Hierarchical Parallel Dynamic Dependence Analysis for Recursively Task-Parallel Programs

Nikolaos Papakonstantinou, Foivos S. Zakkak, Polyvios Pratikakis
Foundation of Research and Technology – Hellas
Institute of Computer Science
Heraklion, Crete, Greece
{nikpapac,zakkak,polyvios}@ics.forth.gr

***Abstract*—This work presents a hierarchical, parallel, dynamic dependence analysis for inferring run-time dependencies between recursively parallel tasks in the OmpSs programming model. To evaluate the dependence analysis we implement PARTEE, a scalable runtime system that supports implicit synchronization between nested parallel tasks. We evaluate the performance of the resulting runtime system and compare it to Nanos++, the state of the art OmpSs implementation, and Cilk, a high performance task-parallel runtime system without implicit task synchronization. We find that *i)* PARTEE is able to handle more fine grained tasks than Nanos++; *ii)* PARTEE's performance is comparable to that of Cilk; *iii)* in cases where task dependencies are irregular, PARTEE outperforms Cilk by up to 103%.**

***Keywords*-task parallelism; dynamic dependence analysis; nested parallelism; runtime systems;**

I. Introduction

Task-parallelism offers a high-level abstraction for expressing parallelism to the programmer compared to threads and processes. Hence, task-parallel programming models gain increasing traction with parallel programmers. Early task-parallel programming models [1]–[4] required manual synchronization using locks and barriers, whereas recent approaches support implicit synchronization employing dependence analysis algorithms to discover and resolve dependencies between tasks [5], [6].

OmpSs [5] is one such programming model. OmpSs is designed as an extension to OpenMP and some of its features are slowly getting adopted by the OpenMP standard. As a result, due to the popularity of OpenMP, OmpSs is of special interest. OmpSs is implemented in Nanos++, a runtime system designed to be modular and support multiple architectures, schedulers, dependence analysis algorithms etc. Nanos++, however, fails to scale with the number of cores for task-parallel applications with fine grained tasks—tasks that take less than 2 milliseconds to complete, as we present in Section IV.

To evaluate the performance of Nanos++ we compare two parallel implementations of the mergesort algorithm. The first implementation is the *multisort* benchmark from the OmpSs Dependency Benchmark Suite [7] and the second one is the *cilksort* benchmark from the Cilk benchmark suite [1]. The main difference between multisort and cilksort is that the latter is implemented with task nesting, whereas multisort spawns all tasks from a single thread. We port cilksort to OmpSs, and compare their scalability on Nanos++ [8], the state of the art OmpSs implementation. For reference, we also measure cilksort on Cilk [1] to demonstrate that it is scalable. Cilk is a high performance task-based programming language that does not support implicit synchronization and requires manually inserted barriers to produce correct results. We present our measurements in Figure 1. On the y-axis we plot the speedup and on the x-axis we plot the number of cores. Our measurements show that both cilksort and multisort on Nanos++ fail to scale with the number of cores. We also observe that cilksort achieves better performance than multisort on Nanos++. This is expected as nested tasks distribute the overhead of task-scheduling to different cores. However, Nanos++ still fails to get speedup. We believe that this is due to the overhead of the Nanos++ dynamic dependence analysis.

In an effort to bridge the performance gap between Nanos++ and the Cilk runtime system, we present a parallel dynamic dependence analysis that enables implicit synchronization in recursively task-parallel applications written

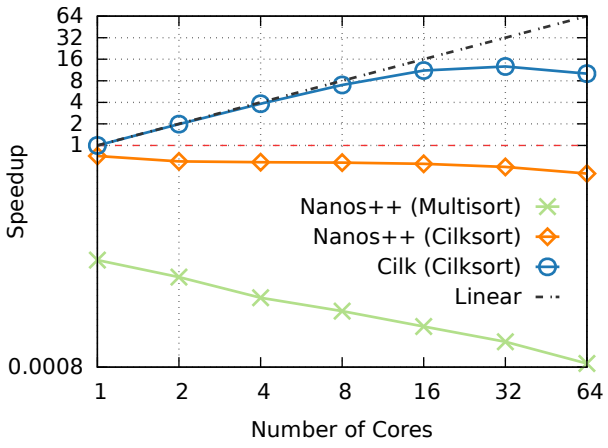

Figure 1. Cilksort vs Multisort on Multiple Runtime Systems

using the OmpSs programming model. Our approach aims to distribute the dynamic dependence analysis overhead to different cores and nesting levels in the same way nested-tasks distribute the scheduling algorithm overhead—by recursively creating new tasks that can concurrently create other tasks. We implement our dynamic dependence analysis in PARTEE, a runtime system implementing the OmpSs programming model, and evaluate its performance in respect to that of Cilk and Nanos++. We find that: *i)* PARTEE is able to handle more fine grained tasks than Nanos++; *ii)* PARTEE's performance is comparable to that of Cilk, while it provides more flexibility to the programmer; *iii)* in cases where task dependencies are irregular, PARTEE outperforms Cilk by up to 103%. Specifically, the contributions of this work are:

- A hierarchical parallel dynamic dependence analysis that infers run-time dependencies between tasks in recursively task-parallel programs.
- PARTEE, a runtime system implementing the proposed dynamic dependence analysis using a parallel region-based allocator.
- The evaluation of PARTEE in respect to the Cilk and Nanos++ runtime systems.

