Callisto-RTS : Fine-Grain Parallel Loops

Tim Harris
Oracle Labs

Stefan Kaestle
ETH Zurich

Abstract

We introduce Callisto-RTS, a parallel runtime system designed for multi-socket shared-memory machines. It supports very fine-grained scheduling of parallel loops—down to batches of work of around 1K cycles. Fine-grained scheduling helps avoid load imbalance while reducing the need for tuning workloads to particular machines or inputs. We use per-core iteration counts to distribute work initially, and a new asynchronous request combining technique for when threads require more work. We present results using graph analytics algorithms on a 2-socket Intel 64 machine (32 h/w contexts), and on an 8-socket SPARC machine (1024 h/w contexts). In addition to reducing the need for tuning, on the SPARC machines we improve absolute performance by up to 39% (compared with OpenMP). On both architectures Callisto-RTS provides improved scaling and performance compared with a state-of-the-art parallel runtime system (Galois).

1 Introduction

Callisto-RTS is a parallel runtime system for multi-socket shared-memory machines. We focus on supporting graph analytics workloads such as PageRank [24] and betweenness centrality (BC) [17]. These workloads are increasingly important commercially, and are the focus of benchmarking efforts [1, 29] along with myriad single-machine systems (such as Galois [21] and Green-Marl [12]) plus distributed systems (such as Grappa [20], Naiad [19], and Pregel [18]). It can be difficult to exploit parallelism in these workloads because of the difficulty of achieving good load balance in combination with low synchronization overhead.

As a running example, consider a PageRank superstep (Figure 1). The outer loop (t) ranges over the vertices. Within each iteration, w ranges over the vertices adjacent to t and updates the new PageRank value for t based on the current value for w. Using OpenMP [22] as an example, the pragma indicates that chunks of BATCH_SIZE iterations of the outer loop should be assigned dynamically to threads. Typically, implementations do this assignment using an atomic fetch-and-add on a shared counter.

```
#pragma omp for schedule(dynamic, BATCH_SIZE)
for (node_t t = 0; t < G.num_nodes(); t ++) {
    double val = 0.0;
    for (edge_t w_idx = G.r_begin[t];
         w_idx < G.r_begin[t+1]; w_idx ++)  {
        node_t w = G.r_node_idx [w_idx];
        val += G_pg_rank[w] /
               (G.begin[w+1] - G.begin[w]);
    }
    G_pg_rank_nxt[t] = (1 - d) / N + d * val;
}
```

Figure 1: PageRank loop, with t ranging over vertices.

Setting BATCH_SIZE introduces a trade-off. Setting it too large risks load imbalance with threads taking large batches of work and some threads finishing before others. Setting it too small introduces synchronization overheads. It is difficult to set BATCH_SIZE optimally: The distribution of work between iterations is uneven—for instance, in a social network, a celebrity has millions of times more neighbors than the average. Even if the iterations are divided evenly, the work performed by each thread can be uneven. In some cases, the number of instructions executed by each thread may be the same, but the execution times differ based on differing memory access times and cache locality.

With Callisto-RTS we reduce the need for tuning by making it efficient to select a very small BATCH_SIZE while still achieving good performance and scalability. Concretely, on machines with 1024 h/w contexts, we achieve good performance down to batches of around 1K cycles (compared with 200K cycles using dynamically-scheduled OpenMP loops).

Section 2 describes our programming model. We provide nested parallel loops, with control over how the h/w contexts in the machine are allocated to different levels of the loop hierarchy—for instance, an outer loop may run with one thread per core, leaving additional threads per core idle until an inner level of parallelism is reached. This *non-work-conserving* approach to nesting can lead to better cache performance when iterations of the inner loop share state in a per-core cache.

Section 3 describes our techniques for fine-grained scheduling. We use a combining mechanism to allow threads requesting new work to aggregate their requests before accessing a shared loop iteration counter (e.g., combining requests via local synchronization in a core's L1$). In addition, we introduce a new asynchronous combining scheme in which a thread issues a request for new work while executing its current work: this provides more time for combining to occur. Furthermore, combining can be achieved with ordinary read/write operations, reducing the need for atomic read-modify-writes.

In Section 4 we evaluate the performance of Callisto-RTS. We use a 2-socket Intel 64 system (having 32 h/w contexts). We also use an 8-socket T5 SPARC system (having 1024 h/w contexts).

In addition to comparing with OpenMP, we compare our PageRank results with Galois, a state-of-the-art system based on scalable work-stealing techniques [21]. In contrast to work-stealing, we show that the shared-counter representation we use for parallel work enables single-thread performance improvements of 5%–26%. The asynchronous combining technique enables improved scalability on both processor architectures.

Section 5 discusses related work, and in particular, task-parallel models such as Cilk [8] and Intel Threading Building Blocks (TBB) [27]. Callisto-RTS differs from these systems in two main ways: First, compared with work-stealing, our implementation is specialized to distributing batches of loop iterations via shared counters. We use request aggregation to reduce contention on these counters rather than using thread-local work queues. Our approach avoids reifying individual batches of loop iterations as entries in work queues (as in Galois [21]), or requiring memory fence instructions (as in typical thread-safe work queues).

Second, we exploit the structure of the machine in the programming model as well as the runtime system. Our non-work-conserving approach to nesting contrasts with work-stealing implementations of task-parallelism in which all of the idle threads in a core would start additional iterations of the outer loop. In workloads with nested parallelism, our approach aims to reduce cache pressure when different iterations of an outer loop have their own iteration-local state: it can be better to have multiple threads sharing this local state, rather than extracting further parallelism from the outer loop.

As we say in Section 6, we hope that our techniques can be incorporated in runtime systems for other parallel programming models in the future.

2 Programming model: parallel loops

In this section we introduce the API supported by Callisto-RTS. Our initial workloads are graph analytics algorithms generated by a compiler from the Green-Marl DSL [12]. Therefore, while we aim for the syntax to be reasonably clear, our main goal is performance.

2.1 Flat parallelism

Callisto-RTS is based on parallel loops. As with OpenMP, and other systems, the programmer must ensure that iterations are safe to run concurrently. Loops are expressed using C++ templates, specializing a `parallel_for` function according to the type of the iteration variable and the loop body. Currently, all of the loops we support distribute their iterations across the entire machine (as with OpenMP `dynamic` loops). This reflects the fact that our graph algorithms typically have little temporal or spatial locality in their access patterns. In this setting, we are concerned more by reducing contention in the runtime system, and achieving good utilization of the h/w contexts across the machine and their associated memory.

