Transitive Leap Frogging

Karl-Filip Faxén
Swedish Institute of Computer science
kff@sics.se

Abstract

Task parallel systems that are based on fork-join parallelism need to find useful work for the processor that reaches the join first. While the problem can be solved in a straight forward manner using a cactus stack, implementations that are built on top of a conventional stack, for instance in order to support standard calling conventions, face complications. This has led to to a number of different trade offs between scheduling flexibility, space useage and complexity. This paper presents *transitive leap frogging*, a new trade off that appears quite efficient in practice.

1. Introduction

Nested fork-join (NFJ) parallelism (this is also often referred to as *task parallelism*) is an increasingly popular approach to programming shared memory multiprocessors. It has been implemented by a number of different programming systems including the Cilk family [8, 11], OpenMP 3.0 [1], the Microsoft Task Parallel Library [10], the Intel TBB [14], and Wool [5, 6]. Figure 1 gives an example of an NFJ computation. In NFJ, parallelism in a computation is created dynamically by *fork* operations (nodes a-d in the figure) while *join* operations synchronize two branches created by a previous fork (nodes j-m). The nestedness means that any computation can at any time fork new parallelism, leading to a tree like structure. While the parallel branches of a fork are in general symmetric, most NFJ systems are presented to the programmer using a task construct (we will refer to this as the *NFJ task* model) where one of the branches is *spawned* as a task and the other is executed by the processor executing the fork. Figure 2 gives a simple NFJ task implementation of the naive Fibonnacci function in a syntax close to that of Cilk (Figure 1 describes a kind of parallel execution trace of this function). Forking a computation is achieved by the `spawn` keyword and joining is indicated by `sync`. We have indicated the tasks created by dashed gray lines, making the tree structure explicit.

When implementing an NFJ system, the overhead of the fork and join operations must be kept very low since these are the only synchronization operations available to the user of the system. Hence they will be at least as frequent as locks, semaphores and similar constructs are in a system

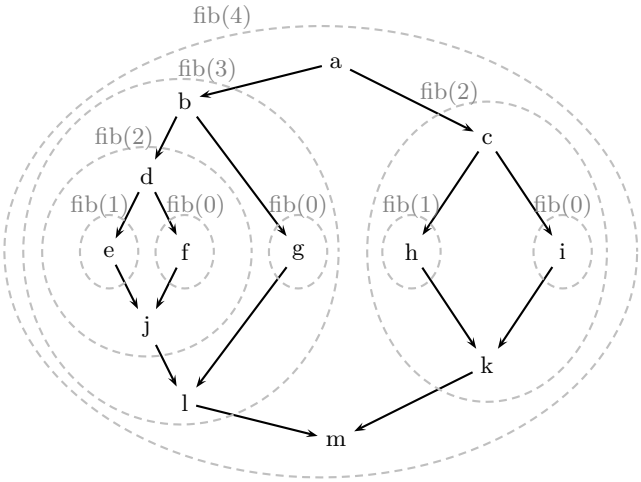

Figure 1. Structure of a nested fork-join compuation

```
int fib( int n ) {
  if( n<2 ) return n;
  else {
    int j = spawn fib( n-1 );
    int k = fib( n-2 );
    sync;
    return j+k;
  }
}
```

Figure 2. The `fib` program

based on threads. The implementation must also accept new parallelism during execution rather than requiring that all parallel activities are declared from the beginning. One popular approach to scheduling NFJ tasks, which meets these requirements, is *work stealing*. In such a scheduler, each processor maintains a private *task pool* containing tasks it has spawned. Processors that have nothing to do choose some other processor and attempt to *steal* a task from the task pool of that processor; in this way load is balanced and all processors are kept busy. A join (sync) is implemnted by checking if the task to join with, the *join target,* (representing the spawned branch of the fork) is stolen or not. If it is not, the joining processor executes the task much as a conventional procedure call. Otherwise, the join *blocks* and cannot complete until the thief has completed the stolen task.

