

## RESEARCH ARTICLE

# Reducing the burden of parallel loop schedulers for many-core processors

Mahwish Arif<sup>1</sup> | Hans Vandierendonck<sup>2</sup> <sup>1</sup>Computer Science Laboratory, University of Cambridge, Cambridge, UK<sup>2</sup>School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast, Northern Ireland**Correspondence**Hans Vandierendonck, Computer Science Building, 16 Malone Road, Belfast BT9 5BN, Northern Ireland.  
Email: hvandierendonck@qub.ac.uk**Present address**

Queen's University Belfast, Belfast, BT7 1NN, Northern Ireland

**Funding information**

Engineering and Physical Sciences Research Council, Grant/Award Numbers: EP/L027402/1, EP/M008495/1; FP7 Information and Communication Technologies, Grant/Award Number: 619706; FP7 People: Marie-Curie Actions, Grant/Award Number: 327744; H2020 Future and Emerging Technologies, Grant/Award Number: 732631

**Summary**

As core counts in processors increases, it becomes harder to schedule and distribute work in a timely and scalable manner. This article enhances the scalability of parallel loop schedulers by specializing schedulers for fine-grain loops. We propose a low-overhead work distribution mechanism for a static scheduler that uses no atomic operations. We integrate our static scheduler with the Intel OpenMP and Cilkplus parallel task schedulers to build hybrid schedulers. Compiler support enables efficient reductions for Cilk, without changing the programming interface of Cilk reducers. Detailed, quantitative measurements demonstrate that our techniques achieve scalable performance on a 48-core machine and the scheduling overhead is 43% lower than Intel OpenMP and 12.1× lower than Cilk. We demonstrate consistent performance improvements on a range of HPC and data analytics codes. Performance gains are more important as loops become finer-grain and thread counts increase. We observe consistently 16%–30% speedup on 48 threads, with a peak of 2.8× speedup.

**KEYWORDS**

parallel computing, shared-memory synchronization

## 1 | INTRODUCTION

While Moore's Law remains active, every new processor generation has an increasing number of CPU cores. Highly parallel processors such as Intel's Xeon Phi Knights Landing<sup>1</sup> provide a high number of less powerful but energy-efficient cores. Moreover, scale-up shared memory machines such as the SGI UV line serve tightly synchronized workloads. Scheduling and distributing work load on large scale shared-memory machines becomes increasingly important in order to make efficient use of the hardware.

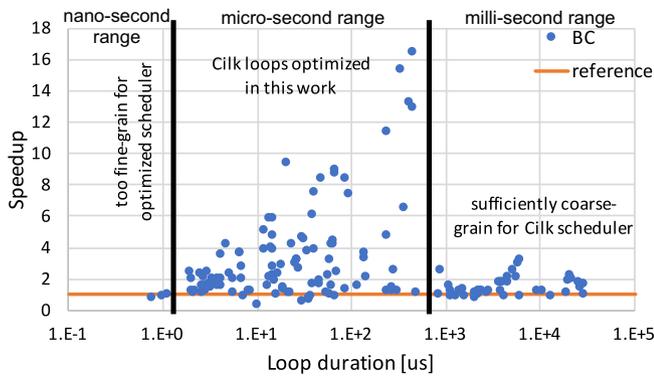
Scheduling and work distribution induce a run-time overhead, called *burden*,<sup>2</sup> that includes the time taken to make scheduling decisions, send the work to other processors and synchronize on the completion status. The scheduler burden has not been widely documented or studied. Creating tasks in Cilk has about 3.63× overhead compared with a normal function call.<sup>3</sup> However, this does not yet involve distributing the task to other processors.

To illustrate the problem, Figure 1 shows the duration of fine-grain parallel loops that occur in the Ligra<sup>4</sup> framework when calculating betweenness-centrality. These fine-grain loops perform operations such as reductions, filtering, and packing of array elements. The loops already execute on 48 threads (see Section 5 for details on the platform) using the Intel Cilkplus runtime.<sup>5</sup> The dynamic range of loop duration is very high, ranging from submicrosecond to tens of milliseconds. This relates to the loop iteration count as well as the amount of work performed per iteration. The vertical axis (Figure 1) shows the speedup obtained by reducing the scheduler burden using the techniques presented in this article, which can

Mahwish Arif was with Queen's University Belfast at the time the research was conducted.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2021 The Authors. *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.



**FIGURE 1** Time take by parallel loops in betweenness-centrality (BC) computation on the Twitter graph (X-axis) and the speedup achieved with a low-burden scheduler over the Intel Cilkplus scheduler (Y-axis)

result in as much as a 16-fold speedup. We will demonstrate that other schedulers, such as OpenMP, are also significantly impacted by their burden.

An immediate consequence of the burden is that some parallel code is *too fine-grain* to make parallel execution worthwhile. What “too fine-grain” means depends strongly on the hardware, the scheduling algorithm and its implementation. This burden, growing with the degree of parallelism, can affect the scalability of schedulers.

This article demonstrates that fine-grain, *micro-second-scale* parallel loops already stress the burden of state-of-the-art parallel schedulers. Micro-scale events have evaded performance optimization for decades, as they are too coarse-grain to overlap with instruction-level parallelism in processors, and too fine-grain to handle with operating system support.<sup>6</sup> However, technological advances have made micro-second-scale events common-place today. As such, it is important to tune the software stack to meet these latencies, else technological advances will be overshadowed by software overhead. This article, specifically, investigates the burden of parallel loop schedulers.

In a bid to reduce the burden, a significant body of research has investigated how to reduce the computational complexity of scheduling algorithms. Near-optimal execution times can be obtained with *greedy* scheduling techniques,<sup>7</sup> which execute work in the order it arrives. Alternatively, *static* schedulers apply an easy-to-calculate work distribution,<sup>8-10</sup> which allows for optimizing the execution order with minimal run-time decision-making. The burden, however, is composed of both decision-making and synchronization delay.

Specialized hardware can reduce the decision-making overhead and synchronization delay,<sup>11,12</sup> Hard-coding a scheduler in hardware, however, removes the option of tuning schedulers to application properties. Hardware-based message queues use dedicated on-chip networks to circumvent the cache coherence protocol.<sup>8,13</sup> However, the benefits of this are modest: The intrasocket ping-pong delay between two threads using the cache coherence protocol on a modern processor is around 60 ns (measured on a 2.6 GHz Intel Xeon E7-4860 v2). Dedicated networks can reduce this delay but the potential benefit is only a fraction of the burden of state-of-the-art software schedulers.

We argue that significant advances can be made to reduce scheduler burden. We pursue this goal by recognizing that fine-grain loops require distinct scheduling policies than coarse-grain loops. These fine-grain schedulers complement the existing coarse-grain schedulers, similar to how schedulers for load-balanced loops complement those for imbalanced loops.<sup>10</sup>

This article makes the following contributions:

- We identify the problem that application performance is hindered by scheduler burden on modern multicores.
- A NUMA-aware algorithm for work distribution of fine-grain parallel loops and its implementation in the OpenMP and Cilk schedulers.
- A hybrid scheduler that dynamically switches between the baseline dynamic scheduler and the fine-grain scheduler. The hybrid scheduler allows the expression of complex nested parallelism in a NUMA-aware manner by splitting off groups of closely collaborating threads.
- Compilation algorithms that enhance the efficiency of fine-grain parallel loops with Cilk reductions.<sup>14</sup>
- Detailed experimental evaluation of the extended OpenMP and Cilk runtimes on programs exhibiting fine-grain parallel loops. The programs are drawn from traditional HPC and modern application domains such as map-reduce workloads and graph analytics (Section 5).

A preliminary version of this work was reported to the sponsor in a project deliverable,<sup>15</sup> and on a poster.<sup>16</sup>

The rest of this article is organized as follows. Section 2 presents our loop-oriented programming model. Section 3 presents our low-overhead work distribution technique and its integration in runtime systems. Section 4 discusses support for generalized, noncommutative reductions. Section 5 presents an evaluation of the proposed techniques and a measurement of the scheduler burden of state-of-the-art schedulers. Section 6 discusses related work.

## 2 | PROGRAMMING MODEL

This section describes the programming abstractions used in **OpenMP** and **Cilk** to express parallel loops and reductions, and presents extensions to identify fine-grain loops.

### 2.1 | The OpenMP language

The OpenMP language<sup>10</sup> provides a directive-based approach to parallel programming. We focus on parallel loops with reductions. Many OpenMP features compose trivially with the ideas presented in this work (e.g., `first/lastprivate`, `nowait`), while others are left for future work (e.g., `teams`).

Figure 2 shows an OpenMP program that accumulates the values of an array using a parallel loop and reduction variable `r`. The pre-processor directive `parallel for` (on line 3) signifies that the following loop is a parallel for-loop. The clause `reduction(+:r)` indicates that the variable `r` is used in a summation, and enables the compiler to create thread-private variables that accumulate per-thread partial sums and to aggregate the partial sums at the end of the loop. The OpenMP standard allows implementations to select suitable runtime mechanisms. GCC 4.9 aggregates the partial sums sequentially after the parallel loop, while Intel OpenMP uses a runtime heuristic to select between atomic additions and pairwise additions incorporated in a barrier-like construct during the parallel loop. Incorporating reductions in the barrier is more efficient at high thread counts, but we will demonstrate that there is scope for optimization of this construct.