A parallel loop to sum the numbers $0 \ldots 10$ is written:

```
struct example_1 {
  atomic<int> total {0}; // 0-initialized atomic
  void work(int idx) {
    total += idx;        // Atomic add
} } e1;

parallel_for<example_1, int>(e1, 0, 10);
cout << e1.total;
```

The `work` function provides the body of the loop. The `parallel_for` is responsible for distributing work across multiple threads. The struct `e1` is shared across the threads. Hence, due to the parallelism, atomic add operations are needed for each increment.

Per-thread state can be used to reduce the need for atomic operations. This per-thread state is initialized once in each thread that executes part of the loop, and then passed in to the `work` function:

```
struct per_thread { int val; };

struct example_2 {
  atomic<int> total {0}; // 0-initialized atomic

  void fork(per_thread &pt) { pt.val = 0; }

  void work(per_thread &pt, int idx) {
    pt.val += idx;       // Unsynchronized add
  }

  void join(per_thread &pt) {
    total += pt.val;     // Atomic add
} } e2;

parallel_for<example_2, per_thread, int>(e2,0,10);
cout << e2.total;
```

In this example the fork function is responsible for initializing the per-thread counter. The work function then operates on this per-thread state. The join function uses an atomic add to combine the results.

Design rationale. We considered whether to use C++ closures for loop bodies. Closures provide simpler syntax for short examples, and permit variables to be captured by reference in the work function. Unfortunately, performance using current compilers appears to depend a great deal on the behavior of optimization heuristics. We hope that future compiler implementations may provide more consistent performance in this regard.

For simplicity we have an implicit barrier at the end of each loop. This reflects the fact that, for our workloads, there is abundant parallel work, plus the fact that our implementation techniques are effective in reducing load imbalance (meaning that threads tend to arrive at the end of the loop at approximately the same time). We assume that Callisto-RTS runs within an environment where it has exclusive use of h/w contexts, and so thread preemption is not a concern.

In more variable multiprogrammed environments, dynamic techniques such as the prior work of Harris *et al.* [10], or abstractions and analyses such as those of Vajracharya and Grunwald may mitigate straggler problems [32].

Implementation. We initially describe the implementation with a single level of parallelism (we discuss nesting in Section 2.2). A set of worker threads is created at startup. A designated *leader* starts the main function. Other *follower* threads wait for work.

The definition of parallel_for instantiates a work_item object and publishes it via a shared pointer being watched by the followers. The work item has a single run function containing a loop which claims a batch of iterations before calling the workload-specific loop body. This repeats until there are no more iterations. A reference to the loop's shared state is held in the work item. Any per-thread state is stack-allocated within run. Consequently, only threads that participate in the loop will need to allocate per-thread state.

The thread which claims the last batch of iterations removes the work item from the shared pointer (preventing additional threads needlessly starting it). Finally, each work item holds per-socket counts of the number of active threads currently executing the item. The main thread waits for these counters to all be 0, at which point it knows that all of the iterations have finished execution.

Process termination is signaled by the leader publishing a designated "finished" work item. This approach means that a worker can watch the single shared location both for new work and for termination.

2.2 Nested parallelism

Parallel loops can be nested within one another, and Callisto-RTS provides control over the way in which h/w contexts are allocated to different levels. The workloads we target have a small number of levels of parallelism, dependent on the algorithm rather than on its input. For instance, our betweenness centrality workload (BC) uses an outer level to iterate over vertices, and then an inner level to implement a parallel breadth-first search (BFS) from each vertex.

Selecting which of these levels to run in parallel depends on the structure of the hardware. In the BC example, parallelizing just at the outer level can give poor performance on multi-threaded cores because multiple threads' local BFS states compete for space in each per-core cache. Conversely, parallelizing just at the inner level gives poor performance when the BFS algorithm does not scale to the complete machine. A better approach is to use parallelism at both levels, exploring different vertices on different cores, and using parallel BFS within a core.

A loop indicates how many levels are nested inside it. That is, a loop at level 0 is an inner loop with no further parallelism. A loop at level 1 encloses one level of parallelism, and so on.

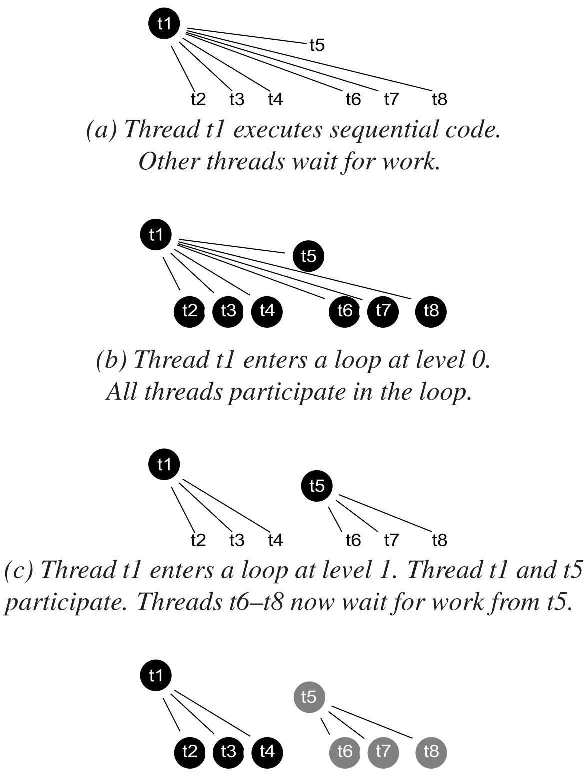
Concretely, writing parallel_for is short for a loop at level 0. For a loop at level N we write:

```
outer_parallel_for<...>(N, ...);
```

Design rationale. This "inside out" approach to counting levels provides composability. A leaf function using parallelism will always be at level 0, irrespective of the different contexts it may be called from.

If we numbered levels "outside in", or assigned them dynamically, then it would not be possible to distinguish (*i*) reaching an outer loop which should be distributed across all h/w contexts, versus (*ii*) an outer loop which should just be distributed at a coarse level leaving some idle h/w contexts for use within it. A given program may have loops with different depths of nesting—e.g., a flat initialization phase at level 0 over all h/w contexts, while a subsequent computation may start at level 1 and just be distributed at a per-socket granularity.

Implementation. Environment variables set how nesting levels map to the machine—e.g., indicating that loops at level 0 should be distributed across all h/w contexts, and that level 1 should be distributed across cores, core-pairs, sockets, or some other granularity. This flexibility lets a program express multiple levels of parallelism on large NUMA machines, but execute more simply on smaller systems.



(a) Thread t1 executes sequential code. Other threads wait for work.

(b) Thread t1 enters a loop at level 0. All threads participate in the loop.