A. The OmpSs programming model

Our work adopts the OmpSs programming model due to its annotation language expressiveness about the tasks' dataflow. OmpSs supports more complex access patterns than those supported by OpenMP and is thus able to properly detect and resolve dependencies in more applications than the OpenMP programming model. As of OpenMP 4.5, most of the other task-related features of OmpSs have been added to the OpenMP specification. In OmpSs, similarly to OpenMP, the programmer uses pragma directives to mark tasks and express their dependencies. Similarly to sequential programs, the execution starts from the `main` function, but at `#pragma omp task` directives the runtime system creates a concurrent task that may run in parallel with the sequential `main`, much like a thread creation. Additionally, at `#pragma omp taskwait` directives the sequential `main` blocks, until all of its spawned tasks reach completion, similarly to thread-join. In task-parallel programs with nested tasks this behavior is recursive—each task may also *spawn* new tasks and *synchronize* with them, through the `omp taskwait` directive. In OmpSs, a task may only be interrupted explicitly through the `omp taskwait` directive or implicitly through the `omp task` directive, in the case of depth-first schedulers. At `omp taskwait` directives child-tasks return the ownership of their memory footprint to their parent-task, essentially creating a communication channel. Respectively, at `omp task` directives the parent-tasks transfer the ownership of a subset of their memory footprint to a new child-task and, in the case of depth-first scheduling, they get interrupted to give priority to the new child-task.

```
1  void qux(int *k, int *l) {
2    *k = *l;
3  }
4
5  void foo(int *x, int *y, int *z) {
6    #pragma omp task in(z) out(x)
7    qux(x, z);
8    #pragma omp task in(z) out(y)
9    qux(y, z);
10 }
11
12 void bar(int *k, int *l) {
13   #pragma omp task in(l) out(k)
14   qux(k, l);
15 }
16
17 int main(void) {
18   // ...
19   #pragma omp task in(z) out(x,y)
20   foo(x, y, z);
21   #pragma omp task in(x) out(k)
22   bar(k, x);
23   #pragma omp task in(m) out(l)
24   qux(l, m);
25   // ...
26 }
```

Figure 2. OmpSs Code Example

In order for the dynamic dependence analysis of the OmpSs implementation to properly detect and resolve dependencies among tasks, it requires the programmer to annotate the parameters passed to each task with the keywords `in`, `out`, and `inout`. The parameters read by the *spawning* task are marked as `in`, those written as `out`, and those that are both read and written as `inout`. We call this information the *memory footprint* of the task. Whenever a new task's memory footprint, or a part of it, is write-owned by another task, the dependence analysis defers its execution until the task owning the memory footprint, or part of it, reaches completion. As a result, in this work we differentiate task spawning from task scheduling, since, in the presence of dependencies, a task may be spawned but not scheduled until the dependencies get resolved. With spawn, we refer to the procedure of creating a new task and detecting any dependencies of it with other tasks, and with schedule, we mean that a task is ready for execution.

Figure 2 presents an OmpSs toy example. In this example, there are three functions, `qux()`, `foo()`, and `bar()`. The `qux()` function takes two parameters and stores the value of the second to the first. The `foo()` function takes three parameters and concurrently stores the value of the third to the first and the second, by *spawning* two task instances of `qux()`. The `bar()` function *spawns* a task instance of `qux()` passing through its parameters. The `main()` function first *spawns* an instance of `foo()`, to store the value of `z` to `x` and `y`; an instance of `bar()`,

to store the value of `z` to `k`; and an instance of `qux()`, to store the value of `m` to `l`.

The rest of this paper is organized as follows. In Section II, we present our hierarchical, dynamic, dependence analysis for inferring dependencies between tasks in recursively task-parallel programs. In Section III we present the key features of PARTEE, our implementation of task-based parallel runtime system that supports implicit synchronization between nested-tasks. In Section IV we evaluate PARTEE and compare it to the Cilk and Nanos++ runtime systems. In Section V we discuss related work and conclude in Section VI.

II. Dynamic Dependence Analysis

Taking advantage of the memory footprint annotations of OmpSs, Nanos++ implements a dynamic dependence analysis to automatically detect and resolve dependencies between tasks at run-time. Mercurium [9] and SCOOP [10] also attempt to infer dependencies at compile-time, using static dependence analysis. However, static dependence analysis cannot detect and resolve all dependencies, limiting the exposed parallelism. As a result, to achieve better performance, it is best to combine a static dependence analysis with the dynamic dependence analysis to reduce the latter's overheads [6], [10], [11].

A. Task Dependency Graphs

Using the memory footprint of the tasks, the runtime system is able to create the task dependency graph of a task-parallel program. In Figure 3 we sketch the task dependency graph of the toy example presented in Figure 2. For each task instance we draw a rounded rectangle. We use solid arrows to represent task *spawns*, and to demonstrate the parent-child relationship between the *spawned* tasks. We use dashed arrows to show the dependencies we need the runtime system to detect and resolve, in order to correctly execute the program. We use dotted arrows to show the actual dependencies present in the program.

Since `bar()` accesses `x` which is written by `foo()` it needs to wait for it to complete, hence the dependency between `Task2` and `Task1`. Examining `foo()`, though, we observe that it never actually writes `x`. Instead, `x` is being written by the first invocation of `qux()` in `foo()`, which is why `Task2` depends on `Task1.2`. Note that in the OmpSs programming model the notion of dependencies is transitive; meaning that for any task t that depends on a task t', all child-tasks of t also depend on t'. As a result, `Task 2.1` depends on both `Task 1` and `Task 1.2`. We further discuss this property below. We also discuss why a single dependency between `Task 1` and `Task 2` is enough to properly execute the code in Figure 2. Finally, note that Figure 3 is a sketch to improve readability and does not reflect the runtime system's internal representation of task dependencies.

