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

Technical Report FORTH-ICS TR-426 February 2012

George Tzenakis Angelos Papatriantafyllou
Foivos Zakkak Hans Vandierendonck Polyvios Pratikakis
Dimitrios S. Nikolopoulos

August 21, 2012

Abstract

Reasoning about synchronization, ordering and conflicting memory accesses makes parallel programming difficult, error-prone and hard to test, debug and maintain. Task-parallel programming models such as OpenMP, Cilk and Sequoia offer a more structured way of expressing parallelism than threads, but still require the programmer to manually find and enforce any ordering or memory dependencies among tasks. Programming models with implicit parallelism such as SvS, OoOJava, or StarSs lift this limitation by automatically inferring parallelism and dependencies, requiring the programmer to describe the memory footprint of each task. Current limitations of these systems require the programmer to restrict task footprints into either whole and isolated program objects, one-dimensional array ranges, or static compile-time regions; all producing over-approximations and false dependencies that reduce the available parallelism in the program.

This paper presents BDDT, a task-parallel runtime system that dynamically discovers and resolves dependencies among parallel tasks. BDDT allows the programmer to specify detailed task footprints on any memory address range or multidimensional array tile. BDDT uses 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 and provides flexibility in array layouts and tile dimensions. We evaluated BDDT using a representative set of benchmarks, and we compared it to SMPSs (the equivalent runtime system in StarSs) and OpenMP. We found that BDDT performs comparable to or better than SMPSs and is able to cope with task granularity as much as one order of magnitude finer than SMPSs. Compared to OpenMP, BDDT performs up to 3.9× better for benchmarks that benefit from dynamic dependence analysis. BDDT provides additional data annotations to bypass dependence analysis. Using these annotations, BDDT outperforms OpenMP also in benchmarks where dependence analysis does not discover additional parallelism, thanks to a more efficient implementation of the runtime system.

1 Introduction

Despite parallel programming becoming increasingly common, it remains complex, difficult and error-prone, as it requires reasoning about synchronization and ordering between many parallel computations. The dominant programming model of parallel threads with shared memory state is nondeterministic and requires the programmer to reason about implicit interactions and conflicts. Moreover, this nondeterminism makes testing and debugging difficult, as no number of correct executions and test cases can prove the absence of error.

Task-parallel programming models [2, 11, 8] offer a more abstract, more structured way for expressing parallelism than threads. In these systems the programmer only 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. This lifts a lot of the difficulty in describing parallel, independent computations compared to the threading model, but still requires the programmer to manually find and enforce any ordering or memory dependencies among tasks. Moreover, these models maintain the inherent nondeterminism found in threads, which makes them hard to test and debug, as some executions might not be easy to reproduce.

Programming models with implicit parallelism [5, 10, 15, 14] extend task-parallel programming models with automatic inference of dependencies, requiring the programmer to only describe the memory resources required in each task. This is easier to use, as programmers need not discover and describe parallelism —which might be unstructured and dynamic— but can instead annotate the program using compiler directives [15, 10] or language extensions [5, 6]; the compiler and runtime system then discover parallelization and manage dependencies

transparently.

Dynamic dependence analysis offers the benefit of potentially discovering more parallelism than is possible to describe statically in the program, as it only synchronizes tasks that actually (not potentially) access the same resources. In order for a dynamic dependence analysis to benefit program performance, it 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.

Most existing systems, however, 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. For example, a task operating on a multidimensional array tile would require marshalling all the relevant row parts 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.

SMPSs with region support [13], a state-of-the-art implementation of the StarSs programming model for shared-memory multicores with implicit parallelism, supports non-contiguous array tiles and non-unit strides in task arguments. This is, however, at the cost of reduced parallelism due to overapproximation of memory address ranges and high overhead for maintaining a complex data structure used to discover partial overlaps.

This paper presents BDDT, a task-parallel runtime system that dynamically discovers and resolves dependencies deterministically among parallel tasks, producing executions equivalent to a sequential program 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. To allow this, 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, this paper makes 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 —even one byte— thus eliminating false positives. 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 implement 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 (for instance, if the program is embarrassingly parallel.)
- We evaluate the performance of our runtime system. On a representative set of benchmarks, BDDT performs comparable to or better than SMPSs while able to handle tasks with one order of magnitude finer granularity, and arbitrary tile sizes and array dimensions. In several benchmarks, dynamic dependence analysis in BDDT discovers additional parallelism, producing speedups of up to $3.9\times$ compared to OpenMP using barriers. BDDT outperforms OpenMP on embarassingly parallel tasks without dependencies, by using hand-added annotations to disable the dependence analysis.

2 Motivating example

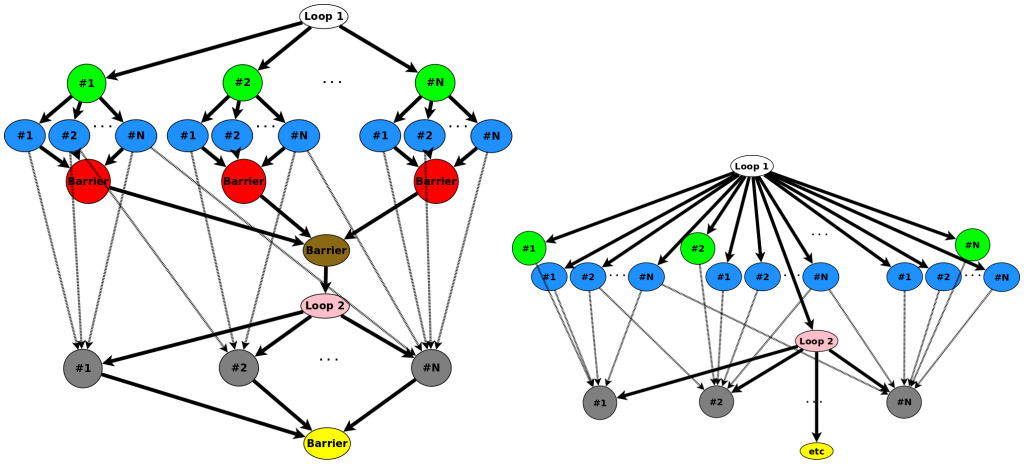
Figure 1 shows an excerpt from an implementation of two-dimensional (2-D) FFT in BDDT. We use `#pragma task` directives to specify that a function call is a BDDT task and define its memory footprint in terms of the function arguments and their sizes.

The program consists of two parallel loops: The first (lines 8–26) transposes the input data in array `A`, using tiles of `TR_BS`×`TR_BS` elements. Lines 11–14 create a parallel task to run function `trsp_blk`, which has an input dependency on its first three arguments, and an input-output dependency on its fourth argument `tile_ii`, a tile of `TR_BS`×`TR_BS` complex doubles, in an array with row size `rowsz`. Specifying tiles as strided memory accesses is a particularly powerful feature of BDDT, as it allows us to access the same data array in distinct ways without copying or transforming the layout of the data. This facilitates migrating code bases to the model as well as interfacing with legacy code. Similarly, lines 20–24 create parallel tasks to swap tile (i,j) with (j,i). The second loop (lines 29–36) performs 1-D FFT on blocks of `FFT_BS` rows.

Parallelization of the same program in OpenMP turns each loop in a parallel loop. However, OpenMP requires a barrier between the loop nests (which is implicitly executed at the end of the parallel loops) in order to enforce dependencies

```
1  void FFT_1D (long N, long N_SQRT, long FFT_BS, long TR_BS,
2            double _Complex A[N_SQRT][N_SQRT])
3  {
4    const size_t rowsz = N_SQRT * 2 * sizeof(double);
5    const size_t longsz = sizeof(long);
6
7    // Loop 1: Transpose
8    for (long I = 0; I < N_SQRT; I += TR_BS) {
9      void *tile_ii = &A[I][I];
10
11     #pragma task \
12       in(N[longsz], N_SQRT[longsz], TR_BS[longsz]) \
13       inout(tile_ii[rowsz][TR_BS|TR_BS*2*sizeof(double)])
14     trsp_blk(N, N_SQRT, TR_BS, tile_ii);
15
16     for (long J = I + TR_BS; J < N_SQRT; J += TR_BS) {
17       void *tile_ij = &A[I][J];
18       void *tile_ji = &A[J][I];
19
20       #pragma task \
21         in(N[longsz], N_SQRT[longsz], TR_BS[longsz]) \
22         inout(tile_ij[rowsz][TR_BS|TR_BS*2*sizeof(double)], \
23               tile_ji[rowsz][TR_BS|TR_BS*2*sizeof(double)])
24       trsp_swap(N, N_SQRT, TR_BS, tile1, tile2);
25     }
26   }
27
28   // Loop 2: First FFT round
29   for (long J = 0; J < N_SQRT; J += FFT_BS) {
30     tile = &A[J][0];
31
32     #pragma task \
33       in(N_SQRT[longsz], FFT_BS[longsz]) \
34       inout(tile[FFT_BS*rowsz])
35     FFT1D(N_SQRT, FFT_BS, &A[J][0]);
36   }
37
38   ...
39 }
```

Figure 1: Parallel FFT in BDDT



(a) OpenMP (b) BDDT

Figure 2: Task graphs showing run-time task execution and dependencies

between the transposition of blocks and the 1-D FFT. This barrier, however, is a significant loss of opportunity for parallelism, especially when the threads become unbalanced. Considering today's non-uniform memory architectures with dynamic power management, it is not unlikely to encounter thread imbalance, since some threads may be performing more remote memory accesses than others, some CPUs may be reaching temperature limits more often, etc. It is here that dynamic task dependence tracking brings benefits, as it allows to overlap those iterations of the second loop that are ready to execute with those iterations of the first loop that, for some reason, are taking longer to execute.

Figure 2(a) shows the task graph for the equivalent OpenMP program in Figure 1. We depict each task as an ellipse, control flow edges as solid arrows, dependencies as dotted arrows, and we abstract over the number of iterations per loop. It should be clear that the control flow edges separating the first and second loops impose much stricter dependencies than the data dependence edges between the tasks in both loops.

