Parallel Computing 40 (2014) 90–106

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Scheduling directives: Accelerating shared-memory many-core processor execution

Oded Green [a,b,*], Yitzhak Birk [a]

[a] Electrical Engineering Dept., Technion, Haifa, Israel
[b] College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

ARTICLE INFO

Article history:
Available online 14 December 2013

Keywords:
Scheduling and task partitioning
Shared memory
Parallel processing
Fine grain parallelism
Data dependencies

ABSTRACT

We consider many-core processors with a task-graph oriented programming model, whereby scheduling constraints among tasks are decided offline, and are then enforced by the runtime system using dedicated hardware. Here, exposing and beneficially exploiting fine grain data and control parallelism is increasingly important. Therefore, high expressive power for stating such constraints/directives, along with the ability to implement them in fast, simple hardware, is critical for success. In this paper, we focus on the relationship among different duplicable (multi-instance) tasks, which are used to express and exploit data parallelism. We extend the conventional Start-After-Complete (precedence) constraint to also be usable between replicas of different such tasks rather than only between entire tasks, thereby increasing the exposable parallelism. Additionally, we propose the parameterized Start-After-Start constraint, which can be used to control the degree of “lockstep” among multiple such tasks, e.g., in order to improve cache performance when the tasks work on the same data. Also, we briefly describe several additional interesting directives. Finally, we show that the directives can be supported efficiently in hardware. Hypercore, a very efficient CREW PRAM-like shared-cache architecture, which is very challenging because it has extremely fast dispatching for basic constraints, is used in the discussion. However, the new directives have broader applicability. Having shown the possibility of simple implementation and indications of benefit, this motivates further exploration of these directives and their implementation in hardware, as well as their support by programming tools.

1. Introduction

1.1. Overview

We consider programs that comprise a set of serial tasks along with a set of scheduling relations among them. These may represent data dependences in order to ensure correct execution, or aim to govern the scheduling for other reasons such as efficient resource utilization. This programming model is sometimes referred to as *task-oriented programming*. It is essentially a coarse grain (task granularity) dataflow-machine model.

The partitioning of a program into tasks aims to expose parallelism and enable the exploitation of many compute cores. When considering two tasks, one of the following holds:

* Corresponding author at: College of Computing, Georgia Institute of Technology, Atlanta, GA, USA..
E-mail address: ogreen@gatech.edu (O. Green).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.12.001

1. The two tasks carry out the same operations but on different data (data parallelism). E.g., summing up different rows of a matrix.
2. The two tasks perform different operations on the same data (program parallelism). E.g., searching for different virus signatures in the same data.
3. The two tasks perform different operations on different data (unrelated tasks).

We consider a shared-memory (no private caches) many-core architecture, essentially a Concurrent Read, Exclusive Write (CREW) PRAM architecture [1]. A program comprises a set of serial tasks along with a set of precedence relations among them, which represent data dependences and ensure correct execution.

For reasons such as programming convenience and reduced code footprint, multiple-instance ("duplicable") tasks are used in data-parallel situations such as summing up the rows of a matrix. Tasks are dispatched to cores by hardware within very few clock cycles and at a very high rate. This is thus a dataflow machine at the inter-task level, with conventional control flow within each task. The Plurality Hypercore [2,3] is such an architecture.

The precedence constraints guarantee correctness, and the absence of private caches obviates the need to consider which core should execute any given task. (This is not the case for platforms with private caches such the x86 and GPUs.) However, one must still decide the dispatching order whenever the number of runnable tasks exceeds that of available cores. This choice among correct execution orders can impact performance: (1) it can mitigate bottlenecks, namely situations wherein a task that must precede many others is scheduled later than it could have been and subsequently causes cores to be idle awaiting its completion, and (2) it can impact the instantaneous memory footprint of the program and its data, thereby affecting the hit rate of the shared cache.

For a given number of cores and a specific program with known task execution times, one could simply add precedence relations in order to enforce the desired scheduling order. This, however, is more difficult in the general case wherein runtime parameters such as execution time are data dependent, and can actually degrade performance when running on a different number of cores. The problem is most acute with duplicable tasks, as the "basic" precedence constraints apply jointly to all task instances.

For synchronization purposes, each duplicable task has an entry point (fork) and an exit point (join), as depicted in Fig. 1. The entry point states that all replicas may be dispatched; this can be thought of as a fork. The exit point refers to the fact that the duplicable task is considered complete only after its replicas have all been completed; this can be thought of as a join.

Using duplicable tasks has several advantages: task graphs are simple to create, changing the number of replicas is simple (allowing portability), and task dispatching is efficient. However, there are also limiting factors when using duplicable tasks instead of regular tasks and applying scheduling constraints jointly to all the replicas of a duplicable task. These include a loss of expressive power, out of order completion and reduced portability whenever optimizations have been made.

This work focuses on scheduling constructs ("directives") that can be used by programmers and by automatic optimization tools to further (beyond mere correctness) direct the runtime dispatcher, with special attention to duplicable tasks. Such constructs must lend themselves to efficient implementation in hardware. We present several such directives, along with small illustrative examples in which they increase performance. We also outline their implementation in the context of a Hypercore-like system, thereby proving them to be practical. This serves to motivate further study of these directives as well as their incorporation into future architectures and programming environments.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents Hypercore in some detail. In Section 4 we consider regular tasks, and present Start-After Start (SAS), a new type of scheduling directive that offers a level of priority that lies between precedence and priority. In Section 5 we present our main contribution, new scheduling directives for duplicable tasks, along with insights and small examples. In Section 6 we present a simple hardware mechanism that supports the new scheduling directives. Finally, Section 7 offers concluding remarks.

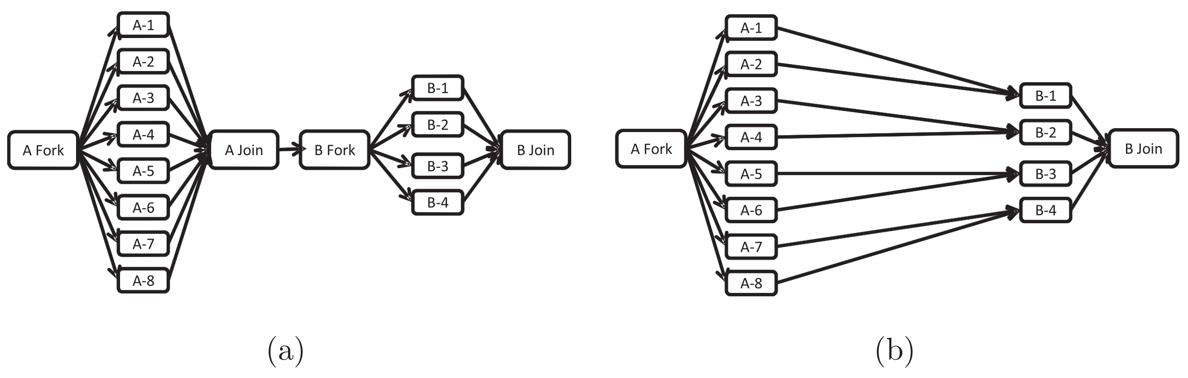

Fig. 1. Duplicable tasks *A* (8 replicas) and *B* (4 replicas): (a) *A* precedes *B*; (b) precedence constraints among replicas of the two tasks.

2. Related work

2.1. Additional many-core platforms

In this work we chose Plurality's Hypercore as our system of interest because of its unique memory hierarchy, which essentially implements the classical PRAM architecture (the concurrent read, exclusive write, or CREW variant); this is different than the one present in the following systems:

1. UMDs XMT (Explicit Multi-Threading) architecture [4,5] supports fast dispatching of duplicable tasks to the many-core system. Each of the XMT's cores, however, has a private cache, so cache coherency protocols are required.
2. The Cray XMT [6] system (unrelated to UMD's XMT) is a highly parallel system with thousands of lightweight threads and Uniform Memory Access (UMA) shared-memory hierarchy. The cores do not have any private caches nor do they have shared caches. Context switching every cycle is used for latency hiding.
3. NVIDIAs CUDA platform [7] platform is a platform for general purpose computing on the GPU. It is commonly referred to as GPGPU, and includes different architectures such as the Tesla, Fermi, and Kepler. These systems consist of multiprocessors (MP) such that each MP has a private cache shared only among its many streaming processors (SP). The MPs also have a second level memory that they share. Similar to the Cray XMT, fast context switching is used for latency hiding.
4. Symmetric Multi-Processors (SMP). These include systems such as the x86 and Tilera [8] that have processors capable of running the Linux operating system. While these systems are different in frequency and power consumption, both have private caches per core and are Non Unifrom Memory Access (NUMA) systems. Thread creation on the x86 can take hundreds to thousands of clock cycles.