The `schedule` clause indicates how the loop should be scheduled. A `static` schedule assigns each thread a predetermined block of iterations, whereas a `dynamic` schedule dynamically assigns blocks of loop iterations to threads.

### 2.2 | The Cilk language

Cilk is a C/C++ language extension that adds support for parallel execution. The `spawn` keyword before a function call indicates that the spawned function may execute in parallel with the *continuation* of the calling function. The parallel execution extends to the next `sync` statement in the calling function or to the end of the function, whichever occurs first.

Parallel loops are indicated with the `cilk_for` keyword when it is used in place of the `for` keyword. Parallel loops are implemented on top of the `spawn/sync` constructs through a recursive decomposition of the loop iteration range. This allows dynamic load balancing but also incurs a higher burden than, for example, OpenMP static scheduling, due to work stealing.

Cilk supports generalized, noncommutative reductions through *reducers*, a type of *hyperobject*.<sup>14</sup> A reducer is defined by a triple (*type, associative operator, identity*). Reducers may have multiple *views* associated to them throughout the execution of a program. A view is an object of the reduction type that is accessible to at most one thread at a time. Each thread collects its views in a *hypermap*, which maps original addresses of reducers to the thread's view. Before accessing a reducer, the client program looks up the current view in the hypermap. If nonexistent, a new view is created. The runtime reduces views as necessary upon completion of spawned functions and `sync` statements. It does so by reducing whole hypermaps at a time. Figure 3 shows a Cilk program that accumulates the values in an array using a reducer `r`. The operation `*r` is a short-hand to lookup the current thread's view for `r` and serves the relevant view for the calling thread.

```

1 int accumulate( int *a, int n ) {
2     int r = 0;
3     #pragma omp parallel for reduction(+:r) schedule(static)
4     for(int i = 0; i < n; i++)
5         r += a[i];
6     return r;
7 }

```

FIGURE 2 An OpenMP loop with a reduction

```

1 int accumulate( int *a, int n ) {
2     cilk :: reducer<cilk :: op_add<int>> > r;
3     cilk_for (int i = 0; i < n; i++)
4         *r += a[i];
5     return r . get_value ();
6 }

```

FIGURE 3 A Cilk loop with a reducer

## 2.3 | Language extensions

We introduce a simple language extension that programmers can use to identify fine-grain loops, and allows the compiler and runtime system to tune their operation to these types of loops. In OpenMP, we introduce a new scheduler specification `schedule (finegrain)`, which instructs the compiler and runtime to use a specific scheduling algorithm for this loop. This mechanism works well for OpenMP as it already supports multiple scheduler types.

In Cilk, we introduce a new pragma “`#pragma cilk finegrain`” that can be added immediately before `cilk_for` loops:

---

```
1 #pragma cilk finegrain
2 cilk_for(int i = 0; i < n; i++)
3 *r += a[i];
```

---

The fine-grain annotations state that the loop is a fine-grain parallel loop and should be scheduled using a specialized low-overhead scheduler. In this work, the burden is reduced by simplifying the scheduling algorithm and by using highly efficient synchronization operations. Other techniques to reduce the burden may be used as well without diminishing the contributions of this work. We assume that no nested parallelism exists in the fine-grain loop. This is reasonable as otherwise the loop would likely not be fine-grain. We demonstrate in this article that the fine-grain scheduler retains the functionality of OpenMP reductions and of Cilk hyperobjects.

## 3 | SCHEDULING FINE-GRAIN PARALLEL LOOPS

We consider scheduling of parallel loops without cross-iteration dependencies, except for the presence of reduction operations. The Intel and GNU OpenMP runtime schedule such loops statically by following four steps. These steps are initiated by the thread that encounters the parallel loop, which we call the master thread:<sup>10</sup>

1. **Scheduling:** The master thread divides the loop iteration range in equal chunks, one for each of the threads that will contribute to the execution of the loop.
2. **Work distribution:** The master thread sends work descriptions to the workers. These include the task (typically identified by a function pointer) and the portion of the loop iteration range assigned to the worker.
3. **Parallel execution:** Worker threads initialize local copies of reduction variables and start executing their work as soon as they have obtained it. The master typically also executes part of the work.
4. **Synchronizing on completion:** The master thread typically needs to wait for the workers to complete. Partial results for reduction variables are reduced into the actual reduction variable.

This template assumes that the loop iteration range is known when the loop is encountered and that there are no unexpected control flow leaving the loop.

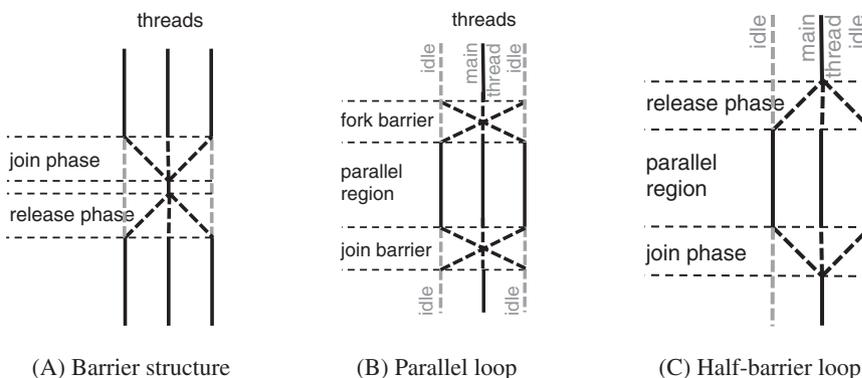
The synchronization in steps 2 and 4 is typically implemented using barriers. Barriers involve a *join* and a *release* phase (Figure 4(A)) that respectively records the arrival of threads, and signals threads to enter the next phase of computation. Step 2 is known as a *fork barrier*. Step 4 is a *join barrier*. Typically, at least two barriers are executed per parallel loop (Figure 4(B)). This is also the case in the GNU and Intel OpenMP runtimes. Additional synchronization may be required, for example, to support dynamic teams, a grouping of worker threads that will be involved in executing the current parallel region.<sup>10</sup> Moreover, the Intel OpenMP runtime implements reductions on top of a barrier-like construct, which effectively introduces an additional barrier.

### 3.1 | Half-barrier pattern

Barriers involve redundant synchronization when synchronizing parallel loops. We build on the assumption that the worker threads are associated to a specific master thread. The master encounters a parallel code region and shares the work with the workers. Under this assumption, which is true in runtime system such as GNU and Intel OpenMP, the following observations can be made:

**Observation #1: The worker threads are available at the start of a parallel region.** If the master is not executing inside a parallel region, then the workers must be idle. As such, it is not necessary for the master to ensure that the workers have arrived at the fork barrier, they have no other task than to wait at that point.

**FIGURE 4** Schematic structure of threads and synchronization in parallel loops and barriers



**Observation #2: The worker threads are independent of one another.** As we focus on loops where tasks are independent and synchronization-free units of work, there is no dependence between workers executing distinct tasks. Combining Observations 1 and 2, we conclude that it is unnecessary to execute the join part of the fork barrier.

**Observation #3: When worker threads leave the parallel region, they are independent of the activities of the master thread during the parallel region.** There is no data or control dependency that needs to be enforced between the master's computation during the parallel region and the workers past the parallel region. As such, the release phase of the join barrier is a redundant synchronization step.

Figure 4(C) illustrates the observations schematically. Compared with Figure 4(B), redundant synchronization is removed as the master thread signals workers that new work has arrived without waiting for acknowledgement of the workers. Similarly, once workers have confirmed the completion of their work, there is strictly no need for the master to acknowledge receipt of the message to the workers. This way, the work distribution and synchronization overhead is reduced to one instead of two barriers.

### 3.2 | NUMA-optimized half-barrier

The performance of barriers is dependent on the read and write patterns to shared memory locations used by the barrier. Scalable algorithms ensure that each shared variable is read by at most one thread and written by at most one thread.<sup>17</sup> The most efficient barrier styles are the dissemination barrier<sup>18,19</sup> and the tree barrier.<sup>17,19,20</sup>

Tree barriers are the most efficient on architectures with broadcast capability in the cache coherence protocol, for example, those using snooping protocols.<sup>20</sup> As our experimental platform uses a snooping bus, we focus on tree barriers.

