A Programming Model For Deterministic Task Parallelism

Polyvios Pratikakis
FORTH-ICS
polyvios@ics.forth.gr

Hans Vandierendonck
Dept. of Electronics and Information Systems, Ghent University
hvdieren@elis.ugent.be

Spyros Lyberis
FORTH-ICS, and University of Crete
lyberis@ics.forth.gr

Dimitrios S. Nikolopoulos
FORTH-ICS, and University of Crete
dsn@ics.forth.gr

Abstract

The currently dominant programming models to write software for multicore processors use threads that run over shared memory. However, as the core count increases, cache coherency protocols get very complex and ineffective, and maintaining a shared memory abstraction becomes expensive and impractical. Moreover, writing multithreaded programs is notoriously difficult, as the programmer needs to reason about all the possible thread interleavings and interactions, including the myriad of implicit, non-obvious, and often unpredictable thread interactions through shared memory. Overall, as processors get more cores and parallel software becomes mainstream, the shared memory model reaches its limits regarding ease of programming and efficiency.

This position paper presents two ideas aiming to solve the problem. First, we restrict the way the programmer expresses parallelism: The program is a collection of possibly recursive tasks, where each task is atomic and cannot communicate with any other task during its execution. Second, we relax the requirement for coherent shared memory: Each task defines its memory footprint, and is guaranteed to have exclusive access to that memory during its execution. Using this model, we can then define a runtime system that transparently performs the data transfers required among cores without cache coherency, and also produces a deterministic execution of the program, provably equivalent to its sequential elision.

General Terms Languages, Performance

Keywords cache coherency, programming model, task parallelism, deterministic parallelism

1. Introduction

As the silicon semiconductor technology reaches its physical limits, the ever-increasing market thirst for processing power has made multicore processors the de-facto standard. Even at modest numbers of cores per chip, the memory hierarchy becomes a bottleneck, leading to increasingly complex interconnects and cache coherency protocols. As the number of cores increases, memory bandwidth and coherency protocols throttle performance.[1] To overcome this, manufacturers relax the coherent shared memory requirements, moving towards NUMA and non-cache coherent systems.[2]

In the past, non-shared memory computers were mostly used in scientific and high-performance applications. To fully take advantage of cluster supercomputers, HPC programmers use explicit message-passing between parallel processes running on separate processors. The message-passing programming model allows the programmer to fully optimize for the given machine, at the cost of increased programming complexity. Message-passing parallel programming is tedious, difficult and costly, often restricted to scientific applications that exhibit regular parallelism, as the programmer needs to reason about load balancing, distributing data among processors, explicit communication and synchronization.

Currently, the dominant programming model for multicore processors is threads that share memory. Even though shared memory parallel programming is more intuitive and less tedious for the average programmer than message-passing, multithreaded programming is still error-prone and difficult to get right. Multithreaded programming is more intuitive because a thread resembles a sequential, deterministic program, although it might interact with other threads in a nondeterministic way by reading and writing to shared memory. In fact, the programmer needs to reason about all possible thread interleavings and interactions through memory, and insert explicit synchronization to avoid unwanted orderings. In this setting, it is easy to make programming mistakes, overlook possible interactions, and make synchronization errors. Such errors are difficult to reproduce, debug and correct, because the nondeterministic program behavior makes testing ineffective [19]: a bug might manifest itself only every few years, with catastrophic consequences [23].

Moreover, as multicore processors relax cache coherency requirements, the shared memory abstraction becomes more expensive to implement. To avoid a possible performance hit on NUMA computers or any other architecture with expensive remote memory accesses, programmers need to carefully consider locality issues, even for shared-memory multithreaded programs. Obviously, the abstraction of shared memory is not as costly in a multicore pro-

MSPC'11, June 5, 2011, San Jose, California, USA.

[1] Spec benchmarks show that memory bus contention limits the performance of the 8-core Xeon processor.

[2] The 12-core Opteron processor is NUMA, and gives better per-core scaling than Xeon. The Cell and Intel SCC processors do not have coherent shared memory among cores.