These systems either have a private caches for the processors or no cache at all. As such, this memory hierarchy is different than the shared-memory hierarchy of the Hypercore, in which the cache is shared among all cores. Our directives may be applicable to some of these systems, yet it is likely that the directives will require modifications.

2.2. Related works on scheduling

Our work presents scheduling directives, not scheduling algorithms or mechanisms. Nonetheless, in view of its clear relationship to scheduling, we next mention several works on scheduling that motivate the need for directives or are somehow related to them.

In [9], it is shown that scheduling anomalies can occur even when there is merely a slight change to the program and system parameters. In [10], it is shown that even some of the simplest scheduling problems are NP -complete. The time complexity of a scheduling scheme is based on multiple parameters and constraints; these include the optimality criteria, the system parameters, and an ability to perform context switches. These characteristics can be reflected by a 3-field problem classification $\alpha|\beta|\gamma$ [11]. In [12,13], many different scheduling schemes based on the above classification can be found. A clear distinction exists between offline and online scheduling schemes. Offline scheduling schemes may have knowledge regarding the tasks that make up the applications. As such, they can make wiser scheduling choices, as they are able to go over the full or partial search space. In contrast, online schedulers have to make the scheduling decisions quickly, as otherwise the system utilization can be significantly reduced, and must therefore be relatively simple, but see data-dependent state.

Applications can be represented using Directed Acyclic Graphs (DAG) in which the vertices represent the tasks and the edges represent the task interdependencies. The weight of a vertex represents task execution time. Additional information may include the amount of data that must be passed from one task to another, and communication cost can be minimized by assigning heavily communicating tasks to the same processor. (In a shared-cache architecture like the one considered in this paper, wherein all the cores are identical and there are no private caches, the choice of processor is not an issue, but data sharing among tasks may affect scheduling because it affects the cache hit rate.)

The critical path of the DAG is defined as the longest path between any entry vertex (a vertex that does not depend on any other vertex) and any exit vertex (a vertex on which no other vertex depends). Two notable papers that deal with critical path scheduling are [14], which assumes a given fixed number of processors, and [15] that permits addition of processors.

The use of an "OR" precedence constraint that allows for dispatching tasks only when some subset of their precedence constraints have been met is discussed in [16].

One can consider the use of redundancy, whereby tasks are executed more than once for the sake of correctness. If a task fails then all the dependent tasks cannot be dispatched. We do not present additional scheduling schemes that use this type of constraint, yet we note that this is a scheduling directive. Various parallel platforms have been designed to simplify parallel programming: OpenMP [17], Intel Threading Building Blocks (TBB) [18], and Intel Concurrent Collections (CnC) [19,20]. Open applications can in fact be stated in the format of a task graph. TBB offers additional parallel primitives, data structures, and algorithms with system portability and support of work stealing. CnC permits an additional abstraction that separates the control flow of an application from its data flow.

3. The Hypercore architecture

Hypercore, developed by Plurality [2,3] based on initial work from [21–23], offers a shared-memory many-core architecture wherein the on-chip cache is fully shared among the cores, but is partitioned into numerous banks; a low-latency, high bandwidth combinational multistage interconnect carries the core-cache traffic. Address interleaving is used for load balancing and collision mitigation. Same-address writes, as well as same-bank accesses (to different addresses within the bank) are serialized by the interconnect, and simultaneous same-address reads are served efficiently by multicast. The memory banks are equidistant from all the cores, so this is a UMA system, (Fig. 2). The absence of private caches (and a large amount of state in them) and the UMA architecture permit any core to execute any compute task with equal efficiency. The shared cache has a number of memory banks that is larger than the number of cores in the system. This, along with the use of interleaving in the address-to-bank mapping and the fact that not all instructions access data memory, results in a low probability of "collision". Hypercore thus features the simplicity of truly shared memory (no inter-core cache coherence issues and performance indifference to the core to which a given task is assigned), while avoiding the would-be core-to-cache communication bottleneck of a shared bus. This greatly simplifies programming and runtime management. Typical shared-cache size is several Megabytes. The memory hierarchy also includes off-chip (shared) memory.

The programming model of Hypercore is a set of serial tasks along with precedence relations among them. These precedence constraints can be represented via a task graph. Any task whose prerequisites have been met is runnable. (It is the programmer's responsibility to provide all the precedence constraints that are required for ensuring correctness.) The use of duplicable tasks (same code operating on different data) simplifies the creation of task graphs, as only a single vertex in the graph needs to be created for a specific operation regardless of the number of replicas needed. Duplicable tasks have additional benefits: reducing the size of the task graph, increasing the portability of the task to additional systems, facilitating the changing of problem size, and simplifying the online scheduler as there are fewer different tasks and thus fewer decisions to be made.

Hypercore can be viewed as the "dual" of a single-core processor with out-of-order execution: the latter is a control flow machine when viewed from afar, and a dataflow machine when one zooms in on the instructions currently in the CPU; Hypercore, in contrast, is a coarse-grain (task granularity) dataflow machine and a fine-grain control-flow machine (each core is typically an in-order pipelined processor).

Plurality has implemented an online hardware scheduler called the "Synchronizer/Scheduler". It receives the task graph along with pointers to the start address of every task, tracks the completion of every task, and dispatches a runnable task as soon as a core becomes available. (We will simply refer to this unit as the dispatcher or the scheduler.)

To enable fast scheduling and dispatching, Plurality created a distribution network between the dispatcher and the cores (Fig. 2). From the moment the dispatcher dispatches a task until the task reaches an idle core, it takes $O(log(cores))$ cycles, which is usually not more than a handful of cycles. The fast task dispatching permits the beneficial exploitation of fine grain

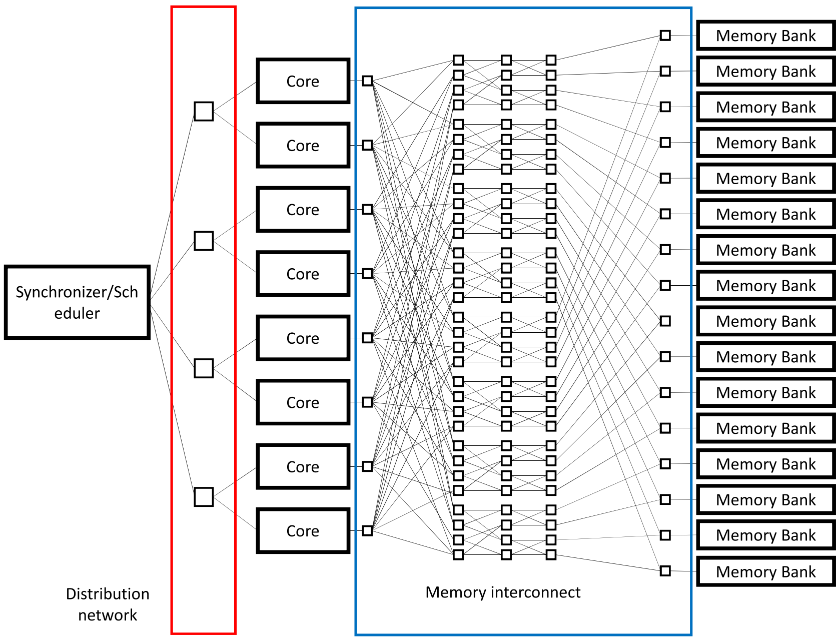

Fig. 2. Schematic view of Hypercore. Of importance is the distribution network that connects the scheduler to the many cores and the UMA memory interconnect. The symmetry of these networks allows for each core to execute any task or replica (for duplicable tasks) with equal efficiency.

parallelism. The dispatch network is a tree rooted at the dispatcher with the cores as the leaves. The dispatcher node's fanout is implementation dependent. Each of the nodes in the scheduler's distribution network can complete its mission in one cycle, sending the dispatch request onward. Hypercore supports two types of dispatching: (1) dispatching a single task on each sub-tree in the distribution network; this limits the number of dispatched tasks per cycle to the number of sub-trees; (2) dispatching a duplicable task with multiple copies on each sub-tree; here, the number of dispatched replicas is limited only by the total number of cores in that sub-tree.

The dispatcher, implemented in hardware, is very fast (in terms of both latency and throughput). The fast dispatcher enables beneficial exploitation of fine-grain parallelism. Together with the UMA, high-bandwidth shared-cache, this yields a very effective, agile and easy to program architecture.

It is Plurality's goal to make this system a low power system. While exact numbers cannot be given as this platform has not yet been fully synthesized, the numbers suggest ~ 4 Watts for 64 OpenSPARC cores at 500 MHz with 40 nm CMOS technology.

Hypercore is extremely attractive from power-performance, applicability and ease of programming perspectives. We therefore chose it as the basic architecture for our work, though much of our findings and suggestions have broader applicability as we discuss in the final section of this paper.