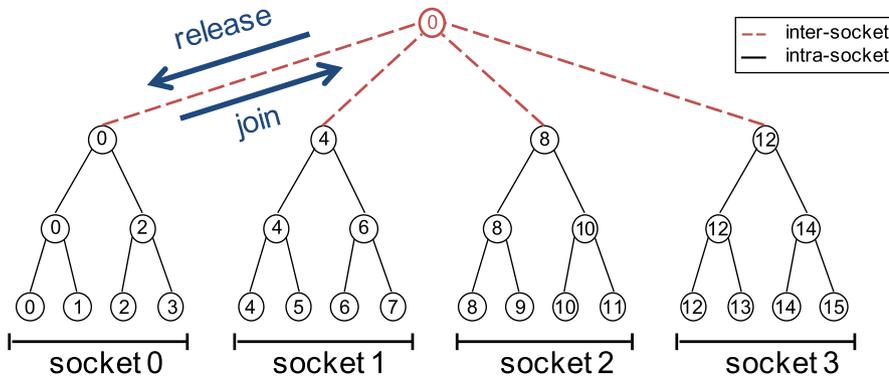
Tree barriers organize mini-tournaments between pairs of threads. The nodes of the tree implement centralized barriers for two threads. Threads perform the join phase of the centralized barriers in each node on the path from their leaf node to the root during the join phase. They traverse the tree from the root down to their leaf during the release phase.<sup>17,19</sup> The tree barrier has a critical message path of  $\lceil \log_2 P \rceil$ , which is similar to the dissemination barrier. However, during each round at most two messages are sent per thread. This makes it more bandwidth-efficient, which is important for bus-based architectures.

We have optimized the half-barrier for best performance on our experimental machine, which is a four-socket multicore (see Section 5 for details). We experimented with combinations of the centralized and tree barriers. The best performing configuration uses a tree barrier with distinct branching factors (Figure 5). At the top-most level, the branching factor equals the number of sockets. Each subtree below this level is bound to a specific socket and has a branching factor of 2. We found no benefit from using different branching factors in the join and reduce phases as previously suggested.<sup>20</sup>

### 3.3 | Supporting nested parallel loops

We design a mechanism for nested parallelism specifically to support NUMA-aware applications, such as the graph analytics code we evaluate. It is not necessary to support nested parallelism in the general case. For a fine-grain loop nested inside an outer parallel loop, the outer loop will contain sufficient work (at least a sequential execution of the fine-grain loop) to merit parallelization. As such, sequential execution of fine-grain loops in the presence of outer-loop parallelism is, for most applications, efficient.

Our design can be extended to nested parallelism for NUMA systems with minor modifications. The purpose of this is to enable NUMA-aware workload distribution. In order to retain efficiency, however, we apply some restrictions on the flexibility of nested parallelism. We design for two levels of parallelism. The same principles can be applied for multiple nesting levels; however, two levels is sufficient for most loop-parallel applications.



**FIGURE 5** NUMA-aware tree barrier. Each leaf node is associated to a thread, numbered 0–15 in this example. Internal nodes of the tree represent point-to-point synchronization actions between the threads listed in the node's children

We structure the applications following the tree barrier (Figure 5). We assume that the outer parallel loop creates one thread per NUMA domain. These threads will act as the master on their respective NUMA domain. When such a thread initiates the inner parallel loop, it is executed by the threads associated to the same NUMA domain. This allows us to reuse the tree barrier for synchronization: the outer parallel loop uses only the intersocket part of the tree, while the inner loop synchronizes on the intrasocket part.

### 3.4 | Workload imbalance

Some loops have workload imbalance, that is, some iterations take more time to complete than others. Loops with workload imbalance cause issues for parallelization as threads assigned less work sit idle while others complete. A commonly applicable resolution to this solution is to schedule the loop dynamically. However, this is not efficient for fine-grain loops. Alternatively, it may be possible to leverage statically known information. For instance, when it is known that some loop iterations require more computation than others, chunks can be chosen such that the work packed in each chunk is balanced. Such an issue occurs when dealing with longer rows in a triangular matrix. In this case, chunk sizes can be chosen at compile time such that the total number of elements per chunk is balanced, as opposed to balancing the number of rows per chunk. Similar issues occur also in graph processing, where partitioning techniques are essential to achieve load balance regardless of scheduling policy.<sup>21</sup>

### 3.5 | Integration with Cilk scheduler

We extend the Cilk work stealing algorithm such that applications can utilize both static scheduling for fine-grain loops and dynamic scheduling for coarse-grain or nested loops. Normally, an idle Cilk worker thread executes the random work stealing algorithm, whereby a worker randomly selects a victim worker and attempts to steal inactive work items from its work queue. The work steal attempt is successful when the victim's queue is not empty. Execution of the stolen stack frame is resumed and, when the worker returns to being idle, the process is repeated.

Scheduling of fine-grain loops is distinct from this process. When a worker initiates a fine-grain loop, it first checks if it needs to be executed in parallel or sequential. Hereto, it checks if the fine-grain loop is initiated from a full frame with outstanding parallelism, or not. If the full frame has outstanding parallelism (i.e., there exist sibling tasks), then the fine-grain loop is nested inside other loops and will be executed sequentially. Otherwise, it will be executed in parallel. In the case of sequential execution of the fine-grain loop, the scheduler executes the fine-grain loop sequentially. Note that a parallel loop is encoded in Cilk using a function pointer, start and end values for the loop iteration range and an opaque pointer to hold application-specific state. Executing the loop sequentially thus consists of calling this function once on the complete iteration range.

In the case of parallel execution of the fine-grain loop, the scheduler sets up control variables, including a task descriptor containing the loop's function pointer, per-thread iteration range and pointer to application-specific state. It then proceeds with executing its own portion of the fine-grain loop. For the other workers to join execution of the fine-grain loop, they need to become aware that a fine-grain loop is to be executed. Hence, when doing random work stealing, the workers will check the control state of the fine-grain scheduler and switch over to contribute to executing the fine-grain loop. When the fine-grain loop is complete, they return to the work stealing policy of the Cilk scheduler.

Figure 6 shows the work stealing loop, where the Cilk work stealing protocol is executed repeatedly by the function `check_for_work`. This loop moreover controls whether workers should sleep in case the degree of parallelism is low (SCHEDULE\_WAIT case), or whether they should exit in case the runtime shuts down (SCHEDULE\_EXIT case). The calls to `fg_slave_protocol` are inserted to alternate a cycle of the random work stealing algorithm with listening on the worker's flag variable in the tree barrier. Eventually, one of these actions will succeed and the worker continues either the fine-grain or baseline scheduling protocol. This incurs a minor overhead on either scheduler type as additional instructions are executed only when the worker is idle.

**FIGURE 6** Hybrid scheduling algorithm for Intel Cilkplus. The function `fg_slave_protocol` polls the worker's flag in the tree barrier and returns immediately in the absence of work

```

1  full_frame *ff = 0;
2  while(!ff) {
3      switch(worker_runnable(w)) {
4          case SCHEDULE_RUN:
5              ff = check_for_work(w); // work stealing protocol
6              if( !ff )
7                  fg_slave_protocol (w); // quickly probe for fine-grain loops
8              break;
9          case SCHEDULE_WAIT:
10             fg_slave_protocol (w); // quickly probe for fine-grain loops
11             suspend thread until work stealing becomes active again;
12             break;
13          case SCHEDULE_EXIT:
14             return NULL;
15         }
16     }
17     // resume ff

```

The placement of the calls to the fine-grain scheduler are motivated by the assumptions we make on the code structure. As fine-grain loops should be top-level loop constructs, we expect not much other parallel activity and workers may often find themselves in a suspended state. Hence, the master in the fine-grain scheduler will wake up all workers after setting up the control variables that describe the fine-grain loop. When workers wake up, they will execute a cycle of the fine-grain scheduler that contributes to the execution of the fine-grain loop. Workers may also remain active (state `SCHEDULE_RUN`) outside of fine-grain parallel loops. This state is determined by the Cilk scheduler based on the success of random work stealing. Hence, we also cycle the fine-grain scheduler upon a failed work stealing attempt. Note that the latter call to the fine-grain scheduler is not necessary for correctness reasons, nor for deadlock avoidance. However, performing the check while in the `SCHEDULE_RUN` state results in more efficient execution compared with cycling the work stealing algorithm until the worker drops down to the `SCHEDULE_WAIT` state.

## 4 | REDUCTIONS

Up till now we have considered loops without reductions. The half-barrier can be leveraged also to optimize reductions. In the case of Cilk, we can optimize further by simplifying the dynamic nature of reducers to a statically defined, and more efficient, pattern.

### 4.1 | Efficient barrier-based reductions

As described above, a statically scheduled OpenMP loop is typically implemented using a fork barrier to initiate processing of the loop and a join barrier to complete the loop (Figure 4(B)). When the loop has reduction variables, the Intel OpenMP runtime executes an additional tree barrier to aggregate per-thread results. The technique is simple: as each thread completes, it seeks to synchronize with other threads following a tree structure as in Figure 5. At each step of the join phase of a tree barrier, the per-thread values of a reduction variable are reduced when the threads synchronize. The resulting value is carried to the next synchronization step until the root of the tree is reached, when only the final value remains.

Our runtime optimizes performance further by merging the reduction operation in the final half-barrier that joins all threads. As such, a parallel loop requires two half-barriers only, whereas the Intel OpenMP runtime performs two full barriers per parallel loop, or three when the loop has reductions. We will demonstrate that such a small difference matters at scale.