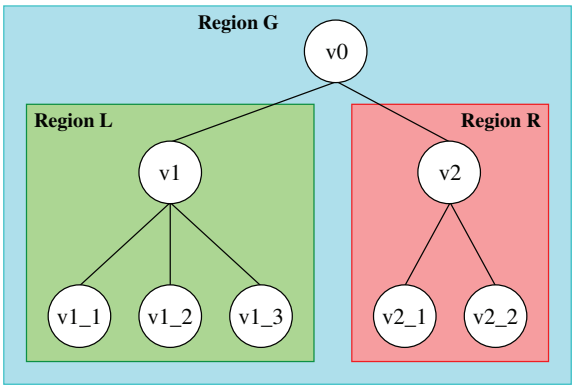

Figure 1. A tree of objects (v0 through v22), two regions (L and R) and a second-level region (G).

```
1  // top called with [inout region G]
2  top(Vertex *v0) {
3    spawn reduce(v0->left, L) [inout region L];
4    spawn reduce(v0->right, R) [inout region R];
5    spawn change(v0->right->right) [output v2_2];
6    wait on [v0->left, v0->right];
7    return v0->left->result + v0->right->result;
8  }
9
10 reduce(Vertex *v, region r) {
11   foreach c in v->children
12     spawn sum(c) [inout c];
13   wait on [region r];
14   foreach c in v->children
15     v->result += c->result;
16 }
```

Figure 2. Sample code for the programming model. Memory footprint annotations are inside square brackets.

cessor as in a cluster supercomputer. However, ignoring the memory hierarchy and inter-core communication costs can still hurt the performance of a parallel program.

Contribution

This paper describes a parallel programming model that aims to bridge the benefits of explicit memory management in message-passing with the intuitive abstractions in shared-memory multi-threaded programming, and combine them with guarantees of deterministic, predictable and testable parallel execution.

To express parallelism, we adopt a task-parallel programming model, similar to Cilk [1] and Sequoia [28], where the program is a hierarchy of recursive, nested, parallel tasks. We remove the need for explicit synchronization by the programmer, using the abstraction that each task is atomic, sequential code. Moreover, we provide a way to control the memory hierarchy, by requiring that all tasks are annotated by their memory footprint. To make this abstraction more expressive and less tedious, we combine task-footprints with region-based memory management [16, 30], so that tasks can operate on dynamic, pointer-based data structures.

Using this model, we can then develop a runtime system that transparently performs the data transfers required among cores without cache coherency. In our model, a distributed, scalable memory allocator keeps track of the memory accessed by each task. Using this data, a scheduler detects any ordering dependencies among tasks that operate on the same memory, and resolves these dependencies deterministically, preserving the sequential program order, similarly to SMPSs [21, 22]. SMP-Superscalar (SMPSs) is a task-based programming model that focuses on scientific computations on arrays and uses annotations on task arguments to dynamically detect dependencies between tasks.

We have developed a preliminary implementation of the programming model that performs comparably to Cilk++, and we have formally proven our model to always produce deterministic executions, equivalent to executions of the sequential program elision, i.e., the sequential program produced by removing all parallelism annotations.

2. Example

2.1 Programming model

Figure 1 shows an example hierarchy of objects and regions in memory. Objects v0 through v2_2 are allocated in the heap and contain pointers which link them in a tree structure. Region L consists of all objects prefixed by "v1" and region R consists of the ones prefixed by "v2". Region G includes vertex v0 and regions L and R.

Figure 2 shows an example program operating on the tree of Figure 1. The entry point is task `top` (lines 2—8), which has input/output access for the whole region G. A task that owns a region can operate on anything contained in it, or it may spawn children tasks that do so. Task `top` spawns three such tasks (lines 3—5). The first applies `reduce` on the left child of v0, reading and writing objects in region L (line 3). The second applies `reduce` on the right child, similarly operating on region R (line 4). The third task applies function `change`, not shown, on an object under the right child of v0, and its memory footprint is restricted to writing object v2_2 (line 5). The first two tasks (lines 3—4) can begin executing at once, as they operate on regions L and R, which are "owned" by the top-level task, as they are sub-regions of G. The third task (line 5) overwrites object v2_2, which is part of region R. Assuming the second task has not finished, region R is not yet owned by `top`, as ownership has passed to the second task. Consequently, the runtime will force the third task to wait.

In the meantime, task `top` continues running but needs to access objects v1 and v2, which will contain the results of the reduction operations. Our programming model requires that if top delegates regions L and R to its children, it cannot anymore access anything belonging to either of them until it "waits" for them (line 6). Task `top` will yield until both data dependencies are resolved, at which time it will resume operation, add the results and return (line 7).

The first two tasks spawned in `top` (lines 3—4) execute function `reduce` (lines 10—16). Function `reduce` takes a node v and for every object that is a child of v in the tree, it spawns a new sub-task `sum` that operate on the child node in place (lines 11—12). To collect the results, task waits for the whole region (line 13); this has the effect of a local barrier, because task `reduce` requests to regain access to the whole region, which includes all of the objects delegated to its sub-tasks. This is granted only when all the children have finished.

