Extending OpenMP to Facilitate Loop Optimization

Ian Bertolacci[1(✉)], Michelle Mills Strout[1], Bronis R. de Supinski[2], Thomas R. W. Scogland[2], Eddie C. Davis[3], and Catherine Olschanowsky[3]

[1] The University of Arizona, Tucson, AZ 85721, USA
`ianbertolacci@email.arizona.edu`, `mstrout@cs.arizona.edu`

[2] Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
`{bronis,scogland1}@llnl.gov`

[3] Boise State University, Boise, ID 83725, USA
`eddiedavis@u.boisestate.edu`, `catherineolschan@boisestate.edu`

Abstract. OpenMP provides several mechanisms to specify parallel source-code transformations. Unfortunately, many compilers perform these transformations early in the translation process, often before performing traditional sequential optimizations, which can limit the effectiveness of those optimizations. Further, OpenMP semantics preclude performing those transformations in some cases prior to the parallel transformations, which can limit overall application performance.

In this paper, we propose extensions to OpenMP that require the application of traditional sequential loop optimizations. These extensions can be specified to apply before, as well as after, other OpenMP loop transformations. We discuss limitations implied by existing OpenMP constructs as well as some previously proposed (parallel) extensions to OpenMP that could benefit from constructs that explicitly apply sequential loop optimizations. We present results that explore how these capabilities can lead to as much as a 20% improvement in parallel loop performance by applying common sequential loop optimizations.

Keywords: Loop optimization · Loop chain abstraction
Heterogeneous adaptive worksharing · Memory transfer pipelining

1 Introduction

Efficient use of the complex hardware commonly available today requires compilers to apply many optimizations. OpenMP already supports many of these optimizations, such as offloading regions to accelerators like GPUs and the parallelization of code regions through threads and tasks. However, OpenMP currently ignores the large space of sequential optimizations, such as loop fusion, loop fission, loop unrolling, tiling, and even common subexpression elimination.

One might argue that the compiler can automatically apply traditional sequential optimizations. This approach has worked reasonably well for sequential code regions. However, the space of sequential optimizations, when to apply them, and in what order is complex and OpenMP parallelization complicates it further. Thus, hints, or even prescriptive requirements, from the application programmer could substantially reduce the complexity of the compiler's navigation of that space.

More importantly, the semantics of several existing OpenMP constructs preclude the use of some sequential optimizations prior to the transformations represented by those constructs. Many proposed OpenMP extensions, such as ones to support pipelining [3] and worksharing across multiple accelerators [7,9], increase these restrictions. Others, such as loop chaining [1,2] complicate the optimization space further. Thus, the ability to specify sequential optimizations and when to apply them could substantially improve the compiler's ability to exploit the complex hardware in emerging systems.

A key issue is supplying sufficient data to keep the computational units busy. Sequential optimizations frequently improve arithmetic intensity and memory performance. OpenMP parallelization can also improve memory performance by reducing the memory footprint per thread and thus freeing CPU cache for other data. Many proposed data and pipelining optimizations also address these issues. Often the programmer understands which optimizations to apply and the best order in which to apply them. Other times, the programmer can easily determine that the order in which they are applied will not impact code correctness. Thus, OpenMP extensions should allow the programmer to provide this information to the compiler.

This paper makes the following contributions:

- an initial review of limitations that existing OpenMP constructs and proposed extensions impose on the use of sequential optimizations;
- proposed syntax to allow OpenMP programmers to request the use of sequential optimizations; and
- a discussion of the performance implications, including an initial experimental study, of the current limitations on and the proposed support for sequential optimizations.

Overall we find that flexible support to specify sequential optimizations could use existing compiler capabilities to produce significantly more efficient code.

The remainder of the paper is structured as follows. In the next section, we discuss the limitations that OpenMP semantics place on the use of traditional sequential optimizations. Section 3 then presents our proposed syntax for an initial set of sequential optimizations to include directly in OpenMP. We then present an initial performance study in Sect. 4 that demonstrates that these sequential extensions could provide significant performance benefits.

2 Existing Limitations

Existing OpenMP constructs can require that their semantics be instantiated prior to performing several important sequential optimizations. These requirements can limit the scope and the effectiveness of those optimizations. Other

```
#pragma omp for schedule(static, 1) nowait
for( int i = 0; i < n; ++i )
  A[i] += B[i] * c;
```

Fig. 1. A simple static loop, with a chunk size that interferes with optimization

optimizations may be complicated by the order in which a specific OpenMP implementation interprets OpenMP semantics and applies sequential optimizations. In this section, we discuss common subexpression elimination, unrolling, pipelining, CoreTSAR, and loop chaining.