### 4.2 | Efficient Cilk reducers

We aim to execute Cilk reducers equally efficient as the reduction variables in OpenMP without compromising the programming interface. To understand how this works, we need to delve deeper into the semantics of reducers and how reducers are implemented.

## 4.2.1 | Semantics of reducers

Cilk supports reductions through a type of hyperobject<sup>14</sup> called *reducers*. Each thread has a local view of the reducer which is either created or passed on by the parent thread at `cilk_spawn`. The semantics of Cilk reducers do not require them to be commutative; they only need to be associative as Cilk ensures that the left-right swap of arguments never occurs during execution. As such, for an operation  $a \oplus b$ , if  $a$  is evaluated after  $b$ , Cilk would first reduce  $b$  into a temporary variable  $l$  (which is initialized with the identity value of the reduction) by calculating  $l \oplus b$ , and after  $a$  has been computed, it will calculate  $a \oplus (l \oplus b)$  which is identical to  $a \oplus b$ .

## 4.2.2 | Application binary interface

The Cilk compiler replaces Cilk keywords with Application Binary Interface (ABI) calls.<sup>22</sup> Figure 7 shows equivalent C++ code for Figure 3 rather than assembly code to ease the exposition. The `accumulate_data` structure is a capture for the free variables in the loop body. The `accumulate_helper` function sequentially executes a subrange of the loop as given by its arguments. An ABI call to `__cilkrts_hyper_lookup` is used to retrieve the current view for the reducer. Note that this call is made from within the `cilk::reducer` class and is not inserted by the compiler. The compiler however hoists this call out of the loop for performance reasons.<sup>22</sup> Finally, the loop is replaced by the `__cilkrts_cilk_for_32` ABI call that enables parallel execution of the loop. It recursively decomposes the loop iteration range and executes the helper function on short iteration ranges.

## 4.2.3 | Reducers in the fine-grain scheduler

Fine-grain parallel loops require two changes in the compilation process (Figure 8): (i) a new ABI function `__cilkrts_cilk_for_static_32` is called; (ii) some preparatory work is performed to optimize the lookup of reducers, and the `__cilkrts_hyper_lookup` calls is replaced by a version tuned to the parallel pattern.

To avoid dynamic memory management overhead, the fine-grain parallel loop ABI method preallocates the views, allocating one array of views for each reducer used in a parallel loop (Figure 9). The views are aligned to cache-line boundaries to avoid cache line sharing between processors, which may lead to ping-pongs in the coherence protocol. Each thread initializes its own views prior to executing the loop body.

All arrays of views are stored in a *hyperarray* to facilitate lookup of views. Client code looks up a view through a new ABI call, `__cilkrts_hyper_array_lookup`, that takes as argument a hyperarray (an array of pointers to reducers) and the index of the requested reducer in the hyperarray. The index is assigned by the compiler. Figure 8 shows the construction of the hyperarray (line 14) and the lookup of views (line 7).

As in the OpenMP case, views are reduced and destroyed as part of pairwise thread synchronization in the join phase of the tree barrier. By pro-actively creating instances of the reduction variable for each thread, we perform exactly  $P - 1$  reduction operations for  $P$  threads. In contrast, Cilk performs as many reductions as successful work steal events,<sup>23</sup> which may be significantly higher than  $P$ . In the case of coarse-grain and potentially imbalanced loops, dynamic management of reducers results in economy. In the case of static scheduling, however, a regularly structured design is more efficient.

This technique applies to Cilk's noncommutative reducers provided that every thread is assigned a contiguous range of loop iterations. The correctness of noncommutative reductions is ensured as the master thread assigns iterations to threads in such a way that at every

```

1 struct accumulate_data {
2     int *a;
3     cilk::reducer<cilk::op_add<int>> *r;
4 };
5 void accumulate_helper( void *data, int start, int end ) {
6     accumulate_data * d = (accumulate_data *)data;
7     int * view = __cilkrts_hyper_lookup ( d->r );
8     for(int i = start; i < end; i++)
9         *view += (d->a)[i];
10 }
11 int accumulate( int *a, int n ) {
12     cilk::reducer<cilk::op_add<int>> r;
13     struct accumulate_data data = { a, &r };
14     __cilkrts_cilk_for_32 ( accumulate_helper, (void *)&data, n, 0 );
15     return r.get_value ();
16 }

```

FIGURE 7 Code transformed by Cilk compiler

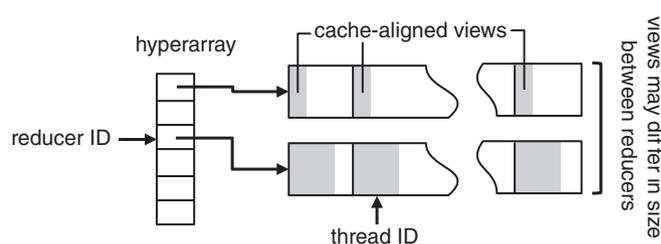
**FIGURE 8** Equivalent code of the accumulation loop with fine-grain pragma

```

1 struct accumulate_data {
2     int *a;
3     __cilkrts_hyperobject_base **hyper_array;
4 };
5 void accumulate_helper( void *data, int start, int end ) {
6     accumulate_data * d = (accumulate_data *)data;
7     int * view = __cilkrts_hyper_array_lookup (d->hyper_array, 0);
8     for(int i = start; i < end; i++)
9         *view += (d->a)[i];
10 }
11 int accumulate( int *a, int n ) {
12     cilk :: reducer<cilk :: op_add<int> > r;
13     __cilkrts_hyperobject_base * obj [1] = { &r }; // r at index 0
14     struct accumulate_data data = { a, obj };
15     __cilkrts_cilk_for_static_32 (
16         accumulate_helper, (void *)&data, n, 0, obj, 1);
17     return r.get_value ();
18 }

```

**FIGURE 9** The hyperarray lays out views in consecutive, cache-aligned memory locations. It is indexed using the reducer's numeric ID within the loop and the thread ID



reduction step, which corresponds to an internal node of the tree barrier, the data produced by the left child results from lesser loop iterations than the data produced by the right child. This constraint needs to be applied recursively while walking up the tree during the join phase.

#### 4.2.4 | Restrictions to fine-grain Cilk loops

It is required that the compiler can list all hyperobjects accessed within a fine-grain loop. This requires that:

- The compiler can inspect all code executed in the loop body. A sufficient solution is that all functions called from the loop body can be disambiguated at compile-time (they are not called through function pointers), and are part of the current compilation unit.
- Reducers may not be created or destroyed during loop execution.
- Addresses of hyperobjects are loop-constant. This implies that the hyper-lookup calls can be hoisted out of the loop.<sup>22</sup>
- All (pointers to) hyperobjects referenced in the loop concern distinct hyperobjects. In practice, most loops have only one reducer.

These constraints are tested at compile-time.

## 5 | EXPERIMENTAL EVALUATION

We evaluate our schedulers on a four-socket 2.6 GHz Intel Xeon E7-4860 v2 machine with 12 physical cores per socket (plus hyperthreading) and 30 MB L3 cache per socket. We pin threads to cores so that only one thread per core is used. The operating system is CentOS 7.0. We use the Intel C/C++ compiler v. 14.0.0 for OpenMP programs and implemented compiler support for the Cilk runtime in clang version 3.4.1. Note that our evaluation system is not very large, but already shows the impact of scheduling burden.

We report results averaged over 15 program runs and calculate speedup against the sequential version of the benchmark with no parallel constructs. This method correctly penalizes parallel runtimes for any overhead they may induce. Moreover, we exclude the runtime startup time in all speedup results.

## 5.1 | Scheduling burden

We use a micro-benchmark (**uforall**) to measure loop scheduling overhead. By varying the amount of work in the parallel loop, we can emulate loops of different granularities. Figure 10 shows the speedup obtained for varying granularity (sequential execution time) of the loop for the OpenMP- and Cilk-based runtimes. Speedup is one for extremely short loops and grows gradually with increasing loop granularity until it reaches maximum speedup. Note that the horizontal axis is displayed on log-scale. The curves do not extend to 48-fold speedup due to instrumentation overhead.

The micro-benchmark results visualize the burden of the scheduler. The OpenMP static scheduler has smaller burden than OpenMP dynamic scheduler, which in turn has smaller burden than the Cilk scheduler. Note that these are properties of the implementation and may vary between implementations of OpenMP.

