Compiler Optimizations For OpenMP

Johannes Doerfert $^{[0000-0001-7870-8963]}$ and Hal Finkel $^{[0000-0002-7551-7122]}$

Argonne Leadership Computing Facility,
Argonne National Laboratory, Argonne IL 60439, USA
jdoerfert@anl.gov, hfinkel@anl.gov

Abstract. Modern compilers support OpenMP as a convenient way to introduce parallelism into sequential languages like C/C++ and Fortran, however, its use also introduces immediate drawbacks. In many implementations, due to early outlining and the indirection though the OpenMP runtime, the front-end creates optimization barriers that are impossible to overcome by standard middle-end compiler passes. As a consequence, the OpenMP-annotated program constructs prevent various classic compiler transformations like constant propagation and loop invariant code motion. In addition, analysis results, especially alias information, is severely degraded in the presence of OpenMP constructs which can severely hurt performance.
In this work we investigate to what degree OpenMP runtime aware compiler optimizations can mitigate these problems. We discuss several transformations that explicitly change the OpenMP enriched compiler intermediate representation. They act as stand-alone optimizations but also enable existing optimizations that were not applicable before. This is all done in the existing LLVM/Clang compiler toolchain without introducing a new parallel representation. Our optimizations do not only improve the execution time of OpenMP annotated programs but also help to determine the caveats for transformations on the current representation of OpenMP.

Keywords: OpenMP · Compiler Optimizations · Alias Analysis · Variable Privatization · Barrier Elimination · Communication Optimization.

1 Introduction

In the LLVM/Clang compiler toolchain [9] the use of OpenMP allows for parallel execution but also introduces immediate drawbacks. Due to early outlining and the indirection though the OpenMP runtime, the Clang front-end introduces an optimization barrier that is impossible to overcome by common middle-end optimizations. As a consequence, the OpenMP-annotated program parts do not benefit from classic compiler transformations like constant propagation or loop invariant code motion. Analysis results, especially alias information, is severely degraded in the parallelized program parts. For these and similar reasons, researchers, industry as well as the general LLVM community are currently looking into alternative representations of parallelism in a modern compiler toolchain [11,14].

In order to provide immediate benefit, and to create a meaningful baseline for these efforts, we investigated the feasibility and limitations of program optimizations based on the current representation of OpenMP programs. In the following we present five distinct program transformations that required reasonable implementation, and thereby maintainability effort. They do however enable some of the most important compiler optimizations. In the process, we were able to determine the caveats for transformations on the current representation of OpenMP programs, which use explicit calls to the OpenMP runtime and early outlined parallel program parts.

The rest of this report is organized as follows. We first present necessary background for our work in Section 2. Afterwards, Section 3 through Section 7 describe the new parallel centric optimizations we introduced in the LLVM pipeline. In Section 8 we show preliminary evaluation results of some of these optimizations on different OpenMP benchmarks taken from the Rodinia benchmark suite [4] as well as the LULESH v1.0 benchmark [7]. Finally, we discuss related work in Section 9 and conclude in Section 10.

2 Background

Clang, the C/C++ front-end of the LLVM compiler framework, immediately lowers OpenMP constructs to runtime library calls. The code in parallel regions (`#pragma omp parallel` and similar) is outlined into separate functions. Their addresses as well as all communicated values (`shared` and `firstprivate` clauses) are then passed to a runtime library call. Inside this library function the outlined parallel region is invoke by each thread in the OpenMP thread team. Due to this opaque indirection, passes that work on LLVM's low-level intermediate representation (LLVM-IR) do not need to be aware of any parallel, or OpenMP specific, semantic. For an example consider the code in Figure 1a which is lowered by Clang to LLVM-IR similar to the pseudo C code shown in Figure 1b.

This early outlining approach allows rapid integration of new features and bears little risk of miscompilations due to the function level abstraction and the indirection through the runtime library. Though, this approach will inevitably prevent any optimization to cross the boundary between sequential and parallel code as long as the semantics of the runtime library are not explicitly encoded.

In this work we present several compiler transformations which are aware of the semantics of runtime library calls that implement OpenMP parallel constructs. These transformations are designed to be also applicable to OpenMP tasks runtime calls as well as other parallel language (extensions) expressed in LLVM-IR.

3 Optimization I: Attribute Propagation

Programmers employ attributes, e.g., `const` or `restrict`, to encode domain knowledge in the source code. This is an explicit contract between the programmer and the compiler that limits the set of defined execution traces in the hope of better transformations. Similarly, the compiler might employ attributes to manifest knowledge that was inferred by analyses passes.