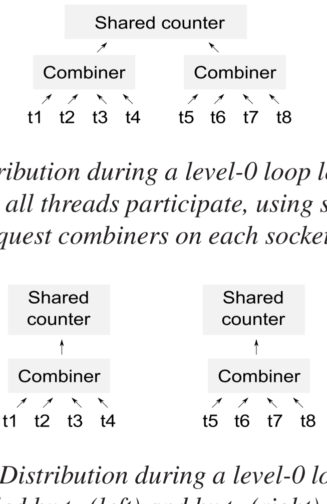
(c) Thread t1 enters a loop at level 1. Thread t1 and t5 participate. Threads t6–t8 now wait for work from t5.

(d) Threads t1 and t5 enter loops at level 0, threads participate in the respective loops.

Figure 2: Allocation of threads to loops. Thread t1 is at the top level, t5 at level 1, and other threads at level 0. This allocation might be appropriate in a 2-socket machine with 4 threads per socket.

Based on this, threads are organized into a tree which selects which threads participate in which loops. Each thread has a *level* in this tree, and a *parent* at the next non-empty level above it (aside from a designated top-level thread which forms the root of the tree). Dynamically, each thread has a *status* (leading or following). Initially, the root is leading and all other threads following. A thread's *leader* is the closest parent with leading status (including the thread itself). A thread at level n becomes a leader if it encounters a loop at level $k \leq n$. A follower at level n executes iterations from a loop if its leader encounters a loop at level $k \leq n$; otherwise, it remains idle.

Figure 2 illustrates this dynamically with a possible organization of 8 threads across 2 sockets. The main thread is t_1 and is the parent to $t_2 \ldots t_4$ in its own socket (level 0), and t_5 in the second socket (level 1). In turn, t_5 is parent to $t_6 \ldots t_8$. Initially t_1 is the only active thread and hence leader to all of the threads $t_1 \ldots t_8$ (Figure 2a). If t_1 encounters a loop at level 0 then all threads participate in the same loop (Figure 2b). If, instead, t_1 encounters a loop at level 1 then just t_1 and t_5 participate (Figure 2c). If t_5 then encounters a new loop at level 0 then it becomes a leader of $t_5 \ldots t_8$ (Figure 2d).



(a) Distribution during a level-0 loop led by t_1 in which all threads participate, using separate request combiners on each socket.

(b) Distribution during a level-0 loop led by t_1 (left) and by t_5 (right).

Figure 3: Work scheduling in different loops. A top-level loop spans the complete machine, with local requests for work being combined in each set of nearby threads. Multiple instances of an inner loop may run concurrently on the two parts of the machine.

3 Work scheduling

We now introduce our techniques for distributing iterations. We take a hierarchical approach to defining work scheduling policies, with a number of basic policies that can be combined to form more complex variants. An individual thread makes a request to the leaves of a tree of work distributors, and the implementation of this may involve a call to a higher level distributor, and so on.

Our hierarchical approach lets us reflect the structure of the machine within the hierarchy used for work scheduling. In addition, it lets us explore a range of complex policies—for instance, exploring whether data structures should be per-core, per-L2$, or per-socket, Figure 3 illustrates this using the example 8-thread machine. Separate work distributors are used for each parallel loop—for instance, the 4-thread loop led by t_1 is handled separately from the 4-thread loop led by t_5.

Shared counter. The simplest work distributor is a single shared counter, initialized with loop bounds, and with threads claiming iterations using an atomic fetch-and-add. We include this initial implementation to reflect the techniques used for dynamically scheduled loops in many OpenMP runtime systems.

Distributed counters. The iteration space is distributed evenly across a number of stripes according to the number of sockets, cores, or threads within the machine. Each thread is associated with a *home* stripe (e.g., with per-socket distribution, this would correspond to the thread's socket). In addition, each thread has a *current* stripe. A thread claims iterations by an atomic increment on its current stripe until that portion of the iteration space has been completed. At that point it moves on to the next stripe, and so on until it returns to its home stripe.

Request combining. Request combiners attempt to aggregate requests for work which are made "nearby" in time and in the machine. Rather than have multiple threads across the machine compete for atomic updates to a single cache line, sets of threads can compete at a finer granularity, and then a smaller number of threads compete at a global level. This reduces the number of atomic read-modify-write instructions, and it increases the likelihood that contention remains in a local cache.

Each thread using a combiner has a *slot* comprising a pair of loop indices (`start/request`, and `end`). For instance, in a 4-slot combiner:

Start / request	0	REQ	REQ	0
End	0	0	0	16
	Combiner lock			

Slot $(0,0)$ is quiescent. Slot $(\mathtt{REQ},0)$ represents a request for work. Slot $(0,16)$ represents supplied work (in this case the iterations $0\ldots16$). In addition, each combiner has a lock which needs to be held by a thread collecting requests to make to the upstream counter. In pseudocode:

```
  my_slot->start = REQ; // Issue request

  while (1) {
    // Try to acquire the combiner lock
    if (!spinlock_tryacquire(&my_combiner->lock)) {
      // Lock busy.  Wait for it to be released, then
      // test if we received work.
      while (spinlock_is_held(&my_combiner->lock)) {
      }
    } else {
      // We acquired combiner lock, collect requests
      // from other threads, issue aggregate request,
      // distribute work, and then release lock.
      ...
      spinlock_release(&my_combiner->lock);
    }
    // Test if request has been satisfied
    if (my_slot->start != REQ) {
      return (my_slot->start, my_slot->end);
} }
```

A thread starts by writing `REQ` in its slot and then trying to acquire the lock. If the lock is already held then the current thread waits until the lock is available, and tests if its request has been satisfied. Note that the `REQ` flag is set without holding the lock, and so the lock holder is not guaranteed to see the thread's request. If a thread succeeds in acquiring the lock it scans the other slots for `REQ` and issues an upstream request for a separate batch of iterations for each requester (for brevity we omit the pseudocode for this). Work is distributed by writing to the `end` field and then overwriting `REQ` in the `start` field. Hence, on a TSO memory model, a thread receiving work sees the start-end pair consistently once `REQ` is overwritten.

If all threads using a combiner share a common L1$ then the request slots are packed onto as few cache lines as possible, Otherwise, each slot has its own line. Combiners can be configured in various ways. For instance, threads within a core could operate with a per-core combiner, and then additional levels of combining could occur at a per-L2$ level (if this is shared between cores), or at a per-socket level. We examine some of these alternatives in our evaluation (Section 4).

Asynchronous combining. With asynchronous combining, a thread sets its request flag *before* executing its current batch, rather than after finishing it. This asynchrony exposes a request over a longer interval: other threads using the same combiner can handle the request while the thread's current batch is being executed.