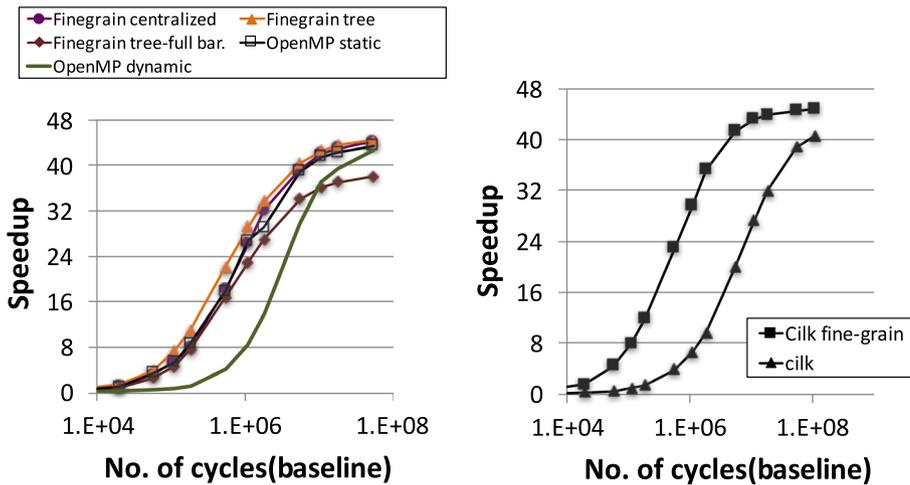
We estimate the scheduling burden using Amdahl's Law:

$$S = \frac{T}{d + T/48},$$

where  $T$  is the sequential execution time,  $d$  is the work distribution time, and  $S$  is the resulting speedup. We perform a least-squares fit between the experimental data and the model to estimate  $d$ . The burden is several micro-seconds (Table 1). The dynamic schedulers have a significantly higher burden than the static ones. In particular, Cilk has a high burden as it is designed for coarse-grain fork/join parallelism with potential load imbalance. It implements parallel loops on top of this construct, leading to measurable overhead.

Our fine-grain scheduler has a burden that is 43% lower than OpenMP and 12.1× lower than Cilk. Moreover, using a half barrier in the fine-grain scheduler reduces the scheduling delay by about half compared with a full barrier.

We furthermore calculate the work required to achieve 5% and 95% of peak speedup for each of the schedulers, and to achieve 24-fold speedup (Table 1). The static schedulers require a sequential loop duration of about 1.5  $\mu$ s to start observing minimal speedup, and of 80–90  $\mu$ s to achieve near-maximum speedup. The dynamic schedulers require a sequential loop duration of around about 16  $\mu$ s to observe any speedup and reach near-maximal speedup for loops longer than 200  $\mu$ s. As such, static schedulers benefit loops up to 200  $\mu$ s long. For example, a loop taking 80  $\mu$ s would see a twofold acceleration when scheduled by a static scheduler (where it achieves near-maximum speedup) compared with Cilk (where it achieves



**FIGURE 10** Speedup of **uforall** micro-benchmark for the OpenMP (left) Cilk (right) runtimes

**TABLE 1** Characterizing scheduler burden

	$d$ ( $\mu$ s)	95% peak ( $\mu$ s)	24× (~50% peak) ( $\mu$ s)	5% peak ( $\mu$ s)
Fine-grain tree	5.67	81.64	21.99	1.49
Fine-grain centralized	7.55	87.13	24.42	1.57
Fine-grain tree with full-barrier	12.00	88.03	28.54	1.62
OpenMP static	8.12	84.30	24.25	1.54
OpenMP dynamic	31.94	209.38	41.54	16.35
Cilk	68.80	264.07	75.58	16.51

24x speedup). Similarly, a sequential loop of about 25  $\mu$ s observes minimal speedup on the dynamic schedulers, but would observe a 24x speedup on a static scheduler.

## 5.2 | Fine-grain parallelism in HPC codes

We analyze the performance of our fine-grain scheduler on three stencil codes with varying characteristics. We use the OpenMP scheduler for these workloads. Table 2 summarizes the characteristics of these workloads. The workloads have high coverage by parallel loops. The majority of loops are fine-grain loops with sequential execution time less than 1 ms.

### 5.2.1 | MPDATA

The Multidimensional Positive Definite Advection Transport Algorithm (MPDATA) is a 2D stencil over an unstructured grid<sup>24</sup> implemented using the scalable Atlas library for numerical weather prediction.<sup>25</sup> This code is used in production by the European Centre for Mid-range Weather Forecasting (ECMWF), where it is applied to a 6.5 million point unstructured grid. The grid is distributed over 1440 MPI tasks with approximately 4800 grid points per task. We model the work performed by one MPI task using a slightly coarser grid with 5568 points and 16,399 edges.

Figure 11 (left) shows the parallel speedup of MPDATA. MPDATA is typically run with an OpenMP parallelism of 4, where a 2.96x speedup is achieved. Parallel speedup stagnates quickly, however, with increasing parallelism, especially from 12 threads onwards.

Scalability is an important issue for numerical weather prediction as these codes need to run under strong scaling conditions. Weak scaling increases the amount of work superlinearly due to an interaction between the granularity of the grid and the time step. Figure 11 shows that the code has already been taken to the limits of strong scalability. Under these conditions, the fine-grain scheduler increases performance by up to 22% over the off-the-shelf Intel OpenMP runtime. The speedup increases steadily as the thread count improves.

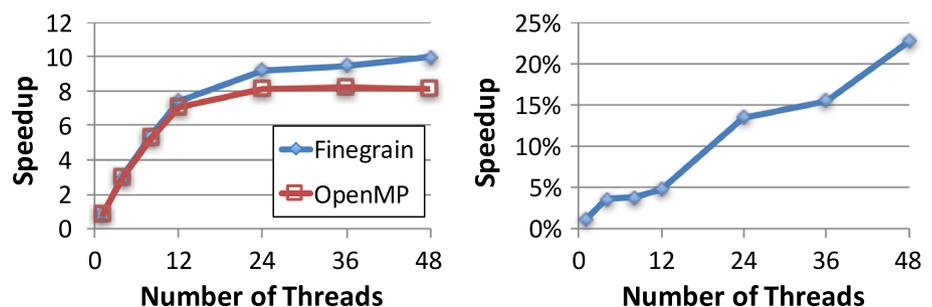
### 5.2.2 | LB3D

LB3D implements a lattice Boltzmann computation over a 3D structured grid (LB3D).<sup>26</sup> Figure 12 (left) shows the parallel speedup of LB3D over a grid of size  $32^3$ . Perfect linear scaling would result in a parallel speedup of 48 for 48 threads, which is typically not achieved in real scenarios. Compared with the Intel OpenMP runtime, our fine-grain scheduler improves overall execution time, and hence the speedup. The improvement moreover increases with increasing thread count as scheduler burden increases with higher degrees of parallelism.

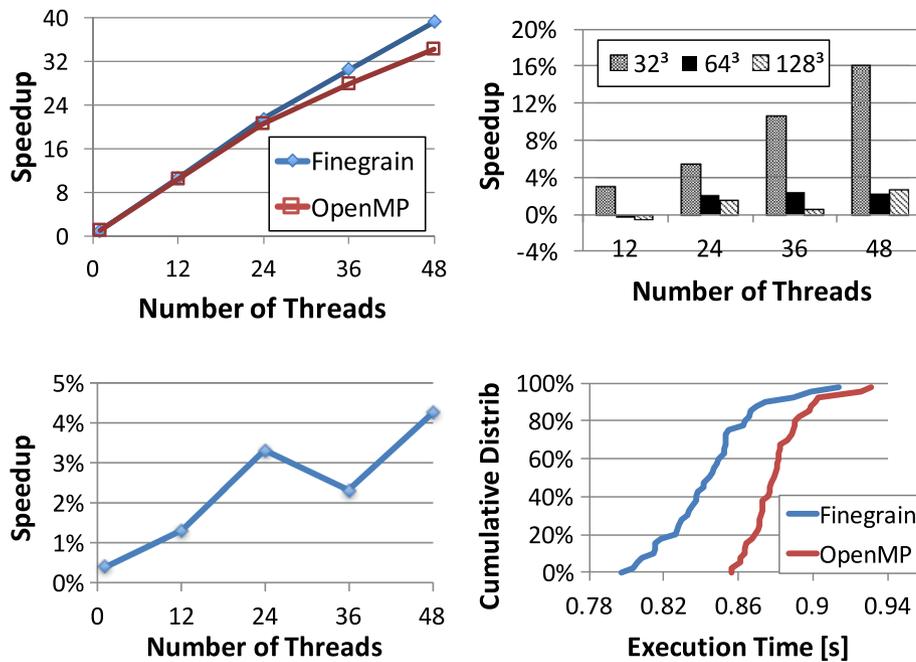
**TABLE 2** Performance characteristics of HPC codes

Method	$T_{seq}$	Parallel loops		Time per loop			Loops <1 ms		
		Number	%Time	Min	Avg	Max	Number	Total time	
CloverLeaf	2D staggered, structured grid	17.29 s	111	98.87	0.95 $\mu$ s	2.71 ms	46.32 ms	69	0.12 s
MPDATA	2D unstructured grid	22.0 s	36	88.64	0.72 $\mu$ s	21.29 ms	753.84 ms	32	17.93 s
LB3D	3D structured grid	4.92 s	2	94.85	45.11 $\mu$ s	2.34 ms	4.63 ms	1	45.11 ms

Note: Statistics are shown for sequential execution time, all parallel loops and fine-grain loops. In this table, fine-grain loops are loops taking less than 1 ms sequential execution time.



