Efficient Work Stealing for Fine Grained Parallelism

Karl-Filip Faxén
Swedish Institute of Computer Science
Stockholm, Sweden
kff@sics.se

***Abstract*—This paper deals with improving the performance of fine grain task parallelism. It is often either cumbersome or impossible to increase the grain size of such programs. Increasing core counts exacerbates the problem; a program that appears coarse-grained on eight cores may well look a lot more fine-grained on sixty four.**

In this paper we present the *direct task stack*, a novel work stealing algorithm with unusually low overheads, both for creating tasks and for stealing. We compare the performance of our scheduler to Cilk++, the `icc` implementation of OpenMP 3.0 and the Intel TBB library on an eight core, dual socket Opteron machine. We also analyze the reasons why our techniques achieve consistent speed ups over the other systems ranging from 2-3x on many fine grained workloads to over 50 in extreme cases and show quantitatively how each of the techniques we use contribute to the improved performance.

I. Introduction

Nested task parallelism is gaining popularity as a programming model for multi(core)processors. Parallelism is expressed by spawning tasks that the implementation is allowed, but not mandated, to execute in parallel. Existing implementations [13], [21], [1] exhibit significant overheads for fine grain computations, forcing application programmers to implement manual cut-offs (avoid spawning a task if the expected run time is small) or manage parallelism in other ways. For instance, in a recent paper on the Intel TBB [8], the authors recommend that tasks with less than 100k cycles worth of work should be handled using a special continuation passing style API (which requires a great deal of code restructuring) to reduce overhead. Not only does this put an extra burden on the programmer (predicting execution times can be difficult or, for some programs, impossible), but it also precludes exploiting fine grain parallelism.

This paper presents the *direct task stack* (Section III-A), a novel work stealing algorithm which virtually eliminates the overhead of task creation for tasks that are never stolen and achieves an overhead for stealing that is at most half of that of state of the art systems such as Cilk++, TBB and the `icc` implementation of OpenMP 3.0. We compare our work stealing scheduler Wool to these three systems and analyze the performance difference in terms of the *task granularity* (the average useful work per task) and *load balancing granularity* (the average useful work per steal) of the computations (Section IV-D).

The effects of these two kinds of fine grainedness are illustrated in Figure 1. On the left, `fib` (with no cutoff) is an example of very small task granularity; it spawns a task for every 13 cycles worth of work. The graph shows how the task management overheads of current implementations outweigh the gains of parallel execution. Wool, on the other hand, achieves speedup from two processors.

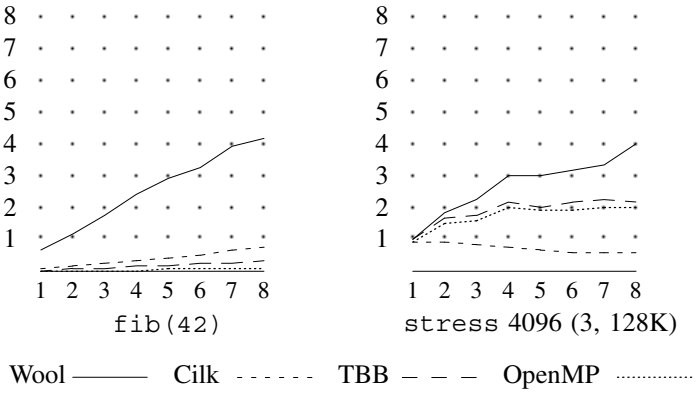


Figure 1. Absolute speedup of `fib(42)` with no cutoff and relative speedup of `stress` (4096, 3, 128K) on Wool, Cilk++, TBB and OpenMP (`icc`)

```
TASK_1( int, fib, int, n ) {
  if( n<2 ) return n;
  else {
    int a,b;
    SPAWN( fib, n-2 );
    a = CALL( fib, n-1 );
    b = JOIN( fib );
    return a+b;
  }
}
```

Figure 2. A simple Fibbonacci function in Wool

On the right, `stress` is a program that repeatedly spawns a balanced tree of tasks with a simple loop at the leaves; execution is serialized between the trees. Because the trees, including the leaves, are relatively small (less than 70k cycles), this program stresses the load balancing performance of the implementation. For some systems, this overhead is large enough that adding processors makes performance worse.

Figure 2 shows the Wool version of the Fibbonacci function (whose performance is shown in Figure 1). `SPAWN` creates a task and adds it to the local work pool of the spawning processor, which then does an ordinary recursive `CALL`, after which it `JOIN`s with the spawned computation. The join does not terminate until the joined task has terminated; this is the main form of synchronization in task parallel models such as

Wool.

The `JOIN` will either find the task still in the local pool, in which case it will invoke the task as an ordinary procedure call (this is often called *inlining* the task), or it will find that the task has been stolen by another processor. In the latter case, the scheduler must find other work for the joining processor until the stolen task is completed.

Work stealing goes back at least to work on parallel functional languages [5], [22] and has since been used in several systems [22], [13], [6], [21] and has also been studied theoretically [4], [23], [3], [14].

Work stealing has also been used in the context of clusters, most recently by Dinan *et.al.* [10] who get very good results on a system with 8K processors. While they do not aim specifically for fine grained performance, they have arrived at some of the same implementation techniques as we.

