LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-670944

Enabling Region Merging Optimizations in OpenMP

T. Scogland, J. Gyllenhaal, J. Keasler, B. de Supinski

May 18, 2015

international workshop on OpenMP
Aachen, Germany
October 1, 2015 through October 2, 2015

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Enabling Region Merging Optimizations in OpenMP ⋆

Thomas R. W. Scogland, John Gyllenhaal, Jeff Keasler,
Rich Hornung, and Bronis R. de Supinski

Lawrence Livermore National Laboratory, Livermore, CA 94551 USA

Abstract. Maximizing the scope of a parallel region, which avoids the costs of barriers and of launching additional parallel regions, is among the first recommendations in any optimization guide for OpenMP. While clearly beneficial and easily accomplished for code where regions are visibly contiguous, regions often become contiguous only after compiler optimization or resolution of abstraction layers. This paper explores the changes to the OpenMP specification that would allow implementations to merge adjacent parallel regions automatically, including the removal of issues that make the transformation non-conforming and the addition of hints that facilitate the optimization. Beyond simple merging, we explore hints to fuse workshared loops that occur in syntactically distinct parallel regions or to apply `nowait` to such loops. Our evaluation shows these changes can provide an overall speedup of $2-8\times$ for a microbenchmark, or 6% for a representative physics application.

1 Introduction

That OpenMP programs should minimize the number of synchronization points and parallel regions is well know optimization guidance. The region's team of threads must be initialized when it begins and joined when it ends, which is a costly synchronization point. Reducing these costs can significantly improve performance, especially for short parallel regions. In the common case, merging contiguous parallel regions with no intervening serial code is a trivial optimization. However, few compilers implement it and instead expect programmers to merge them in the source code. Further, the OpenMP specification does not explicitly allow merging of parallel regions.

While the OpenMP specification does not explicitly allow region merging, a few simple changes would allow it while preserving the semantics of parallel regions. Even with those adjustments however, some cases for which merging is desired would not be facilitated without hints on the parallel constructs that they can be merged safely. This paper proposes the necessary changes to allow region merging, and extensions to the parallel construct to control the merging and generate more efficient code. Specifically, we make the following contributions:

⋆ This material is based upon work supported by the U.S. Department of Energy (LLNL-CONF-670944)

```
// parallel_forall.hpp
template<typename FunT, typename It>
void parallel_forall(It begin, It end, FunT fun){
#if defined(USE_OMP)
#pragma omp parallel for
#endif
  for(It i = begin; i < end; ++i){
    fun(i);
  }
}
// user_code.cc
#include <parallel_forall.hpp>
void foo(double* a, double* b, size_t N){
  std::vector<double> c(N);
  parallel_forall(0, N, [&](int i){
      c[i] = a[i] * b[i];
    }
  );
  parallel_forall(0, N, [&](int i){
      a[i] = sqrt(c[i]);
    }
  );
}
```

Fig. 1. A C++ lambda abstraction over OpenMP

- Specification changes to allow merging of adjacent parallel regions;
- Syntax to guide region merging and synchronization of merged regions;
- Evaluation of region merging benefit with several OpenMP runtimes.

The remainder of the paper is composed as follows. Section 2 describes the proposed optimizations along with the proposed clauses to support them. Section 3 presents our evaluation. Section 4 presents related work.

2 Region Merging and Control

The overhead of starting a thread team has been considered a key performance issue for nearly as long as OpenMP has existed. Recently, however, users have begun to interact with OpenMP and other threading models from higher levels of abstraction. These abstraction levels support portable parallel frameworks across multiple threading models or target devices, including C++ frameworks like RAJA [6] and Kokkos [1]. When using these frameworks, programmers may not even be aware that an OpenMP parallel region exists in their code.

