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

George Tzenakis[1], Angelos Papatriantafyllou[3], Hans Vandierendonck[1], Polyvios Pratikakis[2], and Dimitrios S. Nikolopoulos[1]

[1] Queen's University of Belfast, Belfast, United Kingdom
{gtzenakis01,h.vandierendonck,d.nikolopoulos}@qub.ac.uk
[2] FORTH-ICS, Heraklion, Crete, Greece
polyvios@ics.forth.gr
[3] TU Wien, Vienna, Austria
papatriantafyllou@par.tuwien.ac.at

Abstract. We present 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, multi-dimensional array tile or dynamic region. BDDT uses a block-based dependence analysis with arbitrary granularity. The analysis is applicable to existing C programs without having to restructure object or array allocation, and provides flexibility in array layouts and tile dimensions.
We evaluate BDDT using a representative set of benchmarks, and we compare it to SMPSs (the equivalent runtime system in StarSs) and OpenMP. 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.

Keywords: Compilers and runtime systems, Task-parallel libraries, Middleware for parallel systems, Synchronization and concurrency control

1 Introduction

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 may not be easy to reproduce.

Programming models with implicit parallelism [4,10,14,13] extend task-parallel programming models with automatic inference of dependencies, requiring the programmer to only describe the memory resources required in each task. They are 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 [14,10] or language extensions [4,6]; the compiler and runtime system then discover parallelization and manage dependencies transparently.

Dynamic dependence analysis can discover more parallelism than 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 such 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 or on dynamic linked data structures.

SMPSs [12], 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, multidimensional array tile, or dynamic region. 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 marshaling, 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 memory ranges, partial argument overlapping across tasks, dependence tracking at configurable granularity, and dynamic memory management in tasks. The analysis is tunable to balance accuracy and performance.
- 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.
- We evaluate the performance of our runtime system. On a representative set of benchmarks, BDDT performs comparable to or better than SMPSs and handle arbitrary tile sizes and array dimensions. In several benchmarks, dynamic dependence analysis in BDDT discovers additional parallelism, producing speedups of up to

3.9× compared to OpenMP using barriers. BDDT outperforms OpenMP on embarrassingly parallel tasks without dependencies, by using hand-added annotations to disable the dependence analysis.

2 Dataflow Execution Engine Design

BDDT uses a dataflow execution engine based on block-level dependence analysis for identifying parallel tasks. Task arguments are annotated —at task-issue time— with data access attributes, corresponding to three access patterns: read (**`in`**), write (**`out`**) and read/write (**`inout`**). The 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. Block-based analysis can also 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. Furthermore, tasks can have arguments that are non-contiguous in memory, such as a tile of a multidimensional array or a collection of objects in random memory locations.

There are two potential drawbacks to block-based dependence analysis. First, as the dependence analysis is performed per block, the runtime system must sometimes repeat the same action across all blocks in an argument, increasing overhead. 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 or when the block size is too large. BDDT overcomes both problems. A custom memory allocator integrates the metadata with the application data to eliminate the overhead of metadata lookup, and allows the sharing of metadata between blocks. Moreover, BDDT allows the user to adjust 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: *task issue* performs dependence analysis, queuing the task if any pending dependencies are unresolved; *task scheduling* releases a task for execution when all its dependencies are resolved, selects a worker's queue and inserts the task; *task execution* executes it; and *task release* resolves pending dependencies of an executed task, potentially releasing new tasks for execution. The dynamic dependence analysis induces overheads in the issue and release stages, for checking dependencies and task wakeup, respectively. We design the data structures used in the dependence analysis specifically to minimize these overheads.

Retrieving the metadata that track dependencies for each byte, object, or block of memory accessed by a task can be expensive. A general solution to this is to maintain a hash from memory addresses to metadata [12], although this incurs a large overhead per access. A faster way is to attach the metadata directly to the actual data payload [1,17], but this may make metadata visible to the programmer, require significant additional changes to the program source and memory layout. We achieve the best of both solu-

tions using a custom memory allocator that allows for fast lookup of metadata, while still hiding metadata management in the runtime system. In short, allocation happens 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. BDDT allows multiple blocks to share the same metadata information. Then, critical dependence tracking operations operate on one metadata element instead of multiple, reducing 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 all the blocks in the tile.

To detect task dependencies, we also allow multiple metadata elements to describe the same block, capturing the task order. Specifically, each written task argument (**`out`** or **`inout`** footprint) 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 multiple 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.