Even such a small example demonstrates some of the strengths of our proposed model: *(i)* It supports recursive nesting of tasks: any task can spawn children tasks, similarly to Cilk [1]. *(ii)* It can *scale hierarchically*: to process a large dataset divided into large regions, a task can spawn sub-tasks on the regions, which can in parallel spawn smaller tasks on smaller regions and so on, similarly to Sequoia [28]. *(iii)* It enables data-dependent, *deterministic* parallelism: the programmer needs not to worry about guessing all thread interleavings and resolving all possible conflicts with synchronization, because the runtime system detects and resolves

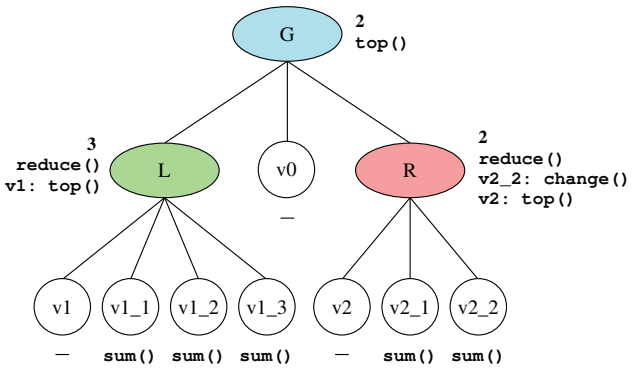

Figure 3. Hierarchical representation of the objects and regions of Figure 1, with activity counters and queues at an instant during the execution of the code in Figure 2.

dependencies, similarly to the SMPSs model [21], but without its restriction on scalability due to non-recursive parallelism.

In comparison, Cilk would not allow the `top` task to spawn `change` at line 5 and continue to line 6, because `change` depends on the previous sub-task. Cilk would thus require a `sync` statement before line 5, reducing parallelism. Similarly, Sequoia would require explicit code in `top` to make the third sub-task wait for the second, and transfer the data required.

2.2 Execution

Figure 3 shows the same hierarchical structure of regions and objects as in Figure 1, augmented with two additional properties. First, objects and regions are each associated with a *dependency queue* containing tasks that need ownership of the object or region. Second, regions (but not objects) keep a *descendant counter* recording how many of their direct children are currently busy. We show the descendant counter next to each region, followed by its dependency queue; for example, region L has 3 busy children, and task `reduce` is at the top of R's queue.

The memory snapshot shown in the figure refers to the instant where task `top` has spawned all of its children and is waiting for ownership of the left and right children of v0, namely v1 and v2. The two `reduce` tasks spawned at lines 3 and 4 have also spawned all their sub-tasks and are waiting for ownership of regions L and R, respectively. Task `change` is blocked waiting for ownership of object v2_2. Each of the six objects v1_1, v1_2, v1_3, v2_1 and v2_2 is owned by a task running `sum`. These six tasks are running in parallel.

At the shown instant, the descendant counter for region L is 3, since objects v1_1, v1_2 and v1_3 are busy, owned by three `sum` tasks. The task queue contains the `reduce` task spawned at line 3 at the front, although it is blocked and cannot take ownership of L until its descendant counter is 0.[3] Next in the task queue for L is task `top`, which has issued a wait statement; note that `top` waits for ownership of v1 but it is in the L queue. This happens because of the semantics of `wait` that requires the waiting task to start at a region it owns and traverse the hierarchy downwards, towards the wanted object. In this case, to get the ownership of v1, `top` starts at the region it already owns when it reaches the `wait` statement, namely G, and traverses the hierarchy down to the requested node v1. Since `top` is first at the queue of G, it can immediately go down to L, where it enters the end of the task queue. We use the notation `v1: top()` to show that `top` is in the L queue to get to v1. At this instant, the path from G to v1 is blocked by the `reduce` task, which is next in line to operate on the whole of L, including v1. Thus, `top` cannot go further down and get ownership of v1 until `reduce` is finished and `top` becomes first in the queue of L.

Similarly, region R has a descendant counter of 2, and a queue containing the `reduce` task of line 4 at the front. The `change` task spawned at line 5 is enqueued after the `reduce` task, waiting to go down to object v2_2. Task `top` waiting for v2 is last in the queue, because it is enqueued by the `wait` at line 6, after `change`.