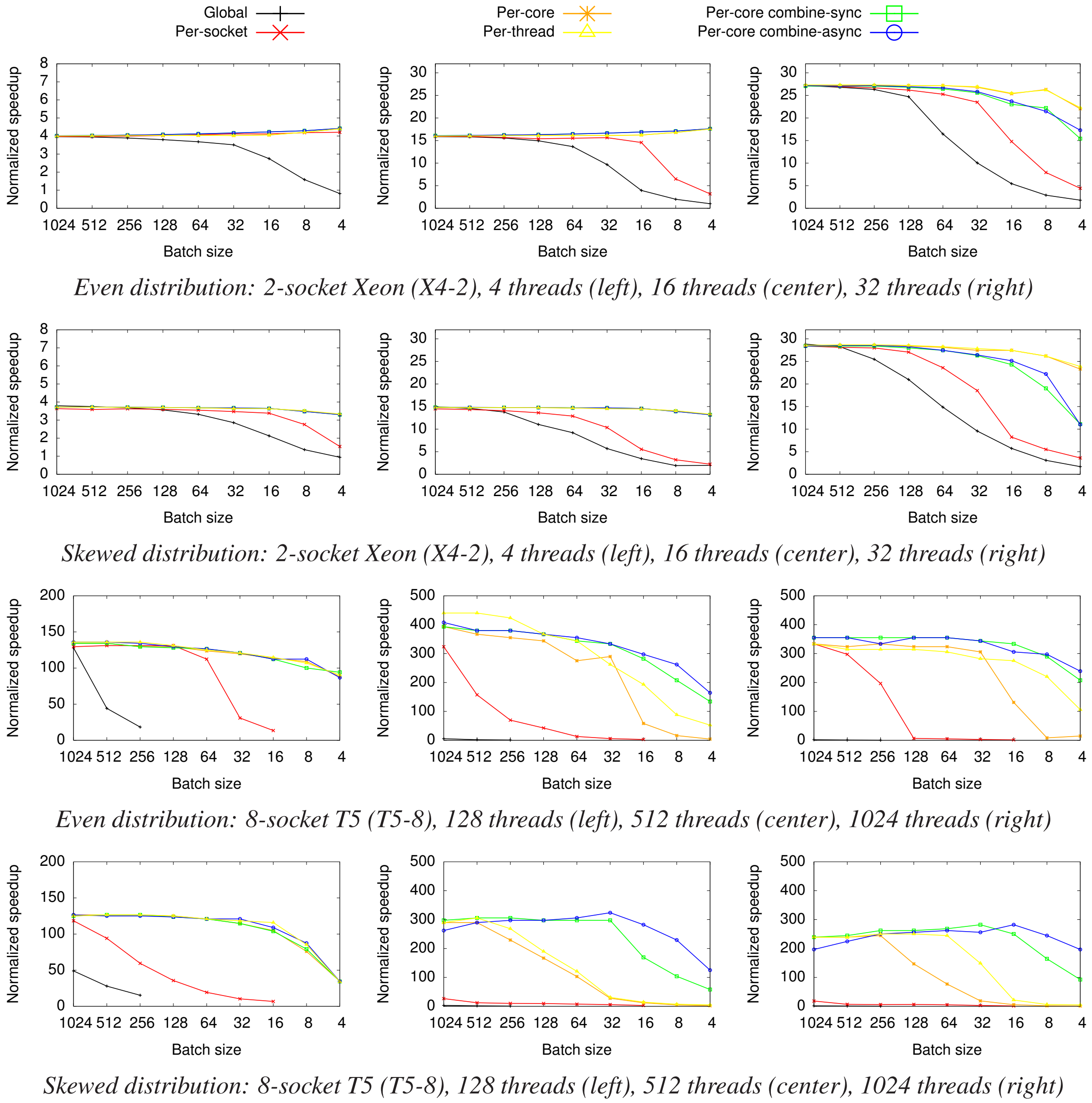
In the best case, in a set of n threads, all but 1 will find they have received new work immediately after finishing their current batches. Furthermore, if additional combining occurs, then it increases the size of the aggregate requests issued from the combiner (reducing contention on the next level in the work scheduling tree), and it reduces contention on the lock used within the combiner (if most threads receive work then they never need to acquire the lock). The fast-path for the $n-1$ threads receiving work is (*i*) reading the work provided to them, and (*ii*) setting their request flag. On a TSO memory model this avoids fences or atomic read-modify-write instructions.

4 Evaluation

We evaluate the performance of Callisto-RTS using machines with two different processor architectures:

Intel 64. We use an Oracle X4-2 machine. This is a 2-socket machine with Intel Xeon E5-2650 IvyBridge processors. Processors have a per-socket L3$, and per-core L2$ and L1$. Each core provides 2 h/w contexts for a total of 32 h/w contexts in the machine.

We use GCC 4.7.4 and Linux 2.6.32. We confirmed a subset of our results on Linux 3.14.33 but saw no difference: the runtime systems are set to employ user-mode synchronization using atomic instructions rather than `futex` system calls.



Even distribution: 2-socket Xeon (X4-2), 4 threads (left), 16 threads (center), 32 threads (right)

Skewed distribution: 2-socket Xeon (X4-2), 4 threads (left), 16 threads (center), 32 threads (right)

Even distribution: 8-socket T5 (T5-8), 128 threads (left), 512 threads (center), 1024 threads (right)

Skewed distribution: 8-socket T5 (T5-8), 128 threads (left), 512 threads (center), 1024 threads (right)

Figure 4: Microbenchmark scalability on X4-2 and T5-8 systems.

SPARC. We use an Oracle T5-8 machine. This is an 8-socket machine with SPARC T5 processors. Each socket has 16 cores, and each core supports 8 h/w contexts for a total of 1024 h/w contexts in the machine. As with the Intel 64 system, the T5-8 has per-socket L3$ caches, and per-core L2$ and L1$. We use Solaris Studio 12.4 on Solaris 11.2.

Both architectures provide atomic compare-and-swap (CAS). The Intel 64 architecture provides additional atomic operations such as fetch-and-add. Conversely, the T5 processor provides a user-mode-accessible `wrpause` instruction which lets a h/w context wait for a configurable number of cycles, avoiding it consuming pipeline resources while waiting. This can be important in the multi-threaded SPARC processor: when only a single h/w context is runnable, that context can issue instructions to multiple pipelines in each clock cycle. On the T5-8 we use `wrpause` for 128 cycles in loops which are expected to unblock quickly (e.g., during request combining), and for approximately 4096 cycles in loops which are expected to unblock less quickly (e.g., waiting on entry to a loop).