The issue of fine grained computations has been studied in the context of hardware support in Carbon [15]. In that paper, the authors propose to implement work stealing entirely in hardware with specialized structures for task pools. Interestingly, they collect all of the work queues in a central location; the cores have to get and put tasks there (but have one element local task buffers for latency). In this way, the centralized stealing logic can find work extremely quickly as long as some of the queues are nonempty.

a) Spawn and steal: In a *steal child* system like Wool or TBB, a spawn makes the newly created task stealable whereas in a *steal parent* system like Cilk, the spawned task is immediately executed and it is the continuation of the parent task that is available for stealing. When a steal parent system is executing on a single processor, a task runs when control reaches the spawn; in a steal child system it is instead run when control reaches the corresponding join. This affects memory consumption:

```
for( ; p != NULL; p=p->next) spawn foo(p);
sync;
```

In the code above, Cilk will use constant space for the task pool, whereas Wool and TBB will use space proportional to the length of the list.

b) Join with a stolen task: When a join finds the task stolen but not yet completed, the join is *blocked* until the joined task completes. In order for the scheduler to be greedy (which amounts to ensuring that there is not simultaneously stealable work and idle processors [4]), it needs to find other work to do for that processor in the meantime. One alternative is to (randomly) steal work from the other workers.

This approach has a couple of drawbacks. First, each worker stack may grow beyond the size needed by a complete sequential execution since stealing will build a new stack on top of the blocked join. Second, a direct style library implementation such as Wool and (in the direct style API) TBB suffers from the *buried join* problem that completion of the join now also depends on completing the stolen work, since the latter sits on top of the join in the processor stack (the join code calls the work stealing code and so on). This may lead to a loss of parallelism if the task the join waits for completes but the stolen work is not completed.

If a work stealer is implemented in a compiler back-end, as is the case for Cilk++, the problem can be avoided by using a *cactus stack* where it is possible to start growing a new stack from an arbitrary activation frame, not just the top one.

Another solution to the buried join is *leap frogging* [25] (which is used by Wool), where the worker executing the join is constrained to stealing only from the worker that stole the task to join with. Then it can not be the case that the stolen code has not completed when the join has become unblocked. Leap frogging also solves the memory problem since the code that is stolen is always code the worker would have executed had there been no steal. However, it suffers from the problem that it only allows the joining worker to steal a (small) subset of the work in the system, which can in the worst case lead to sequential execution [24].

Yet another solution to the buried join problem is to have more than one worker thread per processor so that when a worker becomes blocked at a join, that processor switches to another thread where it can do unrestricted stealing. When the join becomes unblocked, the thread can be resumed by any ready processor. This strategy has been implemented in the Microsoft TPL [17].

II. Task parallelism and granularity

We believe that achieving good performance for fine grain task parallelism is important for several reasons:

- Efficient implementation of fine grained parallelism allows more parallelism to be exploited, which is especially important with the expected increase in core counts in future processors.
- The programming model is simplified if programmers do not need to manually avoid spawning small tasks, which is very difficult when task execution times can not be predicted in advance [19].
- Many (soft as well as hard) real time systems have periodic serialization points when input (eg sensor data) is consumed and output is produced. A natural way to program such a system is to parallelize each interval, which then becomes the parallel region.

We distinguish between two aspects of granularity; *task granularity* G_T and *load balancing granularity* G_L. Task granularity measures the frequency of task creation; for instance, the `fib` program is extremely fine grained in this sense (unless a cutoff is used). However, `fib` makes rather modest demands of a work stealing scheduler and typically receives excellent relative speedup. In work stealing, sub trees of the task tree located close to the root are stolen. Because of the shape of the task tree of `fib`, these sub trees are of similar (large) size, so that few steals are needed to distribute work evenly over the processors. We say that while `fib` has small task granularity, it has a large load balancing granularity. In contrast, programs with small parallel regions and frequent serializations have mostly small subtrees close to the root, requiring a large number of steals and get a small load balancing granularity.

With these considerations in mind, we define task and load balancing granularity as $G_T = T_S/N_T$ and $G_L = T_S/N_M$ where T_S is sequential execution time (with no task overheads), N_T is the number of tasks spawned and N_M is the number of migrations of tasks. On a work stealing implementation, N_M is the number of steals.

The task granularity measure depends only on the program and its input, but the load balancing granularity is implementation dependent (we use it as measured by Wool). We believe that there is an underlying implementation independent property of computations which influences the amount of load balancing needed, but we have at present no good definition. Our measure is valid to the extent that it correlates with the performance of other systems as well.

III. Fine grain work stealing

In order to cope with very fine grained tasks, the spawn and join of an inlined task must be performed with minimal overhead while still making the task available for stealing. Spawning involves allocating space for the task, initializing it and inserting it into the task pool, taking care not to expose partially constructed tasks for stealing. To join, it must be determined whether the task has been stolen, and if not, it must be removed from the task pool atomically, so that it is not simultaneously stolen. The code for the task needs to be invoked and passed the arguments and finally the task must be disposed of.

The direct task stack of Wool reduces overhead for these operations by avoiding all indirections. The task pool is made up of fixed size *task descriptors* (rather than pointers to task descriptors) and memory management is simplified by adhering to a strict stack discipline. This design is very unusual; both Cilk++ and TBB uses free list allocation of task structures, keeping only pointers in their task queues.

A. The Direct Task Stack

Figure 3 shows pseudo code for spawning, joining and stealing for the direct task stack algorithm. In the actual Wool system, the implementation is slightly more complex due to additional optimizations (public/private tasks, for instance) and tests for task pool extension.

The functions `spawn_f`, `wrap_f` and `join_f`, and `f` (the function that actually does the work of the task, not shown in the figure) are *task specific*, generated for the declaration of task `f`, while `RTS_join` and `RTS_steal` are part of the run time system and used by all tasks. The join function shows the advantage of task specific functions most clearly, in that the inlined case does a direct call to `f` which is visible to compiler optimization.

