Compiler Optimizations For Parallel Programs

Johannes Doerfert [0000−0001−7870−8963] and Hal Finkel [0000−0002−7551−7122]

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

Abstract. This paper outlines a research and development program to enhance modern compiler technology, and the LLVM compiler infrastructure specifically, to directly optimize parallel-programming-model constructs. The goal is to produce higher-quality code, and moreover, to remove abstraction penalties generally associated with such constructs. We believe that such abstraction penalties are increasing in importance due to C++ parallel-algorithms libraries and other performance-portability-motivated programming methods.

In addition, we will discuss when, and more importantly when not, explicit parallelism-awareness is necessary within the compiler in order to enable the desired optimization capabilities.

Keywords: Parallel Programming · LLVM · OpenMP · Compiler Optimizations · Intermediate Representation · Programming Models

1 Introduction

Parallel programming, and often heterogeneous programming, is becoming a ubiquitous part of writing high-performance applications for modern architectures. This raises the question how compilers have to adapt to this new reality. To this end, we have to look at several important trends that are intersecting at the present time:

- Parallel processing, and heterogeneous architectures, have become a common reality across much of modern computing technology. Everything from mobile devices to supercomputers offer multiple cores and heterogeneous accelerators.
- Parallel programming models are nowadays commonly used. This includes source-language directives, e.g., OpenMP [6], and OpenACC, and data-parallel languages, e.g., CUDA, and OpenCL [17]. While this additional semantic information should tend to the compiler, the low-level encoding of parallelism in the otherwise sequential compiler intermediate languages generally prevent analyses and optimizations to cross the barrier between sequential and parallel code.
- The use of parallel libraries, including the new parallel C++ STL, but also libraries such as Thrust [4], Kokkos [8], and RAJA [10], is increasing. These libraries provide a way to cleanly integrate parallel and heterogeneous programming constructs into software-engineering practices and, in addition, provides performance-portability benefits.

The result of these trends is that parallel, and heterogeneous, programming is becoming important for a larger class of applications, and moreover, the potential for compiler optimizations in this space increases as well. Because of directives and other language constructs, the compiler can understand the parallel/heterogeneous semantics. At the same time, the level of abstraction is rising thanks to high-level parallel-programming libraries and other performance-portability techniques. To write modular and clean code, a key aspect in modern software-engineering efforts, the description of parallelism is often separated from the actual algorithm. This separation, and other aspects of the aforementioned abstraction, add penalties to the overall performance of the application. It is therefore clear what needs to be done: The compiler should exploit available information to perform optimizations that mitigate common abstraction penalties and aid the programmer's effort to write maintainable, high-performance code.

The work we present here takes place in the context of the LLVM compiler infrastructure [13]. Currently, there are various research groups and companies exploring options to enhance the existing LLVM intermediate representation (LLVM-IR) with parallelism/heterogeneity-aware optimizations. Given that there are already several proposals that show promising results [7,15,20,12], we will primarily focus on a different question, namely: *For what purposes do we require parallelism-aware extensions to the existing code base and when are more general abstractions better suited to enable the desired optimizations?*

To answer this question we will first review some of the fundamental constructs provided by parallel programming models in Section 2. Our focus will be on the "default representation" in the LLVM compiler toolchain and the reasons abstraction penalties occur when these constructs are used. In this context, we elaborate direct consequences of the internal representation as well as additional penalties that arise from otherwise-reasonable uses of modularity, e.g., through parallel libraries. In Section 3 we show how the right abstraction can enable classical compiler optimizations to mitigate abstraction penalties with only marginal changes to their implementation. The limits of existing (sequential) optimization techniques and the need for a specific representation of parallelism in the compiler is afterwards discussed in Section 4. We also provide a brief introduction into related work in Section 5 before we finish with a conclusion and remarks for future research in Section 6.

2 Compiler Representation of Parallel Constructs

Most compilers for non-explicitly-parallel languages are designed with sequential program execution in mind. The LLVM compiler toolchain, on which our work is build, is no exception. When parallelism is present in the input program, e.g., through directive-based language extensions like OpenMP or the (transitive) use of parallel libraries such as pthreads, a layer of indirection in the internal program representation is used to ensure the separation of parallel and sequential program parts. Without this separation, existing optimizations which were written with sequential program execution in mind, and are consequently unaware of the

parallel semantics, will probably miscompile the code. While there are certainly differences in the way this code separation is implemented for parallel libraries and programming models, the general structure is always the same:

- The parallel code is placed into a separate function (or a similar abstraction).
- A runtime-library call is placed at the original location of the parallel code.
- The arguments of the call include the address of the newly-created function as well as a way to access captured variables, e.g., through pointers.

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
{/* Use i, read In, write Out */}
```

(a) Generic parallel loop.

```
int v = ...
#pragma omp task
{/* Use v, read In, write Out */}
```

(b) Generic parallel task.