Avoiding that blocking joins cause processors to become idle is one of the trickiest parts of implementing NFJ tasks. One could think that the joining processor can simply steal

some work from some arbitrary other processor, but that idea quickly runs into a couple of problems:

Consider a join occurring deep in a recursive computation. The worker executing the join has many activation records on its stack (the join might be very close to a leaf of the computation). Now, assume that the task that the worker steals while waiting for the join target to complete is close to the root of the computation (in fact, this is even likely since a thief always steals the oldest not yet stolen task of its victim). While executing this task, a large number of activation records are again pushed on the stack of the worker, on top of the previous ones. Then another blocking join might be encountered, and so on. This may lead to an explosion in stack space used.

Similarly, there is a problem with loss of parallelism when the join target completes execution before the joining worker has completed the task it has stolen. Since the activation record associated with the join is not on top of the stack, it cannot be resumed, and there is thus logically runnable work in the system that is not available to the scheduler.

Leap frogging [15] is an approach to solving this problem. It allows the joining worker to keep busy by stealing work, but restricts the choice of worker to steal from (the *victim*) to be only the worker that stole the join target. In this way, the joining worker steals a task that is part of the computation rooted at the task whose completion it is waiting for. This solves the memory problem since the stolen task will be even deeper (further from the root) in the computation than the task containing the join. This guarantees that the stack will not grow deeper than in a sequential execution of the same program. Similarly, the time problem is solved since the join target can not complete execution while the joining worker executes some other task, since the only other tasks that the joining worker are allowed to execute must be completed before the join target completes.

However, for some unbalanced computations like the Unbalanced Tree Search (UTS) benchmark [12], we have found that leap frogging is not able to solve the problem; it is too restrictive (see the blue line in Figure 6 for the less-than-linear speedup of leap frogging). To overcome this problem, we have generalized leap frogging to *transitive leap frogging* by essentially allowing the joining worker to steal also from workers that have stolen from the original thief and so on. This still avoids both the stack blow up and the loss of parallelism since it is still the case that the joining worker only steals tasks that are descendants of the join target.

2. Transitive leap frogging

The transitive leap frogging algorithm is a generalization of and builds on traditional leap frogging [15]. When blocked on a `sync`, the scheduler first attempts to leap frog, that is, steal a task from the thief. If that fails because all tasks spawned by the thief have been stolen by other workers, the scheduler attempts to transitively leapfrog into these. It can do this by following *leads* left by a thief in the task pool of its victim (in the slot that contained the stolen task). Traditional leap frogging also uses leads; the thief records its own identity in the task pool of the victim.

In order to maintain the invariant that a worker blocked trying to sync with a stolen task T is allowed to steal only tasks that are descendants of T, the blocked worker is only allowed to traverse workers who have stolen tasks that the thief spawned *as part of executing* the stolen join target. These are the tasks that the thief has spawned after the steal operation. Therefore, the lead that the thief leaves for the victim also records the value of its task pool `top` pointer (in the form of an index into the pool). Tasks spawned later will be placed in slots above that position, so these are exactly the tasks available for the transitively leap frogging victim.