2.1 Common Subexpression Elimination

A simple example that we do not address in this work is common subexpression elimination (CSE). For CSE, the optimization could take the form of code hoisting to outside of an OpenMP parallel region and creation of firstprivate copies that are stored in registers for each thread. CSE would be performed before parallelization.

2.2 Unrolling

Of particular interest in the context of our proposed extensions are limitations implied by the loop construct. Consider, for example the loop unrolling optimization. In Fig. 1, the specification of a static schedule with a chunk size parameter of one implies that unrolling cannot be performed prior to implementing the schedule. Specifically, since OpenMP requires iterations to be distributed in chunks of the given size, a single thread cannot be provided with any more than one consecutive iteration in this case.

2.3 Pipelining

The expression of data access patterns can be leveraged to allow for pipelining of data transfers and computation in loop offload. This is a topic that we have explored before in the context of pipelining individual loops [3]. Given annotations of the data access pattern of a given iteration in a loop nest, the implementation can implement multiple buffering and an overlapped pipeline of data transfers and computation for the user. Figure 2 shows a simple example of a stencil code, which uses one possible way to describe the data access pattern, by using the iteration variable to split on the `pipeline_map` clause. Specifically, referring to the induction variable `k` in the `pipeline_map` clause expands to a reference to all iterations of the range of iterations from one to `nz-1` as defined by the loop.

A weakness of the previously proposed approach is that it cannot easily be extended to work for asynchronous loops with dependencies. It must be synchronous with respect to other loops at least, if not with respect to host execution. Since the first loop must fully complete before the next can start, loop

```
#pragma omp target \
   pipeline(static[1,3])\
   pipeline_map(to:A0[k-1:3][0:ny-1][0:nx-1])\
   pipeline_map(from:Anext[k:1][0:ny-1][0:nx-1])\
   pipeline_mem_limit(MB_256)
for(k=1;k<nz-1;k++) {
#pragma omp target teams distribute parallel for
   for(i=1;i<nx-1;i++) {
      for(j=1;j<ny-1;j++) {
         Anext[Index3D (i,     j,     k)] =
            (A0[Index3D (i,     j,     k + 1)] +
             A0[Index3D (i,     j,     k - 1)] +
             A0[Index3D (i,     j + 1, k)] +
             A0[Index3D (i,     j - 1, k)] +
             A0[Index3D (i + 1, j,     k)] +
             A0[Index3D (i - 1, j,     k)])*c1
           - A0[Index3D (i,     j,     k)]*c0;
      } }
}
```

Fig. 2. An example stencil kernel using pipelining.

fusion is effectively impossible, at least unassisted. Similarly pipelining complicates tiling since both modify the loop bounds, possibly dynamically. Given extensions to express how the data flows from one loop to the next, this kind of pipelining might be applied to multiple loops in sequence without having to complete one loop in full before beginning the next.

2.4 CoreTSAR

As with the pipelining extensions, CoreTSAR [7,9], the Core Task Size Adapting Runtime, leverages data access pattern information to coschedule loops across different resources such as CPUs and GPUs or coprocessors. The main abstraction is a loop iteration mapping to the accesses made by any given iteration. Figure 3 shows a simple partitioning of a GEMM kernel by its outer loop, splitting the loop by rows according to the `i` index. CoreTSAR uses a somewhat more abstract syntax to describe the data access pattern. Its `hetero` clause selects the devices and schedule to use, and defines that the associated loop over `i` should be split. Each `part_map` clause overloads the array section syntax to represent `split by loop iter:length`.

The information provided by the user, along with the usual loop trip count information, feeds into a scheduler that selects how much of the given iteration range should be provided to each device, and distributes the data accordingly. This scheduler can be any of a wide range, but the `static` and `adaptive` schedules are most common. The `static` schedule is similar to the `static` schedule on the loop construct but adjusted proportionally by the performance of

```
void runGemm(T **a_a, T **b_a, T **c_a) {
  T *a = *a_a, *b = *b_a, *c = *c_a;
#pragma omp target teams distribute parallel \
         for part_map(c[1:N][0:N])          \
         part_map(to: a[1:N][0:N]) map(to: b[0:N*N])\
         hetero(1, all, adaptive, default, 10)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; ++j) {
      c[(i * N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
      }
    }
  }
}
```

Fig. 3. An example GEMM kernel with CoreTSAR.

the hardware targets, while `adaptive` starts out as static and uses a linear-optimization approach to attempt to complete the work given to each device in as close to the same amount of time as possible while minimizing time. The resulting code precludes loop fusion for the same reason as pipelining does. If the loop transformations to generate the dynamically sized inner chunks are applied before sequential optimizations, tiling is also effectively precluded since the loop distribution extension would require information about the tiling to be correct.