A task descriptor has one field for each argument and an additional field `state` with the following values:

- `EMPTY`, signifying a either a transient state during stealing or that there is no task stored in that task descriptor.
- `TASK(w)`, which means that there is a task with wrapper function `w` there that can be stolen or inlined.

```
spawn_f( T_1 a_1, ..., T_n a_n ) {
  top->a_1 = a_1;
  ...
  top->a_n = a_n;
  top->state = TASK( wrap_f );
  top++;
}

wrap_f( Task_f *t ) {
  t->result = f( t->a_1, ..., t->a_n );
}

join_f( ) {
  State s;
  top--;
  s = swap( &(top->state), EMPTY );
  if( is_task(s) )
    return f( top->a_1, ..., top->a_n );
  else {
    RTS_join( top, s );
    return top->result;
  }
}

RTS_join( Task *t, State s ) {
  do {
    while( s == EMPTY ) s = t->state;
    if( is_task(s) )
      s = swap( &(t->state), EMPTY );
  } while( s == EMPTY );
  if( is_task(s) )
    get_wrapper(s)(t);
  else if( is_stolen(s) )
    while( t->state != DONE )
      steal( get_thief(s) );
  bot--;
}

RTS_steal( Worker *victim ) {
  Task *t = victim->bot;
  State s1 = t->state, s2;
  if( is_task(s1) ) {
    s2 = cas_val( &(t->state), s1, EMPTY );
    if( s1 != s2 || victim->bot != t) {
       if( s1 == s2 ) t->state = s1;
    } else {
      t->state = STOLEN( self_idx );
      victim->bot = t+1;
      get_wrapper(s1)(t);
      t->state = DONE;
    }
  }
}
```

Figure 3. Code for spawn, join and steal

- `STOLEN(i)`, denoting a task that has been stolen by worker `i` (knowing the thief is necessary for leap frogging).
- `DONE`, for a task that has been stolen and where the thief has completed its execution.

This information is readily packed in a single pointer by representing `TASK(w)` with a pointer to the (evenly aligned) wrapper function and use odd integers for the other values. The function `is_task` returns true if its argument is a `TASK(w)` and `get_wrapper` and `get_thief` extracts the wrapper function pointer or worker index from an appropriately formed task state. The per-worker variables `top` and `bot`[1] manage the stack; `top` is private to the owning worker whereas `bot` is shared with thieves.

On a machine that does not reorder stores, such as for instance the x86 and SPARC under TSO, no synchronization is needed for spawn operations as long as the write which makes the task stealable is the last write.

Most algorithms for work stealing [13], [3], [7], [16] synchronize thief and victim based on the values of the `top` and `bot` pointers, but the direct task stack uses the `state` field in the task descriptor instead. At a join, the owner acquires the task using an atomic exchange on `state`, while the thieves use a compare and swap (CAS) operation to atomically write `EMPTY` to `state` if its value is still the same as previously read (`s2`). Most of the algorithms mentioned above use a synchronization method between thief and victim based on an algorithm by Dijkstra [9] which avoids atomic operations but instead depends on sequential consistency (see *e.g.* [2]), something that most processors today do not support. Instead, explicit memory barrier instructions must be used and these are often at least as expensive as atomic instructions. In principle, a CAS operation (or an unconditional atomic exchange) need not be more expensive than a combined load and store since it essentially involves acquiring the cache block in modified (exclusive) state. Both the duplicating queue of the Microsoft TPL and the idempotent work stealing [18] avoid Dijkstra style synchronization in favor of atomic operations (in the latter case by exploiting synchronization elsewhere in the algorithm).

Synchronizing on the task descriptor instead of on the `top` and `bot` pointers confers several benefits. First, it decouples the thieves from the frequently changing `top` pointer, which can be kept private to the owner. Second, in a situation with idle workers and no stealable tasks, the idle workers will have cached all bot pointers and will effectively be polling the task descriptors to which the next spawns will be made. When a worker spawns, only that cache block needs to be transferred to accomplish the steal since the same block contains both the data needed to run the stolen task and the location which signals the availability of the data.

There is no explicit synchronization for the `bot` pointer. Instead, it is implicitly owned by the worker that has stolen (or joined with) the task `bot` points to. This is apparent in `RTS_steal`, where it is only updated after the task has been acquired. Similarly, when the owner has joined with a stolen task, `bot` must point at the task immediately above the joined one and is thus owned by the local worker.

Because of the absence of explicit synchronization, a thief can read the bot pointer, then be arbitrarily delayed (interrupts,...) and then trying the CAS operation. This leads to a potential problem if the owner has already joined with that task and at least one other and then spawned new tasks such that the same task descriptor is again valid with the same kind of task. Then the CAS succeeds and the `bot` update is performed by the thief. However, there may now be unstolen tasks *below* the one pointed to by `bot`, invalidating its implicit synchronization protocol. Hence a thief checks the victim's current value of bot after acquiring the task, and backs off from the steal if that value is wrong. Writing back the old value of `state` is appropriate since the transient value (`EMPTY`) only makes thieves abort and the joining owner wait (see below). These back offs are infrequent, always below 1% of successful steals.

B. Private tasks

The main part of the cost of an inlined task is the atomic swap instruction in the task specific join routine. This instruction costs about 16-20 cycles on current x86-64 implementations, but is necessary in order to coordinate with thieves.

We reduce the cost of this mutual exclusion by making certain task descriptors (elements of the task pool array) *private*, by adding a `public` flag to each task descriptor (this optimization is not reflected in the code in Figure 3). If the flag is set, the task stored there (if any) can be stolen, and mutual exclusion is needed when joining with it. Otherwise, any steal attempt for this task will fail and no mutual exclusion is needed.

Since private task descriptors can not be stolen, having many public task descriptors minimizes the risk that a thief goes unnecessarily idle, but carries a higher synchronization overhead than fewer public descriptors. In general, if the task tree is balanced, fewer public task descriptors suffice to keep all workers busy while very unbalanced trees require more.