The arguably most important use case for attributes are caller-callee boundaries, especially for intra-procedural analyses. Attributes for function parameters provide information about the otherwise unknown inputs while the potential effects of function calls are limited through attributes at the call-site arguments and at the called function.

To improve attributes at the caller-callee boundary of indirectly called parallel program parts, we created an LLVM propagation pass. It communicates the following attributes between pointer arguments in the sequential context and parameter declarations of the parallel work function:

- The absence of pointer capturing[1].
- The access behaviour, thus read-only or write-only.
- The absence of aliasing pointers that are accessible by the callee.
- Alignment, non-nullness and dereferencability information of the pointer.

While all but aliasing information can be simply propagated from the (indirect) call site to the parameter declaration and vice versa, the coarse grained nature of the no-alias attributes, e.g., `restrict` in C/C++ and `noalias` in LLVM's IR, complicates propagation. Even if pointers are known to be alias free in the (sequential) code preceding the parallel region, the `restrict` or `noalias` attribute cannot be simply placed at the parameter declaration to convey this information. First, other arguments could be derived from the alias free pointer which would introduce aliasing opportunities in the parallel work function. Second, the attributes will break dependences that cross barriers thereby allowing code motion across these sequencing constructs. An example to showcase the second problem is given in Figure 1. The OpenMP annotated C source code in Figure 1a is translated by the front-end to LLVM's IR corresponding to the pseudo code shown in Figure 1b. Since the parameter `int* p` is known to be alias free in the (sequential) context of `foo` we want to `restrict` qualify it in the parallel work function as shown in Figure 1c and 1d. This qualification will break the dependence between the accesses to `p` and the call to `bar`, allowing the store-load forwarding performed in Figure 1c. However, `bar` could contain a barrier which would require all increments to be performed prior to any multiplication.

To enable optimizations in the outlined parallel work function we still want to propagate alias information. However, existing analysis and optimization passes might change the semantics if we propagate `restrict`/`noalias` attributes to non-read-only argument pointers. For such alias-free pointer arguments, we have to ensure that their dependences with potential barriers are not eliminated. Since the semantics of both `restrict` and `noalias` are defined based on accesses

[1] A pointer is captured if a copy of it is made inside the callee that might outlive it.

through the syntactic pointer expression, we can prevent any unsound transformation by providing potential barriers `access` to this syntactic expression. This representation is illustrated in Figure 1d. All existing compiler analyses and transformations have then to assume the memory pointed to can be inspected and modified by the potential barrier, ensuring the original memory state when a (potential) barrier is executed.

```
int foo() {
  int a = 0;
#pragma omp parallel shared(a)
  {
#pragma omp critical
    { a += 1; }

    bar();

#pragma omp critical
    { a *= 2; }
  }
  return a;
}
```

(a) OpenMP annotated C source input featuring a call to an unknown function `bar` inside the parallel region.

```
int foo() {
  int a = 0;
  int *restrict p = &a;
  omp_parallel(pwork, p);
  return a;
}
void pwork(int tid, int *p) {
  if (omp_critical_start(tid)) {
    *p = *p + 1;
    omp_critical_end(tid);
  }
  bar();
  if (omp_critical_start(tid)) {
    *p = *p * 2;
    omp_critical_end(tid);
  }
}
```

(b) Pseudo C-style representation of the lowered LLVM-IR produced by Clang for the input in Figure 1a.

```
void pwork(int tid,
           int *restrict p) {
  if (omp_critical_start(tid)) {
    omp_critical_end(tid);
  }
  bar();
  if (omp_critical_start(tid)) {
    *p = 2 * (*p + 1);
    omp_critical_end(tid);
  }
}
```

(c) Unsoundly transformed work function after alias information propagation *if* the call to `bar` contains a barrier.

```
void pwork(int tid,
           int *restrict p) {
  if (omp_critical_start(tid)) {
    *p += 1;
    omp_critical_end(tid);
  }
  bar()[p]; // May "use" pointer p.
  if (omp_critical_start(tid)) {
    *p *= 2;
    omp_critical_end(tid);
  }
}
```

(d) Sound representation after alias information propagation with a pretended use by the potential barrier call `bar`.

Fig. 1: Example illustrating the problematic propagation of `restrict` or `noalias` information from the parallel region context to the parallel work function.

4 Optimization II: Variable Privatization

Writing OpenMP code involves the tedious and error-prone classification of all variables declared outside and used inside the parallel region. Since this classification can have a crucial performance impact we provide a transformation that reclassifies the variables based on their actual usage. Our optimization is performed on low-level LLVM IR and aims to improve both the sequential code as well as the parallel work function. In part, the transformation can be interpreted as strengthening of the OpenMP clauses described below from top to bottom:

– Shared, which indicates any modification might be visible to other threads as well as after the parallel region.
– Firstprivate, which is a private variable but initialized with the value the variable had prior to the parallel region.
– Private, which is a thread-local uninitialized copy of the variable, thus similar to a shadowing re-declaration in the parallel region.