Figure 2(b) shows the task graph obtained with the program expressed in BDDT. The main difference here is the absence of the barrier nodes. These barrier nodes are redundant as the dependence edges enforce a "happens-before" ordering between tasks. This ordering is sufficiently restrictive to produce deterministic program execution, that computes the same result as the sequential elision. But it is also not too restrictive, exposing all existing parallelism in the program.

The SMPSs runtime [13] can also parallelize the same program using dynamic dependencies to increase parallelism. The technique employed by SMPSs, however, is very restrictive on data layout requirements. In this example, the array A must be aligned to an address 2^n, such that 2^n is larger than the size of A and the leading dimension of A must be a power of 2. While the first requirement is acceptable on processors with large virtual address space, the second argument is non-trivial. Programmers cannot generally choose the leading dimension of an array freely, as the choice of leading dimension has a large impact on intra-array conflict misses [**?**]. Indeed, there exist special compiler optimizations that automatically adjust the leading dimension of multi-dimensional arrays to reduce such conflict misses [17].

We performed a number of simple tests to understand the sensitivity of SMPSs's dependence analysis to the data layout. We found that changing the leading dimension of an array induces false dependencies among parallel tasks. For instance, using an 128×128 array with 32×32 element tiles, a change in leading dimension reduces parallelism in the first loop to 3 tasks and the critical path grows from 1 to 13 tasks. The actual DOALL parallelism is recovered only for an exact layout. In contrast, BDDT is insensitive to padding and base address translations, provided it is performed by multiples of the block size.

3 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 four access patterns: read (**`in`**), write (**`out`**), read/write (**`inout`**) and commutative tasks (**`cinout`**). The language runtime system detects dependencies between tasks by comparing the access properties of arguments of different tasks that overlap in memory. To do that, 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 (RAW) or anti- (WAW, WAR) dependencies between blocks by comparing block starting addresses and checking their access attributes.

The benefit of the block-based analysis is that it can detect dependencies between tasks that whole object analysis does not: Partially overlapping arguments are dependencies if the overlapping part is written by at least one task. Further-

more, tasks can have arguments that are non-contiguous in memory —defined as collections of contiguous memory blocks— such as a tile of a multidimensional array or a collection of objects in random memory locations, which the dependence analysis will analyze properly.

There are two potential drawbacks to block-based dependence analysis, which must be offset by the additional parallelism that it exposes. First, as the dependence analysis is performed per block, the runtime system must sometimes repeat the same action across all blocks in an argument. In contrast, whole-object dependence tracking must perform each action only once per argument. Second, false positives may occur when data structures are not properly laid out and/or when the block size is too large. BDDT overcomes both problems by its design and its programming interface. On the design side, BDDT integrates the metadata with the data payload to eliminate the overhead of metadata lookup. Also, it shares metadata between blocks in order to reduce the dependence analysis overhead. The programming interface allows the user to adjust the block granularity to be coarse enough to amortize overhead, yet fine enough to avoid false positives. In our experience, selecting an appropriate block size is quite straightforward.

Each task in the program goes through four stages: the task *issue* stage performs dependence analysis, queuing the task if any pending dependencies are unresolved; the task *scheduling* stage releases a task for execution when all its dependencies are resolved, selects a ready queue and inserts the task in that queue; the task *execution* stage executes a task; the task *release* stage resolves pending dependencies on the executed task, potentially releasing new tasks for execution. Dynamic dependence analysis causes overheads in the issue and release stages for performing dependence analysis and task wakeup, respectively. We design the data structures used in the dependence analysis specifically to minimize these overheads.

One bottleneck lies in retrieving the metadata that track data dependencies for each byte of memory accessed by a task. A general solution to this would be to maintain a hash from memory addresses to metadata [13], but this clearly causes a large overhead per access. A faster way would be to attach the metadata directly to the actual data payload [1, 18], but this makes the management of metadata visible to the programmer, and may require significant additional changes to the program source. We achieve the best of both solutions by designing 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 quite similar to dependence tracking on whole objects. There can be, however, extra overhead, as a task argument may consist of multiple blocks and dependencies must be tracked on each such block. We have thus designed a mechanism that allows multiple blocks to share the same metadata information. Then, critical dependence tracking operations operate on one metadata element instead of multiple, which greatly reduces the overhead of dependence tracking. We use this mechanism in particular to track dependencies on strided arguments—usually multidimensional array tiles: while dependencies are tracked on each block individually, the runtime system registers a single metadata element for the whole sparse region. This way, any later accesses of metadata in the runtime system do not need to scan collections of blocks in order to process a strided argument.

To detect task dependencies, we also allow multiple metadata elements to describe the same block, capturing the task order. Specifically, each written (**`out`** or **`inout`**) task argument creates a new metadata element to describe the argument blocks, and each read (**`in`**) task argument creates one or more metadata element to describe the argument blocks. Read arguments may result in more than one metadata elements, if the relevant blocks were described by more than one metadata elements (fragmentation) before the new task is created; this captures the scenario of a consumer task waiting for multiple producers.

This design allows for an efficient dependence analysis while limiting the complexity of accessing and updating the same data structures for every block. Section 4 discusses this mechanism in detail, while Section 4.2.1 contains an example demonstrating the usefulness of this mechanism.

4 Implementation

A data-flow dependency-aware runtime system dynamically constructs the task graph by deducing task dependencies from the access modes and the data blocks that are accessed by tasks. BDDT is designed so that identifying and retrieving dependent tasks introduces minimal overhead.

4.1 Internal Data Structures

BDDT consists of a custom block-based memory allocator; metadata structures for dependency analysis; and a task scheduler. BDDT metadata includes task elements that model tasks and block elements to structure the ordering and par-

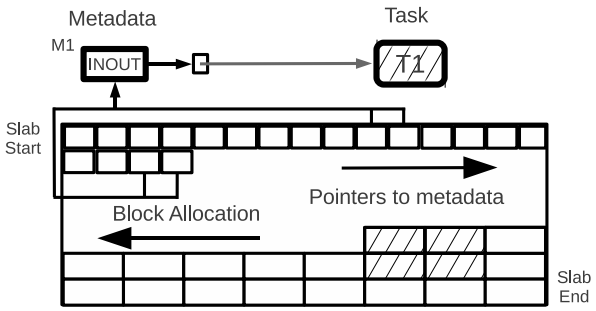


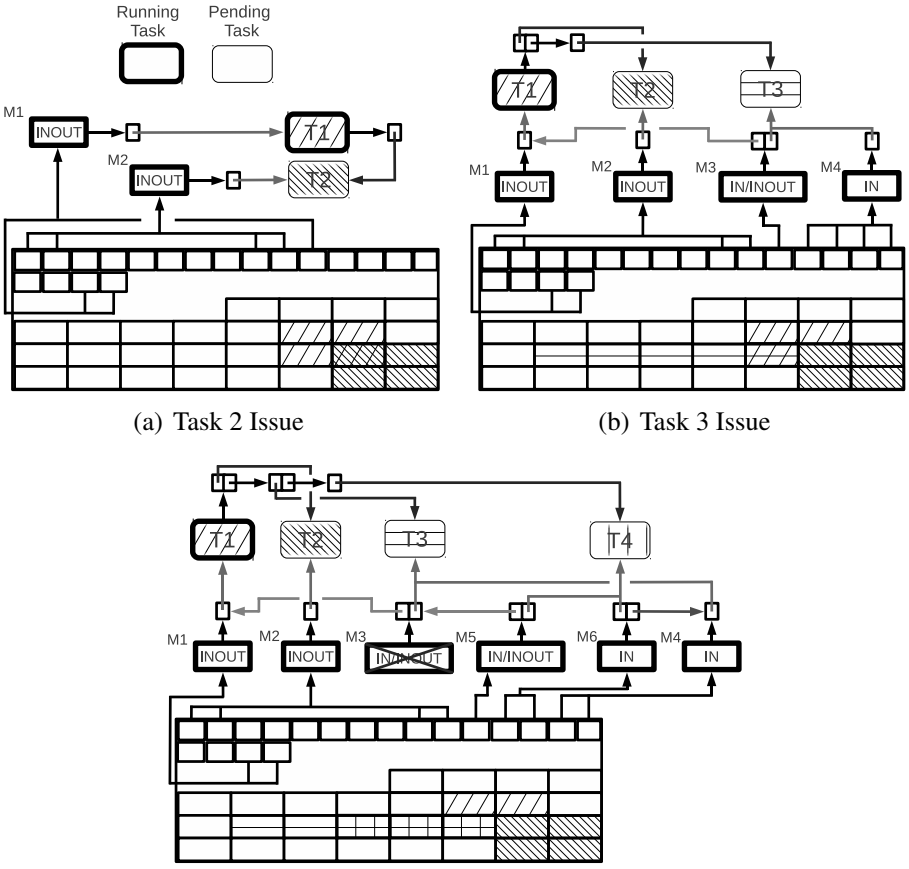
Figure 3: State of the runtime data structures when the runtime issues a task with one **`inout`** argument consisting of four blocks.

allelism of tasks (Figure 3). The memory allocator is designed to facilitate fast lookup of block elements that model the outstanding and executing tasks operating on these blocks, and to coalesce runtime system operations on blocks that are used in the same way, e.g., consecutive blocks forming a single task argument. This key optimization in the design in the runtime reduces redundancy and saves both time and space.

4.1.1 Task Elements

In the runtime system, a task element represents a dynamic instance of a task. Task elements contain all the essential metadata that is necessary to execute a task, including the closure: a function pointer, the number of arguments, and the address, size and access attributes for each argument. BDDT supports strided arguments specified as a base address, the number of elements, and the stride (in bytes) between consecutive elements. Arguments consisting of multiple contiguous blocks are also considered as strided arguments by setting the stride equal to the block size.

Task elements contain a list of dependent tasks; new task elements are appended to this list when additional dependent tasks are issued. The dependent task list is also used during task release to check whether any of the dependent tasks becomes ready to execute. We implement this check using an atomically updated join counter, which tracks the number of task arguments that are not yet ready. The dependent task list facilitates traversal of the task graph as it allows going from each task to any of its dependents.