With a conventional cut off [11], [12], the decision to not make a task can not be revoked because none of the administrative data structures needed for a task are constructed. In the private task strategy we use the fact that the major cost is that of synchronization; the task descriptor is built (which is cheap) while we postpone the expensive synchronization. This allows the cut off to be adjusted dynamically; a task that was marked as private when it was spawned can change status to public later as necessary. However, because no synchronization is used, the change in status must be performed by the owner of the affected task descriptors. It is the responsibility of the thieves to notify the owner of a task pool when more tasks must be made public.

We have implemented the following *trip wire* scheme: The nth highest public task descriptor is marked as a trip wire.

[1]Different authors refer to the pointers into the task pool by different names. We have chosen to emphasize the close relationship to a stack, so that the local worker pushes and pops tasks using the `top` pointer and thieves operate towards the bottom of the stack using the `bot` pointer.

When that task is stolen (which happens only once for that trip wire), the thief notifies the owner, which makes some more task descriptors public. The notification is accomplished by means of a per-worker flag that must be checked regularly (for instance on every spawn and join operation). Note that privacy is independent of validity; `EMPTY` (invalid) elements in the task pool can be marked public.

Changing state from public to private requires that the owner first acquires the task (to avoid race conditions with thieves), making it convenient to do this as part of the join operation. The run time system detects an opportunity for making tasks private when it has inlined many public tasks; this is precisely the kind of situation private tasks are designed to exploit.

Other schemes for cut off have been proposed, for instance by Duran *et.al.* [11] who present an adaptive strategy based on collecting profiling information at run time, but it differs from the present scheme since the cut off decision is not revocable. Thus they have to assess beforehand if creating the task is likely to be beneficial. Dinan *et.al.* [10] present a *split queue* technique similar to private tasks but using worker based synchronization.

IV. Experimental results

This section provides the experimental evaluation of the techniques we have presented. First, we present the benchmarks we use and characterize them in terms of task and load balancing granularity. We then present results concerning the contribution of each of the techniques we have discussed, followed by a comparison with the example task schedulers and an analysis of the reasons for the performance differences.

Except were explicitly noted, all experiments were run on a dual quadcore Opteron server with 16GB of memory running Ubuntu Linux 9.04, kernel 2.28-15.

A. Benchmarks

For our measurements we have chosen mostly simple programs from various sources, and modified them as follows: Since our focus is on fine-grain computations, we run them with unusually small inputs, and to get measurable execution times (at least about a second) as well as offsetting initialization costs, we repeat the benchmark kernels a suitable number of times. This leads naturally to the kind of program structure with a sequence of small parallel regions discussed in section II.

We study the effect of granularity by varying the size of the parallel computation and (inversely) the number of repetitions of the kernel. We thus get several workloads for each program.

`cholesky`: Sparse matrix factorization on a random square matrix using explicit nested tasks. Taken from the Cilk-5 distribution. Parameters are the number of matrix rows and the number of nonzero elements.

`mm`: Dense matrix multiply (not blocked) of square matrices with the outermost loop parallelized. Taken from the Wool distribution. The parameter is the number of matrix rows.

`ssf`: Based on the Sub String Finder example from the TBB distribution. For each position in a string, it finds from which other position the longest identical substring starts. The string is given by the recursion $s_n = s_{n-1}s_{n-2}$ with $s_0 = "a"$ and $s_1 = "b"$ where n is the parameter in the workload.

Table II
Optimizing inlined tasks; single processor executions

Version	Time (s)	Overhead (cyc)
Base	18.9	77
Synchronize on task	7.8	29
Task specific join	5.9	19
Private tasks (no private)	6.0	19
Private tasks (all private)	3.0	3
Serial	2.4	0

`stress`: A micro benchmark written to have a precisely controllable parallelism and granularity. The program creates a balanced binary tree of tasks with each leaf executing a simple loop making no memory references. The granularity of the leaf tasks can be varied by varying the number of iterations of the loop and the granularity of the parallel regions is controlled by that value and the depth of the tree. The parameter is the depth of the tree. We give two sets of workloads with leaf granularity of 256 iterations (512 cycles) and 4096 iterations (8K cycles).

The implementations of the programs for the different systems share the same structure, except for the TBB version where we have used the continuation passing API for efficiency. In the loop based programs (`mm` and `ssf`), the OpenMP implementations use OpenMP parallel for loops rather than using tasks trees to implement loops.

Table I presents some statistics on the benchmark programs. The parallelism is calculated from a span (critical path length) measurement facility in the Wool run time system. Cilk has a similar tool which gives numbers that are close to the Wool numbers. We report parallelism both in an abstract model where load balancing and communication cost nothing and in a more realistic model. Here potentially parallel computations are assumed to be executed sequentially if the savings from parallel execution are less than 2000 cycles. Otherwise, they are assumed to be executed in parallel with an extra cost of 2000 cycles added. RepSz is the average serial execution time per repetition in kilocycles.

We can make several observations from this table:

- Load balancing granularity increases with the parallel region size (in this case the RepSz), and also (as predicted by theory [4]) with parallelism.
- Load balancing granularity decreases with increasing processor counts. While worst case bounds scale linearly in the number of processors [4], we invariably see the number of steals growing faster than the number of processors.
- The parallelism of computations with finer tasking granularity (such as `cholesky` and `stress` with the smaller leaf size) are more affected by realistic overheads in the parallelism computation than workloads with bigger tasks.