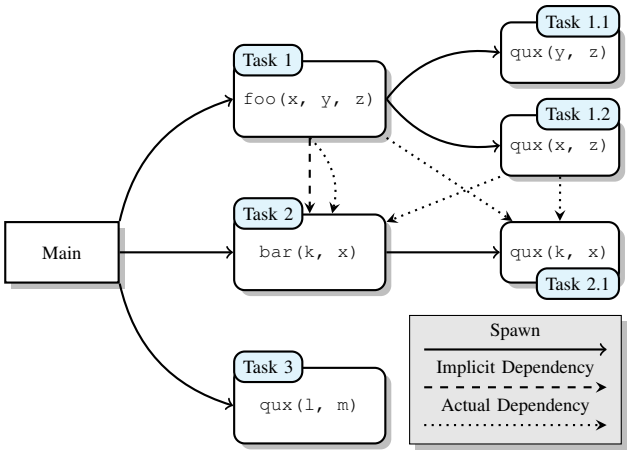

Figure 3. Task Dependency Graph

B. Hierarchical Distribution

The nature of nested-tasks allows us to create a spawn-tree, a tree representation of the tasks that will be spawned at run-time and the relation to their parents. In a spawn-tree, the root is the main function, interpreted as a virtual task. The intermediate nodes are both child-tasks, of the higher level tasks, and parent-tasks, of the lower level tasks in the tree. Finally the leaf nodes are child-tasks that perform no further spawns. Figure 3 includes such a spawn graph for the toy example in Figure 2.

The main advantage of nested-tasks over flat-tasks is that as the spawn-tree width and depth increase, the creation of new tasks gets scattered to different cores, essentially distributing the overhead of spawning and scheduling new tasks. As a result, to detect and resolve dependencies efficiently, the dependence analysis should be distributed in a similar manner. To ensure deterministic execution, the programming model requires parent-tasks to include their child-tasks' memory footprints in their own memory footprint [12]. Thus, if a task owns a memory segment, then all its ancestor-tasks—all the tasks in the path from the root of the spawn-tree to that task—own it as well. That is, for any task t, its memory footprint is the union of its own and its child-tasks' memory footprints. Additionally, for each memory segment there may not exist more than one paths in the spawn-tree with write ownership to it, or part of it. As a result. a task may only be scheduled when none of its sibling-tasks (if any) has a write ownership on any of the memory segments, or parts of them, in its memory footprint, and its parent-task has the ownership of its memory footprint. Note that, by definition, a descendant-task's spawn depends on the scheduling of its parent-task—if its parent-task is not yet scheduled, then it cannot be spawned either. Consequently, when a task is spawned, its parent must have

already been scheduled and own its memory footprint. This property narrows down the conditions to be checked at run-time to one—whether any of the new task's sibling-tasks have write ownership on any part of its memory footprint.

In general, two tasks are said to be *dependent* when the one is not a descendant-task of the other, and the intersection of their memory footprints is not the empty set. Resolving such dependencies is required to ensure the correct execution order of tasks.

C. The Base Algorithm

To check whether a memory footprint or a part of it in a task's memory footprint is already owned by any of its sibling-tasks (if any) we propose a novel algorithm which is the base of our dynamic dependence analysis. The algorithm consists of two parts, one that creates the dependencies and should be run by the parent-tasks whenever they spawn a new child-task, and one that resolves dependencies and should be ran by each task when it reaches completion.

To simplify the presentation of the algorithm, in this section we assume that the dependence analysis operates on objects instead of arbitrary memory segments. Below, in Section III we explain how we extend the algorithm to work with arbitrary memory segments in its implementation.

To keep track of the dependencies and the memory footprint of each task at run-time, the algorithm we propose requires the association of each task with a *task descriptor* comprising of: (*a*) a reference to the code to be executed; (*b*) the *argument list*, a list of argument descriptors (the task's memory footprint); (*c*) the *dependencies counter*, counting how many tasks this task depends on; (*d*) the *resolved dependencies counter*, counting how many dependencies of this task have been resolved—the task owning them, released them; and (*e*) the *notify list*, a list of *task descriptors* for the tasks that have dependencies on this task. Each argument descriptor in the *argument list* consists of: (*a*) a reference to the allocated *object*; and (*b*) the requested *type* of the argument;

Additionally, our algorithm associates each object with a set of attributes that hold the ownership information. This information is (*i*) a reference to the last *owner* of the argument—the task descriptor of the task which was last assigned the argument; (*ii*) the ownership *type* of the argument—the type of the access performed by the last task; (*iii*) the *readers' list*—a list of *task descriptors* that use this argument for *read access* (if any).

At run-time, whenever a new object is allocated, it gets associated with the above set of attributes. Additionally, whenever a new task is spawned it gets associated with a *task descriptor*. Using these associations the dependence analysis is able to trace back which tasks (if any) access an object and the corresponding type of the access (i.e., *in*, *out*, or *inout*).

Figure 4 presents the algorithm that creates the dependencies for new tasks. For each argument, we first check if the argument is not used by any other task (line 3). If true, we assign to the corresponding object the new task as its owner, and store the access type (lines II-C–5). If false, we proceed by checking for read after read (RAR), write after read (WAR), read after write (RAW), and write after write (WAW) dependencies.

In the case of RAR accesses, we simply append the task descriptor of the new task descriptor to the object's readers' list (line 8). In the case of WAR accesses, we go through the object's readers' list and we append the new task to every reader's notify list and increase the local "waitfor" counter (lines 10–14). We then assign the object to the new task and clear its readers' list (lines 15–17). This way, we delay the scheduling of the new task until all the reader tasks reach completion. At the same time, by assigning the object to the new task, we ensure that any future accesses are going to wait for it. In the case of RAW accesses, we try to append the new task to the notify list of the object's current owner (line 19). On success, we proceed to increase the local "waitfor" counter (line 20). On failure, we assume that the last owner has reached completion and thus the dependency is actually a RAR, so we update the object's type for future accesses (line 22). Last, we append the task to the object's readers' list (line 24). In the case of WAW accesses, we attempt to append the new task to the notify list of the object's current owner and then we assign the corresponding object to the new task (lines 26–30).

