Inference and Declaration of Independence: Impact on Deterministic Task Parallelism*

Foivos S. Zakkak, Dimitrios Chasapis, Polyvios Pratikakis, Angelos Bilas
Institute of Computer Science, Foundation for Research and Technology—Hellas
Heraklion, Crete, Greece

Dimitrios S. Nikolopoulos
School of EEECS, Queen's University of Belfast, Belfast, Northern Ireland, UK

ABSTRACT

We present a set of static techniques that reduce runtime overheads in task-parallel programs with implicit synchronization. We use a static dependence analysis to detect non-conflicting tasks and remove unnecessary runtime checks. We further reduce overheads by statically optimizing task creation and management of runtime metadata. We implemented these optimizations in SCOOP, a source-to-source compiler for such a programming model and runtime system. We evaluate SCOOP on 10 representative benchmarks and show that our approach can improve performance by 12% on average.

1. INTRODUCTION

Thread-based programming is a low-level abstraction for describing parallel computations, close to the underlying hardware model of processor cores and communication via shared memory variables. Multithreaded programs require the programmer to reason about all thread interactions and manually synchronize conflicting threads, which is difficult and error prone. Task-based parallelism offers a higher level abstraction to the programmer, making it easier to express parallel computation. Early task-parallel programming models, such as Cilk and OpenMP raise the level of abstraction for expressing parallelism, but still require explicit synchronization. Recent task-based programming models use implicit synchronization, using a task's memory footprint, either inferred by the system or declared by the programmer, to avoid concurrent accesses, or even achieve fully deterministic execution [1, 4].

Statically prohibiting concurrent access to shared memory among parallel tasks is too restrictive, as in many cases only a few instances of conflicting tasks will actually conflict. For this reason, even existing statically verifiable non-dependent task models relax the requirement for strictly non-overlapping task memory footprints. Instead, they use dynamic techniques to detect and correct conflicts at runtime [2].

However, dynamic dependence analysis incurs a high overhead compared to hand-crafted synchronization. It requires a complex runtime system to manage and track memory allocation, check for conflicts, and schedule parallel tasks. Often, the runtime cost of checking for conflicts in pessimistic, or rolling back a task in optimistic runtimes becomes itself a bottleneck, limiting the achievable speedup.

This work aims to alleviate the overhead of dynamic dependence analysis without sacrificing the benefit of implicit synchronization. We use a static dependence analysis to detect and remove runtime dependence checks when unnecessary. We make the following contributions:

- We develop a static analysis that detects independent task arguments and reduces runtime overhead in task-parallel programs with implicit synchronization.
- We implement our analysis in SCOOP, a compiler that extends C with parallel tasks and task-footprint annotations, similar to OpenMP and SMPSs [3]. SCOOP targets the BDDT runtime dependence analysis, improving its performance by inferring and removing unnecessary runtime checks.
- We evaluate the efficiency and precision of the analysis on a representative set of task-parallel benchmarks. SCOOP found 28 out of 31 independent task-arguments in all benchmarks, improving performance by 12% on average and by up to 40%.

2. DESIGN AND IMPLEMENTATION

We have implemented our analysis in SCOOP, a compiler for task-parallel programs with implicit synchronization. We use the BDDT runtime system to schedule tasks and perform the runtime dependence checks that cannot be statically eliminated.

SCOOP is structured as three modules. The first extends the C front-end with support for OpenMP-like `#pragma` directives to define tasks and SMPSs-like syntax to define task footprints. The syntax for declaring task footprints supports strided memory access patterns, so that we can describe multidimensional array tiles as task arguments. When not explicitly given, we assume that the size of a task argument is the size of its type.

The second SCOOP module uses a type-system to generate points-to and control flow constraints, and then solves them to infer argument independence. To increase the analysis precision, we use a context-sensitive, field sensitive points-to analysis, and a context-sensitive control flow analysis.

*The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement n^o 248647.

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.
ACM 978-1-4503-1182-3/12/09.