(a) Task 2 Issue

(b) Task 3 Issue

(c) Task 4 Issue

Figure 4: Continuing Figure 3, this graph shows three additional steps for issuing four tasks with overlapping strided arguments and resolving dependencies between the tasks: (a) The runtime issues the second task and detects an overlapping region with the first task. (b) The runtime issues the third task and detects an overlapping region with the first task. (c) The runtime issues the fourth task which shares some blocks with the third task and overlaps with the first task.

BDDT locks each task element by using a specific spinlock stored inside each task element. The need of the spinlock is due to the concurrent accesses during issuing and releasing phase. There are cases where a task is released by the runtime and at the same time a spawned task detects dependencies with that task. BDDT locks the released task to prevent an unpredicted loss of dependency. The first should wait for the released task to terminate completely in order to proceed to the rest of the issuing phase. In case a task enters the release phase but an already spawned task has detected dependencies with the first task then the released task should wait until it can obtain the lock for proceeding further to the release phase.

4.1.2 Block Elements

Block metadata elements capture the running and outstanding tasks that operate on a particular collection of data blocks. They facilitate the construction of the task graph as dependencies between tasks are derived from the data blocks that they access. In BDDT, a block metadata element may represent a collection of blocks; in contrast, systems implementing object-based dependency tracking would use one metadata element per object. Any collection of blocks may correspond to multiple metadata elements describing operations on that collection. A new metadata element is allocated following each task with a write side effect (i.e., **`out`** or **`inout`**)[1]. These metadata elements are strictly ordered from the youngest (most recently issued) tasks to the oldest (least recently issued) tasks. Moreover, every block metadata element contains a list of tasks that take the corresponding collection of blocks as a task argument. This list is used at task issue to link the newly issued task on the dependent task list of older tasks that have overlapping arguments.

Block metadata elements conceptually include information about the access mode of the collection (**`in`**, **`inout`**, **`out`**). In fact, for space optimization, there can be up to two access modes per block metadata element: an **`in`** mode followed by an **`out`**, or **`inout`** mode. Task elements with **`in`** arguments in the task list of a block metadata element store an additional pointer to the first non-**`in`** task in the list. The reason for this optimization is that in typical usage a set of tasks with **`in`** mode is followed by a single task with **`out`** or **`inout`** mode. This optimization thus saves about half the space overhead of metadata elements in typical operation.

By construction, the metadata elements reveal the parallelism between tasks: tasks listed in the same block metadata element may execute in parallel, while

[1] Tasks with a **`cinout`** mode are placed in the same metadata element because they are allowed to commute.

tasks in a younger block element must wait until all tasks in an older block element have finished execution. Here, the above space optimization must be taken into account by introducing additional synchronization between the two types of tasks.

BDDT creates block metadata elements during issue process. While a single thread operates on issue phase there is no need for locking the metadata elements. During issue time the thread can disable an active metadata element and return it back to the recycle area. An active metadata element can change its state to inactive if it refers to zero number of blocks. Such a situation can occur during issue phase where a spawned task with access pattern **`inout`** accesses all the blocks refered by a metadata element and progressively are being refered by the metadata element created by the latter task. The first metadata element is active without refering to any blocks.

4.1.3 Memory Allocator

BDDT uses a custom memory allocator to embed dependence analysis metadata in the allocator's metadata structures. This key design aspect of BDDT further reduces the overhead of dynamic dependency tracking from memory footprints, in addition to metadata sharing.

BDDT requires that all data that constitutes shared state between parallel tasks is allocated through the custom memory allocator[2]The allocator partitions the virtual address space in *slabs* and services memory allocation requests from such slabs. Memory allocators typically manage multiple slabs and allocate chunks of the same size in the same slab. BDDT divides the slabs in blocks of configurable but fixed size. For every data block in a slab, there is also room provisioned to store a pointer to index the metadata elements, as discussed in Section 4.1.2.

Figure 3 shows the structure of a slab. While data blocks are allocated starting from one end of the slab, pointers to the metadata for these blocks are allocated starting from the opposite end of the slab. Thus, there may be fragmented (unusable) memory in the slab, the amount of which is bounded by the block size. All shared memory and metadata is bulk-deallocated upon completion of all tasks. As such, BDDT does not need special handling for fragmentation. Moreover, by using slabs of fixed size and alignment, we can calculate the address of a block's metadata through very efficient integer arithmetic on the block address. This also

[2]Several parallel runtime systems implement custom memory allocators for performance reasons, e.g. Cilk++ and Intel TBB. This is not a limitation of the usability of the programming model.

increases locality, as the metadata of consecutive blocks are located at neighboring addresses.

The metadata pointers stored in the slab implement collections of blocks: a collection of blocks is a group of blocks that are operated on by the same task and will be available as a task argument together. Thus, we optimize dependency tracking by mapping all blocks in a collection to the same metadata elements. BDDT implements merging of collections of blocks simply by assigning a pointer to the same metadata element to all blocks, and splits collections of blocks by assigning a new pointer to a subset of the blocks.

For example, Figure 3 shows the runtime metadata as it is constructed after spawning four tasks. Task T1 accesses 4 different blocks as an **`inout`** strided argument. When issuing T1, BDDT registers one metadata element (M1) for the four blocks and sets the slab pointers of the blocks to M1. In addition, T1's task element is inserted in M1's linked list of tasks. In the state shown, task T1 is executing or pending to execute.

4.2 Task Graph Operations

4.2.1 Task Issue

During task issue, BDDT identifies dependencies between the new task and older tasks by scanning all data blocks in the arguments of the task and analyzing the corresponding metadata elements. Note that blocks operated on in the same way are mapped to the same metadata element. As such, a task with a large memory footprint may still require only a few of the following actions. Depending on the access mode (**`in`**, **`out`**, **`inout`**), and any outstanding tasks that access the same data, BDDT either immediately schedules the task, or stores it for later scheduling. In case a data block is touched for the first time, BDDT creates an empty block metadata element for each collection of blocks with the same access mode.

Handling `in` arguments: If the most recent block metadata element contains writer tasks, then BDDT iterates through the metadata's list and registers the new task in the list of dependent tasks of all the linked task elements. It also increments the join counter by one in every task element it finds. Next, BDDT creates a new metadata element in the youngest position of the metadata element list for the current collection of blocks, and adds the new task to the new metadata element's task list. Alternatively, if the most recent metadata element contains only reader tasks, then the new task element is simply added to its task list.

Note that the operations on the metadata elements are performed only once for all blocks sharing the same metadata elements, i.e., they have equal pointers in the memory allocators slab at the start of task issue. The equality of slab pointers is maintained after task issue for all blocks accessed by the new task. If the collection contained blocks that are not accessed by the new task, then their slab pointers are not updated which effects a split of the collection.

Handling `inout` and `out` arguments: Such arguments similarly benefit from the optimization of operating on collections of blocks that have the same slab pointer. If the most recent metadata element contains writer tasks, then BDDT iterates through the metadata's task list and adds the new task to the dependent list of all the task elements. It also increments the join counter by one for every task element on the list, creates a new metadata element and inserts the new task in its task list.

If the most recent metadata element contains readers but no writers, BDDT again adds a new metadata element. This is necessary because all blocks in the collection are mapped to this new metadata element. Again, the new task is inserted in the task list in the metadata element, and in the dependent task lists of each task in the previous metadata element.

Merging collections: Collections of blocks are merged when blocks with different metadata elements are passed as part of the same **`out`** or **`inout`** argument. In this case, a new metadata element is added and the slab pointers for each block are set to point to the new metadata element. This effectuates the merge of blocks in a collection to speedup future dependency analysis.

Example: Figure 4 continues the example of Figure 3. Assume, for the sake of the example, that while T1 is running, additional tasks T2, T3 and T4 are spawned. T2 read-writes four blocks in **`inout`** mode, with one block overlapping with the footprint of T1. T2 registers one metadata element (M2). T2 iterates through the linked list of M1 to place itself in the T1's dependency list. T2 creates the first node in the linked list of M2. In addition, T2 alters the slab-pointers to the overlapped blocks along with the pointers of its other three blocks to point to M2. T3 reads five contiguous blocks in **`in`** mode. These blocks partially overlap with the memory footprint of T1. Two new metadata elements are created: M3 that models accesses to the block accessed by both T1 and T3, and M4 that models accesses to the remaining blocks. The slab pointers are updated accordingly, splitting the

collection of blocks accessed by T1 to reflect different subsequent usage. Task T3 is linked in the dependent tasks list of T1.

Finally, T4 reads 3 contiguous blocks in **`in`** mode. This argument overlaps with the T1/T3 footprint intersection (M3) and it partially overlaps with the collection of blocks that is accessed uniquely by T3 (M4). Consequently, two new metadata elements are created. M5 complements the M3 metadata element while M6 models accesses to part of M4. M4 keeps existing, modeling the blocks accessed by T3 but not by T4. T4 is inserted in the list of dependent tasks of T1 because it has a dependence with T1.

Note that metadata elements are recycled when they are no longer used: when the last slab pointer to a metadata element is removed, the metadata element is freed, as is the case here for M3. Note also that, in total, 11 blocks are accessed, but due to the coalescing of metadata elements between blocks that are accessed in the same way, only 6 metadata elements are allocated.

Bypassing the analysis: BDDT allows the user to flag task arguments as safe (free of dependencies), therefore bypassing all dependence analysis paths in the runtime.

4.2.2 Task Release and Scheduling

BDDT is based on a master-worker program model. The master is responsible for task issue and dependence analysis. The workers concurrently perform task scheduling, execution and release. The master can also operate as a worker, as discussed below. On task completion, the finished task walks through its dependence list and decrements by one the dependence counter of every dependent task. The tasks with no pending dependencies are pushed for execution. The released task is free to return to the recycle area for prospective use.

