Manycore Work Stealing

Extended version of the work presented at CF11

Karl-Filip Faxén John Ardelius

Swedish Institute of Computer Science

kff,john@sics.se

Abstract

Manycore processors comprised of many relatively simple cores interconnected with a switched network form one of the main approaches to future massively parallel chip multiprocessors. On the programming side, nested task parallelism implemented by work stealing schedulers is gaining popularity as a programming model. This paper investigates the convergence of the trends; executing task based programs on a 64 core Tilera processor under the high performance work stealer Wool. We measure the performance of (Wool translations of) several programs from the BOTS benchmark suite, as well as microbenchmarks exhibiting extremely fine grained tasks, observing excellent scalability (sometimes over 50 on 63 cores) whenever sufficient parallelism exists.

We also look at two issues that arise when scaling work stealing to larger number of cores: First, while random victim selection provides good load balancing, the number of steals needed and their attendant overhead can be reduced by sampling a few workers and stealing the (heuristically) largest task on offer. Second, for machines where the number of potential victims (ie cores) is large compared to cache and TLB sizes, thieves can benefit from attempting steals from only a subset of the potential victims if multiple thieves are active at the same time.

1. Introduction

As manycore processors, which trade the absolute performance of individual cores for a larger number of cores, are becomming available, the question is how to program them. Do the techniqes that have been proposed for multicores scale up, or are entirely new techniques needed? We address this question by porting Wool, a work stealing implementation of the *nested task parallel* programming model, to the TILEPro64, a recent commercial manycore processor. We measure the performance and scaling of a set of programs from the BOTS benchmark suite, and some microbenchmarks used in previous work on Wool.

We also make a preliminary exploration of some issues that arise when scaling fine grained work stealing to a 64 core system. In particular, we explore alternatives to purely random work stealing and extrapolate their behaviour to larger systems using a simplified simulation model.

In *sampling victim selection* a thief does not steal the first work it finds. Instead, it samples several potential victims and selects the one with the largest task on offer (we use a heuristic that favors tasks close to the root of the computation).

