies, so an explicit task synchronization would be required, as in *Tasks xy nodep*.

Our application is not currently NUMA-aware, which hurts performance on NUMA architectures, including the nodes used in this study. The conventional NUMA awareness for OpenMP tasks can be achieved with the `affinity` clause of OpenMP 5.0 [17]; however, to the best of our knowledge, this clause is not supported on any publicly available compilers as of the time of writing. (In [17], an LLVM runtime with preliminary support of task affinity is implemented. We are currently evaluating our application with this runtime.) In our application, all data is allocated and initialized by a single thread and so will likely reside on a single NUMA domain.

5 Evaluation

The different versions of Minimod are evaluated on Summit (a supercomputer with IBM POWER9 architecture) and Cori and SeaWulf (supercomputers with an Intel architecture).

5.1 Experimental Setup

Computer	**Hardware**		**Software**
Summit	CPUs	2x IBM Power9	LLVM 9.0
	CPU cores	44 (22 per CPU)	
	Memory	512 GB	
	L3	10 MB (per two cores)	
	L2	512 KB (per two cores)	
	L1	32+32 KB	
	Device fabrication	14nm	
Cori	CPUs	2x Intel Xeon E5-2698v3	LLVM 10.0
	CPU cores	32 (16 per CPU)	
	Memory	128 GB	
	L3	40 MB (per socket)	
	L2	256 KB	
	L1	32+32 KB	
	Device fabrication	22nm	
SeaWulf	CPUs	2x Intel Xeon Gold 6148	LLVM 11.0 (git 3cd13c4)
	CPU cores	40 (20 per CPU)	
	Memory	192 GB	
	L3	28 MB (per socket)	
	L2	1024 KB	
	L1	32+32 KB	
	Device fabrication	14nm	

Table 1. Hardware and software configuration of the experimental platforms.

Summit [25] is a computing system at the Oak Ridge Leadership Computing Facility (see Table 1 top panel). Each node also has 6 NVIDIA V100 GPUs; however, we do not use GPUs in this study. We use 42 OpenMP threads in all experiments with each thread bound to a physical core.

Cori [23] is a computing system at the National Energy Research Scientific Computing Center (NERSC) (see Table 1 middle panel). We perform experiments on *Haswell* nodes of Cori. 32 OpenMP threads on the Haswell nodes were used, each thread bound to a physical core (using `OMP_PLACES=cores` and `OMP_PROC_BIND=true`).

SeaWulf is a computing system at Stony Brook University. Details are given in Table 1 (bottom panel). In each run, we use 40 OpenMP threads (one per physical core) with each thread bound to a physical core.

Each simulation is run with grid sizes between 64^3 (64 in each of the three dimensions) and 1024^3. Sizes 512^3 and 1024^3 are reported in this paper. Results with the LLVM compiler on each computer are reported in this paper. Cache statistics were collected using the Perf and HPCToolkit [20] profilers. Execution times are averaged over three trials on Summit and SeaWulf. We were unable to compute a three-run average on Cori due to lack of availability; however, the application shows little variation in run time on the other machines, so it likely would make little difference.

5.2 Results

Execution times for each configuration from Section 4 on all three platforms are shown in Figure 2. For each of the xy-blocked configurations, the time shown is for the block size that gives the lowest execution time for each configuration. On Cori, poor performance is seen from "Loop x static" as compared to other configurations. Performance among the xy-blocked configurations are generally quite similar.

To understand the relative performance and how it relates to the architecture used, we gathered cache use statistics for each configuration. Table 2 shows the L3 miss rate for each configuration on Summit, Cori, and SeaWulf, respectively. On Summit, the miss rate is significantly lower for static configurations than for the other configurations. On Cori and SeaWulf, the L3 miss rate is highest for "Loop x static", and relatively similar among all xy-blocked configurations.

	Grid size 512^3			**Grid size** 1024^3		
	Summit	Cori	SeaWulf	Summit	Cori	Seawulf
Loop x static	16	19	49	12	16	50
Loop x dynamic	42	10	27	35	9	28
Loop x guided	45	15	41	36	15	47
Loop xy static	9	11	47	7	12	43
Loop xy dynamic	26	11	45	27	11	43
Loop xy guided	26	12	47	22	12	44
Tasks xy	27	12	45	27	11	43
Tasks xy nodep	27	11	45	26	11	43
Average	27.3	12.6	43.3	24.0	12.1	42.6

Table 2. L3 miss rate [%] on each computer for each configuration.