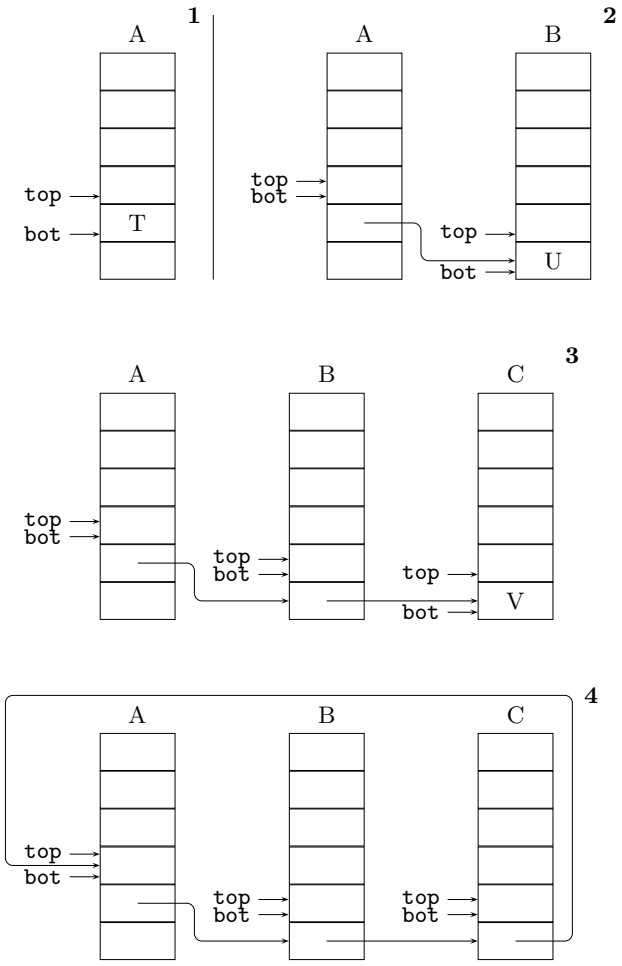

Figure 3. Data structures (example with three workers)

2.1 An example

Consider the example in figure 3: When a worker B steals a task T from another worker A, it records a *lead* in the task pool slot it stole T from. A lead tells

1. who stole the task (B in this case), and
2. the current top pointer in the thief's task pool (t_B)

(shown as an arrow from the task pool slot of the stolen task to the indicated slot in the thief's task pool). In classic leap frogging, A is only allowed to steal from B, so the t_B value is not needed.

When B executes the stolen task T, it will typically spawn additional tasks (such as U in the figure) that are part of the task tree rooted at T. These will always be placed in B's task pool at or above the saved t_B value, allowing A to determine which of the tasks stolen from B's task pool were part of the task tree rooted at T. Transitive leap frogging allows A to steal from workers who have stolen these tasks.

```
1  bool trlf( Worker *thief, int index )
2  {
3    while( index < thief->bot ) {
4      Task *t = thief->pool[index];
5      if( isStolen(t) ) {
6        Worker *nThf = thiefOf(t);
7        int    nIdx = idxOf(t);
8        if( !visited[nThf] ) {
9          visited[nThf] = true;
10         if( steal(nThf) || trlf(nThf, nIdx) ) {
11           return true;
12         }
13       }
14     }
15     index++;
16   }
17   return false;
18 }
19
20 void join( Worker *self )
21 {
22   if( isNotStolen(self->top) ) {
23     run( self->top );
24   } else {
25     while( isStolen(self->top) ) {
26       Worker *thief = thiefOf( self->top );
27       if( !steal(thief) ) {
28         clear(visited);
29         trlf( thief, idxOf(self->top) );
30       }
31     }
32   }
33 }
```

Figure 4. The transitive leapfrogging algorithm

When C subsequently steals the task U from B, the slot of U records the identity of C and its current top pointer value t_C. C then spawns V, bringing us to the situation in sub-figure **3**.

At this point, worker A attempts to join with task T. Since T is not finished, A needs to suspend its current task and find other work to do. In classic leap frogging, A is only allowed to steal from B, which does not have any stealable tasks since it has only spawned U and U was stolen by worker C. In transitive leap frogging, however, A is allowed to follow the lead to C since C is in fact working on the tree rooted at T. Worker A can use the t_C value (the arrow from U's slot to V's slot) to find task V, which it steals. It records its current top value in V's slot and starts executing V, bringing us to sub-figure **4**.

At this point, if worker C tries to join with task W, it will suspend and try to leap frog. However, all tasks in worker A's task pool are already stolen (`top` = `bot`) so classic leap frogging fails. Also, since the t_A value recorded by A when it stole V is also equal to `bot`, indicating that there are no tasks in the tree of V that have been stolen from A, so there are no opportunities for transitive leap frogging either. Note that worker C is not allowed to follow leads from the task pool slots *below* the one designated by the t_A value it follows since the tasks stolen from these slots (for instance T) are from before A stole V can therefore not be part of the task tree rooted at V.