**FIGURE 11** Analysis of MPDATA: parallel speedup of the fine-grain and OpenMP schedulers (left); speedup of fine-grain scheduler over OpenMP (right)



**FIGURE 12** Analysis of LB3D: Speedup for a  $32^3$  grid (left); speedup of the fine-grain scheduler over the baseline OpenMP scheduler (right)

**FIGURE 13** Performance of CloverLeaf: Speedup of the fine-grain scheduler over baseline OpenMP (left), distribution of 40 execution times at 48 threads (right)

The LB3D code has a speedup of 34.1 over sequential execution at 48 threads when using the Intel OpenMP runtime and represents good parallel performance. The fine-grain scheduler improves the speedup further to 39.3 at 48 threads, a 16% improvement.

The techniques proposed in this work are designed specifically for fine-grain parallel loops. As such, we expect small or no benefits for larger problem sizes. Figure 12(right) shows the speedup of the fine-grain scheduler for varying problem sizes. The  $32^3$  case shows good speedups as discussed above. The larger problem sizes see no speedup at 12 threads (there is some variation on the execution time resulting in a 0.58% slowdown for the  $128^3$  grid) and small but consistent speedups for 24 threads and higher. This demonstrates that (i) the fine-grain scheduler need not necessarily lose performance for coarser-grain loops; (ii) higher thread counts expose a higher scheduler burden even for larger problem sizes.

### 5.2.3 | CloverLeaf

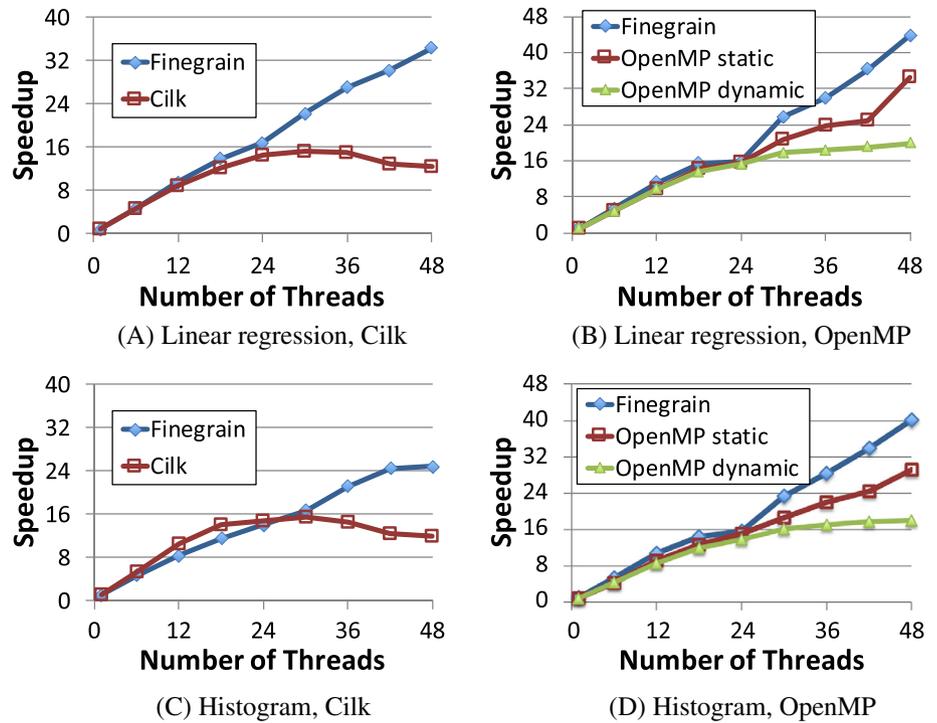
CloverLeaf<sup>27</sup> is a hydro-dynamics mini-app that solves the compressible Euler equations in 2D using an explicit, second-order method. CloverLeaf uses a 2D structured, staggered grid of size  $960^2$ . Speedup over sequential execution comes out at 18.8 at 48 threads with the Intel OpenMP runtime. The performance improvement of the fine-grain scheduler exceeds 4% (Figure 13, left). More importantly, it shows a growing trend with increasing thread count. The distribution of the execution times of the baseline OpenMP and fine-grain schedulers shows that the improvement in execution time is more important than the performance variation (Figure 13, right).

### 5.2.4 | Discussion

We have presented three distinct stencil computations with varying complexity (2D vs. 3D, structured vs. unstructured grid, staggered grid). These codes contains many fine-grain loops, which are repeatedly executed for each time step. Space-time decomposition schemes have been proposed to improve memory locality<sup>28,29</sup> and simultaneously increase the granularity of the parallel loops. These techniques are, however, very hard to apply in real codes as they require a significant restructuring of the time step loop. Such a transformation is feasible, though complex, for a stencil with a single loop (e.g., LB3D). In many other stencil codes, such as MPDATA and CloverLeaf, time-space decomposition becomes overly complex as staggered grids are used and each time step contains over a dozen loops.

## 5.3 | Reductions

We consider two map reduce benchmarks with reductions to evaluate the implementation of reductions in our Cilk and OpenMP runtimes. We evaluate linear regression and histogram workloads from Phoenix++<sup>30</sup> on the medium and small inputs, respectively. In all cases, the burden on the

**FIGURE 14** Performance analysis of map reduce workloads**TABLE 3** Performance of graph analytics workloads and statistics on duration of fine-grain loops

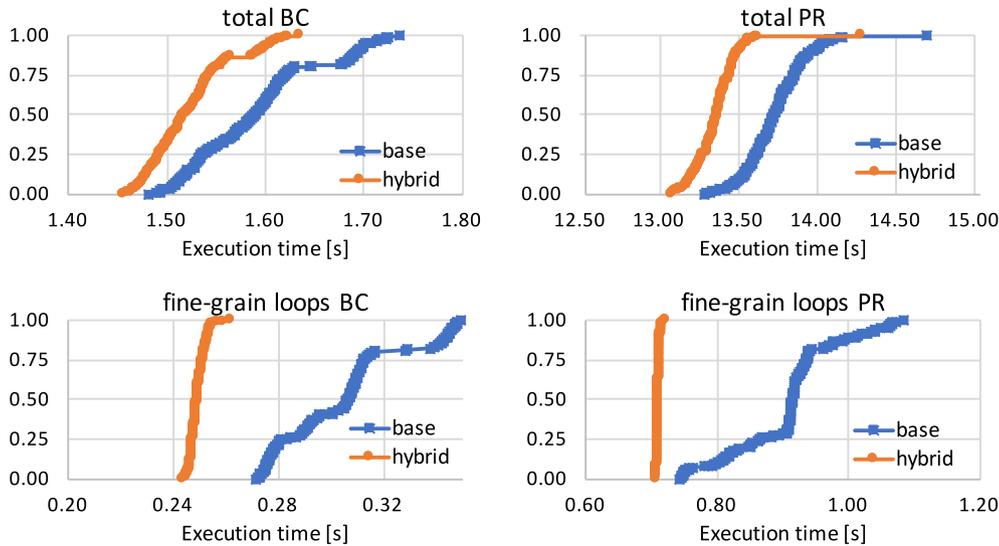
	Fine-grain loops		Time per fine-grain loop			Time Cilk		Time hybrid		Speedup	Speedup
	Number	<0.1 ms	Min	Avg	Max	Total	Fine-grain	Total	Fine-grain	Total	Fine-grain
BFS	73	48	1.84 $\mu$ s	1.27 ms	21 ms	0.387	0.0928	0.376	0.0763	2.75%	21.6%
BC	148	33	0.77 $\mu$ s	2.06 ms	27.2 ms	1.59	0.304	1.53	0.250	4.35%	21.9%
CC	118	93	0.44 $\mu$ s	1.32 ms	21.6 ms	2.13	0.155	2.10	0.129	1.48%	29.6%
PR	82	17	45.9 $\mu$ s	11.1 ms	19.9 ms	13.7	0.907	13.4	0.711	2.73%	27.5%

baseline Cilk and OpenMP schedulers is high, leading to a slow down of speedup as the thread count increases (Figure 14). The fine-grain scheduler is able to maintain the speedup. This is due to a combination of reducing scheduling overhead and efficient implementation of the reduction in the Cilk runtime. This leads to a best-case improvement of 2.8 $\times$  for linear regression.

## 5.4 | Nested parallelism

We apply the hybrid static/dynamic scheduler to Ligra,<sup>4</sup> a graph analytics system. Ligra contains phases of coarse-grain parallelism separated by fine-grain parallel operators such as prefix-scan and reduce over arrays of highly varying length, ranging from tens to millions of elements. We use it to analyze the ability of the Cilk scheduler to switch efficiently between coarse-grain and fine-grain parallelism. We moreover distribute loop iterations in a sparse manner over NUMA nodes. We use this technique to launch one thread per NUMA node, which then acts as the master for a nested fine-grain parallel loop. Other fine-grain loops are launched from sequential code and involve all threads.