Figure 3 shows the effect of block size on execution time for each of the xy-blocked configurations. The given block size is the size of both the x and y

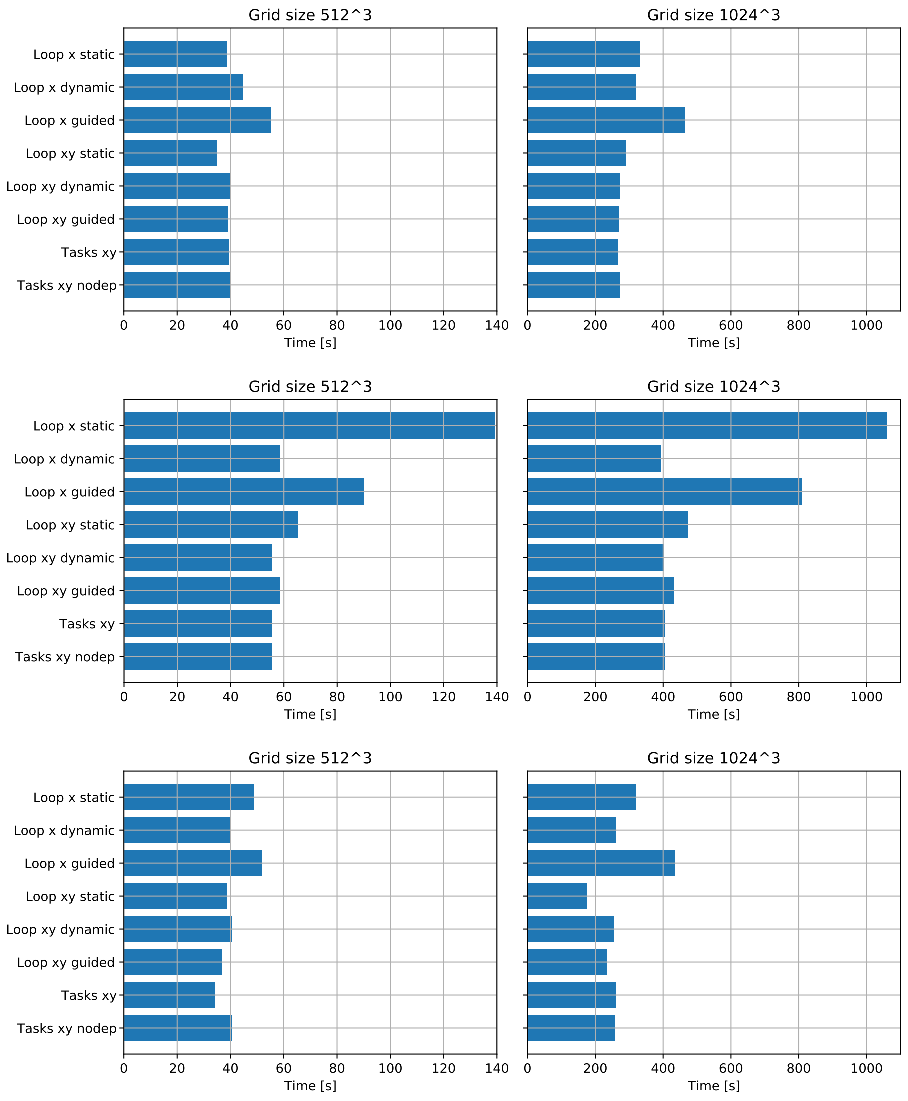

Fig. 2. Execution time (in seconds), top panel Summit, mid panel Cori and bottom panel SeaWulf.

dimensions of each block/task. We frequently see that at a small block size of 4^2, “Tasks xy” does significantly worse than other configurations. Also noteworthy is that at larger block sizes, “Tasks xy” usually outperforms “Tasks xy nodep”, showing the benefit of fine-grained synchronization.

We also ran experiments with other compilers (IBM XL 16.1.1 on Summit, and Intel 19 on Cori and SeaWulf). The general trends discussed here (for the LLVM compiler) also apply to other compilers, indicating that these conclusions are intrinsic to the code and architecture. Due to space constraints, results with the other compilers are not shown here.

6 Discussion

As shown in Table 2, the L3 miss rate on Summit (POWER9 architecture) is lower for static-schedule configurations than other configurations, while for Cori and SeaWulf (Intel architectures) this relationship does not hold. To understand why, we must examine the cache hierarchies of these architectures. On the POWER9 architecture (Summit), the L3 cache is shared between each pair of cores only (Table 1). With a static schedule, the assignment of domain regions to threads does not change between timesteps, and data resident in the L3 cache will be reused at subsequent timesteps. With non-static schedules (including tasks), the assignment of domain regions to threads is arbitrary and can change at each timestep, introducing L3 cache misses (and an expensive fallback to main memory) when a region moves to a different pair of physical cores. On Intel architectures (Cori and SeaWulf), the L3 cache is shared on the entire socket, so movement of regions between timesteps does not cause L3 cache misses unless the movement is between sockets.