We spread software threads as widely as possible within the machine. We use OpenMP with active synchronization (i.e., spinning, rather than blocking in the OS). For each algorithm-machine combination, the fastest result is achieved with active synchronization rather than blocking. We report median-of-3 results.

We use three evaluation workloads: a scalability microbenchmark (Section 4.1), graph algorithms with a single level of parallelism (Section 4.2), and an additional graph workload using nested parallelism (Section 4.3).

4.1 Work scheduling microbenchmarks

We start using a microbenchmark with a single large loop. Each iteration performs a variable amount of work (incrementing a stack-allocated variable a set number of times). We can vary (*i*) the number of increments used in the different iterations, (*ii*) the number of threads, (*iii*) the work scheduling mechanism we use, and (*iv*) the batch size in which threads claim work. We investigate two ways of organizing the work within the loop:

Even distribution. Here, each iteration performs the same amount of work: good load balancing can be achieved by splitting the iteration space evenly. We evaluate six scheduling techniques: a single shared counter, distributed counters at per-socket, per-core, and per-thread granularities, and finally per-core work combiners coupled with per-core counters (Figure 4). For each machine we show a workload with a modest number of threads (left column), and then a workload with 1 thread per core (center column), and a workload with all h/w contexts in use (right column). We plot the speedup relative to unsynchronized sequential code on the same machine.

On the Intel 64 system, a single iteration is around 50 cycles. The per-core and per-thread counters perform well across the experiments. Request combining performs slightly worse than simple per-thread or per-core counters: little combining occurs with only two threads per core.

On the SPARC system, each iteration is around 140 cycles. At large batch sizes, we see good scaling to 512 h/w contexts. The number of instructions per cycle is 0.34 and so, with 2 pipelines per core, we would expect to saturate the cores with 750 threads. Beyond this point, contention between threads for pipeline resources can limit performance. We believe this is an example workload where the user-mode `mwait` instruction in the future SPARC M7 processor [25] could provide improved scalability—unlike `wrpause`, the `mwait` instruction permits a thread to monitor a memory location while waiting, rather than needing to pick a specific interval in advance.

Combining shows slight benefits at high thread counts and low batch sizes. As expected, the CAS loop used to increment the counters starts to need re-execution under higher contention (on Intel 64 we can use an atomic fetch-and-add). Re-execution consumes pipeline resources that could otherwise be used productively.

Asynchronous combining generally aggregates requests from all of the active threads in a core irrespective of the batch size used (e.g., with 256 threads, there are 2 threads per core, and each combined request is for 2 batches). Synchronous combining is effective only when the batch sizes are small, making requests more likely to "collide".

Skewed distribution. With the skewed work distribution, the first *n* iterations each contain 1024x the work of the others. We set *n* so that the total work across all iterations is the same as the even distribution. The aim is to study the impact of different work scheduling techniques when there is contention in the runtime system: threads which start at the "light" end of the iteration space will complete their work quickly and start to contend for work with threads at the "heavy" end.

On the Intel 64 system, per-core and per-thread counters perform well. As with the even workload, two threads per core provides little opportunity for combining.

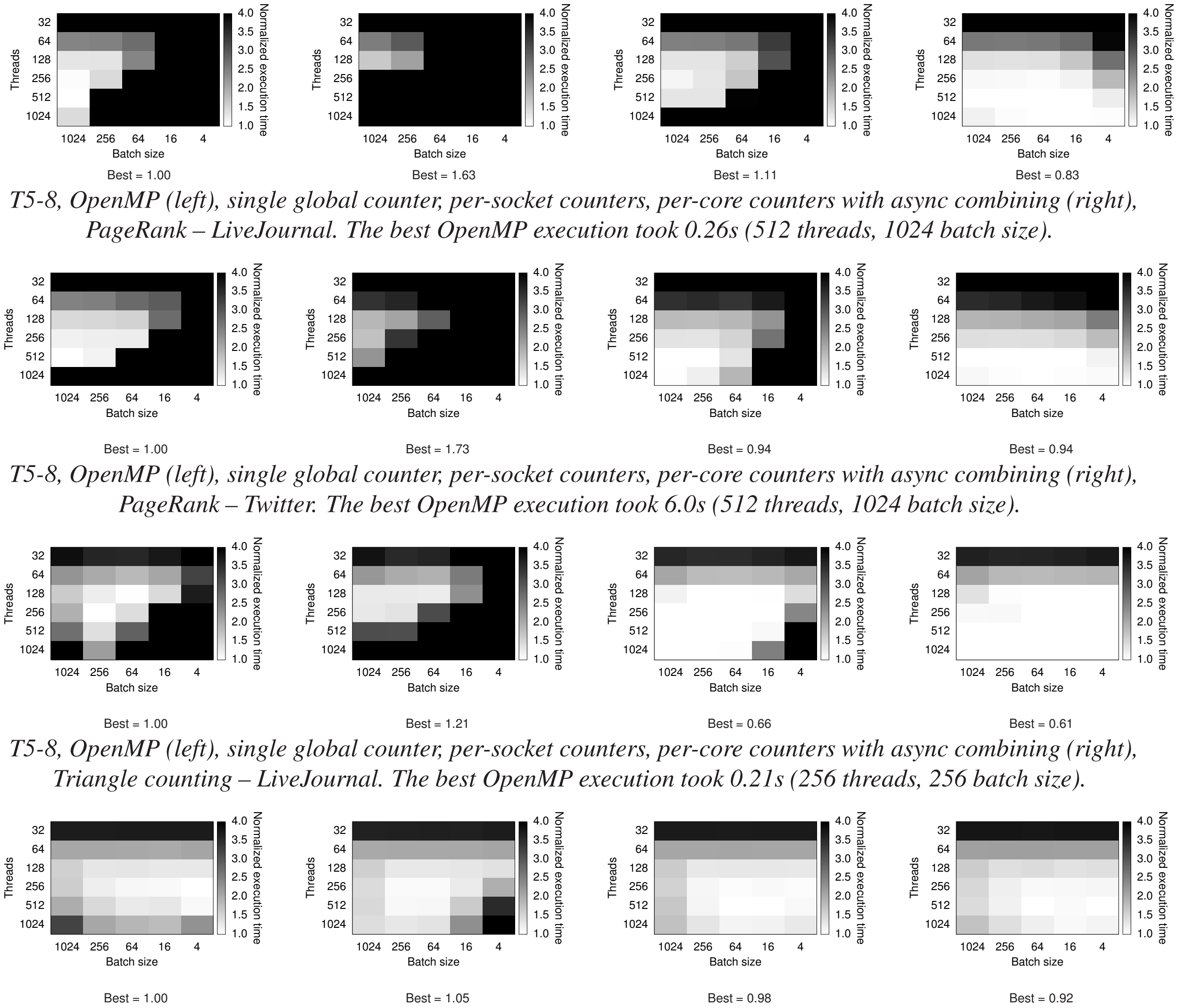
On the SPARC systems, the use of combining has significant benefits at high thread counts (512 or 1024), with some additional benefit from asynchronous combining. The skewed workload means that we see CAS failures and re-execution when incrementing shared counters. In contrast, per-core combining allows most threads to request work by setting their request flag (which remains core-local in the L1$) and then waiting "politely" using `wrpause`.