2.5 Loop Chain Abstraction and Optimization

A *loop chain* is N ($N > 1$) loop nests with no code between them that explicitly share data. Figure 4 shows an example of a simple loop chain. These are common in stencil applications and present an opportunity for both data-reuse optimizations and temporary-storage optimizations. The loop chain abstraction [5] represents a loop chain as a sequence of loops domains $L_1, L_2, \ldots, L_N$, well defined data space domains $D_1, D_2, \ldots, D_M$, read access functions $Read_{l,d} : L_l \rightarrow D_d$, and write access functions $Write_{l,d} : L_l \rightarrow D_d$. This model of a loop chain provides a framework for developing and implementing scheduling and storage optimizations between loops.

The key optimization applied to loop chains is loop fusion, which is used in conjunction with other loop optimizations such as tiling and wavefront parallelism. However these optimizations can be difficult to apply manually to complex scientific codes, obfuscate the primary computation, and may not be portable. Further, there are many combinations of possible optimizations, including the parameterization of tile sizes; a developer would need significant expertise to know ahead of time which optimizations would be useful.

```
for( int i = lb; i <= ub; i += 1 )
  A[i] = (B[i-1] + B[i] + B[i+1]);

for( int i = lb; i <= ub; i += 1 )
  A[i] = A[i] * (1.0/3.0);
```

Fig. 4. Example of simple loop chain.

The loop chain abstraction allows automated loop fusion, tiling, and parallel wavefronting of loop chains through a source-andto-source compiler [1,2]. A major contribution of that work is an OpenMP-style annotation language that allows the developer to state explicitly the loop chain domains, access functions, and a list of desired optimizations to be applied to the loop chain. Figure 5 shows how the loop chain in Fig. 4 would be annotated with this language. The separation of the `schedule(..)` specification in only one location for the entire loop chain enables the use of autotuning across different potential schedules.

The annotations allow the developer to identify the entire loop chain, each loop nest of the chain, the domains of the loops, and the read and write access functions. The annotations can also be applied to sequences of inlineable functions whose bodies contain the loops and accesses. The annotations then provide the transformation tool information about the code (loop domains and data access patterns) required to perform the optimizations, replacing the need for complicated analysis. Additionally, loop chain scheduling operators (`fuse`, `tile`, `wavefront`, `serial`, and `parallel`) allow the developer to list specific optimizations to apply to the loop chain. Combined, the loop chain and related annotations support easy application of complicated optimizations on new and existing code without requiring cumbersome and unnecessary rewriting and redesigning of the applications. These annotations can be written by non-experts, who can easily change the optimizations that are applied. Thus, the developer can use these optimizations and experiment to find the fastest schedule by changing the scheduling annotations and not the application code.

3 Sequential Optimizations

In this section, we discuss some of the most effective sequential optimizations for scientific codes: loop fusion, tiling, and unrolling. Each have been implemented with pragmas in various tools.

3.1 Fusion

Loop fusion is a common optimization that takes two or more loop bodies and combines them into one. The loop domains may also be shifted to respect data-dependencies that occur between the loops. Loop fusion improves caching behavior by reducing the distance between when data is produced and when it is used.

```
#pragma omplc loopchain schedule( ... )
{
  #pragma omplc for domain(lb:ub) \
     with (i) \
        write A {(i)}, \
        read  B {(i-1),(i),(i+1)}
   for( int i = lb; i <= ub; i += 1 )
     A[i] = (B[i-1] + B[i] + B[i+1]);

  #pragma omplc for domain(lb:ub) \
     with (i) \
        write A {(i)}, \
        read  A {(i)}
   for( int i = lb; i <= ub; i += 1 )
     A[i] = A[i] * (1.0/3.0);
}
```

Fig. 5. Example of annotated source code (schedule omitted)

```
for( int i = lb; i <= ub; i += 1 ){
  A[i] = (B[i-1] + B[i] + B[i+1]);
  A[i] = A[i] * (1.0/3.0);
}
```

Fig. 6. Example of a fused loop chain using `schedule( fuse() )`

Loop fusion can also enable reduction of the amount of temporary storage that a computation requires.