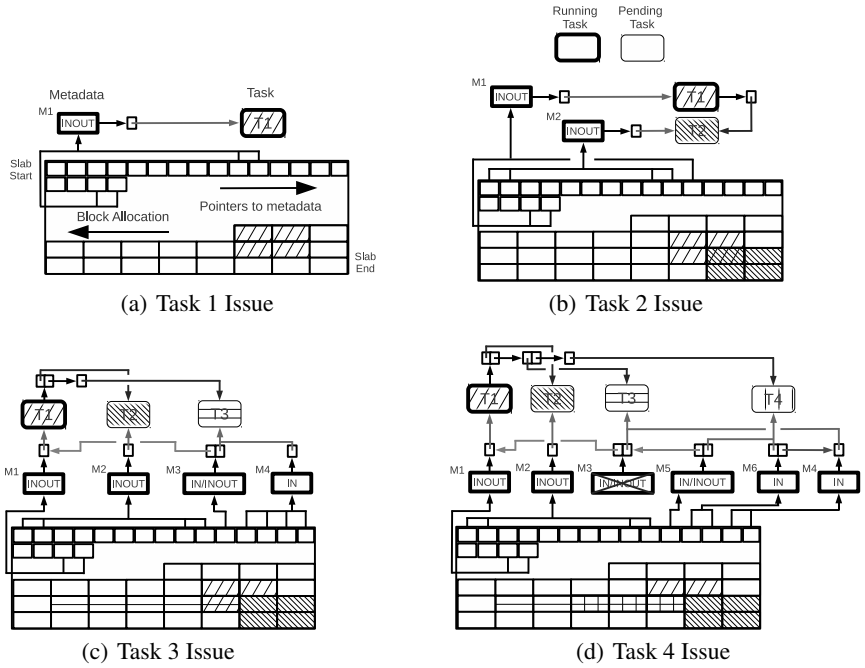
Using solely memory address ranges to describe the memory footprint of a task restricts tasks to finite footprints. Moreover, it prohibits tasks from allocating memory, as the new range is yet unknown at task-invocation time, and thus cannot be part of the footprint. To address these issues and allow tasks to operate on dynamic data structures, we also use a region-based allocator [16]. A dynamic region (or zone) is an isolated heap in which objects can be dynamically allocated. A BDDT task footprint can include dynamic regions, meaning that all memory allocated in a region is in the task's footprint, without having to enumerate the actual ranges. Moreover, a task can then allocate new memory inside a region in its footprint (when it has an `out` or `inout` effect). Dynamic regions are directly linked in the dependence analysis by allocating exactly one metadata element per region, and treating it as exactly one memory location.

3 Implementation

BDDT constructs a task dependence graph dynamically, by deducing task dependencies from the access modes and blocks in task footprints. We design the system so that identifying and retrieving dependent tasks causes minimal overhead.

The runtime system consists of (i) a custom block-based memory allocator; (ii) metadata structures for dependence analysis; and (iii) a task scheduler. The two metadata types include (i) task elements; and (ii) block elements. 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.



(a) Task 1 Issue (b) Task 2 Issue

(c) Task 3 Issue (d) Task 4 Issue

Fig. 1. Memory allocation and dependence analysis metadata

3.1 Task Elements

Each task metadata 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 size and 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.

Each task element contains a list of dependent task: tasks that wait for this task to finish. New task elements are appended to this list whenever a new dependent task is 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. As a task finishes execution and is released, it reduces the counters of all task elements in its dependent list.

3.2 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. A BDDT block metadata element may represent a collection of blocks; in contrast, systems implementing object-based dependence tracking would use one block metadata element per object. Conversely, a collection of blocks may correspond to multiple block elements describing operations on that collection. A new block metadata element is allocated for each task with a write effect (i.e., **`out`** or **`inout`**). These block 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 task elements 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. All task elements with **`in`** effect 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 is that, in typical usage, a series of tasks with **`in`** mode is always followed by a single task with **`out`** or **`inout`** mode.

By construction, the metadata elements reveal the parallelism between tasks: tasks listed in the same block metadata element may execute in parallel, while tasks in a younger block element must wait until all tasks in an older block element have finished execution.

3.3 Memory Allocator

BDDT uses a custom memory allocator to embed dependence analysis metadata in the allocator's metadata structures[4]. 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 3.2.

Figure 1(a) shows the structure of such 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)

[4] 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.

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 increases locality, as the metadata pointers 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. We optimize dependence 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 1(a) shows the state of memory after issuing task T1, which 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.