Table I
WORKLOAD CHARACTERISTICS

Params	Reps	Parallelism		RepSz	Granularities							
		0	2000		G_T	$G_L(2)$	$G_L(3)$	$G_L(4)$	$G_L(5)$	$G_L(6)$	$G_L(7)$	$G_L(8)$
`cholesky`, parameters: number of rows, number of nonzeroes												
250, 1k	4K	22.2	9.0	8653	211	25	12	9	6	5	4	4
500, 2k	1K	48.8	20.9	68349	225	87	39	26	18	14	12	10
1k, 4k	256	87.1	46.5	503035	218	304	117	78	52	40	33	28
2k, 8k	64	116.4	80.7	3878016	216	1148	406	260	168	126	102	87
4k, 16k	16	203.7	183.9	30890437	204	4442	1458	892	554	414	337	275
`mm`, parameters: number of rows												
64	16K	54.3	30.6	976	15486	915	144	211	92	71	59	58
128	2K	117.5	109.8	19496	153513	19247	2582	4050	1558	1165	914	879
256	256	240.3	237.0	226227	887163	69860	18934	43917	14538	11009	8303	7951
512	32	460.8	461.5	2438719	4772444	602153	177968	375368	110023	89959	68787	65060
`ssf`, parameters: number of concatenations												
12	16K	37.5	37.5	552	3858	119	54	65	35	28	25	22
13	8K	79.4	79.6	1432	6172	473	118	165	78	61	52	47
14	4K	152.3	152.5	3936	10469	1000	279	342	167	131	112	100
15	2K	291.1	293.0	10849	17814	2774	770	721	371	278	230	205
16	1K	527.3	523.8	29985	30411	7474	1993	1704	901	696	574	510
`stress`, leaf size 256 iterations, parameters: tree height												
7	128K	54.5	7.1	84	663	51	10	11	7	6	5	5
8	64K	96.6	12.4	147	575	76	17	15	10	8	7	6
9	32K	169.7	21.4	293	573	157	32	27	16	13	11	10
10	16K	290.8	36.8	585	572	244	51	43	26	20	17	15
11	8K	478.9	61.2	1174	573	482	89	78	42	33	27	23
`stress`, leaf size 4096 iterations, parameters: tree height												
3	128K	7.7	4.6	67	9501	65	22	18	15	13	11	10
4	64K	15.1	8.3	133	8891	130	33	29	21	17	16	15
5	32K	29.3	14.0	267	8604	237	53	50	32	26	22	21
6	16K	56.7	25.0	535	8490	527	88	83	50	40	34	31
7	8K	108.1	44.7	1067	8401	898	149	140	78	63	52	47

Params are the parameters of each iteration (see section IV-A), **Reps** is the number of repetitions, **Avg. Parallelism** is T_1/T_∞ assuming stealing overhead zero or 2k cycles, **RepSz** is the size in 1000s of cycles of each repetition, **Granularities** are the average task size in cycles and the average steal interval in 1000s of cycles for 2 to 8 processors. Throughout, k is 1000 and K is 1024.

B. Performance of inlined tasks

In this section we asses the performance we achieve for inlined tasks by comparing it to three alternatives. The **base** alternative uses per-worker locks for mutual exclusion of thieves and victim; a worker takes the lock for join (but not spawn) operations. No state is stored in the task descriptors, instead `top` (which is not private here) and `bot` are compared to determine whether joins and steals succeed. There are no task specific join operations; all of the work is done in the RTS. The **synchronize on task** alternative eliminates the locks and makes `top` private, instead using atomic swap on the task state. It differs from the code in Figure 3 in lacking task specific joins. The **task specific join** version adds these (so it is basically the algorithm presented in Section III-A) while the **private tasks** alternative adds private tasks as described in Section III-B.

We will use `fib` as our guide since it is an extremely spawn intensive program which does very little work in each task. Table II gives execution times on a single processor of `fib(42)` for the different parallel variants as well as for a purely serial version without any tasking overhead. We also give the overhead per task over a procedure call, in cycles, computed as $(T_1 - T_S)/N_T$ where T_S is the execution time (in cycles) of the serial version, T_1 is the execution time of the parallel version and N_T is the number of tasks spawned. We think that this is the relevant comparison, since the alternative to spawning a task typically is a procedure call.

For the private tasks, we show both the best case where all tasks are private (which is what happens when running a single worker) and the worst case where no task is private. Note that the overhead of checking whether a task is public or private is small (the difference between the **no private** and the **task specific join** rows).

We see an approximately six fold speedup from the baseline to the all private tasks results; at that point, the overhead of spawning and joining with a task over the cost of using a procedure call is just three cycles. When running in parallel with some tasks public, the overhead is slightly higher but still closer to three cycles than 19 as it typically is some tasks close to the top of the task tree that are public. This level of overhead virtually eliminates the need for granularity control by the application programmer.

C. Performance of stealing

In this section we assess the effectiveness of our stealer with three alternatives which use per-worker locks for achieving mutual exclusion between thieves and to protect `bot`. This

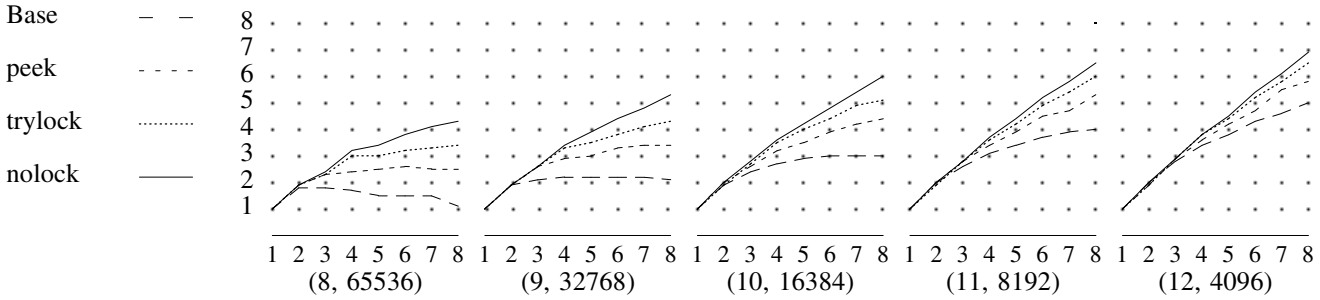


Figure 4. Different implementations of stealing

Table III
COSTS (CYCLES) OF INLINED AND STOLEN TASKS

System	Inlined	2	4	8
Wool	3–19	2 200	5 600	10 400
Cilk++	134	31 050	73 600	110 400
TBB	323	5 800	14 000	30 000
OpenMP	878	4 830	9 200	20 240

lock is taken by thieves when stealing and by victims when joining with stolen tasks. Since bot is protected by the lock, thieves never need to back off.