The abstraction of memory as a tree of regions and objects by the programming model enables the implementation of a very efficient and scalable dependency resolution mechanism. It also allows for parallel and scalable memory allocation and task scheduling algorithms, as the task queues and memory metadata follow a hierarchical structure. Moreover, execution is guaranteed to be race-free, deterministic, and to always yield the same answer as the sequential program. Finally, the scheduler can use the task and memory metadata to optimize for locality, so that tasks operating on the same objects or regions in the tree are scheduled to the same cores.

A requirement of the programming model presented above is object-granularity. Specifically, we have restricted the input and output arguments of tasks to either be whole objects returned by a `malloc()` call or regions. We further assume that the `malloc()` call is part of the runtime system, which builds the hierarchical tree representation including the scheduling metadata. Task spawn and wait statements are then directly mapped to traversals of this tree, one such traversal per task argument. This requirement for object granularity appears more fit towards type-safe and memory-safe languages, although our current, preliminary implementations are C libraries. Note, however, that this programming model targets multicore processors without cache coherency and transparent shared memory. In such machines, the programmer would still need to manually specify data transfers between core memories. In this programming model, instead, data transfer is transparent, at the cost of requiring the program to respect object boundaries in memory, and use only whole objects as task arguments.

3. Determinism proof

We have formalized a simple version of the programming model and proved that it produces deterministic parallel executions, equivalent to the sequential execution of the program. We have omitted support for region-based memory management in this presentation, as it increases the complexity and size of the system, without changing the intuition or the proof technique. We present a summary of the simplified formalism and a proof sketch below.

Figure 4 presents λ^{TASK}, a simple task-parallel programming language. λ^{TASK} is a simply-typed lambda calculus extended with dynamic memory allocation and updatable references, task creation and synchronization. Values include integer constants n, the unit value `()`, functions $\lambda x\,.\,e$ and pointers ℓ. Program expressions include variables x, function application $e_1\ e_2$, memory operations and task operations. Specifically, expression `ref` e allocates some memory, initializes it with the result of evaluating e, and returns a pointer ℓ to that memory; expression $e_1 := e_2$ evaluates e_1 to a pointer and updates the pointed memory using the value of e_2; and expression $!\,e$ evaluates e to a pointer and returns the value in that memory location.[4] Expression $\texttt{task}(e_1, \ldots, e_n)\,\{e\}$ evaluates each e_i to a pointer and then evaluates the task body e, possibly in parallel. The task body e must always return `()` and can only access (via dereference or assignment) the given pointers; if e is evaluated in a parallel task, the expression immediately returns `()`.

[3] Note that earlier in time, `reduce` was in the same position in the queue but was running. When it issued the wait statement, it yielded, waiting for its three children to finish.

[4] We borrow the syntax for dereference and assignment from ML rather than C (not e.g., $*e$), as λ^{TASK} is a simple functional language.

$$
\begin{array}{lrcl}
\text{Locations} & \ell & \in & \mathcal{L} \\
\text{Values} & v \in \mathcal{V} & ::= & n \mid () \mid \ell \mid \lambda\, x\, .\, e \\
\text{Expr.} & e & ::= & x \mid e\ e \mid \texttt{ref}\ e \mid e := e \mid !\, e \\
 & & \mid & \texttt{task}(\vec{e})\,\{e\} \mid \texttt{waiton}\ e \\
\text{Types} & \tau & ::= & \mathit{int} \mid \mathit{unit} \mid \tau \to \tau \mid \tau\ \mathit{ref} \\
\text{Contexts} & E & ::= & [\cdot] \mid E\ e \mid v\ E \mid \texttt{ref}\ E \\
 & & \mid & E := e \mid v := E \mid !\, E \\
 & & \mid & \mid \texttt{task}(\vec{v}, E, \vec{e})\,\{e\} \mid \texttt{waiton}\ E \\
\text{Task ids} & t & \in & \mathcal{T} \\
\text{Task maps} & T & ::= & \emptyset \mid \left(t, \texttt{task}(\vec{\ell})\,\{e\}^{t}\right), T \\
\text{Task queues} & q \in \mathcal{Q} & ::= & t, \ldots, t \\
\text{Dep. maps} & D & :: & \mathcal{L} \to \mathcal{T} \\
\text{Stores} & S & :: & \mathcal{L} \to \mathcal{V} \\
\text{Running tasks} & R & ::= & t \| \ldots \| t
\end{array}
$$