In *set based victim selection*, when there are a significant number of active thieves in the system, each thief only attempts to steal from a subset of the other workers. This reduces the contention for task pools while improving the odds that thieves can keep the relevant data structures of the workers they are trying to steal from in caches and TLBs.

```
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 1. A simple Fibbonacci function in Wool

1.1 Task parallelism

In the world of multicores, nested task parallelism has gained a lot of attention [1, 15, 18, 14]. In this explicitly parallel programming model programs create parallelism by *spawning* tasks with which they later *join* (wait for their completion). Task parallelism is often implemented using *work stealing* where a spawn adds work to a local *task pool*. Idle processors *steal* work from the task pools of randomly selected victims. Join checks if the spawned work has been stolen, in which case it blocks until the stolen work is completed, attempting to find other useful work in the meantime. If the spawned work was not stolen (which is typically the more frequent case), it is performed as part of the join operation by the same processor that spawned it.

In nested task parallel programs the tasks that are executed at run-time form a tree where a task that spawns another task is its parent. In the task pools of the processors, each task represents a subtree of the task tree of the entire program. Executing the task, whether by the owner of the pool or by a thief, unfolds the tree.

Wool [9] is a C library/macro package implementing nested task parallelism by means of work stealing. Wool has unusually small overheads for spawn, join and steal and is therefore capable of efficiently exploiting very fine grained parallelism. It uses a novel work stealing algorithm, the *direct task stack*. Figure 1 gives the Wool version of the well known Fibbonacci function. The code spawns the `n-2` computation and then recursively calls itself with `n-1`. When that call returns, the join checks whether the computation was stolen or whether it should be inlined (executed by the spawning worker).

The direct task stack contains task descriptors, rather than pointers to task descriptors, and treats these in a strict LIFO order, simplifying memory management for tasks. The owner spawns and joins at the top end of the stack using the `top` pointer, while thieves steal from the bottom using `bot`. Synchronization among thieves and between thieves and owner uses state in the task descriptors rather than the more conventional approach of comparing the top and bottom pointers, allowing `top` to be local to the owner. Since task descriptors are aligned to cache line boundaries, this allows a thief to steal touching just two cache lines; the one containing `bot` and the one containing the stolen task.

Wool uses *leap frogging* [21] to keep workers busy during blocked joins. In this scheme, a blocked worker is only allowed to steal tasks that are grandchildren of the task it is trying to join with (that is, it may steal subtrees of the task tree rooted in the joined task). This solves problems with excessive stack growth that occurs if a blocked worker steals a task that unfolds to a tree deeper than that of the joined task [3]. It also ensures that when the computation is finished, the join can unblock and resume execution immediately, which would not be possible if the joining worker had stolen a large computation that was still executing when the joined task completed.

Leap frogging is not a perfect solution to the blocking join problem since it only allows stealing of a subset of the work in the system [20]. If the joined task is sequential, it forces the joining worker to be idle. Some systems, like TBB, provide a continuation passing API to avoid the blocking join operation altogether, while Cilk solves it using a *cactus stack* built from explicitly linked, heap allocated activation records. This solution, while being the only one to avoid both unnecessary idleness and stack overflow in the context of a dirct style API, can only be implemented within a compiler; it is out of reach for library based systems such as Wool and TBB.

1.2 Manycore processors

Manycore processors are not in general scaled up versions of conventional processors, but often embody different design tradeoffs [19]. Where conventional multicore processors first maximize the performance of each core and then pack as many cores as possible, manycore processors maximize performance per mm^2 or per watt, relying instead on thread level parallelism for absolute performance.

Current manycore processors exhibit very different memory models, ranging from cache coherence as in traditional multiprocessors to explicit local memories as in the Cell processor [13] and many GPGPUs. From a programming perspective, cache coherence is attractive, not least becuase it simplifies the porting of legacy code. Caches also give decent performance out-of-the box with code modifications only necessary for improved performance, not for correctness.

On the other hand, it has been argued [12] that coherent caches are wasteful of die area and power, with tag storage as overhead and coherence transactions as an inefficient communications mechanism.

While the hardware trade-off is outside scope of the present work, we attempt to shed some light on the software side if the issue. Does cache coherence allow performance portability of applcations, in particular task based applications?

The results we present in section 5 support this hypothesis; most of the benchmarks we use scale very well.

The TILEPro64 processor used in this work features 64 rather simple in-order 3-way VILW cores (or tiles), each with private split 16Kbyte I and 8KByte D L1 and 64Kbyte unified L2 caches, interconnected with a switched mesh network.

The caches of the TILEPro64 are kept coherent by hardware using a variation of a directory based coherence protocol. Each cache line has a unique *home tile* which is responsible for the coherence state of the line. Writes always update the copy residing in the home tile.

1.3 Contributions

We make the following contributions:

- We give the first analysis of the unbalancing effect of random work stealing and find that sampling can be used to mitigate this effect for systems of over 100 cores but that smaller systems can only benefit if taking a sample is much cheaper than stealing (section 2.1).
- We show how to adapt a low overhead work stealer to many-cores in general and the TILEPro64 in particular, presenting a new work stealing algorithm using only a simple test-and-set operation for mutual exclusion (section 3), and a novel *victim set* algorithm, which consistently outperforms random victim selection (section 2.2).
- We compare the performance of Wool to theoretical bounds finding that the practical implementation is most often at or very close to these bounds for a set of common task parallel benchmarks (section 5).

The rest of the paper is organized as follows: Section 2 presents our nonrandom victim selection heuristics followed in section 3 by a discussion of the port of Wool to the Tilera. Section 4 presents a simulation study of the performance of sampling victim selection while section 5 follows up with the measurements on the Tilera hardware. Finally, section 6 concludes.