For example, in Figure 2, the `parallel for` is conditionally applied in a library header. While the user could simply fuse the loops of this trivial example, more complex cases can make the source level fusion undesirable or hard to detect by eye. Even if the loops cannot be fused, however, the parallel regions could be

```
void foo(double* a, double* b, size_t N){
  std::vector<double> c(N);
#pragma omp parallel for
  for(int i=0; i < N; i++){
    c[i] = a[i] * b[i];
  }
#pragma omp parallel for
  for(int i=0; i < N; i++){
    a[i] = sqrt(c[i]);
  }
}
```

Fig. 2. Lambda code after the templates are inlined

```
//Library function
void mutate_all(double* a, size_t N){
#pragma omp parallel  //Launch thread team
#pragma omp for //Schedule loop onto threads
  for(int i=0; i < N; i++){
    mutate(a[i]);
  }// Implicit barrier followed by join, threads idle or block
}
void foo(double* a, double* b, size_t N){
  mutate_all(a, N);
  mutate_all(b, N);
}
```

Fig. 3. Function call indirection

if they were accessible. After the template instantiation and inlining passes of the compiler, the function is transformed to the one that Figure **??** shows. Since no user ever sees this code, a compiler optimization pass is the only option to merge the regions short of contorting the abstraction to fit OpenMP's fork/join model. While code using C++ lambdas, or Apple's C blocks, in this way is not yet common, the benefits of this sort of abstractions are numerous so we expect the use of lambdas will soon be common in a wide range of applications.

Even without a higher-level abstraction, cases for which source code merging is difficult or impossible can arise from functions or libraries that contain OpenMP regions, such as the example that Figure 3 shows. The two calls into `mutate_all` do not visibly create OpenMP parallel regions at the call-site but, after an inlining pass, as in Figure 4, region merging is clearly desirable.

The rest of this section first discusses the validity of region merging under the current OpenMP specification. We then detail the additional clauses that we propose to assist compilers in making appropriate and efficient decisions when considering regions for merging.

```
void foo(double* a, double* b, size_t N){
#pragma omp parallel  //Launch thread team
#pragma omp for //Schedule loop onto threads
  for(int i=0; i < N; i++){
    mutate(a[i]);
  }// Implicit barrier followed by join, threads idle or block
#pragma omp parallel //Re-activate thread team
#pragma omp for //Schedule loop onto threads
  for(int i=0; i < N; i++){
    mutate(b[i]);
  }// Implicit barrier followed by join, threads idle or block
}
```

Fig. 4. Regions after inlining

```
void foo(double* a, double* b, size_t N){
#pragma omp parallel //Launch thread team
  {
#pragma omp for //Schedule loop onto threads
    for(int i=0; i < N; i++){
      mutate(a[i]);
    }// Implicit barrier, threads idle or block
#pragma omp for //Schedule loop onto threads
    for(int i=0; i < N; i++){
      mutate(b[i]);
    }// Implicit barrier followed by join, threads idle or block
  }
}
```

Fig. 5. Regions after merging

2.1 Region Merging Validity in OpenMP

Continuing the example in Figure 3, we now define the effect of merging the two regions to produce the version of the code in Figure 5. Since the loop construct ends with an implicit barrier, the memory access ordering and synchronization semantics of the merged version remain identical to the original. However, the specification of parallel regions creates a few key differences. First, the number of threads in a parallel region must agree with Algorithm 2.1 of the specification. In cases with complex nested parallelism, a program could observe the difference in behavior, but the worst result would be over- or under-subscription if the program conforms to the specification. Merging would still preserve correctness.

In addition to the algorithm to calculate the number of threads, the specification explicitly states that only the master thread continues execution of the enclosing task region after the join operation. For a case such as Figure 4, one could reasonably argue that eliminating the join, as in Figure 5, does not cre-

```
void foo(double* a, double* b, size_t N){
#pragma omp parallel  //Launch thread team
  {
#pragma omp for //Schedule loop onto threads
    for(int i=0; i < N; i++){
      mutate(a[i]);
    }// Implicit barrier, threads idle or block
#pragma omp master
    {
      orthogonal_serial_work();//serial
    }
#pragma omp barrier

#pragma omp for //Schedule loop onto threads
    for(int i=0; i < N; i++){
      mutate(b[i]);
    }// Implicit barrier followed by join, threads idle or block
  }
}
```

Fig. 6. A merged pair of regions with an embedded serial section

ate an observable difference despite not temporarily returning to master-only execution. Fundamentally, neither this rule nor the selection of the number of threads make region merging necessarily non-conforming, but they do obscure whether the transformation is conforming. Thus, the specification arguably already allows it. However, we suggest that the OpenMP specification explicitly state that region-merging and similar transformations that result in well-defined semantically consistent behavior are conforming.