Table 3 summarizes the execution times of four graph analytics kernels processing the Twitter graph.<sup>31</sup> We execute each workload 200 times and separate out the time spent in the fine-grain parallel loops and compare against an execution using the baseline Cilk runtime. The low-overhead scheduler speeds up fine-grain loops by 21.6%–29.6%, which is a speedup similar to that of MPDATA and LB3D. A key difference with those workloads is, however, that the graph analytics code also contains coarse-grain loops. The fine-grain loops cover between 6.6% and 24% of the execution time, depending on the graph analytics kernel. As such, Amdahl's Law dictates that the whole application speedup is considerably less. Measurements show whole application speedup of 1.5%–4.4%.



**FIGURE 15** Cumulative distribution of total execution time (top) and execution time of fine-grain loops (bottom) for BC and PR using the Cilk scheduler and the hybrid scheduler. The distribution summarizes 200 runs

Fine-grain loops in the graph analytics are very short (Table 3, columns 2–6): the shortest loops take less than a microsecond (2600 CPU cycles on our system) and almost half of the fine-grain loops take less than 0.1 millisecond. A few loops take 20–30 ms, which skews the average loop duration statistic.

Figure 15 shows the cumulative distribution of the time spent in fine-grain loops and total application time across 200 executions. The fine-grain loops are clearly accelerated as all, or nearly all, of the measurements on the hybrid scheduler are smaller than those on the baseline scheduler. Moreover, the variance of execution time on the static scheduler is less as the slope of the curves is steeper.

The distributions moreover demonstrate that the static scheduler does not suffer from stragglers. In fact, the baseline Cilk scheduler is more susceptible to long delays. We believe this results from random work stealing in the Cilk scheduler, as the granularity of stolen work depends on the selected victim, and with it the number of work stealing events and ensuing runtime management.

## 5.5 | Selecting fine-grain loops

The programming interface requires that programmers identify which loops are fine-grain. This may appear hard to judge at first sight. For this work, we have used the following approach: we instrumented every parallel loop to measure its execution time under sequential execution. If the loop takes on average less than 100 ms per invocation of the loop (assuming a sequential execution of the loop), we annotate it as fine-grain. In practice, this implies that all loops in MPDATA and CloverLeaf are fine-grain. LB3D, linear regression and histogram contain just one parallel loop. In the graph analytics code we annotate all loops over (sub)sets of vertices as fine-grain and loops over edges as coarse-grain. Typically there are 10 or more edges per vertex in the target graph, which results in a clean separation of execution times of the two types of loops.

It is not a stretch for HPC programmers to manually identify an appropriate scheduler for parallel loops. OpenMP provides this interface and programmers use knowledge of high-level characteristics of the code (e.g., load imbalance) to select a scheduler. It is also possible to devise runtime techniques that estimate or predict the most appropriate scheduler for each loop, for example, by tracking average loop execution time and load balance.

## 6 | FURTHER RELATED WORK

The need for scheduling fine-grain parallelism is widely recognized in the literature.

Eichenberger and O'Brien<sup>32</sup> demonstrate a fivefold reduction of parallel overhead using a variety of techniques, among others a bit-vector-based thread activation technique to signal active workers. The article, however, does not describe propose a comparable technique for worker threads to communicate back the completion of work.

Task schedulers place inactive tasks on work queues, from where they can be distributed to worker threads through pro-active work distribution or through work stealing. Work queues are implemented with atomic operations to minimize lock contention.<sup>33–35</sup> Active worker threads are isolated from interference of work stealers by operating on distinct ends of a *double-ended queue* (shorthand: deque). This way Cilk avoids acquiring locks on the critical path.<sup>36</sup> Lazy binary splitting<sup>37</sup> attempts to reduce the burden further by creating tasks lazily. Workers periodically check if their deque

runs empty and re-plenish it by breaking off a chunk of the work item they are currently working on. Legion<sup>38</sup> uses logical regions to decouple task dependencies from memory locations. HPX<sup>39</sup> leverages localities to localize synchronization in a distributed system.

Galois is a system for optimistic execution of amorphous parallelism.<sup>40</sup> Its performance is highly dependent on the efficiency of concurrent data structures in the runtime, which have a much higher variety and complexity than dequeues. Campanoni et al. describe a runtime system that uses hyperthreads to prefetch cache blocks used in communication.<sup>41</sup> They have, however, not published their code.

It is possible to alleviate part of the problem by tuning schedulers and runtime systems to application properties. Nguyen et al.<sup>42</sup> propose a language to synthesize concurrent schedulers for irregular algorithms. GRAMPS optimizes the execution of pipeline-parallel programs by differentiating between stateful and stateless pipeline stages.<sup>43</sup> It furthermore performs buffer management to limit runtime overhead. Donack et al. optimize a communication-avoiding LU decomposition algorithm by scheduling a subset of the tasks dynamically. This limits overhead for most tasks but retains load balancing capabilities.<sup>44</sup> This approach, however, makes applications dependent on hardware properties and vulnerable to performance portability issues.

Several researchers have previously combined static and dynamic scheduling in a bid to combine low overhead with scheduling flexibility. DoPE<sup>45</sup> adapts the degree of parallelism by switching dynamically between compiler-generated static schedules. In Reference 46, statically scheduled sets of streaming kernels are connected using a dynamic scheduler to overcome language limitations. In contrast to these works, we make static work scheduling more efficient.

Callisto<sup>47</sup> is a dynamic scheduler designed to be scalable to a high thread count (1024 threads on an Oracle Niagara T2). Work is distributed from a centralized work queue. Threads, however, request work on behalf of neighboring threads in order to reduce contention at the queue.

Bull et al. define a micro-benchmark to analyze the overhead incurred by the *task* construct in OpenMP runtimes.<sup>48</sup> Contrary to our work, they define the benchmark to increase the work of the loop by a factor  $P$  for  $P$  threads. They measure the increase in execution time compared with sequential execution to estimate the runtime overhead. They suggest their analysis can be used to identify a minimum granularity on tasks; however, they kept the task granularity fixed at 0.1  $\mu$ s. While our experiments focus on *strong scaling* of schedulers, their work and the related<sup>49</sup> assume a *weak scaling* scenario.

Several authors consider hardware structures to facilitate interthread synchronization, for example, the synchronization array,<sup>8</sup> hardware task queues and integrated scheduling policies,<sup>11</sup> and asynchronous direct messages<sup>13</sup> that support the efficient exchange of short messages between cores and serves as a basic primitive for building software schedulers. Sanchez et al.<sup>13</sup> furthermore demonstrate that hardware implementation of task queues and scheduling policies provides marginal performance benefits over their software counterparts.

## 7 | CONCLUSION

This article demonstrates that scheduler burden has an important overhead at high degrees of parallelism and already noticeably reduces application performance on current hardware. Addressing scheduler burden is essential to achieve strong scaling in future, highly parallel systems.

We have designed a loop scheduler for fine-grain parallelism with efficient, atomics-free work distribution using a half-barrier. We integrated the fine-grain scheduler in the Intel OpenMP and Cilkplus runtimes and designed compiler support to enhance the performance on reductions. Experimental evaluation demonstrates a consistent speedup for fine-grain loops between 16% and 30% over the baseline OpenMP and Cilk schedulers, with a peak of 2.8 $\times$  speedup. We have moreover observed that fine-grain scheduling reduces performance variability. The speedup grows with increasing thread count, indicating that future many-core processors will be even more susceptible to scheduler burden.

This work presents a first attempt at addressing scheduler burden. It identifies several open problems that should be addressed in future research: (i) Overcoming synchronization overhead imposed by hardware. We measured a 60 ns ping-pong delay between threads on the same chip, and nearly the same delay between hyperthreads. These delays need to be reduced to achieve efficient synchronization. (ii) Designing novel work distribution and synchronization techniques, and static and dynamic scheduling policies for low-burden runtime schedulers.

## ACKNOWLEDGMENTS

This research is supported by the EPSRC under grant agreement no. EP/L027402/1 and EP/M008495/1 mini-projects, by the European Union Seventh Framework Programme under grant agreement no. 327744 (NovoSoft, Marie Curie Actions) and no. 619706 (ASAP), and by the European Union Horizon 2020 Programme under grant agreement no. 732631 (OPRECOMP). The authors are grateful to Willem Deconinck and the European Centre for Mid-range Weather Forecasting (ECMWF) for access to and support with the MPDATA kernel.

## ORCID

Hans Vandierendonck  <https://orcid.org/0000-0001-5868-9259>

## REFERENCES

1. Sodani A, Gramunt R, Corbal J, et al. Knights landing: second-generation Intel Xeon Phi product. *IEEE Micro*. 2016;36:34-46.