2.2 The algorithm

Figure 3 gives the transitive leap frogging algorithm. The function `join()` implements the join operation. It first checks the state of the join target, which is one of **Not-Stolen** if the task has not been stolen, **Stolen** if the task has been stolen and is beeing executed, and **Done** if the task has been stolen but is completed by the thief. If the task is **Stolen**, the identity of the thief is fetched and a steal from the thief is attempted (this corresponds to leap frogging). If the steal fails, the function `trlf()` is called with a pointer to the thief and an index into that its task pool (corresponding to the `top` pointer at the time of the steal), also extracted from the stolen task. The function then loops over those of the task pool slots of the thief that are more recent than the steal (that is, from the given index and up to the `bot` index). For each **Stolen** (in particular, not **Done**) task it finds, `trlf()` extracts the identity of the thief and either steals from it or, if that fails, calls itself recursively. The array `visited` is used to avoid checking any worker more than once. It is cleared before the call to `trlf()` in `join()`. When all workers working on the join target have been visited without a successful steal, `trlf()` returns `false` and as soon as a succesful steal has been made, it returns `true`. The (transitive) leap frogging is then restarted from `join()`.

Transitive leap frogging is not complete in that it also risks losing parallelism, in this case since a worker executing a blocked join is only allowed to steal from a subset of the processors in the system. Hence it may be the case that there is work available at the same time as a worker is idle. Therefore, a work stealer with transitive leapfrogging is still not greedy in the sense of Graham and Brent [2, 9].

2.3 The Done race

There is a race condition in the code above in that the join target may be completed while `trlf()` executes, in which case `trlf()` can steal and execute work that is not part of the join target. This race also exists in ordinary leap frogging; if the joining worker finds the task stolen in line 25 in `join()`, then the thief completes the task (changing its state to **Done**) and then spawns a new task, followed by the joining (and leap frogging) worker stealing the new task in line 27 in `join()`. The crucial property here is that the lead that the leapfrogging worker followed is only valid while the task containing the lead is in state **Stolen**.

This problem can be solved by having the leap frogging worker re-check the state of the join target after it has obtained exclusive access to the task it is stealing. If the join target has changed state to **Done**, the steal is aborted. Otherwise, the worker that stole the join target will not be able to complete the execution of that task until the leap frogging worker has completed the task *it* has stolen.

The problem is similar but slightly trickier in the case of transitive leap frogging. The extra difficulty is that the lead that a transitively leapfrogging worker follows does not lead to a worker with a task that can be stolen to block that worker from invalidating the lead (by changing the state of the task to **Done**). Instead, it leads to a worker that have some further tasks in state **Stolen** which can contain additional leads. We however have the following property: If we follow a lead to a worker where we find another lead while the first lead is still valid, the first lead will stay valid at least as long as the new lead stays valid.

We can use this property by essentially time stamping the leads (we can for instance use a counter for each worker as time stamp). When we follow a lead and find a new lead, we

check that the time stamp of the old lead (in the appropriate slot in the task pool) is still the same as when we first saw it. Then we know that as long as the time stamp of the new lead does not change, the old lead stays valid as well.

2.4 Implementation details

We have implemented transitive leap frogging in Wool and report some experimental results below. Our actual implementation differs from this presentation in some ways:

- We use an explicit work list rather than a recursive function.
- In `join()` we try ordinary leap frogging several times before calling `trlf()` in order to avoid paying for the overhead of the new algorithm when it is not needed.
- The details of how to manage the time stamps and when and how to update them depends heavily on the details of the rest of the work stealer as well as on the consistency model of the underlying processor. Wool is carefully tuned to use a minimal number of coherence transactions and memory barriers and uses a rather elaborate protocol that falls outside the scope of this paper.

3. Experimental evaluation

The experimental evaluation we offer in this section is admittedly minimalistic. The reason is that we to date have found only one program where the limitations of conventional leap frogging noticeably affect performance. The UTS program was introduced by Olivier and Prins [12] in order to stress the ability of OpenMP 3.0 implementations to cope with deep and unbalanced task trees, such as are typically generated by e.g. combinatorial optimization algorithms. The implementations evaluated in that paper did not handle the challenge well; we do not know whether that was due to the same problem we encountered. For the rest of the BOTS benchmarks [4] as well as other programs we have tested, leap frogging is not a limitation; the performance of leap froging and transitive leap frogging is identical. Further experimental comparisons between Wool and other NFJ task systems is given by Podobas et al [13].