3.4 Task Issue

During task issue, BDDT identifies dependencies between the new task and older tasks by scanning all data blocks in the task arguments 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`**, or **`inout`**), and any outstanding tasks that access the same data, BDDT either immediately schedules the task, or stores it for later scheduling. If the task touches a memory block 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.

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 dependence analysis.

For example, assume that while task T1 is running, the program spawns new tasks T2 as shown in Figure 1(b). Task T2 reads and writes four blocks in **`inout`** mode, where one block overlaps with the footprint of T1. To issue T2, BDDT creates a new metadata element (M2), and iterates through the linked list of M1 to place T2 in T1's dependence list. T2 becomes the first node in the linked list of M2. Finally, BDDT alters all (including the overlapping block) slab-pointers corresponding to the blocks in T2's footprint to point to M2.

Figure 1(c) shows the issue of task T3. Task 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, Figure 1(d) shows the issue of task T4. 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.

3.5 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. Tasks with no pending dependencies are pushed for execution.

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 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 in a 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.

3.6 Complexity analysis and discussion

BDDT incurs overhead for task issue and task release. Task release overhead depends on the shape of the task graph: The runtime system receives a finished task from the scheduler and inspects 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. 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.

Task issue has similar complexity. If prior producers of a block are still executing, 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 [12]. This version of SMPSs has less functionality than BDDT: it handles only multi-dimensional

array 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 tiles in their binary representation, taking a number of operations proportional to the number of bits in an address. SMPSs walks a tree data structure to detect overlap with other tiles, updates the tree by adding the tile or updates the metadata of an already existing tile. These operations take O($d_{in} \frac{N}{R} \log T$), where R is the average tile size expressed in blocks and T is the number of tiles in the tree The tile size may be less than the argument size N because tiles must be split to eliminate false positives, with $R = O(N)$ in the worst case.

Although comparisons between block size and array length, and between average collection size C and average tile size R are not trivial, we can 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 Experimental Analysis

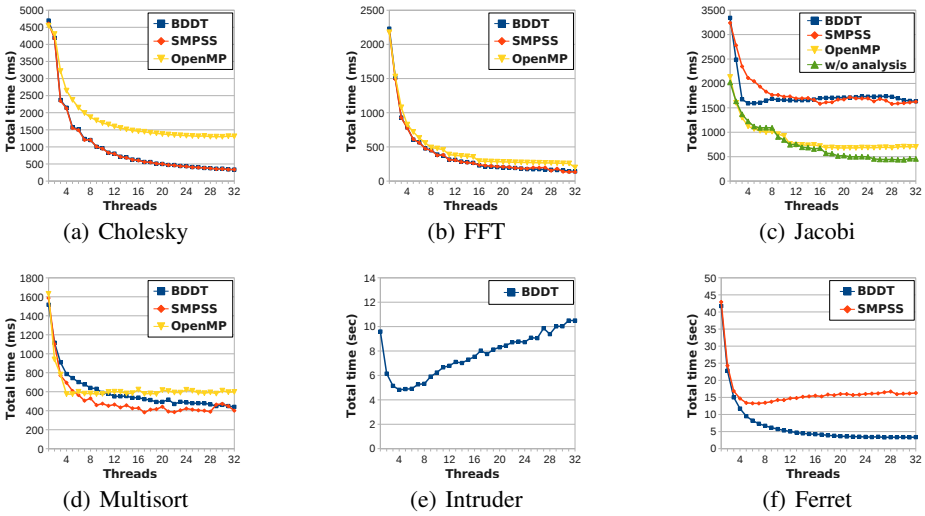
We ran all experiments on a Cray XE6 compute node with 32GB memory and two AMD Interlagos 16-core 2.3GHz dual-processors, a total of 32 cores at 8 cores per processor. Every pair of cores shares one FPU, possibly reducing floating point arithmetic performance. Each 8-core processor has its own NUMA partition, 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.

4.1 Benchmarks

We use a set of task-based benchmarks to evaluate BDDT. All benchmarks use row-major (C-language) array layout. Ferret is taken from the PARSEC benchmark suite [5] and Intruder is from the STAMP benchmark suite [7]. Cholesky, FFT, and Jacobi are SMPSs benchmarks [12] and porting them to BDDT requires only trivial changes.

We compare the performance of the task-based benchmarks with equivalent OpenMP implementations when available, 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 gain of 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. For the block linear algebra benchmarks that work on two-dimensional tiles of the input array, we set the block size to the row



(a) Cholesky (b) FFT (c) Jacobi

(d) Multisort (e) Intruder (f) Ferret

Fig. 2. BDDT, SMPSs and OpenMP on Interlagos

size of a tile. Multisort recursively splits an array until a certain threshold, which we set as the BDDT block size.