If an argument fails to match any of the previous criteria, it means it is marked as *safe* by the SCOOP compiler (line 32). Arguments marked as *safe* are arguments that have been found by the compiler to not cause any dependency during run-time under any execution, thus they can be safely ignored by the dynamic dependence analysis. At the end, we check whether the local "waitfor" counter is zero to decide whether the task can be scheduled or not (line 35). If the "waitfor" value is different than zero, we assign its value to the task's dependencies counter (line 38). A task with a non-zero dependencies counter may not be scheduled until its resolved dependencies counter's value becomes equal to that of its dependencies counter's value.

The use of the local "waitfor" counter, instead of directly increasing the dependencies counter, aims to eliminate the chance of scheduling a task when its resolved dependencies counter reaches the same value as its dependencies counter but not all of its dependencies have been created yet.

Figure 5 presents the algorithm that resolves the dependencies when tasks reach completion. At task completion, we go through its notify list, and, for every task in it, we increase its resolved dependencies counter and then check if it is equal to the dependencies counter (lines 1–8). Note that we use the atomic primitive *fetch-and-add* (FAA) to perform the increase. This is necessary, since more than one

```
1: waitfor ← 0
2: for each arg ∈ task.arguments do
3:    if arg.object.owner = ∅ then
4:       arg.object.owner ← task                    /* First use */
5:       arg.object.type ← arg.type
6:    end if
7:    if arg.type = in and arg.object.type = in then
8:       arg.object.reader_list.append(task)        /* RAR */
9:    else if arg.type ∈ {out, inout}
            and arg.object.type = in then
10:      for each reader ∈ arg.object.reader_list do
11:         if reader.notify_list.append(task) then
12:            waitfor ← waitfor + 1                /* WAR */
13:         end if
14:      end for
15:      arg.object.owner ← task
16:      arg.object.type ← arg.type
17:      arg.object.reader_list.clear()
18:   else if arg.type = in
            and arg.object.type ∈ {out, inout} then
19:      if arg.object.owner.notify_list.append(task) then
20:         waitfor ← waitfor + 1                   /* RAW */
21:      else
22:         arg.object.type ← arg.type              /* RAR */
23:      end if
24:      arg.object.reader_list.append(task)
25:   else if arg.type ∈ {out, inout}
            and arg.object.type ∈ {out, inout} then
26:      if arg.object.owner.notify_list.append(task) then
27:         waitfor ← waitfor + 1                   /* WAW */
28:      end if
29:      arg.object.owner ← task
30:      arg.object.type ← arg.type
31:   else
32:      continue                    /* marked safe by SCOOP */
33:   end if
34: end for
35: if waitfor = 0 then
36:    schedule(task)
37: else
38:    task.deps_counter ← waitfor
39: end if
```

Figure 4. Create Dependencies Algorithm

tasks reaching completion may race each other in increasing the *resolved dependencies counter* of a task that happens to depend on both of them. Additionally, the use of *fetch-and-add* ensures that only a single core may observe that the resolved dependencies counter is equal to the dependencies counter and thus schedule the corresponding task. To achieve this, we use the fetched value added by one and compare it to the value of the corresponding task's dependencies counter (line 4). Note that the *owner* field is never set to ∅ by the algorithms in Figure 4 and Figure 5. The *owner* field may only be equal to ∅ when the corresponding object has not yet been accessed by any task.

Note that the create dependencies algorithm might race with the resolve dependencies algorithm, since a task spawn may race with the completion of a task it depends on. When

```
1: task.notify_list.lock()
2: for each task' ∈ task.notify_list do
3:    deps ← FAA(task'.resolved_deps_counter, 1)
4:    if task'.deps_counter = deps + 1 then
5:       schedule(task')
6:    end if
7: end for
8: task.notify_list.clear()
```

Figure 5. Resolve Dependencies Algorithm

a task being spawned, tries to be inserted in the *notify list* of the task it depends on (Figure 4 lines 11, 24, 26), while the latter tries to empty its *notify list* and notify the tasks waiting for it that it reached completion (Figure 5). To protect the *notify list* in such cases and not allow further additions to it if the resolve dependencies algorithm is running, we use a special node that we baptize *lock*. If the head of the list points to that node, then insertions to that list will fail, meaning that the task owning it reached completion and thus there is no need to add a new node to its *notify list*. To ensure that the head of the list is atomically updated, we use the *compare-and-swap* instruction to insert elements to the head of the list and to lock it. To resolve dependencies we first lock the *notify list* (line 1).

For instance, in the toy example of Figure 2, the `main` function will first spawn *Task 1*, then *Task 2*, and finally *Task 3*. At the spawn of *Task 1* the corresponding *task descriptor* will be created and initialized according to its memory footprint. Then, the create dependencies algorithm (Figure 4) will be run to check for dependencies. In this case, *Task 1* can be directly scheduled, since there are no other active tasks yet, and thus no dependencies (waitfor $= 0$). Later, at the spawn of *Task 2*, similarly, the corresponding task descriptor will be created and initialized, and the create dependencies algorithm will be run to check for dependencies. In that case, assuming *Task 1* is still running, the `x` will be owned by *Task 1* (arg.object.owner $\neq \varnothing$). As a results, the analysis will detect a WAR dependency between *Task 1* and *Task 2*, and increase *Task 1*'s dependencies counter (Figure 4 lines 10–14) and add *Task 2* in the *notify list* of *Task 1* (Figure 4 lines 15–17). Finally at the spawn of *Task 3* we proceed similarly to find that all arguments are not owned and will proceed with the scheduling.

Regarding the nested tasks *Task 1.1* and *Task 1.2*, the process is similar. In the case of *Task 1.2*, the dependence analysis will find that `z` is read-owned by *Task 1.1* and will proceed by appending *Task 1.2* to the readers' list of `z`.