2.2 Syntax Extensions to Support Merging

For regions that are lexically back-to-back, possibly after simple transformations such as inlining as in Figure 3, the compiler can easily determine that the merging transformation is both possible and desirable. However, slightly more complicated scenarios, with serial code between the regions, can still benefit from a slightly more advanced transformation, although the extent to which it should be applied is less clear. For this reason, we propose additional clauses for the parallel construct to guide optimization.

First, we propose a `merge` clause. While we could apply this clause to other constructs, we first limit the proposal to the `parallel` construct and the combined or composite constructs for which it is logically the outermost construct. This clause indicates that the compiler should merge the current region with the next construct in the translation unit. It is a descriptive clause (i.e., it is a hint); merging is not required for conformance. However, if another parallel region can be reached then ideally the compiler will merge the two constructs into a single

parallel region, even across serial code. This merge transformation would result in code such as that shown in Figure 6.

This transformation encapsulates the serial region in a `master` region, which ensures that it only runs on a single thread. While a single region might be more efficient, the master region ensures that the same thread executes the code as in the untransformed case, preserving correctness in the presence of thread local state. This transformation does have some drawbacks. We must insert barriers after the first loop region and after the master region in order to preserve the original semantics. However, in our example, the serial region has no dependence on the results of the first loop, so we could theoretically omit the first barrier. In order to support this potentially common use-case, we propose that the `parallel` and `parallel loop` constructs accept the `nowait` clause. If we include this clause on the first construct in the source code that corresponds to our example, the compiler should apply the `nowait` clause to the first loop construct of the transformed code.

Our proposal still requires an extra explicit barrier after the master region, which our example also does not require. We cannot apply a clause to the first construct to remove this barrier because its semantic meaning would involve dependences of the *second* parallel loop construct of the original code. We have considered support for a third clause that addresses dependence on the preceding code. However, it would need to identify data dependences within the region explicitly or otherwise limit its scope. We leave this issue for future proposals due to its complexity and potentially limited applicability.

3 Results and Evaluation

This section presents results for manually merged parallel regions of representative OpenMP applications in the context of four hardware platforms and five OpenMP runtime libraries. We base our microbenchmark on CLOMP [2], which we modify into two new variants that specifically target the overhead of OpenMP region setup and synchronization. The first variant, referred to as *CLOMPK*, has its inner-loop composed of ten back-to-back parallel loop regions that use a function call to update independent values. The second, *CLOMPKS*, interposes dependent serial work between each of the parallel loop regions. To provide a representative application for testing, we also investigate a version of the LULESH [7] benchmark that has been parallelized with RAJA. All results are presented in terms of speedup over an OpenMP execution of the code with no (manual) merging or annotation applied on the same number of cores unless otherwise noted. Table 1 lists the test environment that we use for our evaluation.

3.1 Back to Back Regions

To investigate the purest case, the results in Figure 7 represent the CLOMPK results, which do not include any serial code between the ten parallel loop regions. In order to explore the space, we tested four different levels of merging in addition

Table 1. Evaluated systems. Note: Power8 tests were conducted in little-endian mode

Name	Processor	OMP Runtimes	Cores	Threads/Core	Sockets
BlueGene/Q	IBM PowerPC A2	default, lomp1/2	16	4	1
RZMerl	Intel Xeon E5-2670	Intel, GNU	8	2	2
RZAlastor	Intel Xeon E5-2680	Intel	10	2	2
RZMist	IBM Power8	GNU	12	8	2
Mic	Intel Xeon Phi 7120P	Intel	61	4	1

to the unmerged baseline. The numbers refer to the number of merged regions at a given level, so two refers to running five parallel regions each containing two workshared loops. Since ten does not divide evenly by three, the three case is actually distributed as $3 \times 3 \times 4$. In nearly all cases, basic parallel region merging improves performance, often substantially. Merging pairs of regions gains 2-5% in most cases, although notably it loses approximately 2% with BlueGene/Q's default OpenMP runtime. When merging into groups of three or more, basic parallel merging always improves performance, from a 15% speedup for some cases on BG/Q to over 2× with the GNU OpenMP runtime on Power8.