```
static void body_fn(int i,
       float** In, float** Out);

omp_parallel_for(0, N, &body_fn,
                 &In, &Out);
```

(c) The loop in part 1a after lowering.

```
static void task_fn(int *v,
          float** In, float** Out);

task = omp_alloc_task(&task_fn, &v,
                      &In, &Out);
omp_add_task(task);
```

(d) The task in part 1b after lowering.

Fig. 1: OpenMP constructs (top) and their representation in LLVM (bottom).

In Figure 1 we illustrate this process through examples that depict the lowering of OpenMP constructs[1] as performed by LLVM's C/C++ front-end Clang. The example in Figure 1a features a generic parallel loop. During the lowering to the LLVM intermediate representation (LLVM-IR) its body is outlined into the function `body_fn` and the loop is replaced by a runtime library call as shown in Figure 1c. Depending on the capture declarations, the variables used in the parallel function are either passed "by-value" (for firstprivate) or as shown, "by-reference" (if unspecified or explicitly declared as shared). Depending on the runtime, variables might be passed directly (as shown) or in a compound object. The latter, which is commonly known from the `pthread_create` method but also employed by various parallel libraries, is similar to the way OpenMP tasks are handled. The lowered version of the generic task shown in Figure 1b is illustrated in 1d. The most important conceptual difference between these two examples is the point at which the parallel code is invoked. In the first example the parallel function was directly called, while in the second example a closure is built and execution is potentially delayed.

Confronted only with a low-level encoding of parallelism through runtime-library calls, a compiler can generally not conclude anything about the interaction of the sequential code in the caller with the parallel code in the outlined function. This includes alias information on the pointer values available at the

[1]It is important to note that we use OpenMP only to improve readability. The same situation arises for various other parallel programming models and library solutions.

call site and also argument usage information that can be derived from the parallel function. As an example, for the latter we could assume that the compiler determines both the `In` pointer and its address, which might be captured by the runtime calls or the parallel functions (`body_fn` and `task_fn`) are only read. However, even if that is determined for these parallel functions, the information could not be used at the call sites, e.g., to pass the value of the `In` pointer directly. From a compiler perspective, the problem with the current encoding of parallelism is less related to the actual parallel execution but stems mainly from the indirection through a function pointer and the runtime-library call. While the uncertainty that is induced by this separation is also the reason we can actually compile parallel programs with compilers that are generally unaware of parallel semantics, the information that is lost will often prevent optimizations in both the caller as well as the parallel function [7].

3 Reuse of Parallelism-Unaware Optimizations

To allow classical, parallelism-unaware optimizations to transform parallel code we need to describe the semantics of the low-level parallelism encoding from a sequential standpoint. To this end, we could state that the `omp_parallel_for` function in Figure 1c will invoke its third argument exactly `N` times, with some value between `0` and `N-1` passed as `i`, and the addresses of the pointers `In` and `Out`. Similarly, `omp_add_task` would eventually result in the invocation of the "task function" stored in the closure. Even if we omit the number of invocations and the value ranges for varying arguments like `i`, this description already suffices to perform important transformations using only existing optimization passes.

As an example we can consider function argument promotion, an optimization that tries to communicate an argument that is only read and not captured "by-value" instead of "by-reference". In the context of OpenMP this transformation would correspond to a declaration change for that variable from `shared` to `firstprivate`. As LLVM already has an implementation for argument promotion, it would be optimal if we could reuse it in this context. Similarly, we want to reuse the analyses that propagate information derived for the arguments of a function to the call site and vise versa. The latter allows for example transformations based on the fact that a pointer argument is only passed through to the transitively invoked parallel function and there only read and not captured.

To perform these kind of optimizations with the existing code base, we introduce *transitive call sites* to LLVM. Similar to the already available, and ubiquitously used, *direct call site* abstraction, transitive call sites allow the user to query information on the callee, caller, arguments, and parameters of a call, without explicitly dealing with the underlying instruction. We currently use manual annotations to identify transitive call sites, thus we mark functions that might invoke one of their function pointer arguments later on. The annotation also describes which arguments to the initial callee are only forwarded to the transitive callee, hence not captured or otherwise inspected. Given this information, which we plan to automatically derive in the future, we can create the transitive call abstractions that relate the initial caller with the transitively called function.

While we are still in the development stage we already have two analyses passes that act on transitive call site information. The first propagates information on the parameters to transitive call sites. If all call sites are known, the second analysis will propagate globally veritable information from arguments to the corresponding parameters in the callee. In addition, we also enabled argument promotion to work with transitive call sites. This change required us to modify less than 50 lines of code, thus less than 5% of the total size. Even with this minimal investment we already achieve speedups similar to the ones presented by Doerfert and Finkel [7], thus more than 10% improvement for the cfd and srad benchmark from the Rodinia suite [5].