In the **base** alternative, the thief attempts to take the lock immediately after selecting a victim (at the start of RTS_steal), while in the the **peek** alternative, a thief first reads the task descriptor pointed to by bot, checking that the task is valid and only attempts to take the lock in that case. The **trylock** strategy, in addition to peeking, uses a trylock operation that fails if the lock is held rather than waiting for it to be released; the steal operation is then aborted. Finally, **nolock** is the direct stack algorithm that has no explicit lock protecting `bot`.

We have evaluated the steal algorithms using the `stress` micro benchmark with a leaf granularity of 256 iterations (512 cycles), presenting the results in figure 4. Since we investigate methods with larger overheads, we have shifted the parallel region sizes one step compared to Table I so that we show data from 64K repetitions to 4K repetitions.

As we move to the right among the plots, the gap between the methods closes, as it should when parallel slack increases [13] and stealing becomes less frequent. The same happens if we go to the left *within* a plot (towards fewer processors).

D. Comparing to the state of the art

In this section we compare the performance of a version of Wool incorporating our optimizations to that of the three state of the art systems. For Cilk++ we use version 4.3.4 (build 7007), for TBB version 2.1, and for OpenMP the Intel C Compiler `icc` version 11.0. The benchmark programs are those discussed in Section IV-A, as well as a micro benchmark measuring inlined task and load balancing performance (taken from Podobas *et.al.* [20] with TBB data added).

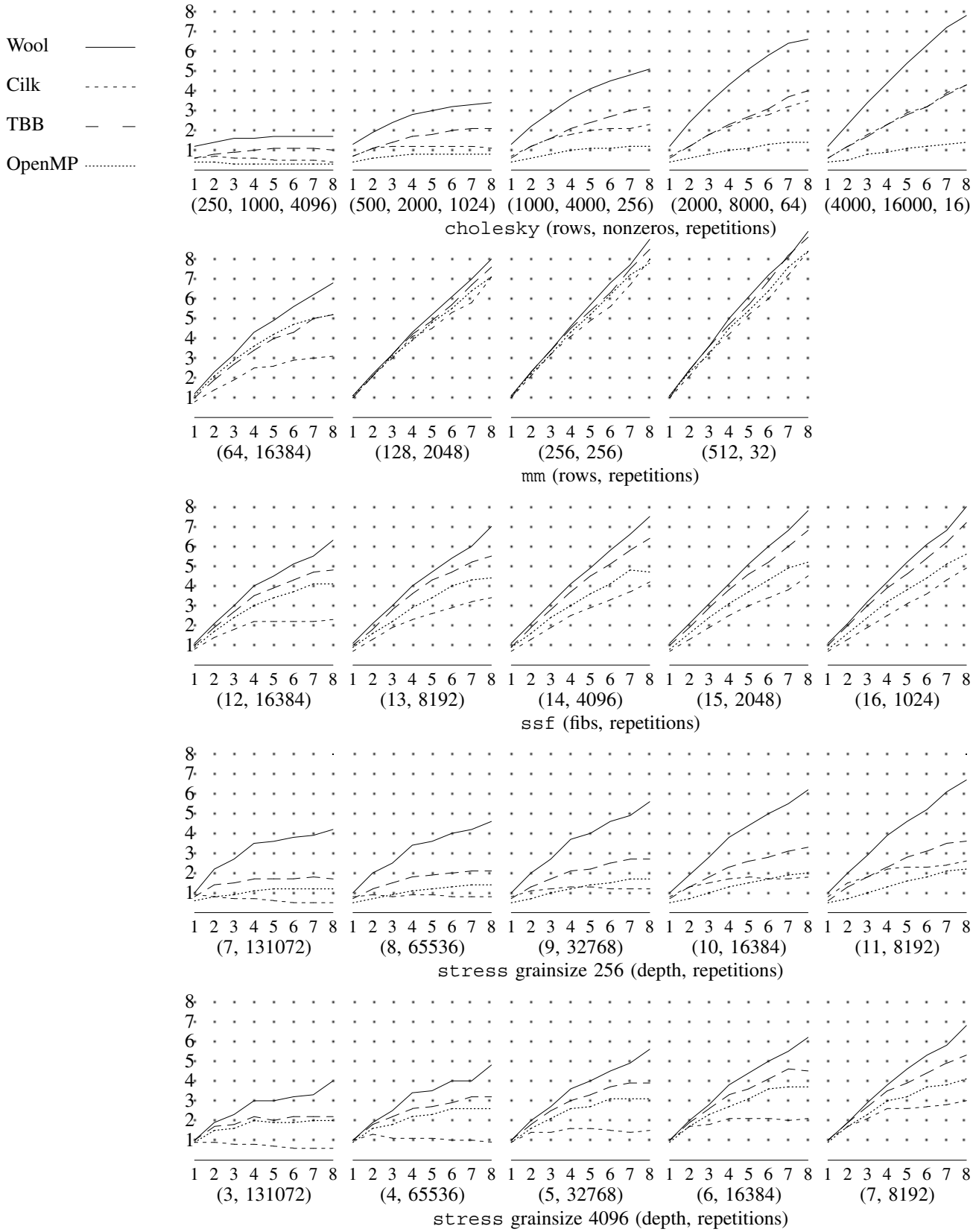
1) Micro benchmark evaluation: We evaluate the cost of inlined tasks using the methodology of Section IV-B and give the results in the Inlined column of Table III. We base our evaluation of load balancing cost on the results and methodology of Podobas *et.al.* [20]. Briefly, we use the `stress` benchmark to measure load balancing overhead as the difference in execution time between a binary tree of tasks with height k, where each of the 2^k leaves is a sequential computation C, on 2^k processors versus the time to run C on a single processor, giving the results in Table III, columns labeled 2, 4 and 8 (processor counts).