In the loop chain transformation framework [1,2], the `fuse()` scheduling operator specifies the fusion of all loops in the loop chain. The tool uses the read/write information to form a dependency graph and to find the smallest shifts required to make fusion legal. Alternatively, the developer can explicitly specify the shifts that they require. Figure 6 shows the result of the schedule `fuse()` applied to the example loop chain in Fig. 5.

3.2 Tiling

Tiling is a common optimization that breaks a loop into contiguous chunks [4,12,13]. Tiling reduces reuse distances in order to improve caching behavior.

In the loop chain transformation framework [1,2], the `tile( )` scheduling operator specifies tiling of the loops in the loop chain. This operator has three required arguments. First, a tuple indicates the tile-size in each dimension. Currently, tile-size is not parameterizable (a current limitation of the polyhedral code generator, ISL [11]), and requires a constant value. The tiling can be of fewer dimensions than the loop being tiled (for example, strip mining is a one dimensional tiling of a two-or-more dimension space). The second argument to

```
#pragma omp parallel
for( int io = lb; io <= ub; io += 10 ){
  for( int i = io; i <= io+10; i += 1 ){
    A[i] = (B[i-1] + B[i] + B[i+1]);
  }
  for( int i = io; i <= io+10; i += 1 ){
    A[i] = A[i] * (1.0/3.0);
  }
}
// clean-up loop
```

Fig. 7. Tiled loop chain with `schedule( tile( (10), parallel, serial ) )`

the tiling operator specifies the schedule to apply *over* the tiles. The current tool supports several scheduling operators: `tile`; `wavefront`; `serial`; and `parallel`. For example, using the `serial` operator over the tiles would lead to each tile being visited serially, whereas using the `parallel` operator allow each tile to be visited in parallel (with no guarantee of order). Similarly, the third argument to the tiling operator is the schedule that is applied *within* a tile; it can also be any one of the scheduling operators (`tile`, `wavefront`, `serial`, or `parallel`). For example, using the `serial` operator within a tile would lead to each each point *within* a tile being visited serially, whereas using the `parallel` operator would allow all points in a tile to be visited in parallel (with no guarantee of order). Figure 7 show the result of the schedule `tile( (10), parallel, serial )`.

3.3 Unrolling

Many compilers support loop unrolling. Unfortunately no common mechanism for invoking loop unrolling is consistently available. For example, the Intel and IBM XL compilers accept `pragma unroll(n)`. On the other hand, gcc supports `pragma GCC unroll n`. In contrast, the PGI and Microsoft C compilers do not offer any unrolling pragma.

The state of standardization for unrolling directives is similar to the range of non-portable parallel directives before OpenMP, and extending OpenMP to support it should carry the same overall benefit. Adding a new `unroll(n)` clause to the loop directive or the same as a loop chain operator can deliver this behavior portably. As a simple example, Fig. 8 shows a simple loop with the unrolling annotation and the result of unrolling it.

3.4 Interaction Between Optimizations

In the loop chain source-to-source transformation framework, each scheduling operator produces some function mapping from one polyhedral space to another. These functions are composed together when performing the complete transformation. In the more general case, any of the optimization operators or clauses

```
// Annotated
#pragma omp for unroll(2) schedule(static, 1) nowait
for( int i = 0; i < n; ++i )
  A[i] += B[i] * c;
// Expanded
#pragma omp for schedule(static, 1) nowait
for( int i = 0; i < n; i+=2 ) {
  A[i] += B[i] * c;
  A[i+1] += B[i+1] * c;
}
```

Fig. 8. Example of applying an unrolling clause to the loop directive.

Table 1. Descriptions for each of the execution schedules presented.

Legend label	Description
Baseline	Original implementation, series of loops
Fuse	Loop fusion
Tiled 8X8X8	Loop fusion then tile
Tiled 16X16X16	Loop fusion then tile
Tiled 32X32X32	Loop fusion then tile

that we have discussed could be composed, and in some cases even repeated. For example, a loop might be tiled for one size, parallelized, then tiled again for an inner cache on a given core. How these optimizations compose together is beyond the scope of this paper.