Figure 4. λ^{TASK}: A simple task-based parallel language

$$
\text{[E-SEQ]}\ \frac{\begin{array}{c} t \in R \qquad T(t) = \texttt{task}(\vec{\ell})\,\{e\}^{t'} \\ \vec{\ell'} = \left\{\ell \in \vec{\ell} \mid D(\ell) = t, q\right\} \\ \left\langle S \downarrow \vec{\ell'}, e\right\rangle \to_s \langle S', e'\rangle \qquad \mathit{dom}\,(S') = \vec{\ell'} \\ T' = T[t \mapsto \texttt{task}(\vec{\ell})\,\{e'\}^{t'}] \\ S'' = \left(S \setminus \vec{\ell'}\right) \cup S' \end{array}}{\langle T, D, S, R\rangle \to_p \langle T', D, S'', R\rangle}
$$

$$
\text{[E-TASK]}\ \frac{\begin{array}{c} t \in R \qquad T(t) = \texttt{task}(\vec{\ell})\left\{E[\texttt{task}(\vec{\ell'})\,\{e\}]\right\}^{t''} \\ \vec{\ell'} \subseteq \vec{\ell} \qquad\qquad t' - \mathit{fresh} \\ T' = T[t \mapsto \texttt{task}(\vec{\ell})\,\{E[()]\}^{t''}][t' \mapsto \texttt{task}(\vec{\ell'})\,\{e\}^{t}] \\ \forall \ell \in \vec{\ell'},\ D(\ell) = q_1, t, q_2 \wedge D'(\ell) = q_1, t', t, q_2 \\ \forall \ell \in \mathit{dom}\,(D) \setminus \vec{\ell'},\ D'(\ell) = D(\ell) \end{array}}{\langle T, D, S, R\rangle \to_p \langle T', D', S, R\rangle}
$$

$$
\text{[E-START]}\ \frac{\begin{array}{c} t \notin R \qquad T(t) = \texttt{task}(\vec{\ell})\,\{e\}^{t'} \\ \forall \ell \in \vec{\ell},\ D(\ell) = t, q \end{array}}{\langle T, D, S, R\rangle \to_p \langle T, D, S, R \| t\rangle}
$$

$$
\text{[E-JOIN]}\ \frac{\begin{array}{c} T(t) = \texttt{task}(\vec{\ell})\,\{()\}^{t'} \\ T(t') = \texttt{task}(\vec{\ell'})\,\{e'\}^{t''} \\ T' = (T \setminus \{t\})[t' \mapsto \texttt{task}(\vec{\ell'} \cup \vec{\ell})\,\{e'\}^{t''}][t'/t] \\ \forall \ell \in \vec{\ell},\ D(\ell) = q_1, t, q_2 \wedge D'(\ell) = q_1, q_2 \\ \forall \ell \in \mathit{dom}\,(D) \setminus \vec{\ell},\ D'(\ell) = D(\ell) \end{array}}{\langle T, D, S, R_1 \| t \| R_2\rangle \to_p \langle T', D', S, R_1 \| R_2\rangle}
$$

Figure 5. Example parallel semantics rules

Finally, expression `waiton` e evaluates e to a pointer and blocks the execution until no child task has that memory.

Figure 5 presents four indicative transition rules of the small-step operational semantics for the parallel execution of λ^{TASK} programs. Small-step judgments have the form

$$\langle T, D, S, R\rangle \to_p \left\langle T', D', S', R'\right\rangle$$

where T is a map from task identifiers t to the task definitions $\texttt{task}(\vec{\ell})\,\{e\}$, D is a map from every memory location ℓ to a queue q of task identifiers, S is a map of memory locations ℓ to the values v they contain, and R is a set of the identifiers of all the tasks currently running in parallel as defined in Figure 4.

Rule [E-SEQ] allows the parallel execution to revert into serial execution for any expression in the program. The first premise selects any task identifier t from the running tasks. The second premise looks up the body $\texttt{task}(\vec{\ell})\,\{e\}$ of the running task in the task map T. The third premise selects a subset $\vec{\ell'}$ of the task memory $\vec{\ell}$, containing all the locations ℓ that are (still) owned by task t, i.e. t is the first task in the dependency queue of ℓ. The fourth premise evaluates the task body e sequentially to e' using the subset $\vec{\ell'}$ of memory, creating a new store S'. The definition of sequential evaluation $\to_s$ is standard. The fifth premise requires the resulting store S' to contain the same locations as S, so as to forbid memory allocation in the sequential code. We use this premise to restrict parallel evaluation to only allocate memory through the parallel version of the allocator. The next premise creates the new task map to reflect that task t has taken a step to e'. Finally, the last premise creates the new global store S'' by integrating the changes made in the task memory S' with the original store S.