The experiments in this paper were performed on a TileExpress board featuring a 700MHz TILEPro64 running Tilera Linux. We used a hypervisor configuration with 63 generally useable tiles and 1 tile dedicated to running the driver for the PCI interface over which the board communicates with the host.

The times we measure are wall clock times excluding startup and shutdown of the library itself. Timing is started when all worker threads are running and have initialized per worker data structures and ends when the application code returns to the library, that is, when the `main` task returns. We exclude startup and shutdown since the overheads affect execution time for short running programs and they are orthogonal to the user level scheduling that is the focus of this paper. Startup time for 63 workers (tiles) is about half a second, while the shutdown time, which is essentially for writing back execution statistics, is about 0.1 seconds.

The Unbalanced Tree Search program has been used to benchmark OpenMP task implementations [12] and as a challenge for a distributed memory work stealer [3]. The program builds a deep and unbalanced tree using the SHA-1 algorithm as a splittable random number generator. A node in the tree has either n or 0 children, the former with a probability very close to $\frac{1}{n}$. We use two workloads, T3 with

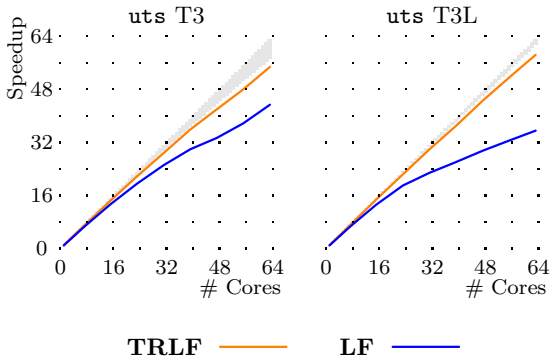

Figure 6. Comparing leap frogging and transitive leap frogging for UTS

4112897 nodes and depth 1572 and T3L with 111345631 nodes and depth 17844., in both cases running with the smallest computational granularity (one iteration of the SHA-1 algorithm).

Figure 5 shows total CPU time for different numbers of processors, normalized to that of the single processor execution. Each bar has three parts, from the bottom *work*, representing time executing application code (this is the only nonempty part for single processor execution), *overhead*, time spent execution steal operations, and *idle* time (which is spent in unsuccessful steal attempts). The middle part is totally insignificant in all of the plots as very little time is spent in the steal and join code itself. Each part is further divided into the part where the worker blocked in a join and leap frogging (top, gray column in the middle of the bar) and the part where it is not (bottom, no gray column). Since the bars represent total execution time, summed over the processors,

- a program with linear speed up would have a flat profile with all bars equally high (and the two top parts empty),
- a program with locality problems that experience extra cache misses in the application as a consequence of stealing would have its application parts increasing, while finally
- a program with a load balancing (or lack of parallelism) problem would show an increase in the idle time category.

This last case is what we see here. We also see that this problem is related to joins since the gray column extends almost the full height of the idle part (as it does also in the work part).

The question to answer now is of course whether transitive leap frogging can solve the problem (recall that it is not complete, as discussed above). We see from Figure 6 that transitive leap frogging does solve the performance problems of the UTS benchmark, achieving close to linear speedup.

4. Related work

As mentioned, several systems implement the NFJ task model, exhibiting a variety of join implementations. The Microsoft TPL uses more than one OS thread per processor, switching to a different thread when one thread blocks. This requires extra space in the form of several stacks per