Cholesky is a factorization kernel solves normal equations in linear least squares problems. The kernel can be decomposed into four tile operations, each of which corresponds to a task in the benchmark. Dependencies among tasks create an irregular task graph, requiring the OpenMP implementation to use barriers between phases. This limits parallelism across outermost iterations of the code. Both SMPSs and BDDT overcome this limitation using dynamic dependence analysis. Figure 2(a) shows the performance of Cholesky for a 4096×4096 double precision 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 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 2(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 is a common method for solving linear equations. Each task in Jacobi works on a tile of the array. The kernel is an iterative method, so we keep an input and an output array and swap arrays from one iteration to the next. 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 kernel is data parallel, communication bound and memory intensive, and thus highlighting the analysis overhead. 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 selectively apply or turn off the dependence analysis per task argument. The "w/o analysis" line in Figure 2(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 is a parallel sorting algorithm originating from the Cilk distribution [11]. The algorithm is a parallel extension of ordinary Mergesort. Multisort recursively divides an array in halves up to a threshold, sorts each half, and merges the sorted halves, with each merge task working on overlapping parts of the array. OpenMP requires barriers between merge phases of the algorithm. Figure 2(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.

Intruder is a signature-based network intrusion detection system. It processes network packets in parallel in three phases: capture, reassembly, and detection. The reassembly phase uses a dictionary that contains linked lists of packets that belong to the same session. The lists are allocated in BDDT dynamic regions, allowing task footprints to include whole lists and tasks to allocate new elements. Each packet issues a task that inserts the packet into a list and possibly packs the list. The task footprint contains the whole region where the list is allocated. Figure 2(e) shows the performance of Intruder on 16384 sessions with max 512 packets per session. The figure only contains BDDT data, because SMPSS cannot express task footprints that contain dynamically linked

lists. Intruder scales up to 2 on 4 cores and then speedup falls to 0.9 on 32 cores. Note that while the Intruder port to BDDT outperforms the sequential code by up to a factor of 2×, the original software transactional memory implementation [7] fails to get any speedup over sequential runs.

Ferret is an image similarity search engine. We issue parallel queries to the search engine, where each query corresponds to a task. We used 1,000 images to issue queries to a database which contains 59,695 images. Figure 2(f) shows the performance of Ferret. BDDT scales up to 15.5× on 32 cores while SMPSS reaches maximum scalability of 3.4× on 6 cores.

5 Related Work

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.

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 [15] 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 [4] 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 [14] is a task-based programming model for scientific computations that uses annotations on task arguments to dynamically detect argument dependencies between tasks. SMPSs [12] 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.

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.

6 Conclusions

We present 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 or dynamic data structures, and produces deterministic executions. BDDT outperforms OpenMP by up to a factor of 3.8$\times$ in benchmarks where dynamic dependence analysis can exploit distant parallelism beyond barriers, and similarly or better than OpenMP in data-parallel benchmarks when dynamic dependence analysis is deactivated. Compared to SMPSs, a state-of-the-art task-based model based on dynamic dependence analysis, BDDT has lower overhead, supports dynamic memory management in tasks, and allows the dependence analysis to be applied (or disabled) on individual task arguments.

Acknowledgments. We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the TEXT (FP7-ICT-261580) project. This research is also supported by EPSRC through the GEMSCLAIM project (grant EP/K017594/1).

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. *TPDS*, 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 *PPoPP*, 2011.
4. 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.
5. C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, October 2008.
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 *OOPSLA*, 2009.
7. C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, September 2008.
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 *SC*, 2006.
9. M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.
10. J. C. Jenista, Y. H. Eom, and B. Demsky. OoOJava: Software Out-of-Order Execution. In *PPoPP*, 2011.
11. C. E. Leiserson. The Cilk++ concurrency platform. *TJS*, 51(3):244–257, 2010.
12. J. M. Pérez, R. M. Badia, and J. Labarta. Handling Task Dependencies under Strided and Aliased References. In *ICS*, 2010.
13. J. M. Pérez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it Easier to Program the Cell Broadband Engine Processor. *IBMRD*, 51(5):593–604, 2007.
14. J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *IJHPCA*, 23(3):284–299, 2009.
15. M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *TOPLAS*, 20(3):483–545, 1998.
16. M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2), 1997.
17. H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *HotPar*, 2011.