In principle, the `nowait` option only removes the implicit barrier between parallel loop regions, so it should always improve performance. On all of the Linux-based platforms, including the Intel Xeon Phi accelerator, adding `nowait` provides the expected benefit, producing as much as a 5× speedup over the baseline on Power8. However, the BG/Q default and LOMP1 results suggest that adding the nowait clause can result in more complicated side-effects. With the BG/Q default runtime, it results in a slowdown of as much as 75%. With LOMP1, it remains faster than the baseline, but is a slowdown compared to parallel region merging. With LOMP2, however, the `nowait` option provides benefits on BG/Q similarly to the Linux-based platforms. The overall runtimes also track this trend, with LOMP1 outperforming default, and LOMP2 outperforming LOMP1 in overall runtimes with nowait.

Since each iteration of each of the loops is dependent only on the value produced by the same iteration in the previous loop, we can fuse the loop regions into one loop region. While we do not expect most compilers to implement this optimization, it is valid for back-to-back loop regions that use the static schedule and no cross-iteration dependences. This optimization provides significant performance benefit on BG/Q, achieving as much as a 2× more speedup in one case, but only slightly outperforms the `nowait` option on the Linux-based platforms.

3.2 Parallel Regions with Intervening Serial Regions

Using the `merge` directive discussed in Section 2.2, users can request that compiler merge consecutive parallel regions even with intervening serial code. As previously discussed, parallel region merging is performed with the serial code encapsulated in a master region followed by an additional barrier directive. Figure 8 shows results with this optimization, using the untransformed code, including the intervening serial region as the baseline. The serial regions essentially prevent the loop fusion optimization so we omit the LOOP variant. As expected in

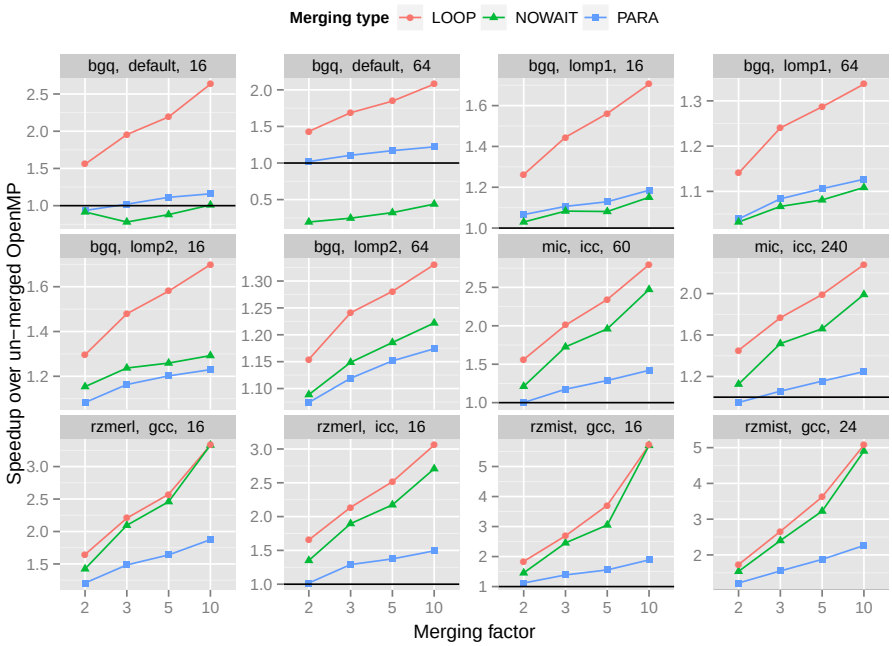

Fig. 7. Performance of back-to-back microbenchmark: parallel region merging (PARA); loop barrier elimination (NOWAIT); and loop fusion (LOOP).

a case with additional serial overhead, as well as the barriers required to protect the master region, the speedups are not as large as with the back-to-back tests. Notably, both the default BG/Q OpenMP runtime and the Intel MIC OpenMP runtime lose performance in some cases. Specifically the BG/Q results for 16 threads slow down by as much as 25%, and on 64 threads by as much as 80% for the use of nowait. This pathological weakness of the default BG/Q OpenMP runtime is not shared by the LOMP runtimes on the same platform. In fact, even with the default runtime as much as a 10% speedup is achieved in the 64 thread case with basic parallel region merging. Other runtimes behave approximately as they do for the back-to-back case. The largest benefits are a speedup of 2.25× for the GOMP runtime on rzmist, the IBM Power8 system.