Note that the clause strengthening from `shared` or `firstprivate` to `private` allows the use of separate variables for the sequential and parallel program parts, which enables additional optimizations in both parts. This kind of variable privatization is legal if all of the below stated legality conditions hold:

– The variable is (re-)assigned on all paths from the end of the parallel region that might reach a use,
– Each use of the variable inside the parallel region is preceded by an assignment that is also part of the parallel region.
– There is no potential barrier between the use of a variable and its last preceding assignment.

In addition, we try to communicate variables by-value instead of by-reference. This is sound if they are live-in (`firstprivate` or `shared`) but not live-out nor used for inter-thread communication. Thus, if only the first and last of the above conditions holds we pass the value of the variable instead of the variable container, e.g., the stack allocation.

Finally, non-live-out variables that might be used for communication inside the parallel region can be privatized prior to the parallel region. Hence, if the first of the above conditions holds we replace the variable in the parallel region with a new one declared in the sequential code. This new one is initialized with the value of the original variable just prior to the parallel region. This transformation decouples the variable uses in the two code region and thereby allows for further optimization of the original one in the sequential part.

Note that all transformations have to be aware of potential aliases that could disguise a user variable. In addition, by-value privatization requires the involved type changes to be legal and potentially even a register file transfer[2].

[2]The kmpc OpenMP library used by LLVM/Clang communicates variables via variadic functions that require the arguments to be in integer registers. When a floating point variable is communicated by-value instead of by-reference we have to insert code that moves the value from a floating point register into an integer register prior to the runtime library call and back inside the parallel function.

5 Optimization III: Parallel Region Expansion

Parallel regions introduce an optimization barrier at their boundary. In addition, the start and end of parallel execution can, depending on the hardware, add significant cost. As an example consider the code shown in Figure 2a.

In each iteration of the sequential outer loop two *new* OpenMP thread teams are started to work on the current value of `ptr`, first in forward and then in backward direction. Due to the early outlining, there is no analysis information transfer between the outer loop and the parallel regions nor between the two parallel loops. Furthermore, starting and ending the task teams will eventually accumulate non-trivial cost on the critical path. To decrease this cost and to improve intra-procedural analyses we extend adjacent parallel regions as shown in Figure 2b.

To eliminate the task spawning overhead further, the parallel section can be expanded around sequential constructs as well. This is only possible if the sequential constructs can be guarded appropriately and they do not interfere with the parallel semantics, i.a., they do not throw exceptions. The final code after parallel region expansion is illustrated in Figure 2c.

If a new expanded parallel region is created the contained existing parallel regions are flattened. Thus, the original *`#pragma omp parallel`* annotations, or alternatively the indirection through the corresponding runtime calls, contained in the extended region are removed. Since there is an implicit barrier at the end of a parallel region, we insert an explicit *`#pragma omp barrier`*, or an appropriate runtime call, when parallel regions are flattened. This allows later passes to remove the former implicit, and thereby irremovable, barriers. However, it is important to note that the expanded parallel region will introduce a new implicit barrier at its end.

Accounting for the two implicit barriers after the parallel for loops, the number of barriers in this example increased by one compared to the original code. However, there is now only one parallel region that starts a thread team and there is far more context in the parallel region to enable further optimizations.

6 Optimization IV: Barrier Elimination

Barriers are synchronization points that can be placed manually by the programmer or occur implicitly due to the use of certain OpenMP annotations, i.e., *`#pragma omp parallel for`* without the `nowait` clause. Since synchronization can significantly increase the runtime it should always be used with caution. However, the minimal placement of barriers is an inherently hard and error-prone task even for expert programmers. Since precise dependency information is required to argue about the need for a barrier, and program transformations might be necessary to obtain such information, compilers are well suited to perform this task. To this end, we implemented an OpenMP barrier elimination pass that uses alias information to remove redundant barriers. A barrier is considered redundant if there is no dependence crossing it, thus from the code after

```
while (ptr != end) {
  #pragma omp parallel for firstprivate(ptr)
  for (int i = ptr->lb; i < ptr->ub; i++)
    forward_work(ptr, i);
  #pragma omp parallel for firstprivate(ptr, a)
  for (int i = ptr->ub; i > ptr->lb; i--)
    backward_work(ptr, a, i - 1);
  ptr = ptr->next;
}
```