BDDT schedules a task for execution whenever all its dependencies are satisfied. Each worker thread has its own queue of ready tasks. Queues have finite length and are implemented efficiently, as concurrent arrays. The master thread has its own task queue and can operate as a worker when the queues of all workers are full.

The master issues ready tasks to worker queues in round-robin. Workers issue tasks to their own task queues to preserve memory locality. If a worker's task queue becomes full, the worker issues tasks to task queues of other workers in round-robin. In case there is no empty slot in any task queue, the task is executed

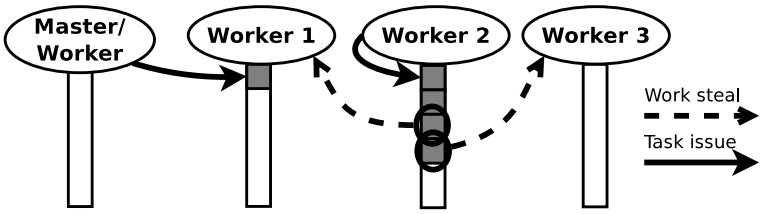


Figure 5: Ready task queues

synchronously by the issuing thread. Any thread can steal tasks from any other thread's task queue in case its own task queue is empty.

Task queues are allocated with the NUMA aware, first-touch policy. NUMA aware allocation is important to reduce remote memory accesses inside the critical path of the worker thread. Ready task queues support lock-free dequeuing with the utilization of atomic primitives. A bit vector indicates the free slots in the queue. We use the atomic "*bit scan forward*" and "*bit test and set/reset*" instructions of x86 to manipulate this vector. The queue allows any number of dequeue operations and up to one enqueue operation to occur concurrently. Enqueue operations must therefore be mutually exclusive by means of a spin-lock. Each task queue has a fixed size of 32 slots which is imposed by the atomic primitives used to implement the task queues.

Figure 5 shows a master thread with three worker threads and their task queues. The master is issuing a task to worker's 1 task queue. Workers 1 and 3 have completed all tasks in their queues and steal tasks from the task queue of worker 2. Concurrently, worker 2 issues a ready task in its own task queue. We do not need to lock the entire task queue to enqueue or dequeue tasks. Utilization of atomic primitives allows these operations to take place in parallel, thus reducing synchronization overhead.

4.2.3 Complexity analysis and discussion

BDDT introduces overheads in two parts: task issue and task release. Task release has an overhead that depends on the shape of the task graph: The runtime system receives from the scheduler the finished task and then it inspects all tasks registered in its dependent task list to locate ready tasks. We assume that the average out-degree in the task graph is d_{out}, at least in the part of the task graph that is dynamically generated. Task release then takes O(d_{out}) operations. Also note that

BDDT shares metadata elements between blocks in the same collection, so the dependent task list is scanned only once per collection. For the remaining blocks, only the slab pointers may have to be updated. Assuming an average collection size of C blocks and N blocks per task, then task release takes O($d_{out}\frac{N}{C}$) operations on average. In practice, sharing of metadata elements reduces task release overhead by more than 50% for arguments having more than 64 blocks. BDDT dependence analysis can be performed with any block size, even a block size of one byte.

Task issue has similar complexity. If prior producers of a block are still in-flight, then the runtime system locates the metadata of a block with a single operation on the bits of the block address in O(1) time and a new metadata element is created. The issued task is linked to all tasks in the last-issued task list of the prior metadata element, taking O(d_{in}) operations assuming an average in-degree d_{in} in the task graph. Furthermore, the slab pointers of all blocks in a collection are updated. In total, task issue takes O($d_{in}\frac{N}{C}$) operations on average. Note that the overhead of merging and splitting collections is included in the presented formulas as they are realized by setting the slab pointers.

To put the overheads in perspective, we compare against SMPSs with region support [13]. The comparison is not entirely fair because this version of SMPSs has less functionality: it handles only multi-dimensional array regions (tiles), encoded with a binary representation, thus disallowing arbitrary pointer arithmetic. The representation is approximate and subject to aliasing and alignment constraints, which restricts the acceptable tile and array sizes to powers of two and is prone to false positives.

Dependence detection in SMPSs requires encoding of regions in their binary representation, taking a number of operations proportional to the number of bits in an address. It also requires walking a region tree data structure to detect overlap with other regions, and update the tree by adding the region or updating the metadata of an already existing region. These operations take O($d_{in}\frac{N}{R}\log T$), where R is the average region size expressed in blocks and T is the number of regions in the region tree (we are assuming here that the metadata is structured in a comparable way to BDDT to facilitate the comparison). The region size may be less than the argument size N because regions must be split to eliminate false positives, with $R = O(N)$ in the worst case.

Although a comparison between block size and array length, and between average collection size C and average region size R are not trivial, we conclude that BDDT has the advantage that the appropriate metadata elements are identified in O(1), while SMPSs requires O($\log T$) time to locate metadata elements of

overlapping task arguments.

4.3 Multi-issuing Functionality

BDDT can perform concurrently dependence analysis for multiple tasks. The runtime separates tasks in two different categories: 1) leaf tasks and 2) parent tasks. The leaf tasks are the computational tasks where the application's computational part takes place. The parent tasks are intermediate tasks which have not any computational part and their main functionality is to group leaf tasks having some common features, like accessing the same memory footprint. The programmer should use the parent tasks in order to activate the multi-issuing mechanism, but should be used wisely because of runtime overhead reasons that we are describing later. By having more than one parent tasks, BDDT can issue concurrently tasks located under different parent tasks.

The runtime performs the same dependence analysis mechanism for both leaf tasks and parent tasks. The main difference is based on the scheduling part. In case we have dependent parent tasks BDDT behaves as these tasks are not dependent on each other and forwards all the parent tasks into the scheduler. As a result, the scheduler will try to execute concurrently all the parent tasks. The leaf tasks located under different parent tasks will be issued in parallel but will not be sent to the scheduler until their parent task's dependencies are zero. The task graph created in the multi-issuing version of BDDT keeps an extra information. The dependent parent tasks are linked to each other and every parent task keeps an additional piece of information for every of its ready leaf tasks. These auxiliary links helps the runtime to locate the dependent parent tasks and its leaf tasks that are ready for execution.

As we have already mentioned, the new programming model is based on a task-tree model. Each node of the tree is a task. The children tasks of a parent task can access a portion or whole of its memory footprint. The programmer must use carefully the multi-issuing mechanism and not create task-trees that have a lot of intermediate tasks. By creating large trees the runtime produces more overhead as the application spawns even more tasks. This runtime system can guarantee progress by using this nesting dependencies programming model in cases where the single master thread spends more time to issue tasks rather than execute them. The saturation of the master thread leads to enable more tasks to aid on issuing process.

4.3.1 Task Elements Extensions

In the multi-issuing version of the runtime we keep some additional information for the parent tasks. The parent tasks have their own block-element array indexing which is an array of pointers point to block elements. This array has been created in order to distinguish the block elements created by different parent tasks which access the same memory footprint. The amount of pointers that every array can contain is tuned to the number of blocks that every parent task is accessing. In addition, BDDT keeps information for the ready leaf tasks of each parent task. We keep a linked list for all the ready leaf tasks and in case a parent task has zero dependencies then we iterate through its ready leaf tasks and BDDT sends them to the scheduler for execution.

4.3.2 Task Issue

During task issue the runtime checks for the category of the corresponding task. In case the task is a parent task then BDDT creates the block-element array indexing by allocating space defined by the number of blocks indicated in the task's memory footprint. The task follows the dependence analysis mechanism as a regular task would perform in the non-extended version. Each parent task is a ready task by default. Thus, even a dependent parent task can be executed. The first issued leaf task of a parent task which has zero dependencies is a ready task. The first issued leaf task of a dependent parent task has dependencies only with the task that its parent is dependent on. As a result, during spawning leaf tasks BDDT checks the number of dependencies of their parent task. In case the parent task is dependent, then the runtime creates a linked list. Each node in the list points to the leaf tasks that have zero dependencies. We encounter dependencies on tasks that belong to the same tree level. In case a parent task is dependent, its leaf tasks do not have knowledge about that information.

4.3.3 Task Release and Scheduling

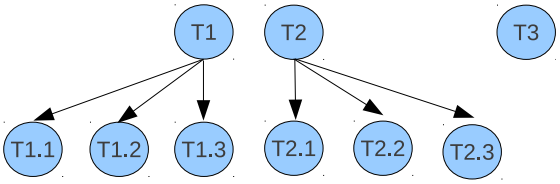
During parent task release the runtime does not remove the task from the task graph to the recycle area. The parent task holds its position until all of its leaf tasks are released. The parent task keeps information about the number of its released leaf tasks. When the counter becomes zero and all the leaf tasks are executed then the parent task can be completely released. At that point, the parent task iterates through its dependency list to free all the tasks that are ready for execution. In case BDDT encounters dependent tasks that are parent tasks and ready for execution

then it iterates through their ready-leaf-tasks list. All the leaf tasks located in that list are pushed to the scheduler. On the other hand, the procedure that addresses the release of leaf tasks is the same as pre-mentioned in the non-extended version of the multi-issuing functionality.