D. Limitations

Although compliant with the OmpSs programming model specification, only detecting dependencies between sibling-tasks imposes a limitation to the expressiveness of the programming model. Figure 6 gives an example where a code segment (line 4) of a task depends on the output of one

```
1 // ...
2 #pragma omp task out(x) in(z)
3 qux(x, z);
4 *z = *x;
5 #pragma omp task out(y) in(z)
6 qux(y, z);
7 // ...
```

Figure 6. Task depending on the output of its child

of its child-tasks (line 3). In such cases, our analysis fails to detect this dependency, and requires the developer to add this code segment in a child-task, so that the dependence analysis will be able to detect and resolve the dependency. Alternatively, an explicit `taskwait` directive between the child-task and the corresponding code segment also suffices. Depending on the level of available parallelism in the task, one case might be preferred over the other. For instance, if the task spawns many independent child-tasks and this is one of the few existing dependencies, the creation of a new child-task should be preferred.

E. Synchronization

In OmpSs a task may only yield explicitly through the *taskwait* directive or implicitly in the case of depth-first schedulers, through the *task* directive. In our work, we also assume that each task performs an implicit *taskwait* at its end. This implicit *taskwait* directive ensures that all of its child-tasks reached completion before the task itself reaches completion. We find this behavior more intuitive to the programmer and safer, since it allows annotated programs with nested-tasks to implicitly synchronize without the need of any explicit *taskwait* directives placed by the programmer. Furthermore, from our experience from porting benchmarks to OmpSs, an explicit *taskwait* at the end of each nested task is almost always required to produce correct results.

To implement the *taskwait* directive, we extend the *task descriptor* with a counter holding the number of child-tasks of the corresponding task, and a reference to its parent-task's descriptor. Whenever a task spawns a child-task, it also atomically increases this counter. Similarly whenever a child-task reaches completion it atomically decreases its parent-task's counter.

Note that the explicit *taskwait* primitive is supported only for compatibility with the OmpSs programming model and to handle limitations like those described in Section II-D. For the correct synchronization of the application, our analysis relies only on the correct annotation of the tasks' memory footprints and the implicit *taskwait* directives.

III. The PARTEE: PARallel Task Execution Engine

We implement our dynamic dependence analysis in a new runtime system, called PARTEE, because OmpSs is focused on being modular at the cost of decreased efficiency (as we show in Section IV), and the Cilk runtime system and scheduler are not straightforward to extend in a way to support task reordering. To rectify this, we design PARTEE to use *task descriptors*, as described in Section II, which are straightforward to reorder.

A. Task Scheduling

PARTEE uses Blumofe and Leiserson's *work-stealing* algorithm [13] for scheduling tasks. The runtime system consists of P software threads, each pinned to one of the P available hardware threads. We call these software threads virtual processors (VP). Each VP maintains a task-queue where it can spawn and execute tasks to and from. In case its task-queue is empty, it tries to steal work from another VP. We implement the task-queues using a non-dynamic variant of the lock-free deque proposed by Chase and Lev [14].

To *spawn* a new task, a VP needs to create a new *task descriptor* and perform the dependence analysis on it, as described in Section II. For this process we use SCOOP [10], a source-to-source compiler which enables us to use `#pragma` directives for task annotation. SCOOP, using the information from the task annotations, generates code that creates the *task descriptor*, initializes it, and finally passes it to the dependence analysis. Additionally, SCOOP performs a static analysis on the task footprints and is able to exclude arguments that are safe to omit from the dynamic dependence analysis—they do not create any dependencies.

When *taskwait* is invoked, the executing VP waits for all of the current task's child-tasks to reach completion. To avoid spinning or idling, VPs waiting at a *taskwait* execute tasks from their task-queue or try to steal from other task-queues if the latter is empty.

B. Block-Based Analysis

To improve readability, in Section II we assume that the dependence analysis algorithm operates per object. However, such an analysis would fail to detect dependencies between arbitrary memory accesses through pointer arithmetic, and overlapping tile and block array accesses. To properly handle such dependencies, we adopt and extend the block-based approach of Tzenakis et al. [15]. We conceptually slice all arguments into blocks of a predefined block size. Each block is associated with a set of attributes that keep its ownership information, as presented in Section II for objects. Instead of going through each argument, the analysis goes through each block of each argument and performs the steps presented in Figure 4. If two arguments of two different tasks contain the same block, then they overlap and may create a dependency.

Since our dependence analysis only checks for dependencies between sibling-tasks, we implement a distinct look-up table (LUT) per task, stored in its *task descriptor*. This LUT holds the associations between blocks, owned by child-tasks, and their set of attributes. Employing this hierarchical design, when *spawning* a new task, PARTEE only needs to

query the *spawning* task's LUT to get each block's owner, in the new task's memory footprint. Since tasks are atomic, the spawn of each child task will be executed by the same VP, thus there is no need to synchronize with other VPs. That also holds for updates to the LUT about a block's owner, as well as, its *readers' list*. This behavior, combined with thread pinning (for the VPs), results in increased spatial and temporal locality, consequently increasing the number of cache hits and thus significantly improving the performance and energy efficiency of the dependence analysis.

To further reduce the overhead, we implement LUTs as two-level array-based *tries* [16]. A *trie* is an ordered *tree* data structure that is used to store a dynamic set or associative array where the keys are usually strings. In our case, we use the memory address of each argument as the key. To perform a look-up we mask the memory address and use its x most significant bits to index the first level of the trie, next we use the following y bits to index the second level. Tuning x and y values allows us to configure PARTEE's block size and detect dependencies on different granularity. For instance, assuming a 64-bit address space, setting x to 28 and y to 28 enables PARTEE to operate on cache line granularity—block size equal to 64 bytes. That is, if two tasks access the same cache line, then PARTEE will detect a dependency. Finer granularity results in increased run-time overhead while coarser granularity may result in false dependencies, and thus reduced exposed parallelism.