While our initial results are already promising and we strongly believe other existing interprocedural optimizations can be similarly easy generalized to transitive call sites, there still is the closure abstraction that has to be overcome. In fact, most parallel runtimes employ at least argument aggregation, e.g., as known from `pthread_create` function. For lowered OpenMP tasks (ref. Figure 1d) the closure even contains the parallel function pointer. To cope with these additional complications we are looking into different possible extensions of our work, including interprocedural memory tracking.

4 The Need for Parallelism-Awareness

Classically, compilers are written with a sequential execution model in mind. If we want to reuse existing analysis and optimization capabilities for parallel programs, we therefore have to rephrase our problems to match the original sequential mindset. While this is certainly possible for many low-level optimizations, this approach is infeasible for transformations that have to explicitly deal with the parallel semantic. Thus, if we want the compiler to optimize parallel task granularity, eliminate explicit and implicit barriers, or determine cutoff values for parallel execution, we will need to introduce new analyses and transformations.

Most of the currently ongoing work in this area (that we are aware of) is in part considering new optimizations to explicitly alter parallel program execution. However, this effort is often mixed with concerns about the reuse of existing scalar analyses and transformations through the embedding of parallel code into the sequential CFG [12,15,20]. While this is can certainly lead to good solutions, they might be more complex and less focused on their main task, namely to perform explicit parallelism-aware transformations. Especially if we assume we can continue to introduce abstractions that allow the reuse of existing scalar optimizations for parallel programs, it seems non-essential to keep such "reuse" as a requirement in the design of a parallelism-aware compiler extension.

Going forward, we will explore how these ideas can be employed in the heterogeneous setting. Currently, for example, when Clang targets GPUs using OpenMP offloading, the frontend itself decides on the code-generation strategy and generates multiple LLVM modules at this early stage in the pipeline (a module for the host code and modules for each accelerator target). So-called "late outlining" approaches have been discussed that will delay this module splitting

and allow for compiler optimizations to take place across the host/accelerator boundary prior to that point. These may be important because, for example, deciding how to map OpenMP code onto a GPU kernel might depend on what OpenMP features are actually used in that kernel (and that may not be known until after inlining and/or other inter-procedural analyses, plus analysis-enabling optimizations, are employed). How to best adapt the compiler's internal representation to enable this kind of functionality is yet unknown.

5 Related Work

Various techniques have been proposed to enable compiler optimizations for parallel programs. Most of them involve some native embedding of parallelism that allows or simplifies the use of existing transformations [11,12,21,20,15,18,19,16]. In addition, there is a vast body of research on explicitly parallelism-aware optimizations [1,2,3,9,7,14].

In contrast to these efforts, we put our focus on simple abstractions that facilitate the reuse of existing analyses and optimizations. We believe that such abstractions are, when applicable, superior to most parallelism-representation schemes. We base this assessment on the required implementation effort for the already proposed approaches, but also the fact that any change to the compiler's internal program representation induces a non-trivial cost as potential interactions with existing analysis and transformation have to be checked.

6 Conclusion And Future Work

We believe our initial result show that certain optimizations for parallel programs are well within reach of a parallelism-unaware compiler. We will continue to explore the use of transitive call sites and we also plan to investigate new abstractions to facilitate the optimization of scalar and parallel programs alike.

Since our work is still in a prototype state, we refrained from a dedicated evaluation. However, our initial results for the cfd and srad benchmark are already on a par with the improvements reported by Doerfert and Finkel [7]. We consequently believe that new abstractions, and increased use of the existing one, will eventually lead to similar results on various benchmarks.

To facilitate the adaption of this work, and to create an incentive for further refinement, we already proposed parts of our implementation to the LLVM community. While a verdict on the integration was not yet reached, we hope that our minimal intrusive proposal will foster the development of optimizations that cross the current optimization barrier between sequential and parallel code.

7 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. Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for cuda. In: GPU computing gems Jade edition, pp. 359–371. Elsevier (2011)
5. 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
6. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
7. Doerfert, J., Finkel, H.: Compiler optimizations for openmp. In: International Workshop on OpenMP (IWOMP). Springer (2018)
8. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202–3216 (2014)
9. 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
10. Hornung, R.D., Keasler, J.A.: The raja portability layer: overview and status. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA, USA (2014)
11. 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
12. 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
13. 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

14. 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
15. 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
16. Stelle, G., Moses, W.S., Olivier, S.L., McCormick, P.: OpenMPIR: Implementing OpenMP Tasks with Tapir. In: Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2017, Denver, CO, USA, November 13, 2017. pp. 3:1–3:12 (2017), http://doi.acm.org/10.1145/3148173.3148186
17. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science and Engineering **12**(3), 66–73 (2010), https://doi.org/10.1109/MCSE.2010.69
18. 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
19. 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
20. 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
21. 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