4. Scheduling directives for regular tasks

We next present scheduling directives. For convenience, we express them in the context of task B that depends in some sense on a set A of "prerequisite" tasks. $Prec(B)$ denotes the set of precedences of task B. Also, we use p_T to denote the priority of task T; a larger number represents higher priority.

4.1. Conventional directives (existing)

Start After Complete (SAC). This is simply the precedence relation, whereby B may be dispatched only after all tasks in A have been completed. We used $Prec(B)$ to denote the set of tasks on which task B has a SAC dependence.

Priority. Here, if both B and some task in A are runnable but there is only one available core, A will be dispatched (assuming A has a higher priority than B). If, however, only B is runnable, it need not wait for A.

We next present a new scheduling directive for regular tasks.

4.2. Start After Start (SAS)

The SAS(B,A) directive

For SAS, task B may not be dispatched until task A has been dispatched. SAS expresses an intermediate degree of prioritization, with SAC being more strict and Priority being less strict. Note that SAS is not "work conserving," as a core may be kept idle despite the fact that there is a runnable task.

A motivating example for this is a situation wherein B is runnable and A is not. Also, certain other tasks have a SAC dependence on A, whereas none depend on B. Suppose that a single core becomes available, B is runnable and A becomes runnable shortly thereafter. Although B does not depend on A, letting B grab the core may delay the dispatching of A if A becomes runnable but there is no core to run it on. In this case, it may be desirable to require that B be dispatched only after A has been dispatched, hence the term "Start after Start".

SAS implementation

SAS(B,A) can be expressed using SAC and Priority as follows:

1. Set $Prec(B) \leftarrow Prec(B) \cup Prec(A)$
2. Set $p_A > p_B$

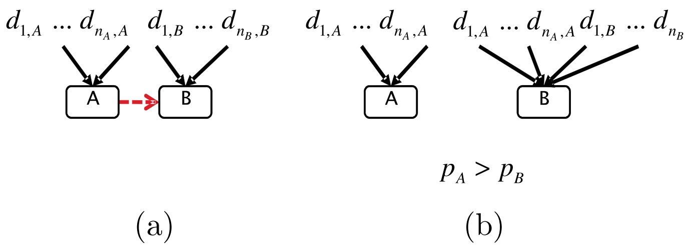
$d_{1,A}$ ... $d_{n_A,A}$ $d_{1,B}$ ... $d_{n_B,B}$
A
B
$d_{1,A}$ ... $d_{n_A,A}$
$d_{1,A}$ ... $d_{n_A,A}$ $d_{1,B}$ ... d_{n_B}
A
B
$p_A > p_B$
(a)
(b)

Fig. 3. (a) Desired graph. The dashed red arrow represents the SAS requirement between tasks A and B. (b) SAS implementation, adding A's precedences to those of B. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The first step ensures that B does not become runnable before A, and the second step ensures that if both are runnable then A is dispatched first. Clearly, both elements are necessary. SAS implementation can thus be based on the existing mechanisms for SAC and priority. See Fig. 3.

5. Scheduling directives for duplicable tasks

5.1. Need and challenges

SAC at the entire-task granularity suffices for guaranteeing correctness. However, it may be overly restrictive and limit parallelism, as illustrated by the following example. Unless stated otherwise, we use a letter to refer to a duplicable task and a subscript value for each of its replicas.

Example 1. Suppose that B_i only depends on data computed by A_i and by A_{i-1}. Clearly, there is no need to wait with the dispatching of B_1 until all replicas of A have been completed. It is therefore desirable to express the SAC constraints between duplicable tasks more precisely in order to expose more parallelism.

Even in the absence of data dependence among duplicable tasks, it may be important to coordinate the dispatching of their replicas. One possible reason is memory performance, as illustrated by the following example.

Example 2. Consider A and B, duplicable tasks that both access the same elements of a data array X in the same order. X is larger than the shared cache. If all of A's replicas are executed prior to any of B's replicas, every data element would have to be brought into the cache twice, as it would drop out of the cache prior to its use by B. If, instead, replicas of A and B were executed concurrently (in lockstep or nearly so), this situation could be avoided.

Remark. One may wonder why the work should be partitioned into two tasks in the first place. The answer is that so doing exposes more parallelism, which may be beneficial especially with fast task dispatching. This is important when the number of cores exceeds the number of data elements, and also serves to reduce the required cache size in support of any given level of computational parallelism. Another use case is when the tasks cannot be fused together, as discussed in Section 5.2.

In view of the above, it is clearly desirable to be able to coordinate the execution of replicas of duplicable tasks with finer granularity. Also, it may be desirable to "throttle" the dispatching of replicas of a single duplicable task for reasons such as total instantaneous memory footprint.

One could conclude from the above that perfect lockstep is the solution, at least SAS lockstep (i.e., controlling the dispatch order). However, this is somewhat simplistic. In Hypercore, for example, the rate at which same-task replicas can be dispatched is much higher than the dispatch rate of different tasks. It is therefore desirable to permit bursts of same-task replica dispatching, but to control burst length.

The challenge is to try and provide sufficient expressive power for stating the desired inter-replica constraints and pacing while permitting sufficiently simple implementation in terms of both speed and the amount of dynamic state information that must be kept. We next present such extensions of SAC and SAS, as well as a limit on the number of active replicas. Before proposing our scheduling directives, we next consider and assess several approaches.

5.2. Duplicable tasks – limitations and workarounds

In this subsection we present several workarounds to the inherently limited expressive power available when a duplicable tasks is treated as a single entity.

Treating each replica as an independent task. E.g., SAC(B_j, A_i). This, however, has major drawbacks: (1) bloated task graph, often becoming impractical for efficient hardware implementation, (2) the task graph becomes parameter dependent: a change in the number of replicas (a parameter change for the application) requires (for performance, not correctness) a change to the graph; (3) the dispatching of regular tasks is usually less efficient than the dispatching of duplicable tasks, because the scheduler [3] can dispatch several replicas of a duplicable task in a single cycle vs. a single regular task per cycle. This approach is therefore impractical.

Fusing two duplicable tasks *A*, *B* to form a single task C. This can sometimes be done offline by the task graph designer. However, task fusion suffers from several deficiencies: (1) the duplicable tasks may not have an equal number of replicas (Fig. 1), complicating the fusion; (2) even simple relationships between the replicas of the two duplicable tasks A and B can be hard to fuse into a new duplicable task C. For example, given that B_i depends on A_i and on A_{i+2} , which replica of C should compute A_j? Should C_j or C_{j+2} compute it? Both C_j and C_{j+2} can compute A_i and A_{i+2}. This causes redundancy in operations. For the scenario that B_i depends on a large number of tasks, this approach is intolerable. Another solution is to let C_j compute A_{j+2}. However, due do to out-of-order completion, C_{j+2} now becomes dependent on C_j, and therefore it cannot be dispatched until the completion of C_j. While task fusion may be suitable for some problems, specifically when B_i only depends on a single A_i, it is not suitable for many scenarios.

We next present our proposed approach and specific scheduling directives. We begin by stating the required priority and state information to which we have elected to restrict our proposed directives. (This is a sensible yet subjective, self-imposed complexity constraint.) We also present our taxonomy.

5.3. Priority and state information

Priority

A replica of a duplicable task inherits the priority of its task. Additionally, it has an intra-task priority (relative to other same-task replicas), which is normally highest for the lowest-number replica. We furthermore assume in-order dispatching of same-task replicas. Specifically, in-order submission to the dispatch dissemination tree-like interconnect. (The exact order in which they obtain cores is not critical for correctness.)

State information

For each duplicable task (e.g., A), we keep the following state information, which is updated by the online scheduler:

1. $A.n$ – total number of replicas (static). This is supported by Hypercore.
2. $A.s$ – number of dispatched/started replicas (both active and completed). This is supported by Hypercore.
3. $A.c$ – number of completed replicas. This is supported by Hypercore.
4. $A.es$ – the "earliest" (lowest index) replica that has been started (dispatched) but has yet to be completed. In Section 6. We will show how to implement its computation efficiently in a manner that is similar to a re-order buffer.

Lemma 1. *Consider a duplicable task A. If $A.es \geqslant A.i$, then all replicas preceding A_i have completed.*

Proof. The in-order dispatching of same-task replicas ensures that these replicas had all been dispatched, and the fact that *es* is the index of the earliest replica that has not been completed ensures that there are no active lower-index replicas. □

The number of active replicas of a given task A is $(A.s - A.c)$. The indication for entire duplicable task completion is $(A.c = A.n)$. In the upcoming subsections, the following functions will be used on replicas of A to determine their status:

1. $S(A_i)$ – returns true iff (A_i) has started,
2. $C(A_i)$ – returns true iff (A_i) has completed,
3. $D(A_i)$ – returns true iff (A_i) may be dispatched.