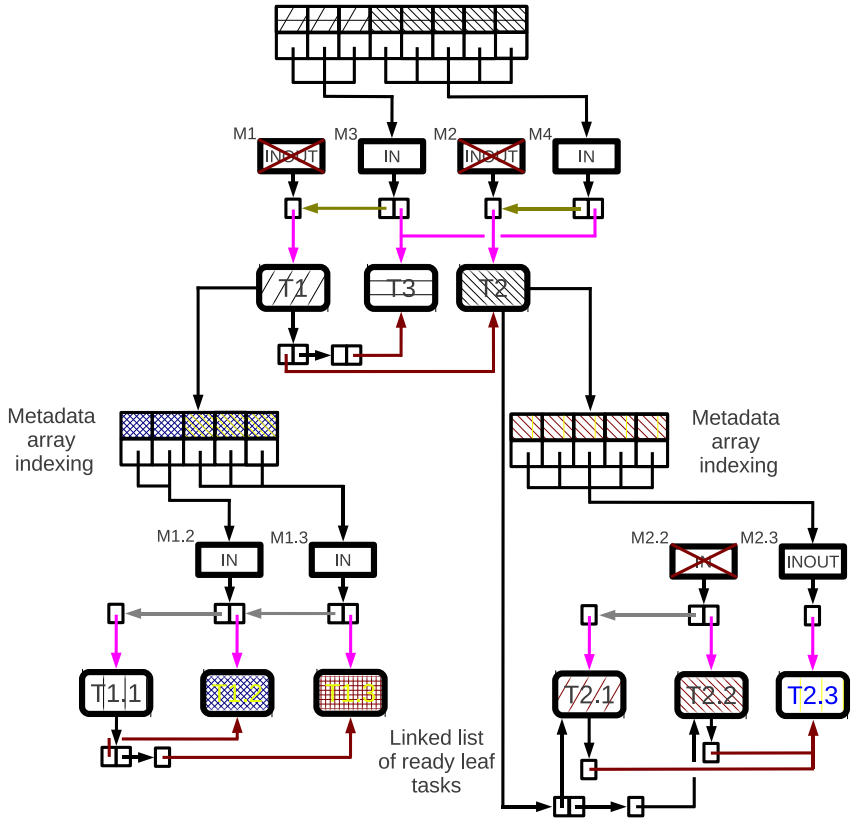
Example: Figure 6(a) shows an example where the runtime detects two parent tasks and a leaf task spawned and belonging at the same task-tree level. Furthermore, each parent tasks spawns three leaf tasks. In addition, Figure 6(b) shows the final state of the task graph. We state all the steps the runtime executes for each parent task in Figure 7. Some of the steps happen to occur concurrently, resulting the parent task T2 and the leaf tasks of parent task T1 to be issued in parallel.

Figure 7 shows the steps followed by BDDT to issue the leaf tasks of T1, the leaf task of T2 and the parent tasks. The example starts from state (a) where parent task T1 accesses five different blocks as an **`inout`** argument. While issuing T1, BDDT registers one metadata element (M1) for the five blocks and sets the slab pointers of the blocks to M1. In addition, T1's task element is inserted in M1's linked list of tasks. In the state shown, task T1 is executing. While another thread assigned to execute the parent task T1, the first thread that issued T1 is issuing parent task T2. Considering the fact that the subsequent T2 is accessing some of the blocks accessed concurrently by the T1's leaf tasks, BDDT uses for its parent tasks a unique block-element array indexing. The size of the array is determined by the number of blocks accessed by the parent task.

We proceed to explain (b) the issuing procedure of T1's leaf tasks. Leaf task T1.1 accesses five contiguous blocks as an **`inout`** argument. While issuing T1.1, BDDT registers one metadata element (M1.1) for the five blocks and sets the slab pointers of the blocks to M1.1. In addition, T1.1's task element is inserted in M1.1's linked list of tasks. Assume that T1.1 is ready for execution. While T1.1 is running additional leaf tasks T1.2 and T1.3 are spawned. T1.2 reads five contiguous blocks in **`in`** mode where all the of the five blocks are overlapping with the footpring of T1.1. T1.2 registers one metadata element (M1.2). T1.2 iterates through the linked list of M1.1 to place itself in the T1.1s dependency list. T1.2 creates the first node in the linked list of M1.2 and connects the node to the linked list of M1.1. In addition, T1.2 alters the slab-pointers to the overlapped blocks to point to M1.2. Task T1.3 reads also three contiguous blocks in **`in`** mode which are overlapping with T1.1's footprint. T1.3 registers one metadata element (M1.3). T1.3 iterates through the linked list of M1.1 to place itself in the T1.1's dependency list. T1.3 creates the first node in the linked list of M1.3 and connects

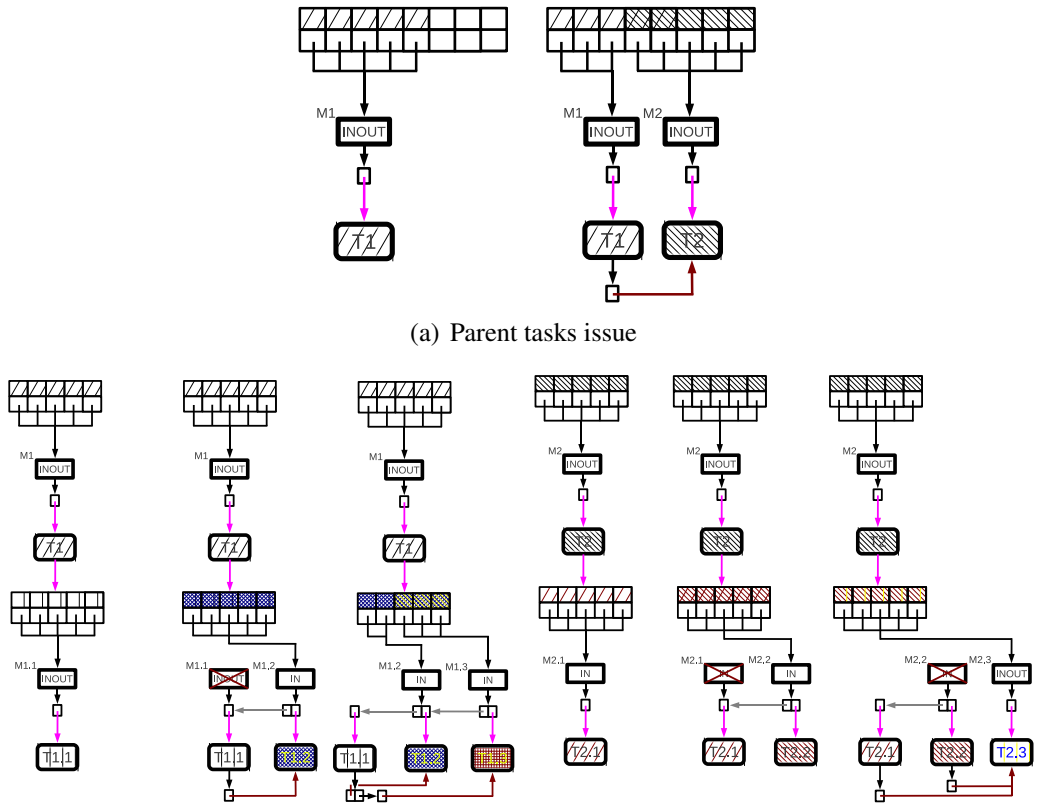


(a) Two parent tasks with their leaf tasks



(b) State of the task graph when issuing all tasks

Figure 6: State of the runtime data structures when the runtime issues parent tasks and leaf tasks.



(a) Parent tasks issue

(b) Leaf tasks issue under parent task T1

(c) Leaf tasks issue under parent task T2

Figure 7: Continuing Figure 6, we have three different states of the runtime system that happens to occur concurrently: (a) The runtime issues two parent tasks and a leaf task, (b) The runtime issues three leaf tasks under the parent task T1, (c) The runtime issues three leaf tasks under the parent task T2

it to the linked list of M1.2. T1.3 and T1.2 share partially three contiguous blocks. For that reason, BDDT has created metadata element M1.3 which corresponds to both T1.3 and T1.2 by splitting the collection of the five contiguous blocks in two metadata elements. The runtime alters also the shared slab-pointers to point to M1.3.

We are changing now to the previous thread (a) which was occupied with issuing the second parent task. T2 read-writes five contiguous blocks in **`inout`** mode where two of them are overlapping with T1's footprint. Parent task T2 registers a new metadata element (M2). In addition, T2 iterates through the linked list of M1 to place itself in the T1's dependency list. T2 creates the first node in the linked list of M2. T2 alters the slab-pointers to the overlapped blocks to point to M2.

Finally BDDT in state (a) has to issue concurrently a leaf task which is located at the same level along with the parent tasks and to dispatch another thread for issuing the leaf tasks of the second parent task. T3 is spawned which tries to read ten contiguous blocks in **`in`** mode. T3 registers two metadata elements (M3) and (M4). M3 models accesses to the block accessed by both T1 and T3 and M4 models accesses to the blocks accessed by both T2 and T3. T3 iterates through both the linked lists of M1 and M2 to place itself in the dependency lists of T1's and T2's respectively. T3 alters the slab-pointers to the overlapped blocks to point to M3 and M4 accordingly.

Another thread, as depicted in (c), is dispatched and tries to issue all the leaf tasks located under the parent task T2. T2.1 is spawned first and reads five contiguous blocks in **`in`** mode. T2.1 registers a new metadata element (M2.1). In addition, T2.1's task element is inserted in M2.1's linked list of tasks and set all the slab-pointers to point to M2.1. We assume that T2.1 cannot be executed because its parent task is dependent on T1 parent task and T1's leaf tasks have not finished yet. When T2 is released then the first ready leaf task will be T2.1. The next spawned task is T2.2. T2.2 reads five contiguous blocks in **`in`** mode. In addition, T2.2 registers a metadata element (M2.2). T2.2 alters the slap-pointers to point to M2.2. T2.2 cannot be executed for the same reason the T2.1 cannot. The third and final leaf task is spawned called T2.3. T2.3 read-writes five contiguous metadata elements in **`inout`** mode. In addition, T2.3 registers a metadata element (M2.3). T2.3 iterates through the linked list of M2.2 to place itself in the dependency list of T2.1 and T2.2. T2.3 alters the slab-pointers to the overlapped blocks to point to M2.3.

5 Experimental Analysis

5.1 Hardware Platform

We ran the experiments on a Cray XE6 compute node with 32GB memory and two AMD Interlagos 16-core 2.3GHz dual-processors, a total of 32 cores at 8 cores per processor. Every pair of cores shares one FPU, possibly reducing floating point arithmetic performance. Each 8-core processor has its own NUMA partition, yielding a total of 4 NUMA partitions with 8 GB of DRAM per partition. To uniformly distribute application data on all NUMA nodes, we initialize input data in parallel. Each core allocates and touches a part of the input array(s) used in each benchmark, so that all NUMA partitions perform approximately the same number of off-chip memory accesses during execution. We compile the benchmarks with GNU GCC 4.4.5 using the -O3 optimization flag.