A notable trend in the block size plots (Figure 3) is that for very small block sizes (i.e., 4^2), there is a large overhead seen in “Tasks xy”. This sensitivity is usually not seen in the other configurations (although on SeaWulf a similar time increase occurs in the “Tasks xy nodep” configuration). This indicates that the LLVM OpenMP runtime has a significant overhead associated with scheduling small tasks. The difference between “Tasks xy” and “Tasks xy nodep” suggests that there is also a significant overhead associated with handling the dependencies between tasks for fine-grained synchronization. The bulk synchronization of “Tasks xy nodep” (task synchronization at the end of each timestep) has less overhead.

Most of the block size experiments in Figure 3 show that there is a “minimum point”, usually around a square block size of 16-32, where the execution time is minimized. In general, there is a trade-off with respect to choosing a block size. Small block sizes expose more parallelism to the runtime, resulting in more opportunities for load balancing. However, as each block is a task that must be scheduled for execution, small block sizes incur increased runtime task scheduling overhead. It is interesting to see that the minimum point for block size is relatively similar across computers in Figure 3.

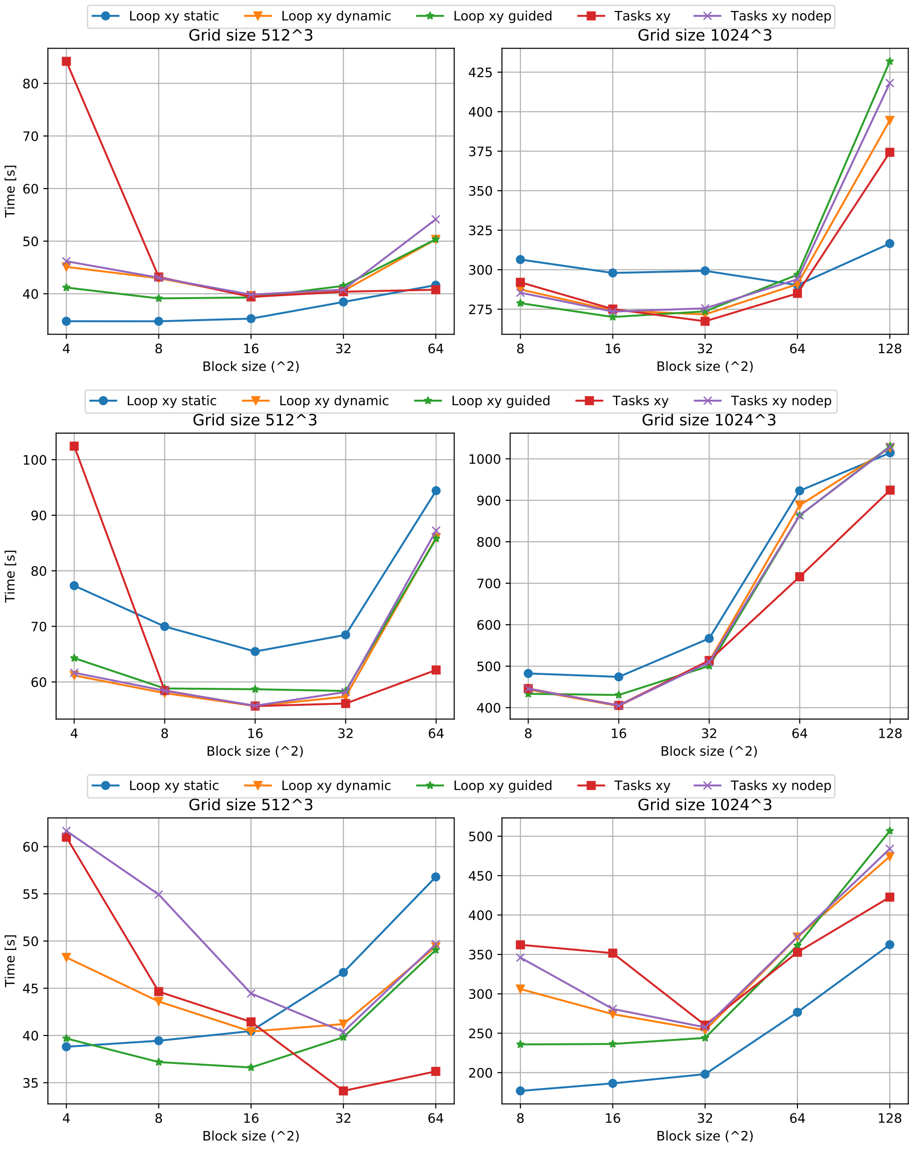

Fig. 3. Effect of block size on execution time using the LLVM compiler on Summit (top), Cori (middle), and SeaWulf (bottom).