In addition to the per-task state, two parameters, l_{min} and l_{max}, are optionally used to constrain the permissible progress "gap" between any two duplicable tasks. Different directives will use the variables with slightly different meanings, so more precise definitions will be given in context.

5.4. Start After Complete (SAC) for duplicable tasks

The extensions represent a trade-off between complexity and expressive power. They are moreover intended mainly for situations in which the set of task A replicas on which B_{i+1} depends is obtained from the set on which B_i depends by adding one to the index of every replica in the latter set. Also, we assume in-order dispatching of same-task replicas, though out-of-order completion is permitted.

Consider duplicable tasks A, B such that B_i depends (SAC) on some set of $\{A_j, A_k \ldots A_{max}\}$ of A's replicas, where A_{max} is the replica with the highest id on which B_i depends. (B_{i+1} depends the set of $\{A_{j+1}, A_{k+1} \ldots A_{max+1}\}$ and so on). Our SAC directive simplifies this by requiring that all replicas of A up to and including A_{max} be completed as a prerequisite for the dispatching of B_i. The price is "false" constraints, but correctness is maintained. Moreover, we will later show a potential performance advantage, namely the ability to dispatch bursts of same-task replicas. In Hypercore, this is much more efficient that dispatching individual replicas or regular tasks.

We next develop the underpinnings of a simple expression and implementation of this constraint.

Theorem 2. *If B_i depends on some subset of A's replicas whose highest index is at most max, $A.es \geqslant max$ and A_{max} has completed, then B_i may be dispatched.*

Proof. by Lemma 1, $es \geqslant max$ ensures that A's replicas with a smaller index than *max* have completed, and together with the fact that A_{max} has also completed this guarantees that all the prerequisites for the dispatching of B_i have been met. □

Corollary 3. *Correctness can be guaranteed by way of a single SAC constraint, namely $B_i \leftarrow A_{\max}$ along with an indicator for the additional constraint on the value of es (whose value equals max). Moreover, whenever the precedence constraints are relative (a* *fixed function of i) and satisfaction of the constraints for B_i implies that they have been satisfied for all earlier replicas of B, the constraints can be updated on the fly for different values of i.* □

Remark. Due to the possibility of out-of-order completion of same-task replicas, *es* may increase in arbitrary increments. A reorder-buffer technique can be used for handling updates to *es*. We will return to this in Section 6.

The $SAC(B,A,l)$ **directive**

Let l denote the difference between the index of a replica of B and that of the highest-index replica of A on which it has a SAC dependence. This is a "relative" or "sliding window" equivalent of the situation presented earlier in this subsection (see Fig. 4).

The directive is stated formally in (1) and (2). The former ensures in-order dispatching of A's replicas; the latter ensures in-order dispatching of task-B replicas and expresses the actual constraint. Dependencies on negative-index replicas are taken as having been met.

$$D(A_i) \leftarrow \{S(A_{i-1})\}; \tag{1}$$

$$D(B_j) \leftarrow \{S(B_{j-1}) \wedge C(A_{j+l})\}. \tag{2}$$

Creating hardware to compute these rules is simple, as A's replicas are dispatched like any duplicable task. We denote $dispatch_B$ as the number of B's replicas that may be dispatched at any given time. $dispatch_B$ is computed based on (2) :

$$dispatch_B = A.es - B.s - l. \tag{3}$$

$(A.es - B.s)$ refers to the distance between the earliest active replica in A and the last replica to start in B. This distance has to be at the very least l for there to be replicas of B that can be dispatched.

Remark. This directive can use the distribution network efficiently to dispatch multiple replicas concurrently, as the value of $dispatch_B$ can be greater than one.

Multiple constraints $SAC(B,A,l_{\mathbf{min}},l_{\mathbf{max}})$

Replicas of one task may depend on those of several other tasks. In fact, there may even be bidirectional dependence among replicas of two given tasks. Fig. 5 depicts an example.

For any given task, its constraints are simply the union of all the SAC constraints that express its dependence on other tasks (or even on its own earlier replicas, for that matter). The set of dispatchable replicas of such a task is the intersection of the subsets determined by the individual constraints.

Deadlock

Deadlock can and should be checked for statically (no need for dynamic checking) using the established techniques (looking for loops in the replica-granularity SAC-dependency graph). Special caution must be taken only whenever the limited expressive power of our SAC constraints results in the implicit addition of (false) constraints, which may cause deadlock in situations that were originally fine.

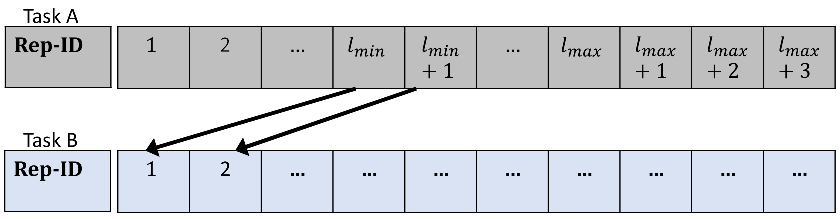

Fig. 4. SAC Type 1 for duplicable tasks. Edges are precedence constraints. Note that the precedence constraints are in one direction.

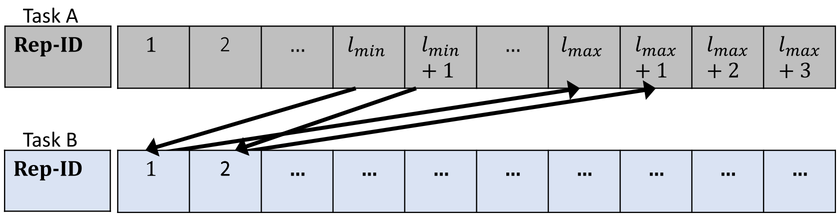

Fig. 5. SAC Type 2 for duplicable tasks. Edges represent precedence constraints. Note that the precedence constraints are in both directions.

5.5. Start After Start (SAS) for duplicable tasks

Given two duplicable tasks, A and B, possibly with no data dependence between their replicas, $SAS(B, A)$ is aimed at specifying the level of synchronization ("lockstep") between the two tasks. Specifically, it specifies the permissible range of the number of replicas by which A's dispatching may advance over B's dispatching. Using the parameters $l_{\min}$ and $l_{\max}$, the definition of this directive is as follows.

Definition. $SAS(B, A, l_{\min}, l_{\max})$

1. The next replica of B may be dispatched only if $A.s - B.s > l_{\min}$,
2. The next replica of A may be dispatched only if $A.s - B.s < l_{\max}$,
3. $l_{\max} \geqslant l_{\min} \geqslant 0$. Negative numbers can be thought of as switching the roles of A and B.
   We refer to ($l_{\min}$, $l_{\max}$) as the *range*.

As stated at the beginning of this section, the purpose of SAS is not correctness. Rather, it is aimed at improving resource utilization and efficiency of operation. In addition to the aforementioned memory-access advantages, SAS can be used to increase the likelihood of being able to take advantage of the burst dispatching capability for same-task replicas. This will be mentioned later in some more detail.

Example 3. Perfect lockstep with priority to replicas of A: range = (0,1) and set $p_{Ai} > p_{Bi+1} \ \wedge \ p_{Ai+1} < p_{Bi+1}$. Note that the priorities alternate between the duplicable tasks and that the indices used in this example are intended for this example alone. (In this case the tasks themselves would receive identical priorities.)

Note that dispatching task B replicas depends on the dispatching of A's replicas and vice versa. Without the former, it would be possible to dispatch all the replicas of B without dispatching a single replica of A, and vice versa.

The SAS constraints are stated formally below, and are illustrated in Fig.6. Eq. (4) enforces the in-order dispatching of replicas of A, and prevents A from getting ahead of B by more than $l_{\max}$ replicas (in terms of dispatching, not completion). Note that in Fig.6, $l_{\max}$ is with respect to upper bound on B_1 (remember that each replica has a different *max* task based on its own index). Similarly, (5) ensures in-order dispatching of replicas of B, and prevents B from trailing A by fewer than $l_{\min}$ replicas. In both cases, a dependence on a negative-index replica is taken as having been satisfied.