5.2 Benchmarks

We use a set of task-based benchmarks that use array tiles from the SMPSs distribution [13] to evaluate BDDT and compare against SMPSs. All benchmarks use row-major (C-language) array layout. We use microbenchmarks to directly compare the overhead of the dependence analyses between BDDT and SMPSs.

We also compare the performance of the task-based benchmarks with equivalent OpenMP implementations, so that both use the same parallelization strategy and parameters, modulo the removal of barriers in the task-based version. We compare against OpenMP in two contexts: First, we measure the performance that dataflow execution gains from dynamic dependence analysis in applications where OpenMP requires barriers to enforce dependencies. Second, we measure the overhead cost of the dynamic dependence analysis using applications with ample task parallelism and few or no dependencies.

The block size used in BDDT to partition task arguments affects the overhead and accuracy of the dynamic dependence analysis. As all benchmarks except Multisort use block linear algebra algorithms that work on two-dimensional tiles of the input array, we set the block size to the row size of a tile. Multisort recursively splits an array until a certain threshold, which we set as the BDDT block size.

Cholesky: The Cholesky factorization kernel is used to solve normal equations in linear least squares problems; it calculates a triangular array from a symmetric positive definite array. The kernel can be decomposed into four tile operations,

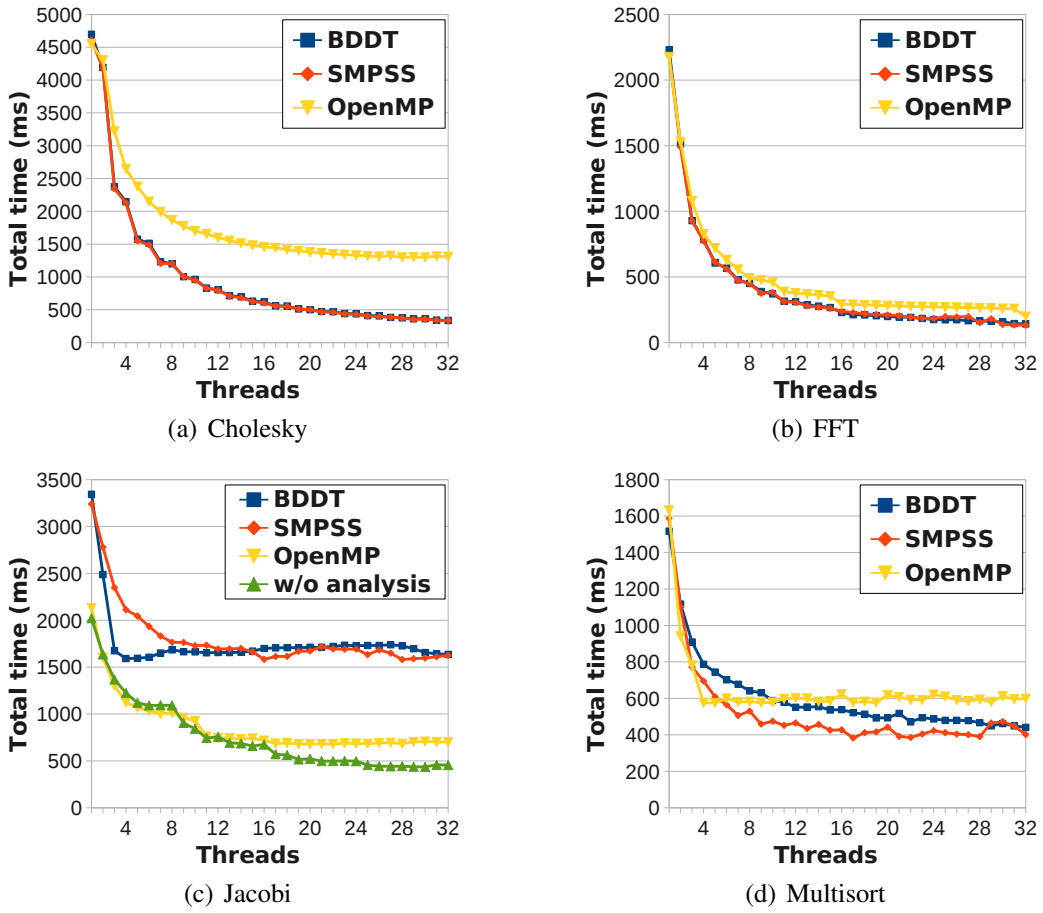


Figure 8: BDDT, SMPSs and OpenMP on Interlagos

each of which corresponds to a task in the benchmark: `POTRF`, which computes the Cholesky factorization of the diagonal block; `TRSM`, which computes the column panel; `SYRK`, which computes the row panel; and `GEMM`, which updates the rest of the array. We used the AMD-optimized ACML library for the computation kernels. Dependencies among tasks create an irregular task graph, requiring the OpenMP implementation to use barriers between the update of diagonal blocks and the panel updates. This limits parallelism across outermost iterations of the code. Both SMPSs and BDDT overcome this limitation using dynamic dependence analysis.

Figure 8(a) shows the performance of Cholesky for a 4096×4096 double pre-

cision matrix and 128×128 tiles. BDDT performs 3.9× better than OpenMP with 32 cores, due to the extraction of additional parallelism. Moreover, BDDT matches the SMPSs performance with less than 2% deviation. Both BDDT and SMPSs versions of the benchmark achieve a top speedup of 14 on 32 cores, whereas the top speedup of the OpenMP version is 3.5.

FFT: The FFT kernel involves alternating phases of transposing a two-dimensional array and computing one-dimensional FFT. We use the FFTW library for the 1-D FFT computations; FFTW requires a row-wise layout in memory for the input array, which forces each FFT calculation task to operate on an entire row of the array. In contrast, transposition phases can break the array into tiles, so the transpose tasks' arguments are non-contiguous array tiles. Because of this difference in the memory layout of task arguments, the OpenMP version must use barriers between phases to ensure correctness. Dynamic dependence analysis in BDDT overcomes this limitation and exploits parallelism across phases, thus permitting transpose and FFT tasks to overlap.

Figure 8(b) shows the performance of a 2-D FFT on 16M complex double-precision elements with BDDT and OpenMP. The input array is 4096×4096 elements and the transpose tile size is 128×128. OpenMP outperforms BDDT by up to 3% when the code runs with up to 4 threads, due to the cost of dynamic dependence analysis. Using more than 4 threads, BDDT extracts more parallelism than OpenMP and performs up to 50% better at 32 threads. BDDT still manages an overall speedup of 16× using 32 threads, whereas OpenMP achieves a maximum speedup of 11×. Furthermore, SMPSs has a performance advantage over BDDT by 3% on 32 threads.

Jacobi: Jacobi is a common method for solving linear equations. We use the Jacobi kernel implementation from the SMPSs distribution which uses row-major array layout. Each task in Jacobi works on a tile of the array. The kernel implements a 5-point stencil, which requires passing the boundaries of the tile as input to each task. Moreover, the Jacobi kernel is an iterative method, so we keep an input and an output array from one iteration to the next. In the end of each iteration we swap the references of the array and begin the next iteration. Dynamic dependence analysis allows tasks from consecutive iterations to execute in parallel by keeping two arrays as input and output and swapping them from one iteration to the next. In contrast, the OpenMP implementation must issue a barrier between outermost iterations of the kernel.

We tested Jacobi using a 4096×4096 array and 128×128 tile size. The Jacobi kernel is communication bound and memory intensive. The kernel performs 5 operations per 6 double precision floating point elements fetched from memory, giving a 0.1 ops/byte ratio. Furthermore, all tasks in a single iteration of the kernel can run in parallel. Therefore, the overhead of dynamic dependence analysis is noticeable. In BDDT and SMPSs, overheads dominate execution time, yielding a 2.3× slowdown compared to OpenMP with 32 threads. The scalability of the BDDT version of the code is also inferior to that of OpenMP: maximum speedup with BDDT reaches 2.1 vs. 3.1 with OpenMP.

BDDT allows the programmer to disable dependence analysis on selected task arguments and replace it with traditional synchronization methods. This feature proves useful in Jacobi, where the analysis overhead shadows any gain from overlapped execution of tasks across multiple outer iterations. The "w/o analysis" line in Figure 8(c) shows the performance of BDDT with dependence analysis disabled for all arguments, via data annotations. BDDT performs identical to OpenMP for up to 16 threads. For 16 threads or more, BDDT outperforms OpenMP by 15% to 45%, increasing with the thread count. The result indicates that BDDT's implementation of the runtime system is efficient, scalable, and can be used by both conventional task-based models and advanced models with out-of-order task execution capabilities.

Multisort: Multisort is a parallel sorting algorithm originating from the Cilk distribution. The algorithm is a parallel extension of ordinary Mergesort. Multisort recursively divides an array in halves and sorts each half using a divide-and-conquer approach. Recursion stops when the size of the sub-array to be sorted falls below a predefined threshold, which we also use as the block size in BDDT, at which point it uses sequential quicksort to sort the sub-array. Multisort then merges the sorted halves, with each merge task working on overlapping parts of the array. Multisort uses a 1-D array and task arguments that are contiguous in memory, however the arguments of different tasks may overlap. The OpenMP version requires barriers between phases of the algorithm. BDDT and SMPSs remove these barriers and extract more parallelism with dynamic dependence analysis.