Rule [E-TASK] evaluates the expression $\texttt{task}(\ell)\,\{e\}$ by creating a new task and inserting it in the dependency queues. The first premise selects a parent-task t from the list R of running tasks whose next instruction is $\texttt{task}(\vec{\ell'})\,\{e\}$, as stated in the second premise. Here, $\vec{\ell}$ is the memory footprint of the parent-task, and $\vec{\ell'}$ is the memory footprint of the new task, which needs to be a subset of $\vec{\ell}$ as stated in the third premise. The fourth premise creates a new, "fresh" task identifier t' for the new task. The fifth premise binds the new task body to t' in T and also takes a step in t, evaluating the expression $\texttt{task}(\vec{\ell'})\,\{e\}$ to $()$, as in the sequential execution. Finally, the last two premises create a new dependency map D' from D by inserting the new task t' immediately before t in the queues of all the locations in $\vec{\ell'}$.

Rule [E-START] evaluates the starting of a ready task, which can happen any time its dependencies are satisfied. So, the starting state $\langle T, D, S, R\rangle$ only differs from the ending state $\langle T, D, S, R\|t\rangle$ in that task t is now running in parallel with the other running tasks R. The first premise requires that the task t that is to be started is not already running. The second premise looks up the body $\texttt{task}(\vec{\ell})\,\{e\}$ of task t. The third premise checks whether all the memory locations in $\vec{\ell}$ are owned by t, i.e., t is the first task identifier in the dependency queues of all locations in $\vec{\ell}$.

Rule [E-JOIN] defines the removal of a finished task. The first premise checks that task t is finished, i.e., it has been evaluated to $()$. The second premise looks up the parent task t' of t. The third premise removes t from the task map, then adds the footprint of t to the footprint of t'. If t had allocated no new memory locations, then $\vec{\ell} \subseteq \vec{\ell'}$ and this union makes no difference. We need the third premise to enforce any memory locations allocated by t to now be in the footprint of t'. The last two premises construct a new dependency map D' by removing the finished task t from all the queues of its footprint $\vec{\ell}$. Finally, in the resulting state we remove task t from the, otherwise unaffected, list of running tasks.

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

THEOREM 3.1. *Sequential equivalence If*

$$\langle \{t_0, \texttt{task}(\ell)\,\{e\}\}, \emptyset, \emptyset, t_0\rangle \to_p^* \langle \{t_0, \texttt{task}(\ell)\,\{v\}\}, D, S, t_0\rangle$$

then $\langle \emptyset, e\rangle \to_s^* \langle S, v\rangle$

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

parallel execution trace to a parallel execution trace whose first step satisfies the sequential order.

4. Related Work

Parallel programming models There are several programming models and languages for writing parallel programs. The dominant parallel programing models for shared memory systems use threads. Multithreaded programming is hard and error prone, as the programmer needs to reason about all implicit thread interleavings and interactions through memory.

OpenMP [7] is oriented towards parallelization of sequential code, using compiler directives to express shared memory parallelism for loops and tasks, implemented usually in a runtime system that hides thread management from the programmer. The programmer is still responsible to avoid races and insert all necessary synchronization.

Intel's Threading Building Blocks (TBB) [24] is a library of Object-Oriented design patterns that can be used to parallelize sequential C++ programs for shared-memory systems. TBB does not use compiler directives, instead providing a library of thread-safe containers and schedulers, and a set of Object-Oriented patterns that hide some synchronization from the programmer and can be used to implement data and pipeline parallelism. Again, the programmer is responsible for inserting any necessary synchronization to avoid races over shared data and to enforce any ordering dependencies among parallel tasks.

Cilk/Cilk++ [1] is a parallel programming language that extends C++ with recursive task-based parallelism for shared memory systems. Cilk expresses parallelism using a *spawn* statement to state that a function invocation can be computed in parallel, and synchronization using the *sync* statement to state that a parent task should wait until all its spawned children have finished. Cilk tasks can be very fine-grained without incurring much overhead, because Cilk only executes *spawn*s in parallel if necessary, using a work-stealing scheduler. Again, the programmer is responsible to use *sync* or any other synchronization mechanism to avoid data races and enforce specific task orderings. Cilk uses a dynamic analysis to detect nondeterminism [13].