Summary. Based on these results, we use per-thread counters as the default on Intel 64, and per-core counters with asynchronous combining on SPARC.

In addition to the results shown here we explored two-level combining schemes in which threads combine requests within a core, before using a further level of combining within a socket. We saw combining occurring at both levels, but the the overall benefits from reduced contention did not offset the cost of the additional operations used. Per-core combining with per-core counters performed better across all workloads, and so we omit the two-level results.

4.2 Graph algorithms

We now evaluate Callisto-RTS using the PageRank and Triangle Counting algorithms from Green-Marl [12] (Figure 5). In Section 4.3 we use a betweenness central-



T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), PageRank – LiveJournal. The best OpenMP execution took 0.26s (512 threads, 1024 batch size).

T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), PageRank – Twitter. The best OpenMP execution took 6.0s (512 threads, 1024 batch size).

T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), Triangle counting – LiveJournal. The best OpenMP execution took 0.21s (256 threads, 256 batch size).

T5-8, OpenMP (left), single global counter, per-socket counters, per-core counters with async combining (right), Triangle counting – Twitter. The best OpenMP execution took 55.3s (256 threads, 4 batch size).

Figure 5: Graph algorithms on LiveJournal (4.8M vertices) and Twitter (42M vertices). Execution times normalized to the best OpenMP result. Below each plot we show the ratio of the best configuration's execution time to the best OpenMP result.

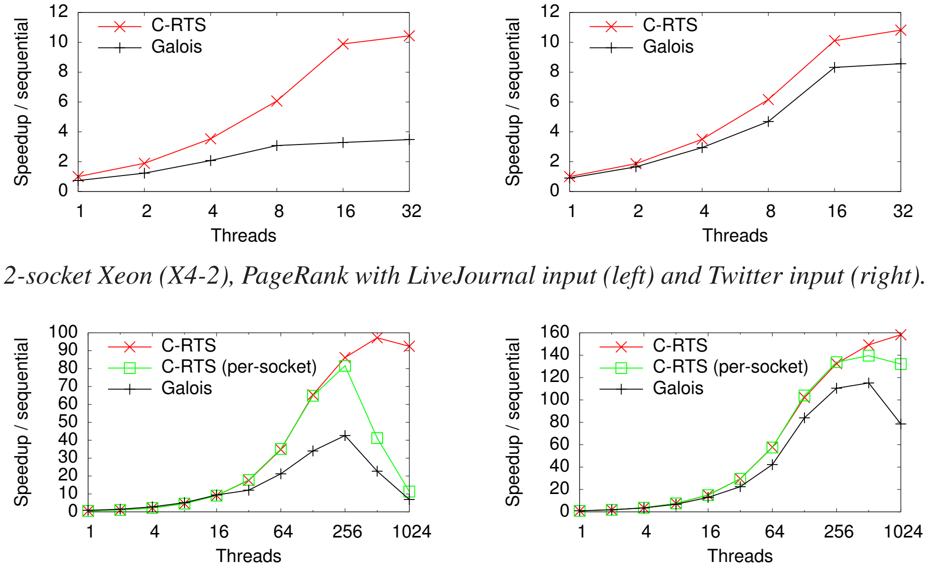
ity algorithm [17] as an example with nested parallelism.

We use the SNAP LiveJournal dataset (4.8M vertices, 69M edges) [16] and the Twitter data set of Kwak *et al.* (42M vertices, 1.5B edges) [14].

We focus on the SPARC machine. As the microbenchmark results illustrated, the smaller 2-socket Intel 64 system does not exhibit a great deal of sensitivity to work scheduling techniques with per-thread counters.

For each machine-algorithm combination we show: the original OpenMP implementation, and then Callisto-RTS using a single global counter, per-socket counters, and per-core counters with asynchronous combining. Each plot shows the performance of the given technique across thread counts (32…1024), and batch sizes (1024…4). Each square shows the execution time, normalized to the best OpenMP result. Below each plot, we show the time of the best configuration, normalized to the best OpenMP result. Note that the dark rows at the top of the plots indicate there are insufficient threads to perform well on these scalable workloads, even with perfect work scheduling and no overheads.

On the LiveJournal input, careful tuning is needed to get good performance with OpenMP or with a single counter: different numbers of threads are best for the dif-



2-socket Xeon (X4-2), PageRank with LiveJournal input (left) and Twitter input (right).

8-socket T5 (T5-8), PageRank with LiveJournal input (left) and Twitter input (right).

Figure 6: PageRank on Callisto-RTS and Galois.

ferent algorithms, and there is a sharp fall-off in performance if the best configuration is not selected.

The OpenMP implementations often perform better than Callisto-RTS using a single global counter. This is because they use `static` scheduling on some loops where work is known to be distributed evenly (e.g., copying from one array to another). Static scheduling works well on such loops, but not on the main parts of the algorithm.

Using a single global counter leads to poor performance at small batch sizes, and work imbalance with large batches. Per-socket counters provide significant improvement at smaller batch sizes. As in the microbenchmark, per-core counters with asynchronous combining provide good performance over a wide range of configurations. We see similar trends on the Twitter input.

Comparison with Galois. The Galois system is a lightweight infrastructure for parallel in-memory processing. In prior work, Nguyen *et al.* demonstrated that Galois has good performance and scalability across a range of graph benchmarks [21]. We use version 2.2.1. We adapted the Galois PageRank code to use the same in-memory compressed sparse row representation as with Callisto-RTS. Compared with the Galois original, this modified implementation is faster across every test. We disabled concurrency control and confirmed that we obtained identical performance between Galois and Callisto-RTS. Thread placement is identical between the two runtime systems. We use Galois' default batch size (32) in both systems.

Figure 6 shows the resulting performance on the X4-2 and T5-8. All results are normalized to the single-threaded implementation without concurrency control. Callisto-RTS performs better on both machines and both inputs.