The constraints on the replicas are formalized as follows:

$$D(A_i) \leftarrow \{S(A_{i-1}) \wedge S(B_{i-l_{\max}})\}. \tag{4}$$

$$D(B_j) \leftarrow \{S(B_{j-1}) \wedge S(A_{j+\min})\}. \tag{5}$$

In (6) and (7) we show that dispatching replicas meeting the constraints of (4) and (5) can be done efficiently. The following two expressions compute the number of replicas from each of the duplicable tasks that may be dispatched:

$$dispatch_A = l_{\max} - (A.s - B.s) \tag{6}$$

$$dispatch_B = (A.s - B.s) - l_{\min}. \tag{7}$$

Note that when the gap is inside its permissible range, replicas of either task may be dispatched. Here, by assigning higher priority to one of the tasks, burst dispatching will be used whenever possible. Specifically, if the number of available cores is smaller than the difference between the current gap and the relevant limit, a single burst of same-task replicas will take place, which is the most efficient.

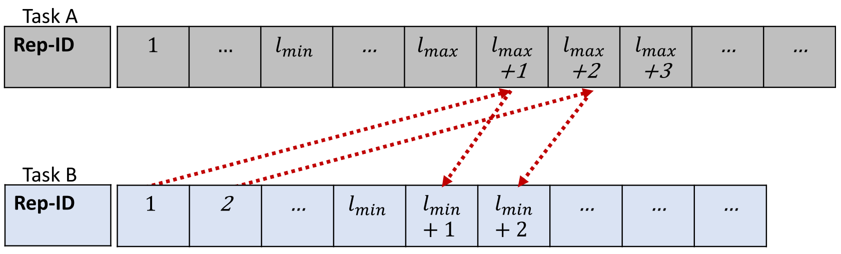

Fig. 6. SAS for duplicable tasks. The SAS (dashed red) edges are in both directions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Remark. It is possible to design the scheduler such that, when replicas of multiple tasks are dispatchable, it would favor tasks based on the number of dispatchable replicas, the number of available cores, and the distance to the relevant limit on the gap. Details are beyond the scope of this paper.

Example 4. In this example we show a use case for SAS that allows for utilizing the scheduler's efficient burst scheduling. For simplicity, we assume a system with two cores. There are two duplicable tasks, A and B, such that all their replicas execute in unit time. Assume that $|A| = |B| \gg 2$. We set $l_{\min} = 2$ and $l_{\max} = 4$. At time $t = 0$, we dispatch A_1 and A_2. At time $t = 1$, we check if replicas of B can be dispatched: $dispatch_B = (2 - 0) - (2) = 0$ and $dispatch_A = 4 - (2 - 0) = 2$. Therefore, at time $t = 1$ we dispatch A_3 and A_4. At time $t = 2 : dispatch_B = (4 - 0) - (2) = 2$ and $dispatch_A = 4 - (4 - 0) = 2$. At time $t = 2$ we dispatch B_1 and B_2. At time $t = 3 : dispatch_B = (4 - 2) - (2) = 0$ and $dispatch_A = 4 - (4 - 2) = 2$. At time $t = 3$ we dispatch A_5 and A_6. This will repeat itself until all the replicas have been dispatched.

The above example is somewhat artificial, as the probability of cycle-accurate lock-step is very low. However, the situation wherein it suddenly becomes possible to dispatch several tasks is quite plausible. It arises, for example, when implementing a barrier. Until the barrier condition is met, cores may become idle as no post-barrier tasks may be dispatched. Once the final remaining barrier condition is satisfied, those cores become usable simultaneously. Using our terms, if both task A replicas and task B replicas have a SAS dependence on task C, a high-priority task that has a SAS dependence between C_i and C_{i-1} and is thus executed sequentially, the completion of a replica of C would possibly render multiple replicas of A and B dispatchable simultaneously to a set of idle cores.

In summary, SAS is a useful construct for pacing the relative progress of different duplicable tasks, in support of fair resource allocation, reduced memory footprint, and more effective dispatching.

5.6. More on SAS and memory performance

The fact that SAS can reduce the instantaneous memory footprint and/or cache miss rate has already been mentioned, as has the fact that perfect lockstep is not necessarily the best approach because of the inefficiency of dispatching individual replicas. Interestingly, in certain cases there is also a memory related reason for permitting some slack. For example, if task B wishes to access a cache line that was just requested by task A, and the latter incurred a cache miss (possibly a compulsory one), B would not incur a miss; however, it would still have to wait for the data to arrive. Therefore, the memory access time experienced by B would be very similar to the miss time. If, instead, B were delayed some, the data would already be in the cache.

This phenomenon is illustrated schematically in Fig. 7. The abscissa is the gap size, and the two ordinates are cache miss rate and average memory access time. We see that whereas the miss rate is monotonically non-decreasing with gap size, average memory access time has a sweet range. The figure is for illustration purposes, not representing actual results, and is intentionally not calibrated.

The following is suggested as a rule of thumb for selecting $l_{\min}$ for the case that the memory needed by each replica is considerably smaller than the shared memory and the replicas of both duplicable tasks have the same execution times:

$$\tilde{l}_{\min} = 2 \cdot |Cores|. \tag{8}$$

In view of the above, there are several good reasons for imposing both an upper limit and a lower limit on the progress gap between two tasks that operate on the same data.

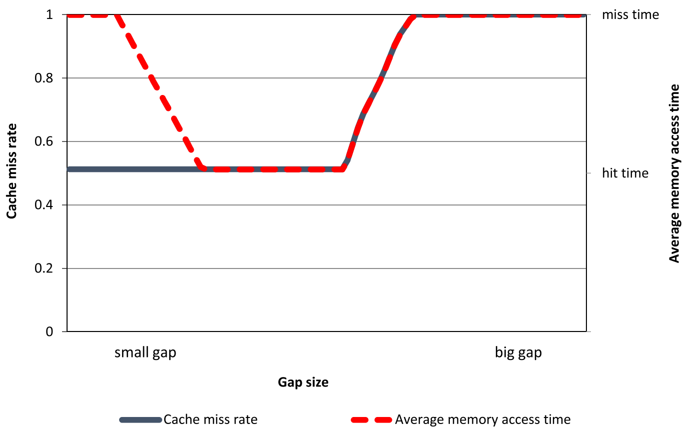

Fig. 7. Synthetic plots that illustrates the effects of gap size on cache performance.

5.7. A basic simulation study

In this section, we present a simple performance evaluation. Due to limited access to an actual hardware implementation, which did not allow us to implement our directives as part of the hardware scheduler, we created a simulator to test SAS and the additional directives. We used a queue-based simulator that dispatched duplicable tasks to a shared-memory many-core system similar to the Hypercore. Using Plurality's cycle accurate software simulator, we were able to obtain cycle counts for memory accesses in the cache, and for the execution of both floating point and integer operations. Plurality's cycle accurate simulator simulates only the first level of the shared memory and does not simulate the DRAM memory, which limits the problem size of the applications that can be tested. Currently, the software simulator API does not support tracing the actual memory access pattern. This too limits the ability to test real applications with actual access patterns. Getting access patterns from other systems such as the x86 is not relevant because of the architectural difference: private L1 caches, cache coherency, different clock frequencies, different pipelines, out of order execution, and more. For DRAM access time, we used actual DRAM times of current technologies. These numbers were passed on to our simulator. Our simulator used the following parameters: 64 cores with a 2*MB* shared cache. We used 32 byte cache lines and a direct-mapped cache. The latency for fetching data out of the DRAM was 20 cycles and fetching from the cache was 2 cycles. Should the 20 cycles latency be an under estimation, this would result in the performance being even more sensitive to cache misses.

The application that we tested was the computation of x (row) and y (column) partial derivatives of a 2000×2000 matrix of single byte elements, which is typical of gray-scale image processing. The size of the array is 4 MB, so the array cannot fit into the shared on-chip cache. The first duplicable task computed the x derivative, and the second duplicable task computed the y derivative for the same matrix. The duplicable tasks were implemented at a fine granularity – element level. Thus, each duplicable task had $\mathrm{n} = 4 \cdot 10^6$ replicas, which is the number of elements in the array.

In Figs. 8 and 9, we present two plots of the number of cache misses and the runtimes for an application, respectively. The dashed curve corresponds to the two duplicable tasks not running concurrently. The solid curve corresponds to executing the tasks concurrently, governed by the SAS directive. The abscissa is the SAS-imposed gap size. The ordinate in Fig. 8 is the total number of cache misses, and in Fig. 9 it is the number of cycles required to complete the application.

It can readily be observed that the use of SAS with a sufficiently small gap offers a noticeable improvement relative to the serialization of the two tasks. This is due to the reduction in cache miss rate. When the gap is large, there is no performance improvement, because data brought into the cache by A is evicted before B has a chance to use it, so B also incurs misses.

5.8. Start After Merged Completion (SAMC)

Start After Merged Completion (SAMC) is used to state that the prerequisites between the duplicable tasks are such that each $\mathrm{B_i}$ depends on the completion of M consecutive task-A replicas. Different task-B replicas are dependent on disjoint subsets of task-A replicas. Given two duplicable tasks, A and B, the dependency between the replicas can be defined as $B_j \leftarrow A_{M\cdot j}, A_{M\cdot j+1}, \ldots, A_{M\cdot(j+1)-1}$. This directive would be useful in implementing a task graph for Merge-Sort [24]. Fig. 10 depicts an example of SAMC where $M = 2$.