Figure 8(d) shows the performance of Multisort on an array of 32M integers, with a threshold of 128K elements for stopping recursive subdivision. BDDT extracts more parallelism than OpenMP and achieves up to 35% better performance at 32 threads. Specifically, BDDT is 3.5× faster at 32 threads, while the top speedup of the OpenMP version is 2.7. SMPSs presents a performance advantage

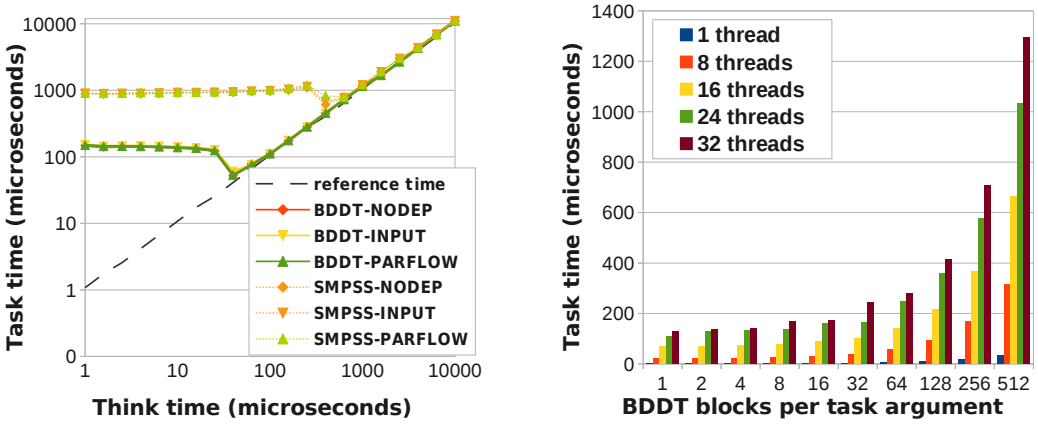
of 20% on average for small number of threads but it deteriorates for higher thread counts, falling to 5% on 32 threads.

5.3 Microbenchmarks

We use a set of carefully crafted microbenchmarks to compare the overhead and efficiency of dynamic analyses between BDDT and SMPSs. Our microbenchmarks spawn a number of tasks and wait for them to complete at a sync operation. We measure the time elapsed from spawning the first task to completing the sync operation, and compare this time against the running time of a reference, sequential version, where tasks are converted to function calls. For parallel runs, we normalize the reference time to the number of threads. Furthermore, we use tasks with different data access properties to evaluate the impact of dependence analysis on running time, and introduce "think time" (artificial delay) in tasks to evaluate the impact of task granularity on runtime overhead. Finally, note that microbenchmark results give an optimistic estimation of runtime overhead, as the cache hierarchy and memory subsystem are dedicated to internal data structures of the runtime system.

To isolate synchronization overhead, we use a "NODEP" microbenchmark, where tasks have no arguments. To evaluate the overhead of the dependence analysis, we use two microbenchmarks: an "INPUT" microbenchmark, where tasks have a single **`in`** argument, and a "PARFLOW" microbenchmark, where tasks have a single **`inout`** argument. In both cases, the microbenchmark passes the same parameter to all tasks. This will cause true dependencies among all tasks in "PARFLOW", forcing them to execute in sequential order. Note that all tasks spawned in the "NODEP" and "INPUT" microbenchmarks are able to execute in parallel as they are independent. Moreover, adding more threads would have no effect on the "PARFLOW" microbenchmark; to evaluate a more representative scenario, we create as many sequential chains of tasks as the number of threads available.

Figure 9(a) shows the impact of task granularity on the overhead of the runtime systems. The X-axis is the artificial "think time" per task, and the Y-axis is the average running time per task, averaged over 8000 tasks. The microbenchmarks shown use 32 threads. Ideally, microbenchmark time should track the reference time, implying that the runtime system incurs no overhead. In practice, microbenchmark time is higher when tasks are fine-grain. BDDT tasks are able to match the reference time with a granularity of 40 μs for the "NODEP" benchmark. For the "INPUT" and "PARFLOW" benchmarks BDDT tasks match the reference



(a) Delay

(b) Blocks

Figure 9: Microbenchmarks

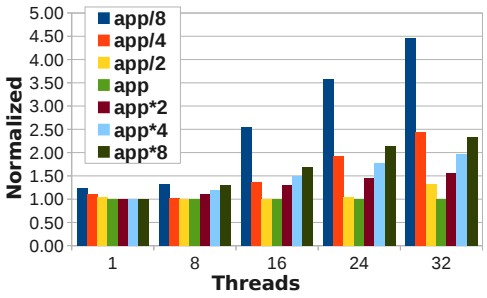
time at 50 μs and 60 μs respectively. The SMPSs runtime system requires a task granularity of at least 600 μs to match the reference time.

Figure 9(b) shows how the number of BDDT blocks per task argument affects the average running time of null-tasks for various numbers of threads. The Y-axis shows task running time for the "INPUT" microbenchmark, averaged over 1000 tasks. Note that task completion time increases with the number of blocks per task argument; it is more than double compared to 1 block for 64 blocks per argument, for all numbers of threads; it reaches around 10$\times$ for 512 BDDT blocks. We believe that task running time increases with core counts as synchronization becomes more expensive accross NUMA partitions.

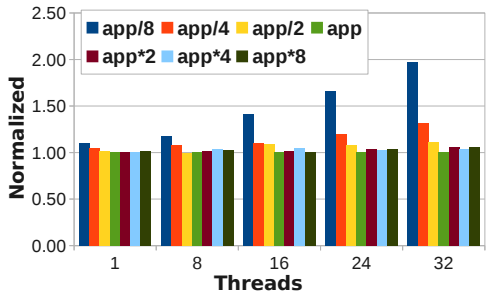
5.4 Block layouts

In the above experiments we have restricted array and block sizes to be powers of two because it is required by SMPSs, which behaves erratically otherwise; BDDT is able to handle arbitrary BDDT block sizes for dependency tracking granularity. Here we explore the hypothesis that many applications may need more flexibility in array layouts and tile dimensions.

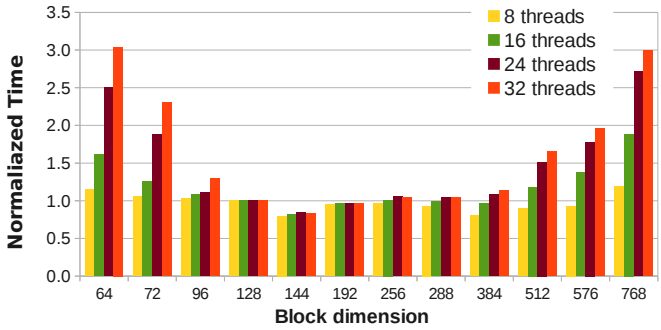
Figure 10(c) shows how the Cholesky kernel performance varies for various tile dimensions with BDDT. We normalize BDDT's total execution time to the total time of 128×128 elements tile which is the best case for SMPSs. Note that



(a) Cholesky, BDDT block sizes



(b) FFT, BDDT block sizes



(c) Cholesky for various tile dimensions

Figure 10: Impact of BDDT block size.

the 144×144 tile gives on average 18% better performance.

We expect the programmer (or a compiler analysis) to define an appropriate BDDT block size. A smaller than optimal BDDT block size introduces unnecessary overhead for dependence tracking. Conversely, larger block size than optimal may introduce false dependencies and reduce parallelism. Figures 10(a) and 10(b) show the performance of the Cholesky and FFT benchmarks for block sizes varying from $8\times$ smaller to $8\times$ greater than optimal. At worst, they perform $4.5\times$ and $2.0\times$ worse than optimal for $8\times$ smaller block size, respectively. The false dependencies introduced by large block sizes do not seem to affect FFT, as they only occur among tiles of the same row, still allowing parallel processing among different tile rows.

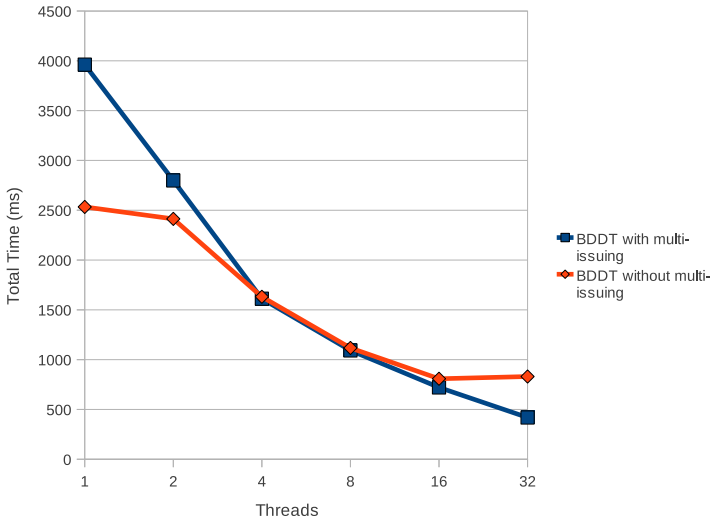


Figure 11: A comparison between Jacobi parallelized with BDDT multi-issuing and Jacobi parallelized with BDDT without multi-issuing.

5.5 Multi-issuing

Jacobi: We tested Jacobi in the extended version of BDDT by using a 4096×4096 array and 256×256 tile size. The Jacobi kernel is communication bound and memory intensive. Jacobi is constituted of three nested for loops. We have task-ified the first nested for loop level which iterates through the x-axis. Each task produces leaf tasks for the y-axis. By following that parallelization technique, the master is not saturated and the other threads can issue tasks in parallel. We have seen in Figure 11 that Jacobi, parallelized with BDDT multi-issuing, scales up to 32 threads. The other threads do not have to wait for the master thread to dispatch them tasks and BDDT can overlap issuing process with execution process. The results indicate that the extended version of BDDT is efficient, scalable and can be used at least to applications that are commucation bound and stretch the runtime for the most of the execution time. The runtime system resolves the overhead produces by the saturation of the master thread. In addition, the Figure 11 shows the difference in performance for both the Jacobi parallelized with BDDT multi-issuing and the Jacobi parallelized with original BDDT. Jacobi in BDDT multi-issuing scales better for more than 16 threads. For 1 to 4 threads we observe that Jacobi in BDDT without multi-issuing performs better. This behaviour

is due to a more complicated dependence analysis which adds more overhead in the runtime system. Moreover, the Jacobi in BDDT multi-issuing spawns more tasks than the Jacobi in the original BDDT. The increment of tasks is based on the fact that we created groups of tasks under parent tasks in order to enable more issuers and to produce more parallelism. Thus, more tasks yields extra work for the runtime.

