BDDT: Block-level Dynamic Dependence Analysis for Deterministic Task-Based Parallelism

George Tzenakis[†] Angelos Papatriantafyllou[†] John Kesapides[†] Polyvios Pratikakis[†]
Hans Vandierendonck[†‡] Dimitrios S. Nikolopoulos[†]

[†]Institute of Computer Science, FORTH
[‡]Department of Computer Science, Ghent University
{tzenakis,angelpap,kesapid,hvdieren,dsn}@ics.forth.gr

Categories and Subject Descriptors D [*1*]: 3

General Terms Performance

1. Introduction

Task-parallel programming models [1, 4, 6] offer a more abstract, more structured way for expressing parallelism compared to threads. In these systems the programmer describes the parts of the program that can be computed in parallel, and does not have to manually create and manage the threads of execution. Such models still require the programmer to manually find and enforce any ordering or memory dependencies among tasks, and also maintain the inherent nondeterminism found in threads, which makes them hard to test and debug, as some executions might not be easy to reproduce. Implicitly parallel models [2, 5, 7, 8] further extend task-parallel models with automatic inference of dependencies; the programmer annotates the program [2, 3, 5, 8]; the system then discovers parallelization and manage dependencies transparently.

Dynamic dependence analysis can discover more parallelism than is possible to describe statically in the program, as it only synchronizes tasks that *actually* access the same resources. To benefit program performance, a dependence analysis must (i) be accurate, so that it does not discover false dependencies; and (ii) have low overhead, so that it does not nullify the benefit of discovering extra parallelism. Existing systems require the programmer to restrict task footprints into either whole and isolated program objects, one-dimensional array ranges, or static compile-time regions. This may cause false dependencies in programs where tasks have partially overlapping or unstructured (irregular) memory footprints, or disallow tasks that operate on a multidimensional tile of a large array. To solve these issues, existing systems use copies, such as marshalling all the relevant row parts of a multidimensional array tile into a buffer, and unmarshalling the result back into the array after the task is done. These techniques incur high overhead, and are cumbersome and error-prone for the programmer to use.

This poster abstract presents a task-parallel runtime system that dynamically discovers and resolves dependencies deterministically among parallel tasks, producing executions equivalent to a sequential execution. Lifting the above restrictions of existing systems, BDDT allows the programmer to specify detailed task footprints on any, potentially non-contiguous, memory address range or multidimensional array tile. We use a block-based dependence analysis with arbitrary granularity, making it easier to apply to existing C programs without having to restructure object or array allocation, introduce buffers and marshalling, or change the granularity of task arguments.

Overall, we make the following contributions:

- We present a novel technique for block-based, dynamic task dependence analysis that allows task arguments spanning arbitrary –potentially non-contiguous– memory ranges, argument overlapping across tasks, and dependence tracking at any granularity. The analysis is tunable, and facilitates balancing accuracy and performance. It is also deterministic, in that it always preserves the sequential program order for all read-after-write memory accesses.
- We have implemented this dependence analysis in BDDT, a runtime system for scheduling parallel tasks. Our implementation is adaptive, the programmer can enable or disable the dependence analysis for each task argument independently to minimize overhead when the analysis is not necessary.
- We have evaluated the performance of our runtime system on a representative set of benchmarks; BDDT performs better than SMPSs (an existing state-of-the-art task runtime that implements dynamic dependence analysis), and is able to handle tasks with one order of magnitude finer granularity than SMPSs (Figure 1(a)). In several benchmarks, dynamic dependence analysis in BDDT discovers additional parallelism, producing speedups of up to $3.6\times$ compared to OpenMP (Figure 1(b)).

2. Dataflow Execution Engine Design

BDDT uses a dataflow execution engine based on block-level dependence analysis for identifying parallel tasks. We assume a programming language where tasks are annotated –through language keywords or directives– with data access attributes, corresponding to three access patterns: read (IN), write (OUT) and read/write (INOUT). The language runtime system detects dependencies between tasks, by comparing the access properties of arguments of different tasks that overlap in memory. BDDT splits arguments into virtual memory blocks of configurable size and analyzes dependencies between blocks. Similarly to whole-object dependence analysis used in tools such as SMPSs and SvS, block-based analysis detects true or anti-dependencies between blocks by comparing block starting addresses and checking their access attributes.

Block-based analysis can detect dependencies between tasks that whole object analysis does not; partially overalapping argu-

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

ments are dependencies if a task writes on the overlapping part. Moreover, it supports tasks with non-contiguous arguments in memory, such as a tile of a multidimensional array. BDDT allows the programmer to adjust block granularity to be coarse enough to amortize overhead, yet fine enough to avoid false positives.

Each task in the program goes through four stages: task *issue* performs dependence analysis; task *scheduling* releases a task for execution when all its dependencies are resolved and inserts the task in a worker's queue; task *execution* executes it; task *release* resolves pending dependencies on the executed task, potentially releasing new tasks. Dynamic dependence analysis causes overheads in the issue and release stages. We design the data structures used in the dependence analysis specifically to minimize these overheads.