Task-A replicas are unconstrained except for in-order dispatching. The constraints on task-B replicas are as follows:

$$S(B_j) \leftarrow \left\{ \begin{matrix} S(B_{j-1}) \wedge C(A_{M\cdot j}) \wedge \\ C(A_{M\cdot j+1}) \ldots \wedge C(A_{M\cdot(j+1)-1}) \end{matrix} \right\} \tag{9}$$

The number of B's that may be dispatched is $dispatch_B = \frac{A.ea}{M} - B.s.$

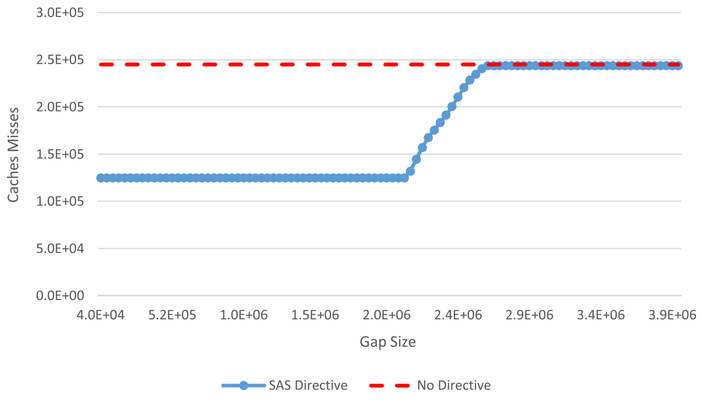

Fig. 8. Total number of cache misses vs. the SAS-imposed gap size for two duplicable tasks with a similar (not exact) access pattern (dashed). The case of no SAS, for which there is no notion of a gap, is brought as a baseline for comparison (solid line).

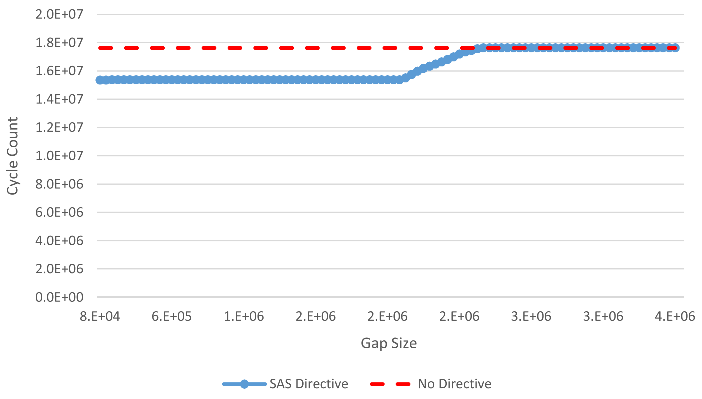

Fig. 9. Execution time vs. SAS gap size for two duplicable tasks with a similar (not exact) access pattern (dashed). The case of no SAS constraints (solid) is the baseline.

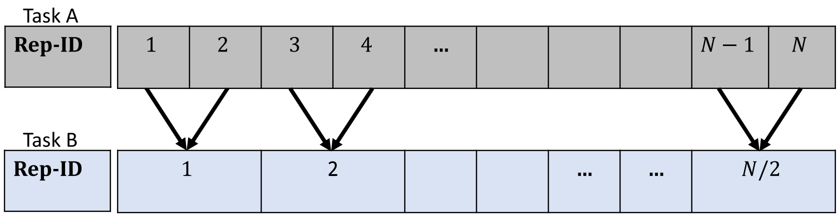

Fig. 10. Start After Merged Completion (SAMC) for duplicable tasks. Edges between A and B are precedence constraints. In this example $M = 2$.

Computing this expression is more demanding than computing the dispatch expressions for the other directives, as this expression requires a division.

The duplicable tasks can have different priorities, which are as follows:

1. $p_A > p_B$: The replicas of A will be dispatched before any of B are disptached.
2. $p_A = p_B$: The replicas of both tasks may be dispatched. For the sake of simplicity and practicality, the replicas of A will be executed before the replicas of B. In effect, this scenario is similar to the previous one.
3. $p_A < p_B$: The replicas of B have higher a priority and will be dispatched before those of A as long as precedence constraints are maintained.

For each of these priority scenarios, a different schedule can be created for dispatching the tasks/replicas.

Example 5. Consider the task graph in Fig. 11 where A has 5 replicas, B has 3 replicas and $M = 2$. For simplicity, assume that these are equal length tasks, as occurs in a parallel prefix summation. Fig. 11 (b) depicts the scheduling of these replicas without any new scheduling directives. Fig. 11 (c) depicts the scheduling for the case that $p_A \geqslant p_B$. Note that while the order in which the replicas are dispatched is the same as in Fig. 11 (b), the schedule length has been reduced. Fig. 11 (d) depicts the scheduling for $p_A < p_B$. Note that the order of dispatching has been changed, and that replicas of A and B may be dispatched concurrently so long as precedence constraints are maintained.

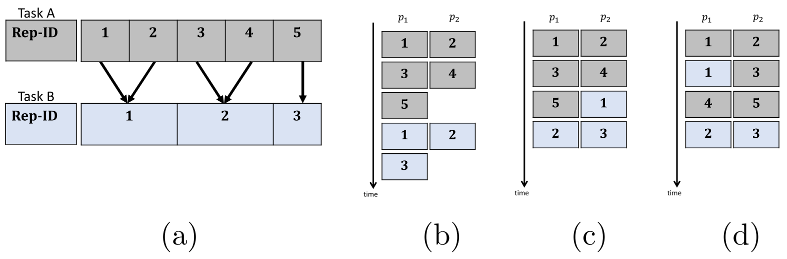

Fig. 11. SAMC example: $|A| = 5, |B| = 3, M = 2$. Assuming equal length tasks. (a) Task graph. (b) Scheduling of graph without new SAMC directive. (c) Schedule of directive when $p_A = p_B$. (d) Schedule of directive when $p_A < p_B$.

We prove that using the SAMC directive with $p_A \geqslant p_B$ ensures that execution time of the program is not greater than the execution time when B is dependent on the completion of all of A's replicas (i.e., the standard model). We do this in two steps. While the first lemma (Lemma 4) may seem intuitive, its proof is necessary for the second (Lemma 5). We add the following notations:

1. $ST(i)$ – Start (Dispatch) Time of replica i. This is dependent on the actual scheduling.
2. $CT(i)$ – Completion Time of replica i. This is dependent on the actual scheduling.
3. $ET(i)$ – Execution time (length) of replica i. This is equivalent to $CT(t) - ST(t)$, but is independent of an actual scheduling and is a property of the task.
4. SL – Schedule length, essentially is the execution time of the program. This is also equal to the CT of the last replica to finish its execution.
5. Std – refers to the standard model of using only duplicable tasks.

Lemma 4. *The order of dispatched replicas is the same when using only duplicable tasks and when using the SAMC directive for the case that* $\mathrm{p_A} \geqslant \mathrm{p_B}$ *(cases 1 and 2).*

Proof. Given the in-order dispatching of replicas, the replicas of A and B are dispatched in the following order: $A_1, A_2, \ldots, A_{N_A}, \; B_1, B_2, \ldots, B_{N_B}$.

When dispatching according to the SAMC directive:

$\mathrm{p_A} > \mathrm{p_B}$ – due to difference in priority, all the replicas of A are dispatched before a single replica of B is dispatched. This ensures that the order of dispatching is $A_1, A_2, \ldots, A_{N_A}, B_1, B_2, \ldots, B_{N_B}$.

$\mathrm{p_A} = \mathrm{p_B}$ – For the same reason, the same order is obtained.

Therefore, the order of dispatching is the same for standard model and for SAMC when $\mathrm{p_A} \geqslant \mathrm{p_B}$. □

Lemma 5. *The schedule length for dispatching the replicas of A, B using SAMC is smaller than or equal to that using only duplicable tasks for the cases* $\mathrm{p_A} \geqslant \mathrm{p_B} : SL_{SAMC} \leqslant SL_{std}$.

Proof. Assume by contradiction that there exists a replica $\hat{b} \in B$ such that $CT_{Std}(\hat{b}) < CT_{SAMC}(\hat{b})$. As the time of $\hat{b}$ is the same for both schedules, $ET_{Std}(\hat{b}) = ET_{SAMC}(\hat{b})$, it can be inferred that $ST_{Std}(\hat{b}) < ST_{SAMC}(\hat{b})$.