(a) Example featuring two adjacent parallel regions each containing a parallel for loop.

```
while (ptr != end) {
  #pragma omp parallel for firstprivate(ptr, a)
  {
    #pragma omp for firstprivate(ptr) nowait
    for (int i = ptr->lb; i < ptr->ub; i++)
      forward_work(ptr, i);
    #pramga omp barrier // explicit loop end barrier
    #pragma omp for firstprivate(ptr, a) nowait
    for (int i = ptr->ub; i > ptr->lb; i--)
      backward_work(ptr, a, i - 1);
    #pramga omp barrier // explicit loop end barrier
  }
  ptr = ptr->next;
}
```

(b) Expanded version of the code shown in Figure 2a with a parallel region containing two adjacent parallel for loops.

```
#pragma omp parallel shared(ptr) firstprivate(a)
{
  while (ptr != end) {
    #pragma omp for firstprivate(ptr) nowait
    for (int i = ptr->lb; i < ptr->ub; i++)
      forward_work(ptr, i);
    #pramga omp barrier // explicit loop end barrier
    #pragma omp for firstprivate(ptr, a) nowait
    for (int i = ptr->ub; i > ptr->lb; i--)
      backward_work(ptr, a, i - 1);
    #pramga omp barrier // explicit loop end barrier
    #pragma omp master
    { ptr = ptr->next; }
    #pramga omp barrier // barrier for the guarded access
  }
}
```

(c) Final code after parallel region expansion. The two adjacent parallel for loops as well as the sequential pointer chasing loop are now contained in the parallel region.

Fig. 2: Example to showcase parallel region expansion.

to the code prior. Note that our transformation is intra-procedural and therefore relies on parallel region expansion (ref. Section 5) to create large parallel regions with explicit barriers. In the example shown in Figure 2c it is possible to the eliminate the barrier between the two work sharing loops if there is no dependence between the `forward_work` and `backward_work` functions, thus if they work on separate parts of the data pointed to by `ptr`. If this can be shown, the example is transformed to the code shown in Figure 3. Note that the explicit barrier prior to the *#pragma omp master* clause can always be eliminated as there is no *inter-thread* dependence crossing it.

```
#pragma omp parallel shared(ptr) firstprivate(a)
{
  while (ptr != end) {
    #pragma omp for firstprivate(ptr) nowait
    for (int i = ptr->lb; i < ptr->ub; i++)
      forward_work(ptr, i);
    #pragma omp for firstprivate(ptr, a) nowait
    for (int i = ptr->ub; i > ptr->lb; i--)
      backward_work(ptr, a, i - 1);
    #pragma omp master
    ptr = ptr->next;
    #pramga omp barrier // synchronize the guarded access
  }
}
```

Fig. 3: Example code from Figure 2 after parallel region expansion (ref. Figure 2c) and consequent barrier removal.

7 Optimization V: Communication Optimization

The runtime library indirection between the sequential and parallel code parts does not only prohibit information transfer but also code motion. The arguments of the runtime calls are the variables communicated between the sequential and parallel part. These variables are determined by the front-end based on the code placement and capture semantics, prior to the exhaustive program canonicalization and analyses applied in the middle-end. The code in Figure 5a could, for example, be the product of some genuine input after inlining and alias analysis exposed code motion opportunities between the parallel and sequential part. Classically we would expect `K` and `M` to be hoisted out of the parallel loop and the variable `N` to be replaced by `512` everywhere. While the hoisting will be performed if the alias information for `Y` has been propagated to the parallel function (ref. Section 3), the computation would not be moved into the sequential code part and `N` would still be used in the parallel part. Similarly, the beneficial[3] recompute of `A` inside the parallel function (not the parallel loop) will not happen as no classic transformation is aware of the "pass-through" semantic of the parallel runtime library call.

[3]Communication through the runtime library involves multiple memory operations per variable and it is thereby easily more expensive than one addition for each thread.

```
__attribute__((const)) double norm(const double *A, int n);

void norm(double *restrict out, const double *restrict in, int n) {
  #pragma omp parallel for shared(out, in) firstprivate(n)
  for (int i = 0; i < n; ++i)
    out[i] = in[i] / norm(in, n);
}
```

Fig. 4: Parallel loop containing an invariant call to `norm` that can be hoisted.

The code motion problem for parallel programs is already well known since the example shown in Figure 4 is almost the motivating example for the Tapir parallel intermediate language [11]. Note that the programmer provided (almost[4]) all information necessary to hoist the linear cost call to `norm` out of the parallel loop into the sequential part. Even after we propagate alias information to the parallel function (see Section 3), existing transformations can only move the call out of the parallel loop but not out of the parallel function. Our specialized communication optimization pass reorganizes the code placement and thereby the communication between the sequential and parallel code. The goal is to *minimize* the cost of *explicit communication*, thus variables passed to and from the parallel region, but also the *total number of computations* performed.