On the X4-2, Callisto-RTS scales similarly on both graphs up to 16 threads (1 thread per core), with a slight additional benefit from hyperthreading. Galois scales well on the Twitter graph, with 15-20% overhead compared with Callisto-RTS. Galois does not scale well on the LiveJournal graph with shorter loop iterations. Both differences are due to the way Galois distributes chunks of work. Each chunk is reified in memory as a block listing the iterations to execute, with each thread holding a current working block, and per-socket queues of blocks. On the smaller graph, the iterations are short-running and contention on per-socket queues appears to limit scaling. On the Twitter graph, each iteration is longer and contention less significant. However, the inner loop of fetching an iteration and executing it remains slower than with Callisto-RTS.

We see similar trends on the T5-8. Galois and Callisto-RTS both scale well to 128 threads (1 per core), as does the additional Callisto-RTS variant using per-socket iteration counters. Beyond this point, Callisto-RTS continues to scale well with asynchronous work distribution, whereas the other implementations are harmed by contention between threads when distributing

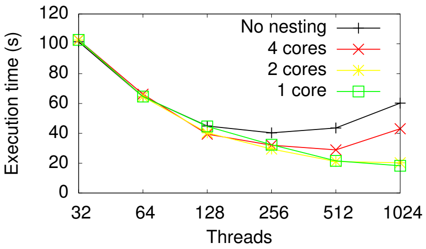


Figure 7: Betweenness centrality using nested parallelism at different levels.

work. On both graphs, Callisto-RTS performs well using the complete machine of 1024 threads.

Summary. Compared with the OpenMP implementation, using per-core counters with asynchronous combining improves the best-case performance in all four of the workloads in Figure 5 by 6%, 8%, 17%, and 39%. In addition, and perhaps more significantly, the performance achieved is more stable over different thread and batch settings, and does not require the programmer to select between static and dynamic scheduling.

4.3 Nested parallelism

Our final results use nested parallelism to compute betweenness centrality [17]. For each vertex, the computation executes breadth-first-search (BFS) traversals. The execution time can be large even for a modestly sized graph. We use the SNAP Slashdot data set [16] (82.1K vertices, 948K edges). Figure 7 compares flat parallelism (in which we process each vertex sequentially), versus nested parallelism at different levels. We use a parallel BFS algorithm with 13 different parallel loops, some initializing per-BFS data structures, and others performing parallel expansion of the next level of vertices. There is a barrier in between each loop (just between the threads executing that BFS, rather than system-wide).

On this workload, flat parallelism scales well to the level of 1 thread per core (128 threads on the T5-8 SPARC system). We see little improvement from further threads, and then some degradation at 512…1024 threads. We recorded values from the SPARC CPU performance counters. With 1 thread per core, 9.8% of load instructions miss in the L2-D$. With flat parallelism, this rises steadily to 29% with 8 threads per core.

We obtain the best performance using nesting within a single core, corresponding to the L2-D$ in this machine. Using nested parallelism, the miss rate rises only slightly to 10.8% when moving from 128 to 1024 threads.

In addition to the results shown here we tried (*i*) nested parallelism at a per-socket level, and (*ii*) parallelism only at the inner level in the BFS algorithm. Both of these alternatives were substantially worse than flat parallelism.

5 Related work

We discuss related work under three sections: programming models providing parallel loops, implementations of task parallelism, and prior work on combining techniques:

Parallel loops. Our techniques could be used in implementations of programming models which include parallel loops. Examples include OpenMP [22], parallel loops in Intel Threading Building Blocks (TBB) [27], and the proposed C Parallel Language Extensions [6]. Currently, the GCC 4.9 OpenMP implementation uses a per-loop shared counter with atomic fetch-and-add. As our results show, this approach requires careful tuning.

Task-parallelism. Systems such as Cilk [8], TBB [27], Wool [7, 26], and the Java ForkJoin framework [15] support task-parallel programming by distributing lightweight tasks using work-stealing systems such as those of Blumofe *et al.* [3] or Chase-Lev [5]. Cilk and TBB provide parallel loops built over task-parallel abstractions, recursively decomposing loops until a minimum size is reached (analogous to the batch size).

Typically, the common execution path involves a thread taking a task from a local work queue, decomposing the task, pushing part of the task back onto the queue, and executing the extracted iterations. While these steps can remain local to a thread, they require an atomic operation or memory fence [2]. Our request combining technique avoids these operations (aside from the one thread performing the aggregate request). Asynchronous combining reduces our fast path to a read of the current batch, followed (without a fence) by a write for a new request.

Tzannes *et al.* [30, 31] observe that a thread can avoid repeated operations on a work-queue by only pushing tasks on to the queue when it is below a threshold size (if the queue is above this size then that indicates that other threads are busy because otherwise items from the queue would have been stolen).

Using work stealing provides the opportunity to benefit from large amounts of prior work on scalable implementations (dating back at least as far as the work of Burton and Sleep [4], and stretching to ongoing work such as that of Tzannes *et al.* [31]). As discussed in our evaluation, Galois is a state-of-the-art example of this kind of implementation, specialized to shared-memory NUMA systems, However, reifying each loop iteration as an entry in a work-stealing queue introduces storage and processing costs, especially when loops contain short iterations.

Combining algorithms. Many systems have used combining techniques in which operations are aggregated to reduce contention. Direct software implementations of early techniques such as combining trees [9] and combining funnels [28] have typically not performed well. Hendler *et al.* illustrate this in the evaluation of their *flat combining* technique for handling requests on a lock-based shared data structure [11]. As in our request combiners, with flat combining each thread has a structure to publish requests for work, and a lock which is held while collecting requests. Unlike our design, flat combining requires threads to watch both the lock and their own request—our approach allows threads to just watch the lock (enabling the use of instructions such as `mwait`), and we make requests asynchronously with working.

Oyama *et al.* described a technique in which a lock protects a data structure and threads add requests to a LIFO queue associated with the lock [23]. Each thread must perform a successful CAS on the head of the list, whereas we allow threads to publish requests by writing to a per-thread flag. We use combining within a core, and empirically the cost of scanning the flags is better than the cost of maintaining a list.

Klaftenegger *et al.* described a queue-based delegation model in which a thread making a write-only request can proceed concurrently with the request's execution [13]. Our specialized use of delegation avoids maintaining an explicit queue, and handles a read-modify-update operation involving aggregation as well as delegation.

6 Conclusions and future work

In this paper we have introduced runtime system techniques for supporting parallel loops with fine-grain work scheduling. We are able to scale down to batches of work of around 1000 cycles on machines with 1024 h/w contexts, and we are able to achieve good scaling with workloads where the distribution of work between loop iterations is skewed. In addition, on an example workload with nested parallelism, we were able to obtain further scaling by matching the point at which we switch to the inner level parallelism to the position of the L2-D$ in the machine. This lets multiple threads execute the inner loops while sharing data in their common cache.