C. Region-Based Allocation

To manage the memory required to store the task descriptors and their LUTs we developed and use a region-based parallel allocator. Our allocator is based on that of Gay and Aiken's [17]. A region is a collection of allocated objects that can be efficiently de-allocated all at once. Thus, the runtime system is able to free a region with all its subregions efficiently and keep the freed memory in a pool for future use. In PARTEE each *task descriptor* owns a region. In this region, it allocates its LUT and all the *task descriptors* of its child-tasks. As a result, whenever a task reaches completion, PARTEE can safely and efficiently free all the memory allocated for its own and its child-tasks needs. Additionally, allocating all the memory related to a *task descriptor* in a single region results in increased memory locality, since the memory within a region is usually contiguous pages.

Gay and Aiken, in their allocator, hold two lists of free pages, which are used for allocation and de-allocation. We found these two lists to be a point of contention when using the region-allocator from within multiple threads. For the needs of PARTEE, we make the free lists distributed. We essentially create n free lists, each protected by a lock. Whenever a VP needs to allocate or free a region, it randomly peaks one of the n lists and tries to lock it. On successful lock it proceeds with the allocation or de-allocation and frees the lock. On failure it randomly peaks one of the n lists again and retries. The value of n depends on the number of available VPs. Experimentally we find that a value of 8 is sufficient to handle 64 VPs. A higher value of n does not impact performance but might affect the total size of memory used by the region-based allocator.

Gay's and Aiken's allocator also creates a tree of regions ensuring that all regions will be de-allocated before the termination of any program. This tree is also used to perform queries about the region that a memory address maps to. To achieve this, Gay and Aiken create a global region used as the root and add new regions as its children. Maintaining this root region up to date in a parallel allocator, however, also requires some kind of mutex exclusion. Since PARTEE does not need to query the region-based allocator about the region of an address, we drop this feature and create all the regions at top level. Regarding de-allocation before program termination, since we free a task's region at completion, it follows from the tree-like task graph that all allocated regions will be eventually freed.

IV. Evaluation

We evaluate PARTEE on a 4-chip NUMA system with 16 cores per chip, totaling 64 AMD Opteron Processor 6272 cores, with 256 GB RAM. We evaluate PARTEE using six benchmarks and compare it with Cilk and OmpSs. We run each benchmark 10 times on varying number of cores, from 1 to 64, doubling the number of cores at each step. We then estimate the geometric mean of the execution time and calculate the speedup over the geometric mean of the sequential executions. To plot the linear scale line, we calculate the sequential execution time using the native benchmark, without using any runtime system. All reported speedups are calculated with the native benchmark execution time as the baseline. Additionally, for every benchmark, we perform a run on a single core using each runtime system to demonstrate the corresponding runtime system's overhead over the native application. To produce the executable files, of each benchmark, for each runtime system, and run them we use the tool versions presented in Table I.

Nanos++ provides five different dependence analyses plugins, each with different characteristics. In this work we aim to provide a single dependence analysis for all cases (e.g., pointer arithmetic, aliasing, tile accesses, plain accesses etc.), thus we compare it only with the *regions* dependence

Table I
Tool Versions

Tool	Version	Flags
GCC	4.4.8	-O3
SCOOP	2.2.0	
Mercurium	1.99.7	-O3 –ompss
Nanos++	0.7.10	–deps=regions –schedule=dbf
Cilk	5.4.6	-O3

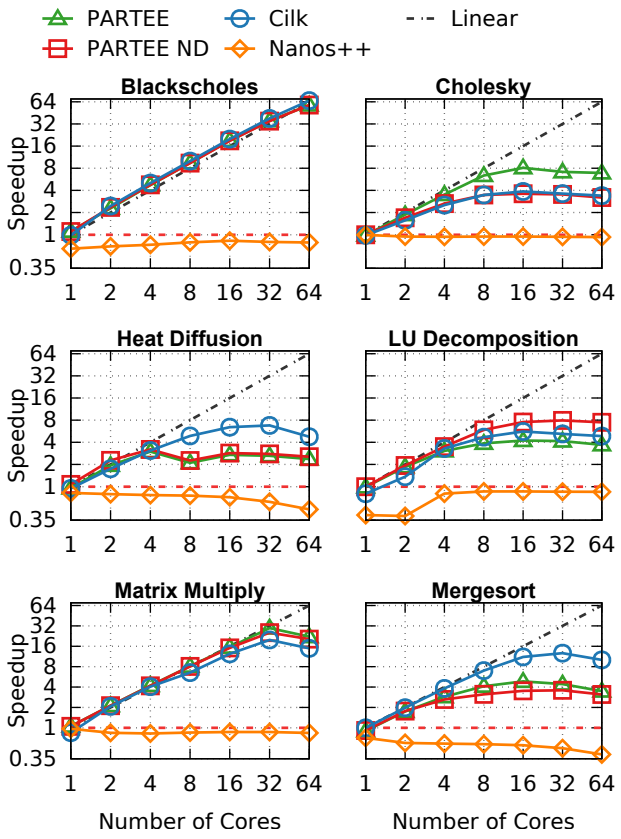

Figure 7. Speedup Over Sequential

plugin of Nanos++ that can also handle such cases. Although the *plain* dependence plugin may be more suitable than the *regions* plugin for some of the benchmarks, we avoid tuning Nanos++ for each application, just as we do for PARTEE. Note, however, that in our measurements even with the *plain* dependence plugin PARTEE outperforms Nanos++.

Figure 7 presents the speedup measurements of the six benchmarks. On the y-axis we plot the speedup and on the x-axis the number of utilized cores. Both axis are in base-two logarithmic scale. A horizontal, dashed, red line helps to separate slowdowns from speedups, and a gray line crossing the plot shows the linear speedup, which is calculated as:

$$\text{linear speedup} = \frac{\text{execution time of native benchmark}}{\text{utilized cores}}$$

We use green triangles to mark speedups of PARTEE; red squares to mark speedups of PARTEE with explicit *taskwait* directives and the dynamic dependence analysis disabled (PARTEE ND), to demonstrate the latter's impact on the speedup; blue circles to mark speedups of Cilk; and orange diamonds to mark OmpSs. Additionally, in Table II we sum up the input arguments for each benchmark.