Our communication optimization pass will generate a weighted flow graph in which all variables are encoded that are executable in both the parallel and sequential part of the program. For our example in Figure 5a the flow graph is shown in Figure 5c[5]. Each variable is split into two nodes, an incoming one (upper part) and an outgoing one (lower part). The data dependences are represented as infinite capacity/cost (c_∞) edges between the outgoing node of the data producer and the incoming node of the data consumer. The two nodes per variable are connected from in to out with an edge that has the capacity equal to the minimum of the recomputation and communication (c_ω) cost. Since recomputation requires the operands to be present in the parallel region, its cost is defined as the sum of operand costs plus the operation cost, e.g., c_{ld} for memory loads. We also add edges from the source to represent this operation cost flow. If an expression is not movable but used by movable expression it will have infinite cost flowing in from the source. Globally available expressions, thus constants and global variables, have zero communication cost and are consequently omitted. Though, loads of global variables are included as they can be moved. Finally, all values required in the immovable part of the parallel region are connected to a sink node with infinite capacity edges.

The minimum cut of this graph defines an optimal set of values that should be communicated between the sequential and parallel code. For the example code in Figure 5a, the communication graph, sample weights and the minimal cut are shown in Figure 5c. After the cut was performed, all variables for which the

[4]We need to ensure that `norm` is only executed under the condition `n > 0`.

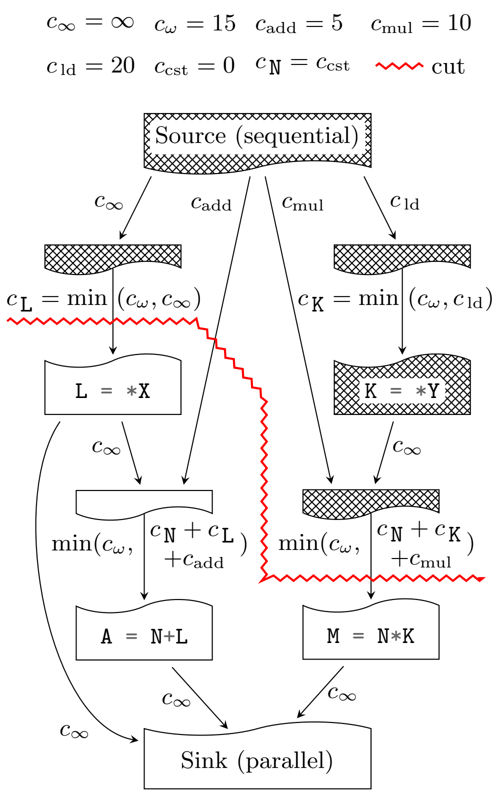
[5] The nodes for `X` are omitted for space reasons. They would look similar to the ones for `L`, though not only allow c_ω flow to the sink but also into the incoming node of `L`.

```
void f(int *X, int *restrict Y) {
  int N = 512;        //   movable
  int L = *X;         // immovable
  int A = N + L;      //   movable
  #pragma omp parallel for        \
      firstprivate(X, Y, N, L, A)
  for (int i = 0; i < N; i++) {
   int K = *Y;        //   movable
   int M = N * K;     //   movable
   X[i] = M+A*L*i;    // immovable
  }
}
```

(a) Function that exposes multiple code movement opportunities between the sequential and parallel part.

```
void g(int *X, int *restrict Y) {
  int L = *X;       // immovable
  int K = *Y;       // c_ld > c_ω
  int M = 512 * K; // c_mul + c_K > c_ω
  #pragma omp parallel             \
            firstprivate(X, M, L)
  {
   int A = 512 + L; // c_add < c_ω
   #pragma omp for                 \
         firstprivate(X, M, A, L)
   for (int i = 0; i < 512; i++)
    X[i] = M+A*L*i; // immovable
  }
}
```

(b) Function shown in Figure 5a after communication optimization.



(c) Communication flow graph[5] for the code shown in Figure 5a. Variable nodes that are partially reachable from the source after the minimal cut are hatched and placed in the sequential part of the result (see Figure 5b).

Fig. 5: Communication optimization example with the original code in part 5a, the constructed min-cut graph in part 5c and the optimized code in part 5b.

incoming node is reachable from the source will be placed in the sequential part of the program. If an edge between the incoming and outgoing node of a variable was cut, it will be communicated through the runtime library. Otherwise, it is only used in one part of the program and also placed in that part. For the example shown in Figure 4 the set of immobile variables would necessarily include `out`, `in` and `n` as they are required in the parallel region and cannot be recomputed. However, the result of the norm function will be hoisted out of the parallel function if the recomputation cost is greater than the communication cost, thus if $c_\omega > c_{\text{call}}$.