We believe that the techniques used in Callisto-RTS are applicable to other parallel programming models. The combining techniques could be applied transparently in implementations of OpenMP dynamically scheduled loops—either with, or without, asynchronous combining.

In addition, the same techniques could be applied to work-stealing systems. It may be profitable to use per-core queues and for threads within a core to use combining to request multiple items at once. As with loop scheduling in Callisto-RTS, this may reduce the number of atomic operations that are needed, and enable asynchrony between requesting work and receiving it. Furthermore, using per-core queues with combining may make loop termination tests more efficient than with per-thread queues (typical termination tests must examine each queue at least once before deciding that all of the work is complete).

Finally, we see the trend toward increasingly non-uniform memory performance making it important to exercise control over how nesting maps to hardware. In Callisto-RTS we do this by explicit programmer control and non-work-conserving allocation of work to threads. Future systems could use feedback-directed techniques, or potentially static analyses.

Acknowledgments

We would like to thank the reviewers along with Gavin Bierman, Callum Cameron, Hassan Chafi, Nawal Copty, Dave Dice, Sungpack Hong, Daniel Goodman, Darryl Gove, Jinha Kim, Martin Maas, Zoran Radovic, Paul Thomson, Vasileios Trigonakis, and Georgios Varisteas for discussions and feedback on earlier drafts of this paper.

References

[1] Angles, R., Boncz, P., Larriba-Pey, J., Fundulaki, I., Neumann, T., Erling, O., Neubauer, P., Martinez-Bazan, N., Kotsev, V., and Toma, I. The Linked Data Benchmark Council: A graph and RDF industry benchmarking effort. *SIGMOD Rec. 43*, 1 (May 2014), 27–31.

[2] Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., and Vechev, M. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL' 11: Proceedings of the 38th Symposium on Principles of Programming Languages* (Jan. 2011), pp. 487–498.

[3] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM 46*, 5 (Sept. 1999), 720–748.

[4] Burton, F. W., and Sleep, M. R. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (1981), pp. 187–194.

[5] Chase, D., and Lev, Y. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the 17th Symposium on Parallelism in Algorithms and Architectures* (July 2005), pp. 21–28.

[6] CPLEX study group, WG14. C extensions for parallel programming. Working draft, 2014-09-18, N1862.

[7] Faxén, K.-F. Wool—a work stealing library. *SIGARCH Computer Architecture News 36*, 5 (June 2009), 93–100.

[8] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1998), pp. 212–223.

[9] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M. The NYU ultracomputer—designing a MIMD, shared-memory parallel machine. In *ISCA '98: 25 Years of the International Symposia on Computer Architecture, Selected Papers* (June 1998), pp. 239–254.

[10] Harris, T., Maas, M., and Marathe, V. J. Callisto: co-scheduling parallel runtime systems. In *EuroSys '14: Proceedings of the 9th ACM European Conf. on Computer Systems* (Apr. 2014).

[11] Hendler, D., Incze, I., Shavit, N., and Tzafrir, M. Flat combining and the synchronization-parallelism tradeoff. In *SPAA '10: Proceedings of the 22nd Symposium on Parallelism in Algorithms and Architectures* (June 2010), pp. 355–364.

[12] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS '12: Proceedings of the 17th Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2012), pp. 349–362.

[13] Klaftenegger, D., Sagonas, K. F., and Winblad, K. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par 14: Parallel Processing – 20th International Conference* (Aug. 2014), pp. 572–583.

[14] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th International Conference on World Wide Web* (Apr. 2010), pp. 591–600.

[15] Lea, D. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 Conference on Java Grande* (June 2000), pp. 36–43.

[16] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[17] Madduri, K., Ediger, D., Jiang, K., Bader, D. A., and Chavarria-Miranda, D. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS '09: Proceedings of the International Symposium on Parallel and Distributed Processing* (May 2009), pp. 1–8.

[18] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A system for large-scale graph processing. In *SIGMOD '10: Proceedings of the International Conference on Management of Data* (June 2010), pp. 135–146.

[19] McSherry, F., Isaacs, R., Isard, M., and Murray, D. G. Composable incremental and iterative data-parallel computation with Naiad. Tech. Rep. MSR-TR-2012-105, Microsoft Research, 2012.

[20] Nelson, J., Myers, B., Holt, B., Briggs, P., Ceze, L., Kahan, S., and Oskin, M. Grappa: A latency-tolerant runtime for large-scale irregular applications. Tech. Rep. UW-CSE-14-02-01, University of Washington, 2014.

[21] Nguyen, D., Lenharth, A., and Pingali, K. A lightweight infrastructure for graph analytics. In *SOSP '13: Proceedings of the 24th Symposium on Operating Systems Principles* (Nov. 2013), pp. 456–471.

[22] *OpenMP Application Program Interface, Version 4.0.* July 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[23] Oyama, Y., Taura, K., and Yonezawa, A. Executing parallel programs with synchronization bottlenecks efficiently. In *PDSIA '99: Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications* (July 1999), pp. 182–204.

[24] Page, L., Brin, S., Motwani, R., and Winograd, T. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[25] Phillips, S. M7: Next generation SPARC. In *HotChips '14: A Symposium on High Performance Chips* (Aug. 2014).

[26] Podobas, A., Brorsson, M., and Faxén, K. A comparative performance study of common and popular task-centric programming frameworks. *Concurrency and Computation: Practice and Experience 27*, 1 (2015), 1–28.

[27] Reinders, J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007.

[28] Shavit, N., and Zemach, A. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing 60*, 11 (Nov. 2000), 1355–1387.

[29] Suzumura, T., Ueno, K., Sato, H., Fujisawa, K., and Matsuoka, S. Performance characteristics of Graph500 on large-scale distributed environment. In *IISWC '11: Proceedings of the International Symposium on Workload Characterization* (Nov. 2011), pp. 149–158.

[30] Tzannes, A., Caragea, G. C., Barua, R., and Vishkin, U. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *PPoPP '10: Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming* (Jan. 2010), pp. 179–190.

[31] Tzannes, A., Caragea, G. C., Vishkin, U., and Barua, R. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS: ACM Transactions on Programming Languages and Systems 36*, 3 (Sept. 2014), 10:1–10:51.

[32] Vajracharya, S., and Grunwald, D. Dependence driven execution for multiprogrammed multiprocessor. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing* (July 1998), pp. 329–336.