To efficiently and transparently maintain and retrieve task-argument metadata, we design a custom memory allocator that allows for fast lookup of metadata while still hiding metadata management in the runtime system. The memory allocator forces allocation in such a way that the location of metadata is efficiently deduced from the memory address.

The dependence analysis on blocks is similar to dependence tracking on whole objects, although a task argument may consist of multiple blocks. We have designed a mechanism that allows multiple blocks to share the same metadata information to reduce overhead. Critical dependence tracking operations operate on one metadata element instead of multiple, which greatly reduces the overhead of dependence tracking. We extend this mechanism to track strided arguments—usually multidimensional array tiles: are tracked on each block individually, the runtime system registers a single metadata element for the whole sparse region.

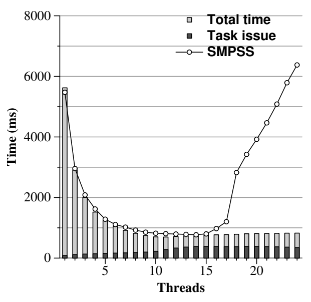
To detect dependencies between tasks, we consider that each block is assigned a new metadata element as new tasks write to it. In principle, each new writing task (i.e., a task that has an output or input/output annotation) creates a new metadata element for the block. All subsequent reading tasks use the same metadata. This divides the task graph into groups of tasks that may execute concurrently and each such group is tracked using a different metadata element. This design allows for an efficient characterization of concurrency while limiting the complexity of the data structures.

3. Implementation

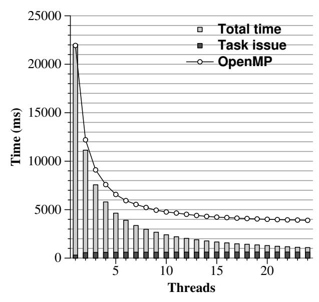
The BDDT block-level runtime system consists of a custom block-based memory allocator; metadata structures for dependence analysis; and a task scheduler.

BDDT Memory Allocator BDDT requires that all memory that constitutes shared state between parallel tasks is allocated through the custom memory allocator. The allocator partitions the virtual address space in *slabs* containing fixed size blocks and services memory allocation requests from such slabs. Data blocks are allocated at the beginning of the slab and the corresponding metadata indices are allocated at the end. By using slabs of fixed size and alignment, we can calculate the address of a block's metadata through very simple and efficient integer arithmetic on the block address, which also increases locality on metadata.

Dependence Analysis The dependence analysis assigns a new version to each block as tasks write new data to it. To do this, BDDT maintains a chain of metadata elements accessed through a metadata array (indexed by the block index stored in the slab), which holds the metadata for each data block. Each metadata element corresponds to a version of the data block, created when a task writes to it. The metadata element includes pointers to the dependent tasks that wait on the corresponding version of the data block, used to resolve task dependencies. Every task element points to metadata elements for all task arguments, an atomic join counter that tracks unresolved dependencies, a function that is the task code, and the actual data, which can be strided array tiles.



(a) BDDT, SMPSs on Jacobi



(b) BDDT, OpenMP on Cholesky

BDDT performs dependence analysis on a per-block basis, depending on the access mode (IN, OUT, INOUT). In each case, the runtime system registers a metadata element for a block when it is first touched. For strided arguments, the runtime system registers a single metadata element for all blocks of the argument, as a single metadata array entry.

Task Scheduling BDDT is based on a master-worker program model. The master is responsible for task issue and dependence analysis, issuing ready tasks to the workers in round-robin order. The workers concurrently perform task scheduling, execution and release. BDDT schedules a task for execution whenever all its dependencies are satisfied. Each worker thread has its own concurrent queue of ready tasks. Upon task completion, the worker walks through the stacks of all of its metadata elements, decrements the join counters of tasks registered with the metadata elements, and releases for execution those tasks with no pending dependencies. Empty workers steal ready tasks from other workers.

4. Conclusions

BDDT is a runtime system that performs dynamic dependence analysis to schedule parallel tasks with memory footprints that span arbitrary memory ranges, producing deterministic execution. BDDT implements efficient and highly concurrent task instantiation, dependence analysis and scheduling techniques. In a set of benchmarks BDDT has similar or better performance than OpenMP, outperforming it by up to a factor of 3.8× at best. BDDT has lower overhead and a more efficient runtime implementation than SMPSs.

References

[1] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE TPDS*, 20(3):404–418, 2009.

[2] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via Scheduling: Techniques for Efficiently Managing Shared State. In *PLDI*, 2011.

[3] R. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[4] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *SC*, 2006.

[5] J. C. Jenista, Y. H. Eom, and B. Demsky. OoOJava: Software Out-of-Order Execution. In *PPoPP*, 2011.

[6] C. E. Leiserson. The Cilk++ Concurrency Platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[7] J. M. Pérez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it Easier to Program the Cell Broadband Engine Processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

[8] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *International Journal of High Perfomance Computing Applications*, 23(3):284–299, 2009.