In summary, this construction will perform constant propagation, communicate arguments by-value instead of by-reference, minimize the number of communicated variables, recompute values per thread if necessary, and hoist variables not only out of parallel loops but parallel functions to ensure less executions.

8 Evaluation

To evaluate our prototype implementation we choose appropriate Rodinia 3.1 OpenMP benchmarks [4] and the LULESH v1.0 kernel [7]. Note that not all Rodinia benchmarks communicate through the runtime library but some only use shared global memory which we cannot yet optimize. The Rodinia benchmarks were modified to measure only the time spent in OpenMP regions but the original measurement units were kept. All benchmarks were executed 51 times and the plots show the distribution as well as the median of the observed values. We evaluated different combinations of our optimizations to show their individual effect but also their interplay. However, to simplify our plots we only show the optimizations that actually changes the benchmarks and omit those that did not. The versions are denoted by a combination of abbreviation as described in Table 1. Note that due to the prototype stage and the lack of a cost heuristics we did not evaluate our communication optimization.

Table 1: The abbreviations used in the plots for the evaluated optimizations as well as the list of plots that feature them.

Version	Description	Plots
base	plain "-O3", thus no parallel optimizations	Figure 6 - 8
ap	attribute propagation (ref. Section 3)	Figure 6 - 8
vp	variable privatization (ref. Section 4)	Figure 6 - 8
re	parallel region expansion (ref. Section 5)	Figure 7
be	barrier elimination (ref. Section 6)	Figure 8

When we look at the impact of the different optimizations we can clearly distinguish two groups. First, there is *ap* and *vp* which have a positive effect on every benchmark. If applied in isolation, attribute propagation (*ap*) is often slightly better but the most benefit is achieved if they are combined (*ap_vp* versions). This is mostly caused by additional alias information which allows privatization of a variable. The second group contains parallel region expansion (*re*) and barrier elimination (*be*). The requirements for these optimizations are only given in some of the benchmarks (see Figure 7 and respectively Figure 8). In addition, parallel region expansion is on its own not always beneficial. While it is triggered for cfd, srad, and pathfinder it will only improve the last one and slightly decrease the performance for the other two. The reason is that only for pathfinder the overhead of spawning thread teams is significant enough to improve performance, especially since barrier elimination was not able to remove the now explicit barriers in any of the expanded regions. These results motivate more work on a better cost heuristic and inter-pass communication. It is however worth to note that the LULESH v1.0 kernel [7] already contained expanded parallel regions without intermediate barriers. This manual transformation could now also be achieved automatically with the presented optimizations.

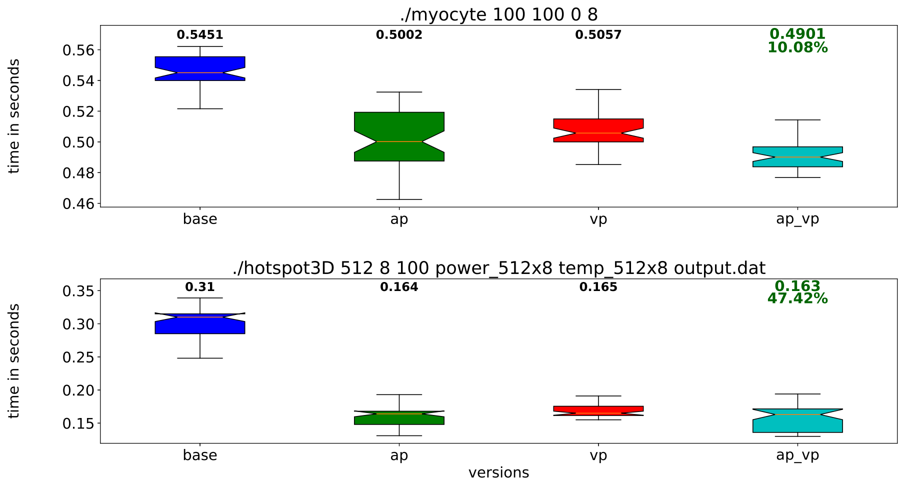


Fig. 6: Performance improvements due to attribute propagation (ref. Section 3) and variable privatization (ref. Section 4).

9 Related Work

To enable compiler optimizations of parallel programs, various techniques have been proposed. They often involve different representations of parallelism to enable or simplify transformations [6,8,15].