Sequoia [11, 28] is a parallel C++-like programming language that can target both shared memory and distributed systems. In Sequoia, the programmer writes a hierarchy of nested parallel tasks, where leafs are atomic and perform simple computations, and inner tasks break down the computation into smaller sub-tasks, and combine their results; a machine description that specifies the various levels in the memory hierarchy, whether memory is shared, etc.; and a mapping file that describes how data is broken and distributed among tasks and their sub-tasks, which tasks are scheduled to run over which level of the memory hierarchy, and when computation workload should be broken into smaller tasks.

StreamIt [14] is a domain-specific language for streaming computations, oriented towards pipeline parallelism. Although not a general-purpose parallel programming language, StreamIt follows a dataflow model where the program is a set of parallel *filters* that form a pipeline.

The Partitioned Global Address Spaces (PGAS) programming model attempts to bridge the gap between MPI explicit communication and shared-memory models. Early approaches include languages like UPC and HPF that use static distribution of global data among processors. Later languages include Chapel [3, 4] and X10 [5, 27], which are not limited to static data distributions, but instead provide mechanisms to manage the *place* or *locale* of data elements, and support NUMA for accessing remote data. Recently, SpiceC [12] uses transactional memory techniques to implement a programming model similar to OpenMP for multicores without cache coherency, where the compiler detects and generates necessary code for data transfers among processor memories.

Dependencies and deterministic parallelism Several programming models and languages aim to automatically infer synchronization among parallel sections of code. Transactional Memory [17] preserves the atomicity of parallel tasks, or transactions, by detecting and retrying any conflicting code. Static lock allocation [6, 18] provides the same serializability guarantees by automatically inferring locks for atomic sections of code. These attempts, however, allow nondeterministic parallel executions, as they only enforce serializability, not ordering constraints among parallel tasks.

StreamIt [14] implements a dataflow model of computation where the program includes static dependencies among the various stages of the pipeline. Since all dependencies among parallel code sections are explicit, the compiler generates all synchronization, data transfer and copying necessary to guarantee deterministic execution that preserves the ordering among parallel tasks.

SMP-Superscalar (SMPSs) [21, 22] is a task-based programming model for scientific computations that uses annotations on task arguments to dynamically detect argument dependencies between tasks. SMPSs does not support nested parallelism, instead, it uses a single master thread to invoke tasks to be executed by a set of worker threads. Each task invocation includes the task memory footprint, used to detect dependencies among tasks and order their execution according to program order. SMPSs has shown dynamic dependency analysis to improve performance in cases of irregular dependencies and unbalanced task workloads. However, it focuses on array arguments, and does not support dynamic data structures.

Recent research has developed methods for the deterministic execution of shared memory, multithreaded programs. Kendo [20] enforces a deterministic order in acquiring locks, using performance counters to decide which thread should next get a lock. This technique produces deterministic executions for programs without races, although it does not guarantee determinism in the presence of races. DMP [8, 9] proposes a combination of hardware ownership tracking and transactional memory that guarantees deterministic execution by enforcing an ordering over thread interactions through shared memory. Both systems produce deterministic executions, although they are not always intuitive to the programmer, as they are not equivalent to a sequential program. Instead, they guarantee the appearance of the same thread interleaving across executions.

Deterministic Parallel Java [2] uses a static type and effect system to enforce memory isolation among tasks, and uses regions to reason about memory effects and infer task memory footprints. However, parallelism is restricted inside `cobegin` statements that implicitly wait for all sub-tasks before returning to the parent task, similarly to Cilk. Moreover, although the implicit inference of task effects reduces the overhead of programmer annotations, it runs the risk of large over-approximation. So, DPJ targets cache coherent, shared memory systems, where a gross overapproximation of task effects only reduces parallelism, without causing communication bottlenecks.

Memory management Programming languages and libraries for parallel systems with explicitly managed memory hierarchies, such as Sequoia and SMPSs use explicit annotations to identify the memory footprint of a piece of code, to help the language runtime make better use of the memory hierarchy. Sandhu et al. [26] use high-level language annotations to define array regions and use these annotations to optimize their implementation of coherent software caches.

Region-based memory management [16, 30] uses growable memory pools or regions, with either static or dynamic lifetimes, to

allocate and deallocate objects effectively, based on their lifetime. Languages like Real-time Java [25], Cyclone [15, 29], and Sequoia use manual, region-based memory management to avoid garbage collection or to explicitly manage the memory hierarchy.