Based on Lemma 4, the order in which the replicas are dispatched is the same and it is known that a replica of B in SAMC has no more dependencies than its counterpart in the standard model. As such, if a replica in the standard model can be dispatched, so can its counterpart in the SAMC directive. This implies that $ST_{Std}(\mathrm{t}) \nless ST_{SAMC}(\mathrm{t})$. Further, by adding $ET(\hat{b})$ to both sides of the inequality $CT_{Std} = ET(\hat{b}) + ST_{Std}(\mathrm{t}) \nless ST_{SAMC}(\mathrm{t}) + ET(\hat{b}) = CT_{SAMC}$. This contradicts the assumption, leading to the conclusion that $CT_{SAMC}(\mathrm{t}_2) \leqslant CT_{Std}(\mathrm{t}_2)$.

Given that the above is correct for all replicas of B, this will also be correct for the last replica of B to complete. Therefore, $SL_{SAMC} \leqslant SL_{Std}$. □

Given the above proofs, SAMC with $\mathrm{p_A} \geqslant \mathrm{p_B}$ will always give an equal or better schedule than the standard model, so a directive such as SAMC is preferable over just using duplicable tasks.

While the above proof discusses theoretical scheduling of SAMC, there are also practical implications of SAMC. One benefit of SAMC is that the replica of B can use data that has already been fetched into the shared cache.

5.9. Additional directives

We have presented and discussed three scheduling directives for duplicable tasks: SAS, SAC and SAMC. We next briefly present several additional structured scheduling directives that we believe to be useful for task graph designers (human or automated tools such as compilers):

1. *Limit Number of Active Replicas (LNAR)* is used in order to limit the number of concurrently active replicas of a duplicable task. This is useful to prevent hogging of all cores by a single task's replicas or for memory and I/O considerations. Setting the limit to K, the first K replicas can be dispatched without any constraint. For all $i > K$, the following constraints are added.

$$S(A_i) \leftarrow \{S(A_{i-1}) \land (A.s - A.c < K)\} \tag{10}$$

The number of replicas that can be dispatched at a given time is:

$$dispatch_A = K - (A.s - A.c) \tag{11}$$

2. *Assign Cores Fairly (ACF)* is used in order to split the cores evenly between two duplicable tasks A and B. This directive is useful when the number of replicas exceeds the total number of cores and it is desirable that not all the cores execute same-task replicas. This directive refers only to the number of started replicas and not to the order of their completion. The constraints on A are:

$$S(A_i) \leftarrow \left\{ \begin{array}{l} S(A_{i-1}) \wedge \\ (A.s - A.c) \leqslant (B.s - B.c) \end{array} \right\} \tag{12}$$

The first constraint on A is the usual in-order dispatching of same-task replicas. The second constraint ensures that A has fewer active tasks than B does, Due to symmetry, the constraints on B are the same. To compute the number of replicas that can be dispatched:

$$dispatch_{A/B} = (A.s - A.c) - (B.s - B.c). \tag{13}$$

In expression (13) the numbers of active of replicas of both tasks are compared. If $dispatch_{A/B} < 0$ then there are more active replicas of B in the system and A may dispatch accordingly the difference. If $dispatch_{A/B} > 0$ then there are more active replicas of A in the system and B may dispatch accordingly the difference. If $dispatch_{A/B} = 0$ then there is an equal number of active replicas and the idle cores should be divided equally between the duplicable tasks.

3. *Limit Number of Replicas after Earliest Started (LNR)* is used in order to limit the span (range of ids) of active replicas of a given duplicable task. This can be seen as a limited size sliding window of dispatched replicas. Until the first replica in the window, *es* is completed, the window cannot be moved forward. This directive is similar to LNAR with the difference being that LNR limits the number of dispatched replicas w.r.t. to the *es* replica dispatched. Furthermore, this directive enforces correctness unlike LNAR. LNR is useful for controlling memory access locality in certain cases.
The first K replicas can always be dispatched. For all $i > K$ the following constraints are added.

$$S(A_i) \leftarrow \{S(A_{i-1}), C(A_{i-K})\} \tag{14}$$

The number of task-A replicas that can be dispatched at a given time is:

$$dispatch_A = K - (A.s - A.ea) \tag{15}$$

6. Thread re-order buffer

In the previous section, we presented the state variable (for each task) *es*, which is the index of the lowest-index replica that has been dispatched but not yet completed. We also showed how *es* can be used in conjunction with in-order dispatching of same-task replicas in order to determine whether any given task-B replica may be dispatched. Finally, we pointed out that, due to out-of-order completion of same-task replicas, the value of *es* may change in arbitrary (positive) increments. Having provided indications for the benefits of having using *es*, the question whether it can be updated efficiently gains importance. In this section, we present a scheme for updating the value of *es*. We refer to the value of *es* of a given task A as $A.es$.

6.1. Replica re-order buffer for updating es

Consider a change of $A.es$ from $A.es_{old}$ to $A.es_{new}$. This can only be brought about by the completion event of A's replica number $A.es_{old}$, with replica number $A.es_{new}$ having been dispatched prior to this event and with all replicas in the range $(A.es_{old} + 1, A.es_{new} - 1)$ having completed prior to it as well. It is readily evident that the update mechanism of *es* is essentially the same as that for controlling the commit phase in processors with out-of-order execution and in-order commit. One can thus employ a reorder buffer (ROB) [25,26] per active duplicable task.

For the implementation of the *es* field to be considered efficient and practical, it must meet the $O(\log_2(|cores|))$ dispatch time of the current scheduler and be low power, small in physical size and scalable.

While the function of our thread ROB is the same as one of the functions of an instruction ROB, there are some important differences. For example, we do not need to actually do anything with completed replicas, other than move a pointer. Therefore, the maximum number of replicas (jointly for all active tasks) equals the number of cores (or, if each core can handle several tasks concurrently, like multi-threading, a small multiple thereof).

We now present a low-power logarithmic time method for computing the *es* replica, initially considering a single active task. As depicted in Fig. 12, we create a tree whose leaves are the cores. Each core provides the index of the replica on which it is working, and a null value if it is idle. Intermediate nodes compute the minimum over the numbers that they receive from their sons and pass it on to their father. The root thus holds the minimum index value of an active replica, which is exactly *es*. This can be constructed as simple combinational logic, or can be pipelined. Fig. 12 depicts the simplest case, namely a binary tree without pipelining.

The extension to multiple concurrently active tasks is as follows. Now, each core provides both the task ID and the replica index. In one possible embodiment, depicted in Fig. 13, the tree is replicated several times (equal to the maximum supported number of concurrently active different duplicable tasks, and the replica ID's are directed to the relevant tree based on their

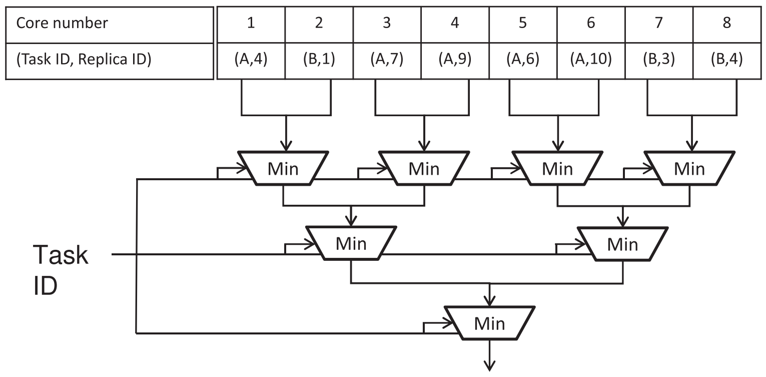

Fig. 12. Schematic diagram of the new hardware. For simplicity, the non-pipelined version is presented. Each core maintains the duplicable task ID and the replica ID of the task that it is executing.

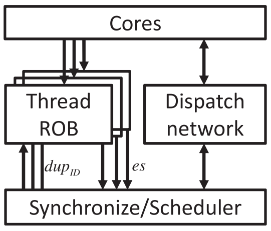

Fig. 13. Block diagram of a the system using a multiple thread re-order buffers.

Table 1
Re-order buffer specifications.

Parameter name	Value
Number of cores	64
Process used	65 nm
Total dynamic power	7.0 mW
Physical size	0.025 mm^2
System frequency	400 MHz
Number of cycles	3

task ID. Alternatively, a single, time-multiplexed tree can be used. In each clock cycle, the cores (or some logic between them and the first layer of internal tree nodes) receives a task id, and only lets the replica ID through if it belongs to the appropriate task.