In addition, there is a vast amount of research on explicit optimizations for parallel programs [1,2,3,5,10]. In contrast to these efforts we introduce relatively simple transformations, both in terms of implementation and analysis complexity. These transformations are intended to perform optimizations only meaningful for parallel programs, but in doing so, also unblock existing compiler optimizations that are unaware of the semantics of the runtime library calls currently used as parallel program representation.

The Intel compiler toolchain introduces “OpenMP-like” annotations in the intermediate representation IL0 [12,13]. While effective, this approach require various parts of the compiler to be adapted in order to create, handle, and lower these new constructs. In addition, each new OpenMP construct, as well as any other parallel language that should be supported, will require a non-trivial amount of integration effort. Partially for these reasons, Intel proposed a more native embedding [14] of parallel constructions (especially OpenMP) into the intermediate representation of LLVM. Similarly, Tapir [11] is an alternative to integrate task parallelism natively in LLVM. In contrast to most other solutions, it only introduces three new instructions, thus requiring less adaption of the code base. However, it is not possible to express communicating/synchronizing tasks or parallel annotations distributed over multiple functions.

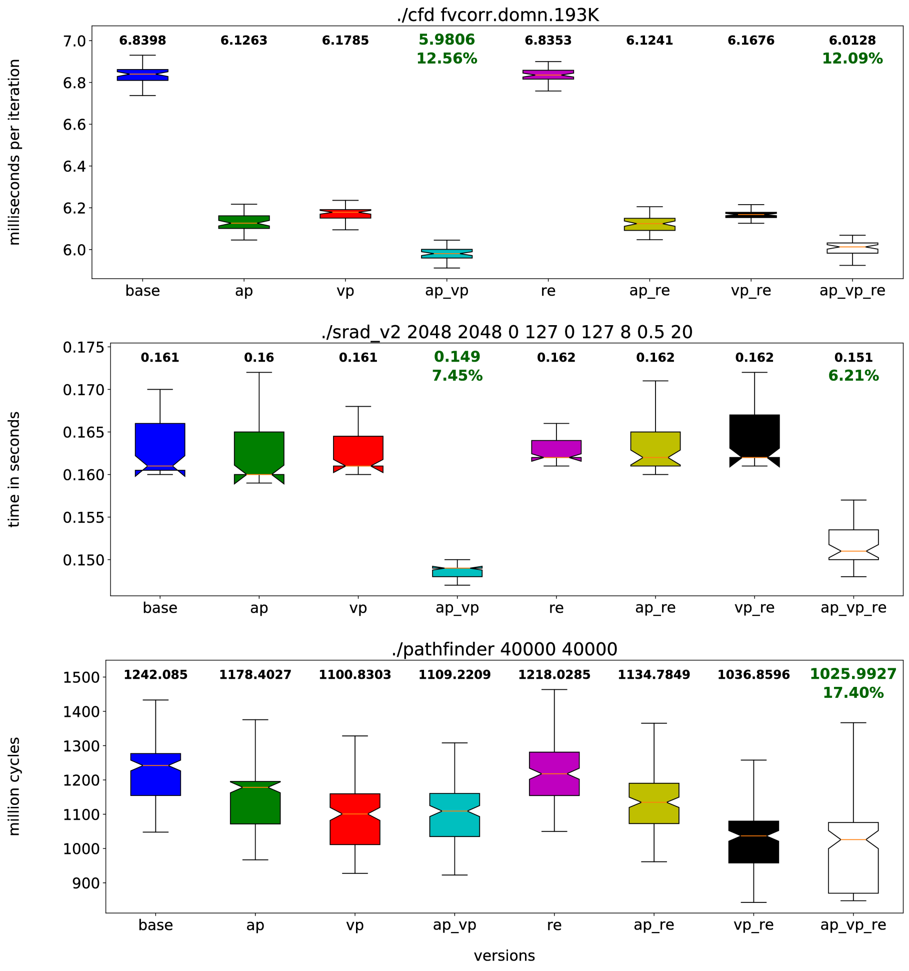


Fig. 7: Performance results for three of our parallel optimizations (*ap*, *vp*, *re*).