3.3 Lulesh

Lulesh is a representative application for shock hydrodynamics that serves as a widely available and comparable algorithm for programming model evaluation. It is also composed of a significant number of parallel loop regions, many of which can be fused into larger parallel regions. Figure 9 shows results of four runs each of the original code, which has forty distinct parallel regions with a version in which we manually merged the forty regions into two larger regions. As each of the forty regions performed a large amount of work, the overall application

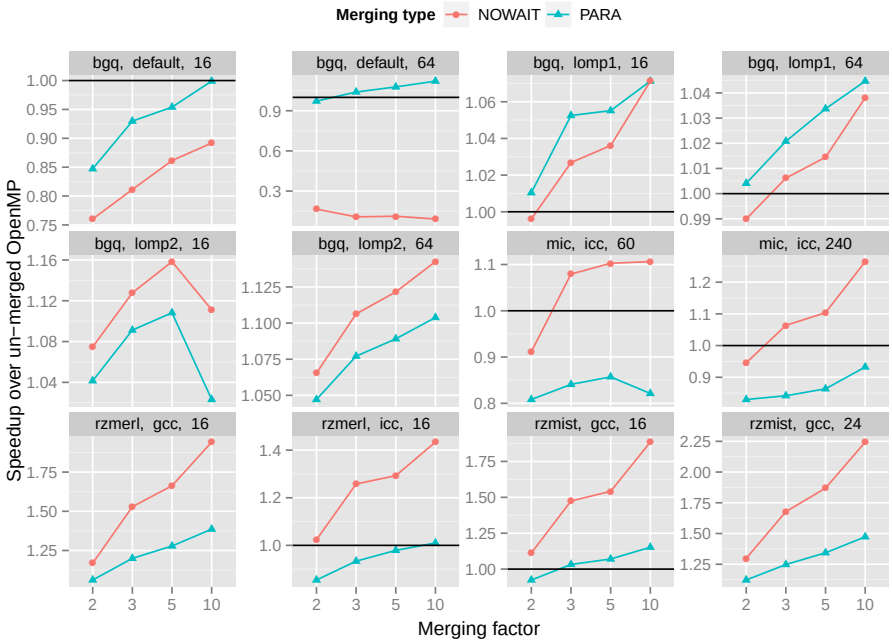

Fig. 8. Performance of intervening serial code microbenchmark: parallel region merging (PARA); and loop barrier elimination (NOWAIT).

speedup is not as large as for the microbenchmarks, but the median speedup remains slightly above 5% on RZAlastor and over 6% on RZMerl. Given that these results apply the optimization to a realistic and reasonably optimized code, 5% is a significant improvement.

4 Related Work

The synchronization, thread launching and scheduling overheads inherent in multi-threaded programming models have been topics of research and discussion since the concept of threading came into being. OpenMP is no exception; the MPCC benchmark suite [3] was published in 1999. At the time, launching a parallel region was approximately as costly as executing a barrier across all threads inside the region. Later work by Müller [8] found that parallel region launch was still as expensive, and recommended merging regions to be as comprehensive as possible. This work also asserts that both the SGI native compilers and the PGI compiler of the time merge contiguous parallel regions based on their performance results. Our tests, discussed in Section 3, show that this optimization does not take place in several current compilers.