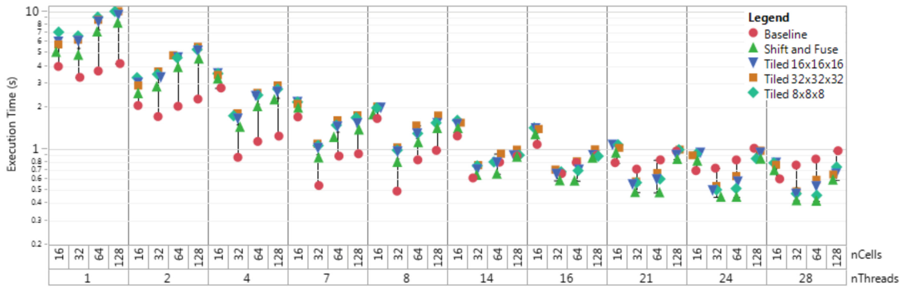
4 Experimental Results

Current results with the proposed loop chain optimizations [2] indicate that stencil applications optimized with loop chain optimizations perform better than the baseline at high thread counts.

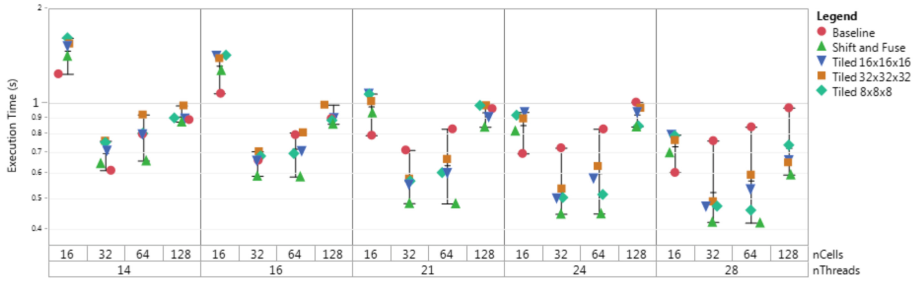
4.1 Loop Fusion and Tiling

MiniFluxDiv[1] captures a subset of the stencil computations implemented in PDE solvers for computational fluid dynamics simulations. It was developed to emulate the behavior of the shared memory portion of code in the Chombo framework [6]. A series of sequential optimizations including loop fusion, iteration space shifting followed by loop fusion, and tiling were applied. The schedules presented here are described in Table 1.

[1] The code for the mini-flux-div benchmark can be found in the Variations on a Theme [10] benchmark repository.



(a)



(b)

Fig. 9. Experimental results of Mini-Flux-Div (a stencil CFD code) micro-benchmark, (a) overall performance results, (b) zoomed view of results for threads 14 through 28

These experiments were run on a single node of the multi-node R2 cluster, at Boise State University. Each node contains a dual socket, Intel Xeon E5-2680 v4 CPU at 2.40 GHz clock frequency with 28 cores (14 per socket).

The sequential optimizations improves performance the most with larger domains and larger core counts. Figure 9 shows that shared memory scaling was improved for the cases with larger domains (128^3). This improvement impacts the full application performance as the interprocess communication can be reduced when larger shared memory domains are utilized.

The fusion optimizations that loop chains can enable also address some of the issues that arise with abstractions that produce many workshared loops nearly in direct contact [8]. They provide an easy way to eliminate the overheads of extra barriers and joins.

4.2 Loop Unrolling

The unrolling optimization can benefit non-chained loops as well as chains. To explore those benefits, we modify the clompk benchmark from the CORAL2 benchmark suite, which measures OpenMP overheads for execution patterns that are roughly consistent with a sparse matrix hydrodynamics code. We produced three unrolling factors for each loop: no unrolling; unroll by two; and unroll by four. We compile with OpenMP in parallel, and separately without any OpenMP directives. Figure 10 presents the performance results.

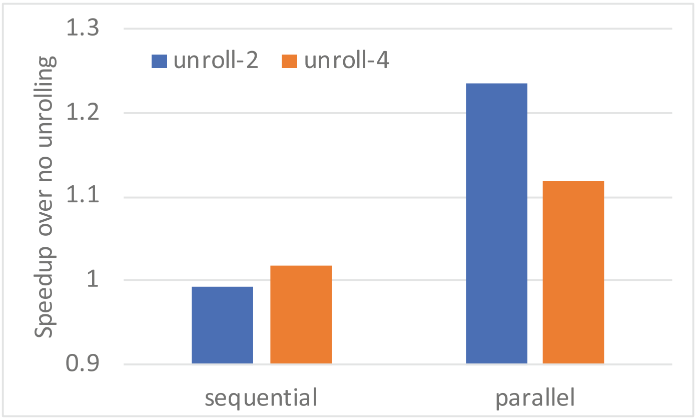


Fig. 10. Performance of clompk with and without unrolling applied.