Since the number of concurrently active duplicable tasks is usually small, and since we are dealing with task granularity, the slight additional delay of the time-multiplexed approach is likely to be tolerable. Finally, one can combine the two schemes so as to support a certain number of concurrent tasks with no penalty while not limiting the number of concurrent tasks that can be supported. In Table 1, the specifications of our “virtual” implementation of the hardware is given for a 64-core system. The implementation is low power, low latency and requires little chip space. Further implementation details are left to implementers.

6.2. ROB-scheduler interaction

Fig. 13 depicts a schematic diagram of the interaction between the new thread reorder buffer and the scheduler and the cores. Given the value of *es* for each task, along with other information available to the Hypercore scheduler (e.g., the next replica to be dispatched for each task as well as the various constraints), the extension of the scheduling logic to make use of *es* and the constraints as described earlier is a simple engineering task using simple logic. Details are therefore omitted.

We note in passing that the new thread reorder buffer has a similar structure to the scheduler’s distribution network, but operates in the reverse direction. In practice, it may be beneficial to co-design the two networks.

7. Conclusions

This work addressed a computing framework whereby scheduling policies are decided offline, and are enforced during run time. Specifically, we investigated the case of duplicable tasks, i.e., tasks that have many instances (data parallelism). This paper did not offer scheduling policies; instead, it offered directives that serve policy makers (human and tools alike) to express their policies.

Our focus was on three directives: Start After Complete (SAC), and Start-After-Start (SAS), Start After Merged Completed (SAMC). However, we presented several additional directives and discussed them briefly. Some of the proposed directives serve to better express correctness constraints, while others aim to enhance performance (e.g., effective use of the memory system) by controlling the relative progress of different duplicable tasks.

In representing (and enforcing) a SAC constraint between replicas of different duplicable tasks, there are two extremes: (1) a SAC constraint between the entire tasks (all replicas of *A* must complete before any replica of *B* is dispatched), and (2) specify the exact inter-replica dependences and enforce them. The former was claimed to potentially reduce performance by hiding too much of the permissible parallelism, and the latter is complex to implement as much state must be maintained and updated. Instead, we proposed a compromise: If a given replica of task *B* depends on a set of replicas of task *A*, treat it as if it depends on the completion of all replicas of *A* with indices less than or equal to that of the highest-index replica of *A* on which it actually depends. We then presented the state variable (for each task) *es*, which is the index of the lowest-index replica that has been dispatched but not yet completed. We also showed how *es* can be used in conjunction with in-order dispatching of same-task replicas in order to determine whether any given task-*B* replica may be dispatched.

The proposed directives represent what we view as a sensible trade-off between expressive power (and resulting benefits) and implementation complexity. To this end, we sketched a power efficient implementation of the main directives.

We presented a small simulation study of a particular application, wherein the performance gain due to the decrease in cache misses (a 50% reduction in miss rate) is around 15%.

Finally, we briefly discussed implementation. We pointed out that, due to out-of-order completion of same-task replicas, the value of earliest started (*es*) value of a task may change in arbitrary (positive) increments, and showed an efficient, ROB-like hardware scheme for updating it efficiently and quickly. We also showed that this and the various constraints can be integrated into an actual system, Plurality's Hypercore system, while maintaining the low power and space envelope using simple logic design.

In our work, we used the Hypercore shared-memory many-core architecture and Plurality's prototype instantiation thereof as a reference system for the incorporation of new scheduling directives, but the new scheduling directives can be used for other systems as well. A variation of our directives might be applicable to NVIDIA's GPU [27]. Such a variation might consider the fact that multiple consecutive CUDA kernels (duplicable task) need to access the same memory that has already been fetched in the private shared caches. Rather than each kernel fetching the data into the shared memory, the scheduler would dispatch the right replica into the right multiprocessor. Our directives may be applicable to the OpenMP [17] standard which would allow independent threads to continue their execution without waiting on a barrier (syncthreads). In fact, it may be possible to implement these directives purely in software as part of the OpenMP standard. A software implementation may reduce their ideal hardware performance yet would increase user expressability. For the x86, it may be possible to create hardware support for these directives, especially since our design is low power. Implementing our directives and our thread reorder buffer for any architectures requires access to the actual hardware scheduler implementation.

The relationship between scheduling and memory performance is very interesting. Some of our proposed directives address this issue, and we provided examples of their beneficial use. However, the treatment of the cache related issues in this paper is only a first step. There are interesting interactions between the use of the directives, including even task replication granularity, and the cache organization (e.g., associativity) and parameters (size, line size, etc.). In fact, there may be a trade-off between the benefits one can obtain by using the directives for a specific configuration and the sensitivity of performance to changes in this configuration.

In this paper, we identified performance related scheduling issues, and identified a set of useful directives, which are both intuitive to a programmer or a tool and can be implemented efficiently. We took another important step, demonstrating the potential benefit on toy examples, and showing that the directives can be supported efficiently in hardware. However, there is clearly more to be done. We hope that this first step will motivate further work in several directions: (1) further study of directives; (2) demonstration on real applications, with special attention to the interplay with the cache; (3) incorporation of support for these directive into the hardware of various architectures; and (4) automatic directive generation by offline tools, as well as support for manual insertion of directives by the programmer.

References

[1] J. Keller, C.W. Kessler, J.L. Traeff, Practical PRAM Programming, Wiley, New York, 2001.
[2] EE Times - Analysis: Hypercore Touts 256 CPUs Per Chip. URL <www.eetimes.com/design/signal-processing-dsp/4017491/Analysis-Hypercore-touts-256-CPUs-per-chip>.
[3] Plurality - HyperCore Software Developers Handbook. URL <www.plurality.com>.
[4] X. Wen, U. Vishkin, Fpga-based prototype of a PRAM-on-chip processor, in: Proceedings of the 5th Conference on Computing Frontiers, ACM, 2008, pp. 55–66.

[5] X. Wen, Hardware Design, Prototyping and Studies of the Explicit Multi-Threading (XMT) paradigm, ProQuest, 2008.
[6] P. Konecny, Introducing the Cray XMT, Seattle, WA, USA, 2007.
[7] C. Nvidia, Programming Guide, NVIDIA Corporation, 2013.
[8] Brief, TILE-Gx8036 Processor Specification, Tilera corporation, 2011, <http://www.tilera.com>.
[9] R.L. Graham, Bounds for certain multiprocessing anomalies, Bell Syst. Tech. J. 45 (9) (1966) 1563–1581.
[10] J.D. Ullman, Polynomial complete scheduling problems, ACM SIGOPS Operating Syst. Rev. 7 (4) (1973) 96–101.
[11] R.L. Graham, E.L. Lawler, J.K. Lenstra, A. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling: a survey, Ann. Discrete Math. v5 (1977) 287–326.
[12] P. Brucker, Scheduling Algorithms, Springer, 2007.
[13] O. Sinnen, Task Scheduling for Parallel Systems, vol. 60, 2007, <Wiley.com>.
[14] G.C. Sih, E.A. Lee, A compile-time scheduling heuristic for interconnection constrained heterogeneous processor architectures, IEEE Trans. Parallel Distrib. Syst. 4 (2) (1993) 175–187.
[15] Y.-K. Kwok, I. Ahmad, Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors, IEEE Trans. Parallel Distrib. Syst. 7 (5) (1996) 506–521.
[16] D.W. Gillies, J.W.-S. Liu, Scheduling tasks with AND/OR precedence constraints, SIAM J. Comput. 24 (4) (1995) 797–810.
[17] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, Comput. Sci. Eng. IEEE 5 (1) (1998) 46–55.
[18] Intel, Threading Building Blocks.
[19] Intel, Concurrent Collections.
[20] Budimlic Zoran, Chandramowlishwaran Aparna, Knobe Kathleen, Lowney Geoff, Sarkar Vivek, Treggiari Leo, Multi-core implementations of the concurrent collections programming model, in: CPC09: 14th International Workshop on Compilers for Parallel Computers, 2009.
[21] N. Bayer, A Hardware-Synchronized/Scheduled Multiprocessor Model (Ph.D. thesis), Technion-Israel Institute of Technology, 1989.
[22] N. Bayer, R. Ginosar, High flow-rate synchronizer/scheduler apparatus and method for multiprocessors, US Patent 5,202,987 (Apr. 13 1993).
[23] P. Avieli, O. Rubenov, N. Bayer, Designing a central synchronization/scheduling unit for multiprocessors, in: The 21st IEEE Convention of the Electrical and Electronic Engineers in Israel, IEEE, 2000, pp. 495–498.
[24] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, New York, 2001.
[25] R.M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, IBM J. Res. Dev. 11 (1) (1967) 25–33.
[26] J.L. Hennessy, D.A. Patterson, Computer Architecture : A Quantitative Approach, Morgan Kaufmann, Boston, 2007.
[27] NVIDIA Corporation, NVIDIA CUDA Programming Guide.