Attempts have been made to construct OpenMP compiler and runtime environments that reduce the overhead of launching a parallel region. Eichenberger [5] proposed a lightweight OpenMP runtime, called LOMP, designed to

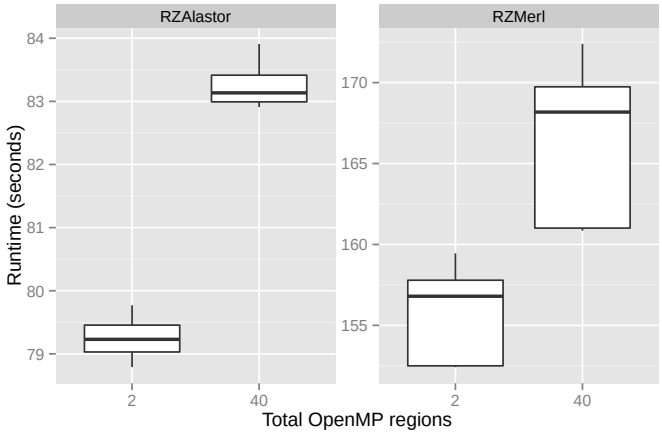

Fig. 9. LULESH runtime with the original 40 regions, or merged into two large regions

cut the overhead to as much as possible on BG/Q while following the OpenMP specification. Some of the optimizations applied included caching thread contexts, distributing region setup to the thread team, the use of low-overhead sleep and wake primitives and creating fast-paths for parallel launches using default parameters. Overall, this work decreased the cost of launching a team of threads across the system, and especially re-entering the region later, but region launch still takes thousands of cycles, and barriers remain a bottleneck. We further evaluate the original LOMP runtime, as well as an updated version, LOMP2, in Section 3.

One exception to our results that show modern compilers do not merge parallel region merging is craycc from the Cray Compiling Environment [4]. It offers the `-h threadn` flag, which when set to a value of two or higher instructs the compiler to attempt to merge, to expand, or otherwise to transform parallel regions. The option documentation implies that these optimizations break strict conformance with the OpenMP specification, which may be why other compilers do not perform parallel region merging.

5 Conclusion

In this paper we re-affirm the common wisdom that merging nearby, and especially contiguous, OpenMP regions improves performance. Further, we identify situations in which regions become contiguous as a result of compiler optimization or evaluation of code at compile time, making the optimization require compiler support. OpenMP as a rule does not require a compiler to perform analysis of the correctness of transformations, and our recommended specification changes maintain that property by ensuring that proper synchronization remains wherever it may be expected. The extended clauses, `merge` and `nowait`, for parallel regions make this possible while giving the user greater control over

the interpretation of a parallel region and the following serial code, if any. Our evaluation shows that for a realistic representative application, region merging can achieve a 5% speedup over the original, and as much as 5× in microbenchmarks. In summary, parallel region merging is an effective optimization with a significant benefit to include in OpenMP compilers.

References

1. Kokkos. `http://trilinos.org/packages/kokkos/`.
2. G. Bronevetsky, J. Gyllenhaal, and B. R. de Supinski. CLOMP: accurately characterizing OpenMP application overheads. In *OpenMP in a New Era of Parallelism*, pages 13–25. Springer, 2008.
3. J. M. Bull. Measuring synchronisation and scheduling overheads in openmp. In *Proceedings of First European Workshop on OpenMP*, volume 8, page 49, 1999.
4. Cray. craycc manual page. `http://docs.cray.com/cgi-bin/craydoc.cgi?mode=View;id=sw_releases-j4spa4zu-1396361754;idx=man_search;this_sort=title;q=;type=man;title=Cray%20Compiling%20Environment%20%28CCE%29%208.3%3a%20C/C%2b%2b/Fortran%20Compiler%20Man%20Pages`.
5. A. E. Eichenberger and K. O'Brien. Experimenting with low-overhead OpenMP runtime on IBM Blue Gene/Q. *IBM Journal of Research and Development*, 57(1/2):8:1–8:8, Jan 2013.
6. R. Hornung and J. Keasler. The RAJA portability layer: Overview and status. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.
7. I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still. Exploring traditional and emerging parallel programming models using a proxy application. *Parallel and Distributed Processing Symposium, International*, 0:919–932, 2013.
8. M. Müller. Some simple OpenMP optimization techniques. In R. Eigenmann and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 31–39. Springer Berlin Heidelberg, 2001.

Prepared by LLNL under Contract DE-AC52-07NA27344.