Especially at larger block sizes, we see a significant improvement of “Tasks xy” over “Tasks xy nodep”. This shows potential for improvement of the fine-grained synchronization provided by task dependencies. However, this improvement is diminished at smaller block sizes. If the overheads of task dependency resolution could be reduced, this approach might also benefit smaller block sizes.

7 Conclusions

In this paper, the Minimod application was ported to use OpenMP tasks. Even for this relatively regular stencil application, task-based parallelism is competitive with traditional loop-based parallelism, and is even better in some experiments. This is a promising result for the effectiveness of OpenMP tasking.

A key finding of this paper is that the movement of domain region computations between timesteps is more expensive on the POWER9 architecture than on Intel architectures due to the difference in L3 cache hierarchy between them (Section 6). This stresses the importance of locality-aware task scheduling and suggests that the optimal policies for such a scheduler may be architecture-dependent. The `affinity` clause introduced in OpenMP 5.0 may help improve the locality of tasks, increasing performance. The OpenMP metadirective, also introduced in version 5.0, could potentially help set scheduling parameters for different target platforms.

As discussed in Section 6, our results indicate the potential for decreasing the overhead associated with handling task dependencies. However, task dependencies currently also have a lack of expressivity (see Section 4). Increasing the expressivity without increasing overhead may prove difficult.

More research is needed to pinpoint the causes of these performance characteristics. For example, we plan to use a profiler to continue to explore OpenMP overheads and barriers for each configuration. We would also like to better understand the extent to which tasks move between threads over the simulation. We hope to see better support for tasks from performance tools.

In future work, this code will be ported to GPUs using OpenMP 4.0+ offloading features, including using tasks to coordinate the work of multiple GPUs. We would also like to extend the code to run on multiple nodes. One possibility is to use MPI to coordinate OpenMP tasks between nodes. We will also add more kernels to Minimod to form a more complete seismic imaging application; in doing so, we expect to further exploit the benefits of task-based parallelism.

Acknowledgements

We would like to thank Total Exploration and Production Research and Technologies for their support of this work. We also thank Vivek Kale at Brookhaven National Laboratory for his help in guiding the experiments in this paper. We used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. We also used resources of the Oak

Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Furthermore, we would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the SeaWulf computing system, which was made possible by a 1.4M National Science Foundation grant (#1531492).

References

1. Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., Kale, L.: Parallel programming with migratable objects: Charm++ in practice. In: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 647–658 (2014). https://doi.org/10.1109/SC.2014.58
2. Atkinson, P., McIntosh-Smith, S.: On the performance of parallel tasking runtimes for an irregular fast multipole method application. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) Scaling OpenMP for Exascale Performance and Portability. pp. 92–106. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_7
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011). https://doi.org/10.1002/cpe.1631
4. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (Nov 2012). https://doi.org/10.1109/SC.2012.71
5. Berenger, J.P.: A perfectly matched layer for the absorption of electromagnetic waves. Journal of Computational Physics **114**(2), 185 – 200 (1994). https://doi.org/10.1006/jcph.1994.1159
6. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. SIGPLAN Not. **30**(8), 207–216 (Aug 1995). https://doi.org/10.1145/209937.209958
7. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J.J.: Parsec: Exploiting heterogeneity to enhance scalability. Computing in Science Engineering **15**(6), 36–45 (2013). https://doi.org/10.1109/MCSE.2013.98
8. de la Cruz, R., Araya-Polo, M.: Algorithm 942: Semi-stencil. ACM Trans. Math. Softw. **40**(3) (Apr 2014). https://doi.org/10.1145/2591006
9. de la Cruz, R., Araya-Polo, M.: Towards a multi-level cache performance model for 3d stencil computation. Procedia Computer Science **4**, 2146 – 2155 (2011). https://doi.org/10.1016/j.procs.2011.04.235, proceedings of the International Conference on Computational Science, ICCS 2011
10. Delannoy, O., Petiton, S.: A peer to peer computing framework: design and performance evaluation of yml. In: Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks. pp. 362–369 (2004). https://doi.org/10.1109/ISPDC.2004.7

11. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel processing letters **21**(02), 173–193 (2011). https://doi.org/10.1142/S0129626411000151
12. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of openmp task scheduling strategies. In: Eigenmann, R., de Supinski, B.R. (eds.) OpenMP in a New Era of Parallelism. pp. 100–110. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79561-2_9
13. Ghosh, S., Liao, T., Calandra, H., Chapman, B.M.: Experiences with openmp, pgi, hmpp and openacc directives on iso/tti kernels. In: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. pp. 691–700 (Nov 2012). https://doi.org/10.1109/SC.Companion.2012.95
14. Gurhem, J., Tsuji, M., Petiton, S.G., Sato, M.: Distributed and parallel programming paradigms on the k computer and a cluster. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. p. 9–17. HPC Asia 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293320.3293330
15. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: Hpx: A task based programming model in a global address space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. PGAS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2676870.2676883
16. Klinkenberg, J., Samfass, P., Bader, M., Terboven, C., Müller, M.S.: Chameleon: Reactive load balancing for hybrid mpi+openmp task-parallel applications. Journal of Parallel and Distributed Computing **138**, 55 – 64 (2020). https://doi.org/10.1016/j.jpdc.2019.12.005
17. Klinkenberg, J., Samfass, P., Terboven, C., Duran, A., Klemm, M., Teruel, X., Mateo, S., Olivier, S.L., Müller, M.S.: Assessing task-to-data affinity in the llvm openmp runtime. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) Evolving OpenMP for Evolving Architectures. pp. 236–251. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-98521-3_16
18. Lee, J., Sato, M.: Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In: 2010 39th International Conference on Parallel Processing Workshops. pp. 413–420 (2010). https://doi.org/10.1109/ICPPW.2010.62
19. Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P.A., Herrmann, F.J., Velesko, P., Gorman, G.J.: Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. Geoscientific Model Development **12**(3), 1165–1187 (2019). https://doi.org/10.5194/gmd-12-1165-2019
20. Mellor-Crummey, J., Fowler, R., Whalley, D.: Tools for application-oriented performance tuning. In: Proceedings of the 15th International Conference on Supercomputing. p. 154–165. ICS '01, Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/377792.377826
21. Meng, J., Atle, A., Calandra, H., Araya-Polo, M.: Minimod: A finite difference solver for seismic modeling. arXiv (2020), https://arxiv.org/abs/2007.06048
22. Moustafa, S., Kirschenmann, W., Dupros, F., Aochi, H.: Task-based programming on emerging parallel architectures for finite-differences seismic numerical kernel. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018: Parallel Processing. pp. 764–777. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-96983-1_54

23. NERSC: Cori, https://docs.nersc.gov/systems/cori/
24. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In: SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–13 (2010)
25. Oak Ridge Leadership Computing Facility: Summit, https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/
26. OpenMP Architecture Review Board: OpenMP Application Programming Interface (Nov 2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, version 5.0
27. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with starss. The International Journal of High Performance Computing Applications **23**(3), 284–299 (2009). https://doi.org/10.1177/1094342009106195
28. Qawasmeh, A., Hugues, M.R., Calandra, H., Chapman, B.M.: Performance portability in reverse time migration and seismic modelling via openacc. The International Journal of High Performance Computing Applications **31**(5), 422–440 (2017). https://doi.org/10.1177/1094342016675678
29. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media (2007)
30. Rico, A., Sánchez Barrera, I., Joao, J.A., Randall, J., Casas, M., Moretó, M.: On the benefits of tasking with openmp. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) OpenMP: Conquering the Full Hardware Spectrum. pp. 217–230. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_15
31. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: A high-productivity programming language for hpc with logical regions. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2807591.2807629
32. Thaler, F., Moosbrugger, S., Osuna, C., Bianco, M., Vogt, H., Afanasyev, A., Mosimann, L., Fuhrer, O., Schulthess, T.C., Hoefler, T.: Porting the cosmo weather model to manycore cpus. In: Proceedings of the Platform for Advanced Scientific Computing Conference. PASC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3324989.3325723
33. Vidal, R., Casas, M., Moretó, M., Chasapis, D., Ferrer, R., Martorell, X., Ayguadé, E., Labarta, J., Valero, M.: Evaluating the impact of openmp 4.0 extensions on relevant parallel workloads. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) OpenMP: Heterogenous Execution and Data Movements. pp. 60–72. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-24595-9_5
34. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of openmp dependent tasks with the kastors benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) Using and Improving OpenMP for Devices, Tasks, and More. pp. 16–29. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_2