For the sequential loop variant, an unroll factor of two produces a slowdown of about one percent, while a factor of four yields a speedup of a little more than one percent. The differences there are nearly in the noise, and show little change from the baseline. On the other hand, the OpenMP loops speed up by over 23% with an unroll factor of two, and almost 12% for a factor of four. These differences underscore a key point of this paper: manual unrolling makes almost no difference for serial execution since the compiler can accurately determine the degree to which to perform the transformation. However, manual unrolling can provide substantial benefit in the context of OpenMP. Thus, the version of clompk with OpenMP active runs 10% slower with one thread than the version compiled without OpenMP.

5 Conclusion

Modern systems are complex. They utilize powerful hardware and diverse architectures. Optimizations on loops, such as heterogeneous adaptive worksharing, memory transfer pipelining, and loop chain scheduling, are necessary to achieve the performance offered by these advanced architectures. However, these, and other necessary optimizations, are difficult to implement in the compiler while maintaining all required invariants of parallel programming models like OpenMP, and can be quite costly if added manually by the developer.

While the scope of OpenMP has traditionally been limited to an easy-to-use API for adding parallelism, we argue that these sequential optimizations could be made more accessible and portable by extending OpenMP to support them. The proposed extensions follow the OpenMP model by providing developers pragmas that explicitly prescribe specific optimizations. We combined them with descriptive information that gives the compiler the necessary information, particularly data access patterns, to perform the optimizations legally and efficiently.

Some of these extensions require the compiler to be more intelligent. For example, the `fusion` optimization in loop chaining requires the compiler to determine the required loop shifts to make fusion legal for stencil computations. However, this intelligence is augmented by the descriptive information provided by the annotations. Thus, developers can easily apply advanced and complex optimizations to their application without restricting their ability to maintain and to improve the application around and beyond these optimizations.

References

1. Bertolacci, I.J., Strout, M.M., Guzik, S., Riley, J., Olschanowsky, C.: Identifying and scheduling loop chains using directives. In: Proceedings of the Third International Workshop on Accelerator Programming Using Directives, pp. 57–67. IEEE Press (2016)
2. Bertolacci, I.J., Strout, M.M., Riley, J., Guzik, S.M., Davis, E.C., Olschanowsky, C.: Using the loop chain abstraction to schedule across loops in existing code. Int. J. High Perform. Comput. Netw. (To be published)
3. Cui, X., Scogland, T.R., de Supinski, B.R., Feng, W.C.: Directive-based partitioning and pipelining for graphics processing units. In: International Parallel and Distributed Processing Symposium, pp. 575–584. IEEE (2017)
4. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of the 15th Annual ACM SIGPLAN Symposium on Priniciples of Programming Languages, pp. 319–329 (1988)
5. Krieger, C.D., et al.: Loop chaining: a programming abstraction for balancing locality and parallelism. In: Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), May 2013
6. Olschanowsky, C., Strout, M.M., Guzik, S., Loffeld, J., Hittinger, J.: A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In: The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November 2014
7. Scogland, T.R.W., Feng, W., Rountree, B., de Supinski, B.R.: CoreTSAR: core task-size adapting runtime. IEEE Trans. Parallel Distrib. Syst. **26**, 2970–2983 (2015)
8. Scogland, T.R.W., Gyllenhaal, J., Keasler, J., Hornung, R., de Supinski, B.R.: Enabling region merging optimizations in OpenMP. In: Terboven, C., de Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 177–188. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24595-9_13
9. Scogland, T.R.W., Feng, W., Rountree, B., de Supinski, B.R.: CoreTSAR: adaptive worksharing for heterogeneous systems. In: Kunkel, J.M., Ludwig, T., Meuer, H.W.

(eds.) ISC 2014. LNCS, vol. 8488, pp. 172–186. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_11
10. Strout, M., Olschanowsky, C., Davis, E., Bertolacci, I., et al.: Varitions on a theme (2017). https://github.com/CompOpt4Apps/VariationsOnATheme
11. Verdoolaege, S.: Integer Set Library (2016). http://isl.gforge.inria.fr/
12. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Programming Language Design and Implementation. ACM, New York (1991)
13. Wolfe, M.J.: Iteration space tiling for memory hierarchies. In: Third SIAM Conference on Parallel Processing for Scientific Computing, pp. 357–361 (1987)