The results show clearly the effectiveness of the implementation techniques discussed in Section III-A and III-B. For Wool, we give a range from "no tasks public" to "all tasks public" where the average typically is close to the lower end for deep trees of small tasks.Even Cilk++, which has the lowest inlined task overhead of the systems, is close to 40 times the low Wool number and about 7 times the high Wool number. This is surprising since Cilk++ is a mature and highly tuned system with compiler support, and it prompted us to study the implementation to find the cause.

Inspection of generated code and the run-time system revealed that Cilk++ uses a cactus stack; stack frames are not contiguous and are allocated from a free list in the worker descriptor. All calls get this overhead, not just spawns. Also, spawning goes through a wrapper function which calls the task function proper that is responsible for checking that the continuation is not stolen (which it does by tail calling the run-time system).

In contrast, the Wool code is very similar to the sequential code, using the stack in the same way. Although the Cilk++ implementation is based on GCC, it appears less willing to inline than the GCC used to compile Wool, which even inlines recursive calls as a kind of unrolling. Hardware profiling indicates that Cilk++ executes more than four times as many instructions, especially memory references and spends around 10% of its time on memory fences.

When it comes to the steal cost measurements (columns labeled 2,4 and 8), the two processor case represents the base cost of stealing one task and then synchronizing with the victim for the join. Since there is only one other processor, this cost does not include costs for searching for work at several victims. Moving to the right, we see that the cost does not scale linearly with the number of levels in the steal/join tree (the logarithm of the number of processors), but grows faster for all of the systems.



The x-axis shows number of processors while the y-axis shows absolute speedup (that is, speedup relative to a purely sequential program) for `cholesky`, `mm` and `ssf` while for `stress` we show speedup relative to single processor Wool executions.

Figure 5. Speedup of fine grained applications on Wool, Cilk++, TBB and OpenMP

Table IV
A SIMPLE STEAL COST MODEL, COMPUTED AND MEASURED SPEED UPS

System	2	4	8
Wool	2.0 (2.2)	3.9 (4.3)	7.1 (6.8)
Cilk++	1.9 (1.4)	2.8 (2.5)	3.2 (3.1)
TBB	2.0 (1.9)	3.7 (3.4)	5.9 (5.2)

We believe that the reason is that multiple thieves interact through the cache system, for instance by invalidating cache lines that other thieves need, and also that the thieves have to search for work at more workers.

The Cilk++ implementation stands out as having a very high steal overhead, more than ten times that of Wool. We believe that the reason is a combination of extensive locking (up to two task descriptors and the victim's worker descriptor) with a large memory footprint for the stealing (possibly due in part to the hyperobject implementation). Of the overhead, more than half is spent in the kernel (suggesting lock contention) with most of the reminder being due to coherence traffic.

2) Benchmark results: We report absolute speedup over sequential versions of the programs with no task management constructs. Because of the optimizations to inlined tasks that our version of Wool incorporates, the single processor Wool execution times are very close to the sequential times.

It is instructive to compare Cilk with TBB; Cilk is faster with inlined tasks but TBB steals much faster. Hence we see Cilk++ catch up when tasks are small and steals less frequent, as when going from left to right (increasing load balancing granularity) for `cholesky`.

a) A simple performance model: The `mm` program with the smallest matrices (64 by 64) is instructive. We can use the steal cost measurements from Table III to make a simple model for the speed up we could expect from the work stealing systems (the OpenMP version is coded with a work sharing loop rather than with tasks). In each repetition, 63 tasks are spawned each of which will do one iteration of the outermost loop. Comparing the repetition size (976k cycles) with load balancing granularity for eight processors, 58k cycles per steal (for Wool), we have almost 17 steals per repetition, on average. Of these, 7 are necessary to distribute work to all processors and roughly correspond to the steal cost micro benchmark discussed in Section IV-D1. The remaining roughly 10 steals would be needed for further load balancing when the first round of steals have not produced the same amount of work for all processors. Assuming that these steals do not happen at the same time and that they find work quickly, the relevant steal cost is the one for the two processor case. For each steal, there are two processors that will have to pay this cost; the thief (obviously) and the victim that will have to join with the thief.

Generalizing this argument we have, for p processors, an approximate cost of $C_p + (W + 2 \times (S_p - (p-1)) \times C_2)/p$ where C_2 and C_p are the steal costs for two and p processors, respectively, W is the sequential work (RepSz), and S_p is the number of steals. Table IV gives the numbers from the model as well as the measured numbers from Figure 5 (within parenthesis). The model typically overestimates the speedup, as is to be expected since the assumptions were systematically optimistic. It is nevertheless interesting since it illustrates the interplay between parallel region size, load balancing granularity and steal cost. It also suggests that load balancing granularity carries over between similar systems; the number of steals for Wool is used for the estimates for all of the systems.

b) Sources of overheads: We have further investigated the sources of overheads in the Wool executions. To this end we have used instrumentation that can distinguish between CPU time that is spent in startup and shutdown (**TR**), application code acquired through leap frogging (**LA**), other application code (**NA**), stealing (**ST**) and leap frogging (**LF**). Figure 6 shows the breakdown for different workloads and processor counts, normalized to the **NA** category for sequential code (which is all of the time in that case). Since the plots show CPU time, increasing times do in general not imply a slowdown when processors are added, just sub linear speedup. Note that different workloads use different scale; the plots for the single processor cases vary in size. This set of measurements have been made on a 12-core machine with two six core Opteron processors and 2.6GHz clock frequency.