Black-Scholes is an embarrassingly parallel benchmark from the PARSEC benchmark suite [18]. We use this benchmark to show the performance and scalability of PARTEE on benchmarks without dependencies. Our measurements show that PARTEE's performance is similar to that of Cilk. Additionally, we observe that both Cilk and PARTEE achieve super-linear speedup, which we attribute to cache effects, since the task memory footprint fits in the L1 cache. On the contrary, we observe that Nanos++ fails to get a speedup. This behavior is consistent in every benchmark. As a result we skip the discussion about Nanos++ for the rest of the benchmarks and give an explanation for this behavior at the end.

Table II
BENCHMARK PARAMETERS

Benchmark	Problem Size	Task Footprint
Black-Scholes	10,000,000	1024 options
Cholesky	4096×4096	256 × 256 doubles
Heat Diffusion	4096×4096	10×4096 doubles
LU	2048×2048	256 × 256 doubles
Matrix Multiply	2048×2048	256×256 & 256×512 floats
Mergesort	4,194,304	8192 elements

Cholesky Decomposition operates on matrices and is commonly used to solve systems of linear equations. The implementation of this benchmark comes from the OmpSs Dependency Benchmark Suite [7] and is not recursively parallel. Our measurements show that PARTEE without the dynamic dependence analysis performs similar to Cilk. With the dynamic dependence analysis enabled, PARTEE is able to expose more parallelism and improve performance. This is an indication that in cases where the parallelism is not only available in phases that can be easily synchronized with *taskwait* directives, dynamic dependence analysis not only eases development, but is also able to expose more parallelism. In this case, PARTEE outperforms Cilk by up to 103%.

The rest four benchmarks are examples from the Cilk distribution that use nested tasks.

Heat Diffusion performs some stencil computation multiple times, to calculate temperature distribution in space. Our measurements show that both versions of PARTEE fail to compete with Cilk after 4 cores. This benchmark uses two matrices, the first one as input and the second one as output. At each step, the two arrays are swapped and the stencil computation is performed again. The stencil computation in each step is embarrassingly parallel, so a *taskwait* directive at the end of each step suffices. PARTEE task creation overhead in this case appears to dominate and decrease the overall performance. This is possible when task arguments consist of many blocks and result in the creation of a large number of entries in the LUTs. For the evaluation of PARTEE we chose a fixed block size of 2 kilobytes, which in this case is much smaller than the argument size, increasing the task creation overhead.

LU Decomposition, similarly to Cholesky, operates on matrices to solve systems of linear equations. Our measurements show that PARTEE with the dynamic dependence analysis disabled and Cilk outperform PARTEE. This is an indication that LU has no interleaved dependencies and can be efficiently expressed using *taskwait* directives. We observe that Cilk's performance falls between that of PARTEE and PARTEE with the dynamic dependence analysis disabled, while all three shape a similar curve.

Matrix Multiply takes two matrices as input and writes their product in a third one. Our measurements show that both versions of PARTEE outperform Cilk. Contrary to LU, Matrix Multiply spawns a single type of task with a few arguments allowing the SCOOP compiler to optimize task-generationWe observe that at 64 cores both runtime systems fail to scale. We attribute this to the architecture design that shares a single compute unit between two cores, and the computation intensive nature of matrix multiply.

Mergesort is a recursively parallel implementation of mergesort. Our measurements show that Cilk outperforms both versions of PARTEE in mergesort. The implementation of this benchmark, follows the map-reduce principle. First binary splits the array and sorts the segments. Then starts merging the sorted segments producing larger sorted segments, until the whole array is sorted. By design this implementation can be sufficiently expressed using the *taskwait* directive. However, PARTEE fails to reach Cilk's performance even with the dynamic dependence analysis disabled, like in heat diffusion.

In all benchmarks we observe that Nanos++ fails to speed up computation as the number of cores increases. We attribute this behavior to the focus of Nanos++ on being modular, for research purposes, at the cost of decreased efficiency. To get an estimation of the overhead introduced by each runtime system we use a micro-benchmark that spawns tasks with different workloads and measures the time they take to complete. When the execution time of a task is longer than its workload, this additional time is the overhead introduced by the runtime system. To measure the actual workload we use the native application and measure the average execution time of each task without the overheads of its creation and handling. Figure 8 presents the results. On the x-axis we plot the actual workload per task, and on the y-axis we plot the measured execution time per task. Both axes are at base-two logarithmic scale. Our measurements show that PARTEE and Cilk are able to handle two orders of magnitude finer grained tasks than Nanos++. To keep the runtime system's overhead bellow 2%, Nanos++ requires tasks of at least 2-millisecond granularity, PARTEE requires tasks of at least 40 microseconds granularity, and Cilk requires tasks of at least 20 microseconds. Finer grained tasks allow for better work balancing and are critical in image processing algorithms that aim to deliver a high number of frames per second (fps). For instance to achieve 60fps the per frame computation needs to complete at approximately 16 milliseconds. This renders OmpSs and its current implementation prohibitive for use in such cases.

V. Related Work

Cilk [1] is a multi-threaded language accompanied by a high performance task-parallel runtime system. Cilk provides a clean and simple way to express parallelism through the *spawn* and *sync* directives. Cilk is not able to perform implicit synchronization to avoid data races and requires explicit synchronization through the *sync* directive. Furthermore, Cilk only allows the spawn-tree to start from the `main` function, and does not allow regular-functions to appear in the call-graph between two spawns. This limitations impact the language expressiveness, especially for non recursively parallel benchmarks. PARTEE combines nested parallelism with a dynamic dependence analysis algorithm to automatically detect and resolve dependencies between tasks and supports the more flexible OmpSs programming model.