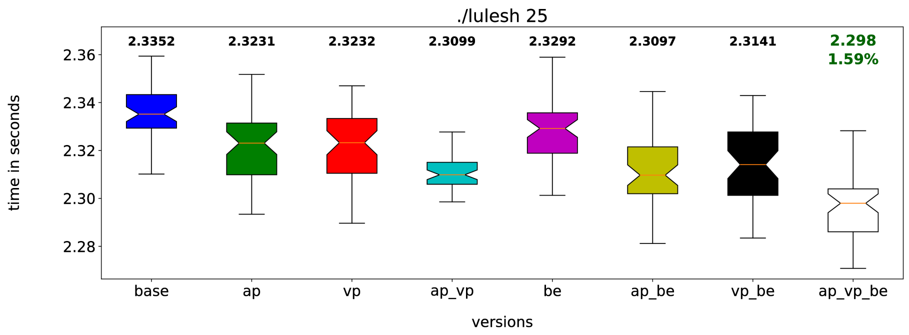


Fig. 8: Performance results for three of our parallel optimizations (*ap*, *vp*, *be*).

10 Conclusion

In this work we present several transformations for explicitly parallel programs that enable and emulate classical compiler optimizations which are not applicable in the current program representation. Our results show that these transformations can have significant impact on the runtime of parallel codes while our implementation does not require substantive implementation or maintainability efforts. While further studies and the development of more robust and cost aware optimizations are underway, we believe our initial results suffice as an argument for increased compiler driven optimizations of parallel programs.

It is worthwhile to note that we are currently proposing to include the presented optimizations into LLVM. To this end we generalized them to allow optimization not only of OpenMP programs lowered to runtime library calls but a more general set of parallel representations.

11 Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

References

1. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel analysis of X10 programs. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007. pp. 183–193 (2007), http://doi.acm.org/10.1145/1229428.1229471
2. Barik, R., Sarkar, V.: Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. In: PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA. pp. 41–52 (2009), https://doi.org/10.1109/PACT.2009.32
3. Barik, R., Zhao, J., Sarkar, V.: Interprocedural strength reduction of critical sections in explicitly-parallel programs. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013. pp. 29–40 (2013), https://doi.org/10.1109/PACT.2013.6618801
4. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA. pp. 44–54 (2009), https://doi.org/10.1109/IISWC.2009.5306797

5. Grunwald, D., Srinivasan, H.: Data flow equations for explicitly parallel programs. In: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993. pp. 159–168 (1993), http://doi.acm.org/10.1145/155332.155349
6. Jordan, H., Pellegrini, S., Thoman, P., Kofler, K., Fahringer, T.: INSPIRE: the insieme parallel intermediate representation. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013. pp. 7–17 (2013), https://doi.org/10.1109/PACT.2013.6618799
7. Karlin, I., Bhatele, A., Chamberlain, B.L., Cohen, J., Devito, Z., Gokhale, M., Haque, R., Hornung, R., Keasler, J., Laney, D., Luke, E., Lloyd, S., McGraw, J., Neely, R., Richards, D., Schulz, M., Still, C.H., Wang, F., Wong, D.: LULESH Programming Model and Performance Ports Overview. Tech. Rep. LLNL-TR-608824
8. Khaldi, D., Jouvelot, P., Irigoin, F., Ancourt, C., Chapman, B.M.: LLVM parallel intermediate representation: design and evaluation using openshmem communications. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015. pp. 2:1–2:8 (2015), http://doi.acm.org/10.1145/2833157.2833158
9. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88 (2004), https://doi.org/10.1109/CGO.2004.1281665
10. Moll, S., Doerfert, J., Hack, S.: Input Space Splitting for OpenCL. In: Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016. pp. 251–260 (2016), http://doi.acm.org/10.1145/2892208.2892217
11. Schardl, T.B., Moses, W.S., Leiserson, C.E.: Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017. pp. 249–265 (2017), http://dl.acm.org/citation.cfm?id=3018758
12. Tian, X., Girkar, M., Bik, A.J.C., Saito, H.: Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs. Comput. J. **48**(5), 588–601 (2005), https://doi.org/10.1093/comjnl/bxh109
13. Tian, X., Girkar, M., Shah, S., Armstrong, D., Su, E., Petersen, P.: Compiler and Runtime Support for Running OpenMP Programs on Pentium-and Itanium-Architectures. In: Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03), April 22-22, 2003, Nice, France. pp. 47–55 (2003), https://doi.org/10.1109/HIPS.2003.1196494
14. Tian, X., Saito, H., Su, E., Gaba, A., Masten, M., Garcia, E.N., Zaks, A.: LLVM Framework and IR Extensions for Parallelization, SIMD Vectorization and Offloading. In: Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, November 14, 2016. pp. 21–31 (2016), https://doi.org/10.1109/LLVM-HPC.2016.008
15. Zhao, J., Sarkar, V.: Intermediate language extensions for parallelism. In: Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '11, Proceedings of the compilation of the co-located workshops, DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, and VMIL'11, Portland, OR, USA, October 22 - 27, 2011. pp. 329–340 (2011), http://doi.acm.org/10.1145/2095050.2095103