These data can shed light on several issues related to possible improvements on Wool. First, the **LF** time gives an upper bound on the possible improvements from a more effective handling of blocked joins. It appears that, for the programs we have studied, leap frogging is adequate. In fact, the **LA** part is small enough that one would say that simply waiting would be adequate. Second, for the programs where CPU time increases most, it is primarily due to stealing and the application time itself. For `cholesky` we have confirmed that the increase comes from coherence misses in the cache hierarchy, that is, from communication.

V. CONCLUSIONS

While it remains true that there is no such thing as a free lunch, we have shown that there is indeed such a thing as an almost free spawn, demonstrating overheads of less than ten cycles. This obviates the need for application level granularity control; any function call can be spawned with negligible overhead. We have shown that by providing synchronization on demand, most join operations can avoid that overhead.

In terms of stealing, we have presented the direct task stack which coordinates thief and victim on the task descriptor under consideration, rather than on the worker descriptor. We have shown that reducing locking is an important factor in achieving good performance.

We have also presented an analysis of the granularity of task based parallelism in terms of the frequency of task creation and load balancing and used that model to shed light on the experimental results.

The resulting system, Wool, significantly outperforms existing implementations for fine grained programs. Cilk++ is able to provide stronger guarantees about memory consumption and

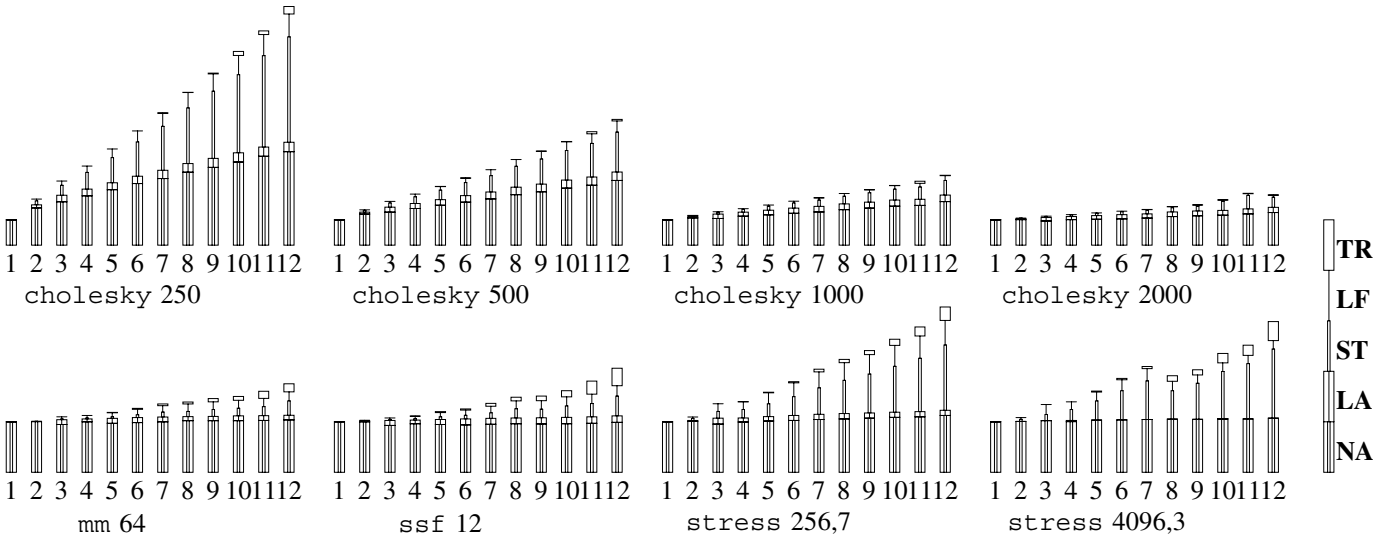


Figure 6. Breakdown of CPU time, selected workloads

scheduling efficiency due to its steal parent execution order and cactus stack but has a stealing overhead on the order of ten times that of Wool. It remains to be seen if this is the inevitable price of the strong guarantees or if it is possible to implement the parent stealing cactus stack more efficiently. In contrast to the other systems, Wool is a C library and therefore does not support C++ exceptions.

REFERENCES

[1] OpenMP application programming interface, version 3.0, May 2008. Available from `www.openmp.org`.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, New York, NY, USA, 1998. ACM.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[5] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194, New York, NY, USA, 1981. ACM.

[6] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.

[7] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM.

[8] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of the intel threading building blocks runtime system. In *International Symposium on Workload Characterization (IISWC 2008)*, September 2008.

[9] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[10] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scalable work stealing. In *SC09*. ACM, November 2009.

[11] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[12] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP Task Scheduling Strategies. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *Lecture Notes in Computer Science: Proceedings of the 4th International Workshop on OpenMP*, volume 5004, pages 100–110. Springer, Springer, May 2008.

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[14] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289, New York, NY, USA, 2002. ACM.

[15] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, New York, NY, USA, 2007. ACM.

[16] D. Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, 2000. ACM Press.

[17] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *SIGPLAN Not.*, 44(10):227–242, 2009.

[18] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. *SIGPLAN Not.*, 44(4):45–54, 2009.

[19] Stephen L. Olivier and Jan F. Prins. Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP*, pages 63–78, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A comparison of some recent task-based parallel programming models. In *MULTIPROG-3*, January 2009.

[21] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[22] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[23] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 91–100, New York, NY, USA, 2009. ACM.

[24] Jim Sukha. Brief announcement: a lower bound for depth-restricted work stealing. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 124–126, New York, NY, USA, 2009. ACM.

[25] David B. Wagner and Bradley G. Calder. Leapfrogging: a portable technique for implementing efficient futures. *SIGPLAN Not.*, 28(7):208–217, 1993.