6 Determinism proof

We have formalized a simple version of the BDDT programming model and proved that it produces deterministic parallel executions, equivalent to the sequential execution of the program. We have omitted support for arrays and arbitrary pointer arithmetic in our formal model, because it greatly increases the complexity and size of the system. Moreover, the model does not differentiate between input and output task arguments. We present a summary of the simplified formalism and a proof strategy.

We define λ^{TASK}, an extension of the simply-typed lambda calculus with dynamic memory allocation and updatable references, task creation and synchronization. Values include integer constants n, the unit value `()`, functions $\lambda\, x\,.\, e$ and pointers ℓ. Program expressions include variables x, function application $e_1\ e_2$, memory operations and task operations. Specifically, expression `ref` e allocates some memory, initializes it with the result of evaluating e, and returns a pointer ℓ to that memory; expression $e_1 := e_2$ evaluates e_1 to a pointer and updates the pointed memory using the value of e_2; and expression $!\, e$ evaluates e to a pointer and returns the value in that memory location. Expression $\texttt{task}(e_1, \ldots, e_n)\,\{e\}$ evaluates each e_i to a pointer and then evaluates the task body e, possibly in parallel. The task body e must always return `()` and can only access (via dereference or assignment) the given pointers; if e is evaluated in a parallel task, the expression immediately returns `()`. Finally, expression `waiton` e evaluates e to a pointer and blocks the execution until no child task has that memory.

To prove that this language produces deterministic parallel executions, we define both its parallel and sequential semantics. The parallel small-step semantics have the form $\langle T, D, S, R\rangle \rightarrow_p \langle T', D', S', R'\rangle$ where T is a map from task identifiers t to the task definitions $\texttt{task}(\vec{\ell})\,\{e\}$, D is a map from every memory location ℓ to a queue q of task identifiers, S is a map of memory locations ℓ to the values v they contain, and R is a set of the identifiers of all the tasks currently running in

parallel. We omit the definition of all semantic rules here for brevity. Sequential small-step semantics are standard, and have the form $\langle S, e\rangle \rightarrow_s \langle S', e'\rangle$.

Using the definition of the parallel and sequential semantics, we can then state sequential equivalence:

Theorem 6.1 *Sequential equivalence*
If $\langle \{t_0, \texttt{task}(\ell)\,\{e\}\}, \emptyset, \emptyset, t_0\rangle \rightarrow_p^* \langle \{t_0, \texttt{task}(\ell)\,\{v\}\}, D, S, t_0\rangle$ *then* $\langle \emptyset, e\rangle \rightarrow_s^* \langle S, v\rangle$

The proof is similar to a confluence proof. In short, we show that given a parallel execution trace, we can construct a sequential execution trace by reordering transitions, so that the initial and final state are the same. The proof is by induction, and reduces any parallel execution trace to a parallel execution trace whose first step satisfies the sequential program order.

7 Related Work

Task-parallel programming models Task parallel programming models offer a more structured alternative to parallel threads, allowing the programmer to easily specify scoped regions of code to be executed in parallel. OpenMP [2] is an API for parallelization of sequential code, where the programmer introduces a set of directives in an otherwise sequential program, to express shared memory parallelism for loops and tasks. OpenMP implements these directives in a runtime system that hides the thread management required, although the programmer is still responsible to avoid races and insert all necessary synchronization.

Cilk [11] is a parallel programming language that extends C++ with recursive parallel tasks. Cilk tasks can be fine-grained with little overhead, as Cilk creates parallel tasks only when necessary, using a work-stealing scheduler; and "inlines" all other tasks at no extra cost. The programmer must use *sync* statements to avoid data races and enforce specific task orderings.

Sequoia [8, 3] is a parallel programming language similar to C++, which targets both shared memory and distributed systems. In Sequoia, the programmer describes (i) a hierarchy of nested parallel tasks by defining atomic *Leaf* tasks that perform simple computations, and *inner* tasks that break down the computation into smaller sub-tasks; (ii) a machine description of the various levels in the memory hierarchy and any implicit (coherency) or explicit communication (data transfer) among memories; and (iii) a mapping file that describes how data should be distributed among task hierarchies, which tasks should run at each level

in the memory hierarchy, and when computation workload should be broken into smaller tasks. Sequoia inserts implicit barriers following the completion of each group of parallel tasks at a given level of the memory hierarchy.

Synchronization, dependencies and determinism Several programming models and languages aim to automatically infer synchronization between parallel computations. Transactional Memory [9] preserves the atomicity of parallel tasks, or transactions, by detecting conflicting memory accesses and retrying the related transactions. Jade [16] is a parallel language that extends C with parallel coarse-grain tasks. In Jade, the programmer must declare and manage local- and shared-memory objects and define task memory footprints in terms of objects. The runtime system then detects dependencies on objects and enforces the sequential program order on conflicting tasks.

SvS [5] is a task-based programming model that uses static analysis to determine possible argument dependencies among tasks and drive a runtime-analysis that computes reachable objects for every task, using an efficient approximate representation of the reachable object sets, resembling Bloom filters. It then detects possible conflicts and enforces mutual exclusion between tasks. SvS assumes all tasks to be commutative and does not preserve the original program order as BDDT. Moreover, it tracks task dependencies at the object level, restricting SvS on type-safe languages. Finally, SvS object reachability sets are approximate, and may include many reachable objects in the program, regardless of whether they are accessed by a task or not. This may hinder the available parallelism, and fails to take advantage of programmer knowledge about the memory footprint of each task.

StarSs [15] is a task-based programming model for scientific computations that uses annotations on task arguments to dynamically detect argument dependencies between tasks. SMPSs [13] is a runtime system that implements a subset of StarSs for multicore processors with coherent shared memory. Similarly to BDDT, in SMPSs each task invocation includes the task memory footprint, used to detect dependencies among tasks and order their execution according to program order. SMPSs describes array-tile arguments using a three-value-bit vector representation to encode memory address ranges. This representation, however, causes aliasing and over-approximation of memory ranges when the array base address, row-size and stride are not powers of 2. Aliasing in turn creates false dependencies which reduce parallelism, and also a high overhead for maintaining and querying a global trie-structure that detects overlapping memory ranges.

In comparison, BDDT uses a transparent block-level dependence analysis with constant-time overhead per block that, with proper choice of block size, eliminates aliasing and false dependencies.

Recent research has developed methods for the deterministic execution of parallel programs. Kendo [12] enforces a deterministic execution for race-free programs by fixing the lock-acquisition order, using performance counters. Grace [4] produces deterministic executions of multithreaded programs by using process memory isolation and a strict sequential-order commit protocol to control thread interactions through shared memory. DMP [7] uses a combination of hardware ownership tracking and transactional memory to detect thread interactions through memory. Both systems produce deterministic executions, even though they may not be equivalent to the sequential program. Instead, they enforce the appearance of the same arbitrary interleaving across all executions.

Out-of-Order Java [10] and Deterministic Parallel Java [6] are task-parallel extensions of Java. They use a combination of data-flow, type-based, region and effect analyses to statically detect or check the task footprints and dependencies in Java programs. OoOJava then enforces mutual exclusion of tasks that may conflict at run time; DPJ restricts execution to the deterministic sequential program order using transactional memory to roll back tasks in case of conflict. As task footprints are inferred (OoOJava) or checked (DPJ) statically in terms of objects or regions, these techniques require a type-safe language and cannot be directly applied on C programs with pointer arithmetic and tiled array accesses.

8 Conclusions and Future Work

This paper presents BDDT, a runtime system for dynamic dependence analysis in task-based programming models. BDDT performs dependence analysis among tasks with memory footprints spanning arbitrary ranges, deterministic execution, and an efficient and highly concurrent implementation of task instantiation, dependence analysis, and scheduling. We demonstrate that BDDT outperforms OpenMP by up to a factor of 3.8$\times$ in benchmarks where dynamic dependence analysis can exploit distant parallelism beyond barriers. In benchmarks with adequate parallelism, BDDT performs similarly or better than OpenMP when dynamic dependence analysis is deactivated, due to an efficient implementation of the runtime system. Furthermore, BDDT has lower overhead and a more efficient runtime implementation than SMPSs, a state-of-the-art task-based model based on dynamic dependence analysis.

There are several opportunities for future research in BDDT that we plan to explore, among which are: (i) adding support for irregular data accesses and dynamic memory allocation (tasks operating on lists, graphs, etc.); (ii) combining the dynamic analysis in BDDT with static analysis and user annotations in a compiler framework, to reduce runtime overhead; (iii) scale the runtime to many cores and tune it to architectural features, such as NUMA or NUCA; and (iv) adding support for Sequoia-like nested tasks.

References

[1] C. Augonnet, S. Thibault, and R. Namyst. StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines. Tech Report RR-7240, INRIA, March 2010.

[2] 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 Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[3] M. Bauer, J. Clark, E. Schkufza, and A. Aiken. Programming the Memory Hierarchy Revisited: Supporting Irregular Parallelism in Sequoia. In *Principles and Practice of Parallel Programming*, 2011.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, 2009.

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

[6] 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 *Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

[7] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *Architectural Support for Programming Languages and Operating Systems*, 2011.

[8] 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 *High Performance Networking and Computing*, 2006.

[9] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, 1993.

[10] J. C. Jenista, Y. H. Eom, and B. Demsky. OoOJava: Software Out-of-Order Execution. In *Principles and Practice of Parallel Programming*, 2011.

[11] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[12] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Architectural Support for Programming Languages and Operating Systems*, 2009.

[13] J. M. Pérez, R. M. Badia, and J. Labarta. Handling Task Dependencies under Strided and Aliased References. In *International Conference on Supercomputing*, 2010.

[14] 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.

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

[16] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, 1998.

[17] G. Rivera and C. W. Tseng. Data transformations for eliminating conflict misses. In *Programming Language Design and Implementation*, 1998.

[18] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *Hot Topics in Parallelism (HotPar)*, 2011.