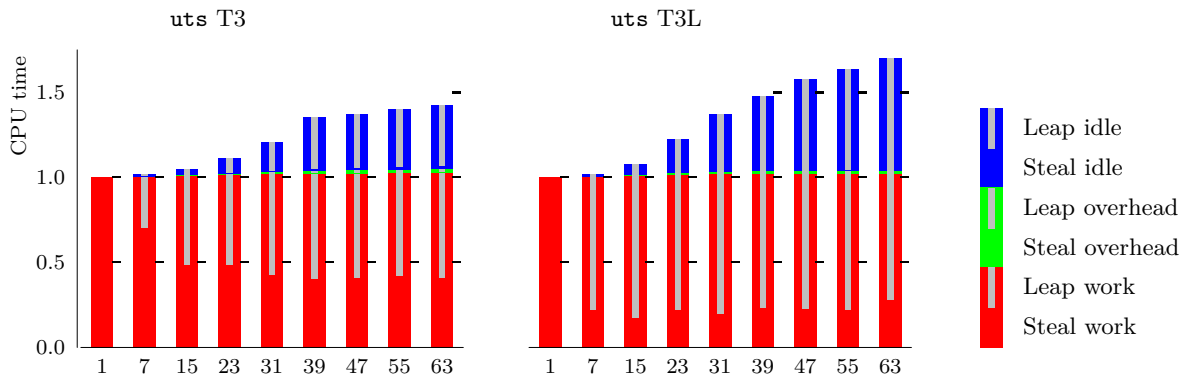

Figure 5. Total CPU time by category

processor but limits each stack size to no more than in a sequential execution. Of course, there is no guarantee that any particular number of extra threads are enough, and in addition the thread switches generally have much higher overheads than the more limited work stealing operations.

The Intel TBB keeps track of the depth of each task in the tree and handles blocking joins by stealing from an arbitrary worker, but only tasks which are no closer to the root of the computation than the join target. This is a weaker condition than that of transitive leap frogging; there are tasks that satisfy the depth condition but are not part of the stolen subtree, but not the other way around. Thus, the TBB depth condition ensures that the stacks do not grow beyond the sequential stack size, but the potential for loss of parallelism is still there since unrelated tasks are allowed. On the other hand, this strategy allows to steal more tasks than transitive leap frogging, although the choice is still limited.

Cilk++ [7] solves the join problem by using a cactus stack, which is a stack where the activation records are not contiguous in memory, but dynamically allocated from a heap and explicitly linked using pointers. There is one base of the stack and an arbitrary number of stack tops. This kind of stack allows to resume execution of a procedure activation record that is not on top of the stack and allows to reclaim memory that is also not on top of the stack. This allows the system to steal procedure activations rather than tasks, so that `spawn` is implemeted by the spawning processor immediately executing the new task while making the saved activation record stealable. This changes the roles of thief and original spawner in the execution of a `sync` (join) so that whoever reaches it first simply records that its branch is finished and goes on to steal work from an arbitrary processor while the last to arrive completes the join and continues with the following code. In this way, a blocked join can always be resumed by any processor. This also solves the stack space problem since in Cilk++ the stack will have at most one top per processor giving an upper bound on stack space as $N \times S$ where N is the number of processors and S is the maximum sequential stack space.

This solution, while powerful, is not available to schedulers like TBB, Wool, TPL and others which are implemented as libraries since it requires the code generator to be changed to use the non standard stack management. In fact, Cilk Plus, which is integrated into the Intel C compiler (`icc`), cannot use this strategy either since it must be compatible with that compiler's calling conventions. Also, the cactus stack is more expensive to maintain than a conventional stack, impacting the performance of the system by adding overhead.

5. Conclusions and further work

We have presented transitive leap frogging, a generalization of the classical leap frogging algorithm of Wagner and Calder. The new algorithm achieves nearly linear speedup for the UTS program, which has much lower speedup under classic leap frogging.

For the future, we will look for other programs for which classic leap frogging is not good enough to see if the transitive version makes a difference. We would also like to prove the correctness of the time stamp protocoll sketched in section 2.3.

References

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

[2] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

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

[4] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP*, pages 124–131, 2009.

[5] Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, 2008.

[6] Karl-Filip Faxén. Efficient work stealing for fine grained parallelism. In *Proc. of 39th International Conference on Parallel Processing*, San Diego, 2010.

[7] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 79–90, New York, NY, USA, 2009. ACM.

[8] 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.

[9] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM JOURNAL ON APPLIED MATHEMATICS*, 17(2):416–429, 1969.

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

[11] Charles E. Leiserson. The cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

[12] 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.

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

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

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