5. Conclusions

This position paper presents two ideas aiming to build a scalable, deterministic programming model for multicore computers without coherent shared memory. First, we adopt a program model of recursively nested, parallel, atomic tasks. Tasks cannot communicate with other tasks, apart from spawning a child task and waiting for ownership of an object or memory region. Second, we require the programmer to annotate each task with its memory footprint. The runtime system and scheduler, then, can detect and resolve dependencies, and transparently perform any data transfers necessary to maintain a shared memory abstraction without hardware cache coherency. To facilitate programming in this model, and support dynamic data structures, we use region-based memory management. We briefly explain a scheduling algorithm that discovers and resolves dependencies, guaranteeing a deterministic execution of the program, provably equivalent to its sequential elision.

Acknowledgments

This work was supported in part by the ENCORE project [10].

Hans Vandierendonck is a Postdoctoral Fellow of the Research Foundation – Flanders (FWO). He is supported by a Travel Grant of the FWO.

References

[1] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. In *PPoPP '95: Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[2] BOCCHINO, JR., R. L., HEUMANN, S., HONARMAND, N., ADVE, S. V., ADVE, V. S., WELC, A., AND SHPEISMAN, T. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL '11: 38th annual ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages*.

[3] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications* (August 2007).

[4] Chapel Language Specification 0.796. `http://chapel.cray.com/spec/spec-0.796.pdf`, Oct. 2010.

[5] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: 20th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*.

[6] CHEREM, S., CHILIMBI, T., AND GULWANI, S. Inferring locks for atomic sections. In *PLDI '08: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[7] DAGUM, L., AND MENON, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering* (January 1998).

[8] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. Dmp: deterministic shared memory multiprocessing. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*.

[9] DEVIETTI, J., NELSON, J., BERGAN, T., CEZE, L., AND GROSSMAN, D. RCDC: A relaxed consistency deterministic computer. In *ASPLOS '11: Proceedings of the sixteenth international conference on Architectural Support for Programming Languages and Operating Systems*.

[10] ENCORE: ENabling technologies for a programmable many-CORE. `http://www.encore-project.eu`.

[11] FATAHALIAN, K., HORN, D. R., KNIGHT, T. J., LEEM, L., HOUSTON, M., PARK, J. Y., EREZ, M., REN, M., AIKEN, A., DALLY, W. J., AND HANRAHAN, P. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the ACM/IEEE conference on Supercomputing*.

[12] FENG, M., GUPTA, R., AND HU, Y. Spicec: scalable parallelism via implicit copying and explicit commit. In *PPoPP '11: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*.

[13] FENG, M., AND LEISERSON, C. E. Efficient detection of determinacy races in cilk programs. In *SPAA '97: Proceedings of the ninth annual ACM Symposium on Parallel Algorithms and Architectures*.

[14] GORDON, M. I., THIES, W., KARCZMAREK, M., LIN, J., MELI, A. S., LAMB, A. A., LEGER, C., WONG, J., HOFFMANN, H., MAZE, D., AND AMARASINGHE, S. A stream compiler for communication-exposed architectures. In *ASPLOS '02: Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*.

[15] GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. Region-based memory management in Cyclone. In *PLDI '02: ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[16] HANSON, D. R. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience* (January 1990).

[17] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: 20th annual International Symposium on Computer Architecture*.

[18] HICKS, M., FOSTER, J. S., AND PRATIKAKIS, P. Lock inference for atomic sections. In *TRANSACT '06: First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing*.

[19] LEE, E. A. The problem with threads. *Computer 39*, 5 (2006), 33–42.

[20] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*.

[21] PEREZ, J., BADIA, R., AND LABARTA, J. A dependency-aware task-based programming environment for multi-core architectures. In *ICCC '08: Proceedings of the IEEE International Conference on Cluster Computing*.

[22] PEREZ, J. P., BELLENS, P., BADIA, R. M., AND LABARTA, J. CellSs: making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development* (September 2007), 593–604.

[23] POULSEN, K. Tracking the blackout bug. `http://www.securityfocus.com/news/8412`, 2004.

[24] REINDERS, J. *Intel Threading Building Blocks*, first ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[25] The Real-Time Specification for Java. `http://http://www.rtsj.org`.

[26] SANDHU, H. S., GAMSA, B., AND ZHOU, S. The shared regions approach to software cache coherence on multiprocessors. In *PPoPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*.

[27] SARASWAT, V., BLOOM, B., PESHANSKY, I., TARDIEU, O., AND GROVE, D. X10 Language Specification v2.1. `http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf`, Feb. 2011.

[28] The Sequoia programming language. `http://http://sequoia.stanford.edu`.

[29] SWAMY, N., HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. Safe manual memory management in Cyclone. *Science of Computer Programming* (October 2006), 122–144.

[30] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. *Information and Computation* (1997), 109–176.