2. He Y, Leiserson CE, Leiserson WM. The cilkview scalability analyzer. Paper presented at: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA '10; 2010:145-156; ACM, New York, NY.
3. Frigo M, Leiserson CE, Randall KH. The implementation of the cilk-5 multithreaded language. Paper presented at: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation PLDI '98; 1998:212-223; ACM, New York, NY.
4. Shun J, Blelloch GE. Ligma: a lightweight graph processing framework for shared memory. Paper presented at: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP '13; 2013:135-146; ACM, New York, NY.
5. <https://www.cilkplus.org>.
6. Barroso L, Marty M, Patterson D, Ranganathan P. Attack of the killer microseconds. *Commun ACM*. 2017;60:48-54.
7. Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. Paper presented at: Proceedings of the 35th Annual Symposium on Foundations of Computer Science; 1994:356-368; IEEE Computer Society; Washington, DC.
8. Rangan R, Vachharajani N, Vachharajani M, August DI. Decoupled software pipelining with the synchronization array. Paper presented at: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques PACT '04. Antibes Juan-les-Pins, France; 2004:177-188.
9. Gordon MI, Thies W, Karczmarek M, et al. A stream compiler for communication-exposed architectures. Paper presented at: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-X. San Jose, CA; 2002:291-303.
10. OpenMP Application programming interface, version 4.5; 2015. <http://www.openmp.org/>.
11. Kumar S, Hughes CJ, Nguyen A, Carbon: architectural support for fine-grained parallelism on chip multiprocessors. Paper presented at: Proceedings of the 34th Annual International Symposium on Computer Architecture ISCA '07; 2007:162-173; ACM, New York, NY.
12. Etsion Y, Cabarcas F, Rico A, et al. Task superscalar: an out-of-order task pipeline in Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture MICRO '10; Vol. 43 2010:83-100; IEEE Computer Society, Washington, DC.
13. Sanchez D, Yoo RM, Kozyrakis C. Flexible architectural support for fine-grain scheduling. Paper presented at: Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems ASPLOS XV; 2010:311-322; ACM, New York, NY.
14. Frigo M, Halpern P, Leiserson CE, Lewin-Berlin S. Reducers and other Cilk++ hyperobjects. Paper presented at: Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures SPAA '09; 2009:79-90; ACM, New York, NY.
15. Vandierendonck H, Murphy K, Arif M, Sun J, Nikolopoulos DS. ASAP D2.3: program analysis and transformation Technical report.FP7 Project ASAP; 2017.
16. Arif M, Vandierendonck H. POSTER: reducing the burden of parallel loop schedulers for many-core processors. Paper presented at: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2018:383-384; ACM, New York, NY.
17. Lubachevsky BD. Synchronization barrier and related tools for shared memory parallel programming. *Int J Parallel Prog*. 1991;19:225-250.
18. Brooks ED. The butterfly barrier. *Int J Parallel Prog*. 1986;15:295-307.
19. Hensgen D, Finkel R, Manber U. Two algorithms for barrier synchronization. *Int J Parallel Prog*. 1988;17:1-17.
20. Mellor-Crummey JM, Scott ML. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans Comput Syst*. 1991;9:21-65.
21. Sun J, Vandierendonck H, Nikolopoulos DS. GraphGrind: addressing load imbalance of graph partitioning. Paper presented at: Proceedings of the International Conference on Supercomputing. Chicago, IL; 2017:1-10.
22. Intel Corporation Intel cilk plus application binary interface specification. Revision 1.1. Document number 324512-002US; 2011.
23. Leiserson CE, Schardl TB. A work-efficient parallel breadth-first search algorithm (or How to Cope with the Nondeterminism of Reducers). Paper presented at: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA '10; 2010:303-314; ACM, New York, NY.
24. Smolarkiewicz PK, Deconinck W, Hamrud M, et al. A finite-volume module for simulating global all-scale atmospheric flows. *J Comput Phys*. 2016;314:287-304.
25. Deconinck W, Bauer P, Diamantakis M, et al. Atlas: a library for numerical weather prediction and climate modelling. *Comput Phys Commun*. 2017;220:188-204.
26. McIntosh-Smith S, Curran D. Evaluation of a performance portable lattice Boltzmann code using OpenCL. Paper presented at: Proceedings of the International Workshop on OpenCL 2013 & 2014 IWOCL '14; 2014:2:1-2:12; ACM, New York, NY.
27. Mallinson AC, Beckingsdale DA, Gaudin WP, Herdman JA, Levesque JM, Jarvis SA. CloverLeaf: preparing hydrodynamics for exascale in Cray User Group; 2013.
28. Strzodka R, Shaheen M, Pajak D, Seidel HP. Cache accurate time skewing in iterative stencil computations. Paper presented at: Proceedings of the 2011 International Conference on Parallel Processing ICPP '11; 2011:571-581; IEEE Computer Society, Washington, DC.
29. Tang Y, Chowdhury RA, Kuszmaul BC, Luk CK, Leiserson CE. The Pochoir stencil compiler. Paper presented at: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). San Jose, CA; 2011:117-128.
30. Talbot J, Yoo RM, Kozyrakis C. Phoenix++: modular MapReduce for shared-memory systems. Paper presented at: Proceedings of the Second International Workshop on MapReduce and Its Applications MapReduce '11; 2011:9-16; ACM, New York, NY.
31. Kwak H, Lee C, Park H, Moon S. What is Twitter, a social network or a news media? Paper presented at: Proceedings of the 19th International Conference on World Wide Web; 2010:591-600; ACM, New York, NY.
32. Eichenberger AE, O'Brien K. Experimenting with low-overhead OpenMP runtime on IBM Blue Gene/Q IBM. *J Res Develop*. 2013;57:8:1-8:8.
33. Arora NS, Blumofe RD, Plaxton CG. Thread scheduling for multiprogrammed multiprocessors. Paper presented at: Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA '98; 1998:119-129; ACM, New York, NY.
34. Chase D, Lev Y. Dynamic circular work-stealing deque. Paper presented at: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures SPAA '05; 2005:21-28; ACM, New York, NY.
35. Morrison A, Afek Y. Fast concurrent queues for x86 processors. Paper presented at: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP '13; 2013:103-112; ACM, New York, NY.
36. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: an efficient multithreaded runtime system in Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming PPOPP '95; 1995:207-216.
37. Tzannes A, Caragea GC, Barua R, Vishkin U. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. Paper presented at: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP '10; 2010:179-190; ACM, New York, NY.

38. Bauer M, Treichler S, Slaughter E, Aitken A. Legion: expressing locality and independence with logical regions. Paper presented at: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis SC '12; 2012; IEEE Computer Society Press, Los Alamitos, CA.
39. Kaiser H, Heller T, Adelstein-Lelbach B, Serio A, Fey D. HPX: a task based programming model in a global address space. Paper presented at: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models PGAS '14; 2014:6:1-6:11; ACM, New York, NY.
40. Kulkarni M, Pingali K, Walter B, Ramanarayanan G, Bala K, Chew LP. Optimistic parallelism requires abstractions. Paper presented at: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation. San Diego, CA; 2007:211-222.
41. Campanoni S, Jones TM, Holloway G, Reddi VJ, Wei GY, Brooks D. HELIX: automatic parallelization of irregular programs for chip multiprocessing. Paper presented at: Proceedings of the International Symposium on Code Generation and Optimization (CGO). San Jose, CA; 2012:84-93.
42. Nguyen D, Pingali K. Synthesizing concurrent schedulers for irregular algorithms. Paper presented at: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS '11; 2011:333-344; ACM, New York, NY.
43. Sanchez D, Lo D, Yoo RM, Sugerman J, Kozyrakis C. Dynamic fine-grain scheduling of pipeline parallelism. Paper presented at: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques PACT '11; 2011:22-32; IEEE Computer Society, Washington, DC.
44. Donfack S, Grigori L, Gupta AK. Adapting communication-avoiding LU and QR factorizations to multicore architectures. Paper presented at: Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS). Atlanta, GA; 2010:1-10.
45. Raman A, Kim H, Oh T, Lee JW, August DI. Parallelism orchestration using DoPE: the degree of parallelism executive. Paper presented at: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '11; 2011:26-37; ACM, New York, NY.
46. Soulé R, Gordon MI, Amarasinghe S, Grimm R, Hirzel M. Hitting the sweet spot for streaming languages: dynamic expressivity with static optimization. Technical report TR2012-948, New York University; 2012.
47. Harris T, Kaestle S. Callisto-RTS: fine-grain parallel loops. Paper presented at: Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference USENIX ATC '15; 2015:45-56; USENIX Association, Berkeley, CA.
48. Bull JM, Reid F, McDonnell N. A microbenchmark suite for OpenMP tasks. Paper presented at: Proceeding of the International Workshop on OpenMP; 2012:271-274; Springer, New York, NY.
49. Bull JM. Measuring synchronisation and scheduling overheads in OpenMP. Paper presented at: Proceedings of 1st European Workshop on OpenMP Lund, Sweden; 1999;8:49.

**How to cite this article:** Arif M, Vandierendonck H. Reducing the burden of parallel loop schedulers for many-core processors. *Concurrency Computat Pract Exper*. 2021;33:e6241. <https://doi.org/10.1002/cpe.6241>