Nanos++ [8] is the state of the art implementation of OmpSs. Our measurements show that PARTEE outperforms Nanos++ and that it can handle more fine grained tasks than Nanos++. Both runtime systems support nested-tasks and implicit synchronization through dynamic dependence analysis. Additionally, the dynamic dependence analyses of both runtime systems exhibit similar limitations. Nanos++ does not handle unaligned memory addresses [19, §4.4] and requires the size of the argument to be a power of two. If these two requirements are not met, Nanos++ may detect false dependencies between tasks and reduce performance. Similarly, PARTEE's block-based dependence analysis, may detect false dependencies if the argument size is not a multiple of the block-size of the block-based analysis or not aligned to it.

BDDT [15] is a task-parallel runtime system that employs a block-based dynamic dependence analysis to dynamically discover and resolve dependencies between tasks. In BDDT,

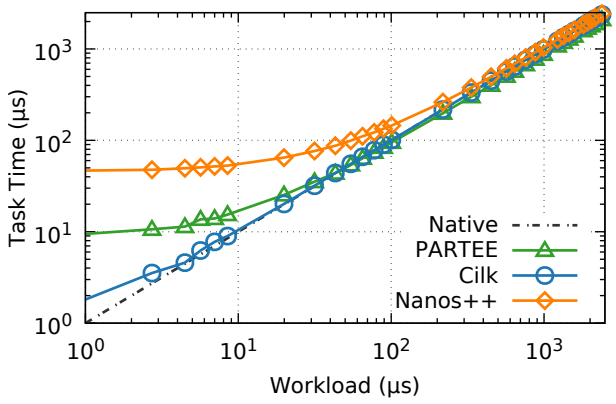

Figure 8. Overhead vs Task Granularity

however, only one thread creates tasks and it does not support nested parallelism. PARTEE employs a similar approach to BDDT to detect dependencies, but in a distributed manner, and supports nested parallelism. Additionally, PARTEE implements the association of blocks and their metadata with the use of a *trie* instead of the BDDT's custom allocator.

Sequoia [20] is parallel programming language that requires the programmer to describe: *a*) the task graph as a hierarchy of nested parallel tasks; *b*) the memory hierarchy of the targeted machine; and *c*) the data distribution among tasks. Sequoia then inserts implicit barriers to ensure the correct execution of the application. We believe that PARTEE's programming model is more intuitive and portable. Additionally, in applications with irregular dependencies we expect PARTEE to expose more parallelism than Sequoia.

VI. Conclusions

In this work, we present the design of a hierarchical dynamic dependence analysis for recursively task parallel runtime systems, and its implementation in the PARTEE runtime system. We also present the design and implementation of a custom, parallel, region-based memory allocator we use to increase locality and concurrency of our runtime system, PARTEE. We evaluate PARTEE on a set of representative benchmarks and find that, in one case where task dependencies are irregular, PARTEE outperforms Cilk, a task-parallel runtime system without implicit task synchronization, by up to 103%.

Acknowledgements: This work was supported in part by the European Commission in the context of FP7 ASAP project (619706). We would also like to thank Christi Symeonidou for her valuable reviews on this work, and the BSC pm-tools team for their responses to our questions regarding OmpSs and Nanos++.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Principles and Practice of Parallel Programming*, 1995.

[2] G. Tzenakis, K. Kapelonis, M. Alvanos, K. Koukos, D. S. Nikolopoulos, and A. Bilas, "Tagged procedure calls (tpc): Efficient runtime support for task-based parallelism on the cell processor," in *High Performance Embedded Architectures and Compilers*, 2010.

[3] D. Leijen, W. Schulte, and S. Burckhardt, "The Design of a Task Parallel Library," in *Object Oriented Programming Systems Languages and Applications*, 2009.

[4] Intel, "Threading building blocks," 2014, version 4.2, https://www.threadingbuildingblocks.org/.

[5] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[6] J. C. Jenista, Y. h. Eom, and B. C. Demsky, "OoOJava: Software Out-of-order Execution," in *Principles and Practice of Parallel Programming*, 2011.

[7] "OmpSs Dependency Benchmark Suite," https://pm.bsc.es/projects/bar/wiki/dependency_benchmarks, Sep 2015.

[8] "Nanos++," https://pm.bsc.es/nanox, Sep 2015.

[9] S. Royuela, A. Duran, and X. Martorell, "Compiler Automatic Discovery of OmpSs Task Dependencies," in *Languages and Compilers for Parallel Computing*, 2013.

[10] F. S. Zakkak, D. Chasapis, P. Pratikakis, A. Bilas, and D. Nikolopoulos, "Inference and Declaration of Independence in Task-Parallel Programs," in *Advanced Parallel Processing Technology*, 2013.

[11] R. L. Bocchino, Jr., 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.

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

[13] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 356–368.

[14] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Symposium on Parallelism in Algorithms and Architectures*, 2005.

[15] G. Tzenakis, A. Papatriantafyllou, H. Vandierendonck, P. Pratikakis, and D. Nikolopoulos, "BDDT: Block-level Dynamic Dependence Analysis for Task-Based Parallelism," in *International Conference on Advanced Parallel Processing Technology*, 2013.

[16] E. Fredkin, "Trie memory," *Communications of ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960.

[17] D. Gay and A. Aiken, "Memory management with explicit regions," in *Programming Language Design and Implementation*, 1998.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Parallel Architectures and Compilation Techniques*, 2008.

[19] J. M. Perez, R. M. Badia, and J. Labarta, "Handling Task Dependencies Under Strided and Aliased References," in *International Conference on Supercomputing*, 2010.

[20] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Supercomputing*, 2006.