Benchmark	LOC	Tasks	Total Args	Scalar Args	Analysis Time (s)	Graph Nodes	Safe Args	BDDT (ms)	SCOOP (ms)	Speedup
Black-Scholes	3564	1	8	1	3.17	1790	7	1618	963	1.68
Ferret	30145	1	2	0	699.05	85128	2	3344	3344	1.00
Cholesky	1734	4	16	8	1.06	7571	0	983	981	1.00
GMRES	2661	18	72	20	2.21	7957	9	16640	13947	1.19
HPL	2442	11	63	35	1.47	9330	0	1628	1574	1.03
Jacobi	1084	1	6	0	0.74	3980	0	11499	11588	0.99
SMPSs-FFT	2278	8	36	25	1.17	9270	0	1077	1042	1.03
SPLASH-FFT	2935	4	12	4	1.72	9750	7	2028	1849	1.10
Multisort	1215	2	8	4	1.02	4016	0	3683	3446	1.07
Intruder	6452	1	5	1	19.89	16855	3	12572	10332	1.22

Table 1: Benchmark description, analysis performance, and impact on execution

Moreover, in several benchmarks tasks within loops access disjoint parts of the same array. However, the points-to analysis treats all array elements as one abstract location, producing false aliasing and causing such safe arguments to be missed. To rectify this, in part, we have implemented a simple loop-dependence analysis that discovers when different loop iterations access non-overlapping array elements. This (orthogonal) problem has been extensively studied in the past, resulting in many techniques that can be applied to improve the precision of this optimization.

The final SCOOP phase transforms the input program to use BDDT for creating tasks, disabling BDDT's runtime dependence checks for inferred or declared independent arguments. During code generation, we optimize the output code by producing custom code to interact with the dependence analysis and scheduler, instead of using generic BDDT API calls. This way, SCOOP is able to optimize away loops, `va_args`, some conditionals, and treat scalar arguments specially (by value), further reducing the runtime overhead of creating a task, compared to a generic API for task creation.

BDDT uses a region-based allocator to support dynamic memory allocation in tasks, and allow for tasks that operate on complex data structures. Hence, SCOOP handles task footprints that include dynamic regions specially. Specifically, the task footprint language allows several task arguments to belong to a dynamic region; the task footprint then includes the region instead of the individual arguments, and SCOOP registers only the region descriptor with the dependence analysis.

3. EVALUATION

We evaluate SCOOP on 10 benchmarks and measure the precision and performance of the analysis, as well as the impact of the resulting optimization on execution time and scalability. The left part of Table 1 shows the characterization of each benchmark. The first columns show the size of the source code processed by SCOOP, the number of static task definitions, the number of the total arguments used by the tasks, and the number of the scalar arguments, which are not processed by the static analysis, respectively.

The middle part of Table 1 shows the performance and precision of the static analysis. Namely, the next three columns show the total running time of the static dependence analysis, the number of nodes in the constraint graph, and the number of independent task arguments inferred by the analysis. Note that Ferret, the largest benchmark, creates the largest constraint graph, causing an analysis time of over 11 minutes, as context sensitive analysis is cubic in the size of the constraint graph. To assess the precision of the analysis, we manually examined all benchmarks for independent arguments. We discovered three additional task arguments we believe are independent in GMRES. SCOOP fails to detect these because GMRES uses complex array indexing expressions that cannot be handled by the simple "doall" analysis implemented in SCOOP.

The right part of Table 1 shows the effect of SCOOP on the total running time of all benchmarks on 32 cores. Specifically, the last three columns show the total running time in milliseconds when using BDDT to perform runtime dependence analysis on all task arguments of all tasks, the total running time for the benchmark compiled with SCOOP, where the runtime dependence analysis is disabled for any arguments found safe by the static analysis, and the speedup factor gained by applying SCOOP for each benchmark.

Overall, the average performance improvement from applying SCOOP on all benchmarks is 12%, although this value is not representative, as actual impact greatly varies among benchmarks; the dependence analysis is able to infer safe task arguments only in Black-Scholes, Ferret, GMRES, SPLASH-FFT and Intruder. In these benchmarks, inferring independent arguments has a large impact on the overhead and scalability of the dependence analysis, producing substantial speedup over the original BDDT versions for four benchmarks. The reduction of dependence analysis overhead is not noticeable in Ferret because the tasks are very coarse grain. On the other hand, the analysis did not discover any safe task arguments in the other five benchmarks, which do not gain any significant benefit from SCOOP, other than that resulting from custom code generation.

4. REFERENCES

[1] M. J. Best et al. Synchronization via scheduling: Techniques for efficiently managing shared state. In PLDI'11.

[2] R. L. Bocchino et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[3] J. Perez et al. A dependency-aware task-based programming environment for multi-core architectures. In *ICCC*, 2008.

[4] G. Tzenakis et al. BDDT: block-level dynamic dependence analysis for deterministic task-based parallelism. In *PPoPP*. Poster paper.