2. Nonrandom victim selection

2.1 Sampling

In nested task parallel programs the tasks that are executed at run-time form a tree where a task that spawns another task is its parent. In the task pools of the processors, each task represents a subtree of the task tree of the entire program. Executing the task, whether by the owner of the pool or by a thief, unfolds the tree.

In programs which have relatively balanced trees (e.g. divide and conquer computations), subtrees closer to the root are typically larger; work stealing exploits this property by stealing the oldest task in the victim's pool. Of the tasks in the pool, this one is always the one closest to the root of the computation. Much of the appeal of work stealing comes from its ability to steal a single task[1] representing a large computation.

When a thief is about to steal work, different potential victims can provide different amounts of work depending on the size of the subtree rooted at the oldest task in their respective task pools. Typically, thieves select victims at random. This is not necessarily optimal. The bigger the stolen sub tree, the fewer steals will be needed for the entire computation, minimizing overhead. In fact, if there are more workers with small amounts of work, a thief is likely to pick a victim with a less than average sized oldest task, which will make work even more unevenly distributed.

One possible response to this problem is sampling several workers before stealing and choosing the one with the largest available sub tree. This is similar in spirit to other load balancing algorithms that use sampling [16], although we do it to find a large pool to extract work from rather than finding a small queue for inserting work into.

While the run times of the tasks are not known in advance, a possible heuristic for balanced computations is their distance from the root of the tree. This simply extends the heuristic of stealing the oldest task of the given victim to the selection of the victim itself. Another heuristic that has been proposed is the number of tasks in the pool [6].

We have implemented a sampling victim selector for Wool, finding that the overhead of sampling (about half of the overhead of a steal) is significant enough that sampling should be disabled

[1] In a cache coherent system, executing the stolen task may yield extra cache misses for transferring the working set of the task to the thief's cache, but for many programs, the amount of extra misses grows more slowly than the task size due to data reuse within tasks.

in situations with low stealing success rate. That is, when most workers have no tasks to offer, many samples will contain no useful information. Instead, it is better to steal the first task one finds. We implement this strategy by aborting sampling when an empty task pool is sampled.

2.2 Victim sets

One consequence of emphasizing the number of cores over the resources devoted to each core is that the amount of cache and TLB space in each core shrinks in relation to the number of potential stealing victims of that core. For a work stealer like Wool that polls aggressively for work, this may generate a large number of cache and TLB misses with an attendant increase in memory and inter core traffic, especially in situations with little available work and many failed steal attempts. In addition, the number of sharers of each cache line increases, increasing invalidation traffic. It would be better if each thief only polled as many workers as fits in its cache and TLB.

In situations with many simultaneously active thieves, it is thus beneficial for each thief to look at only a subset of potential victims, as long as all workers that might spawn work are frequently visited by at least one thief. As more parallel work becomes available and the thieves become fewer, each thief should expand its focus, until the last thief polls all of the other workers. The optimal number of victims to poll thus depends on both machine parameters and the number of active thieves and must be allowed to grow and shrink dynamically during execution.

We choose a set size as $\max(W, \frac{m \times p}{t})$ where W is the constant 12 for this machine (which has 16 data TLB entries), m is a multiplicity factor (how many active thieves we want to poll each worker; we use 4 by default), p is the number of workers and t is the number of thieves. It is beneficial to have $m > 1$ for two reasons. First, since we use randomly selected sets, it is not certain that every worker is visited by some thief; with four thieves on average, the probability of missing a worker is acceptably small. Second, workers often spawn several tasks in rapid succession, in which case we want several thieves to find them, without the delay involved in expanding their focus. We have investigated the performance sensitivity of these parameters and found them to be relatively insensitive (similar values work just as well) as well as suitable for all of the benchmarks we have used.

The *victim set* algorithm implements this strategy as follows Each worker has a private random permutation P of the indices of the other workers (which are its potential victims). When it initiates stealing, it picks a random starting point S in the permutation and attempts to steal from the workers it finds while proceeding circularly through the permutation. When it has encountered at least m thieves *and* made at least W steal attempts, it starts over from S for a possibly longer or shorter walk through P. If it wraps around to the position S again, it starts counting afresh.

There is always a small probability that a set of thieves I becomes *isolated* from the rest of the workers by picking victim sets contained in I itself. This leads to a deadlock like state where each thief waits for another thief to spawn, which it cannot do since it is a thief. The problem manifests itself as a massive load imbalance (not a total standstill; there is always at least one worker making progress, otherwise there would be deadlock even if the victim set scheme was not used). For this reason, we reinitiate stealing, choosing a new random starting point S, after 1000 failed attempts.

3. The two field direct task stack

The TILEPro64 does not support the compare and swap primitive assumed by many work stealing algorithms [2, 11, 4], including the original direct task stack [9]. To bridge the gap, we modified

```
VOID_TASK_2( tree, int, n, int, d )
{
  if( d>0 ) {
    SPAWN( tree, n, d-1 );
    CALL( tree, n, d-1);
    SYNC( tree );
  } else {
    loop( n );
  }
}
```

`loop(n)` computes for 2n cycles, making no memory references.

Figure 2. The `tree` task of the `stress` program

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

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

RTS_join( Task *t ) {
  State s = t->state;
  Alarm a = t->alarm;
  do {
    while( s == EMPTY && a == NOT_STOLEN ) {
      s = t->state;
      a = t->alarm;
    }
    if( s != EMPTY )
      s = test_and_set_to_EMPTY( &(t->state) );
  } while( s == EMPTY && a == NOT_STOLEN );
  if( s != EMPTY )
    get_wrapper(s)(t);
  else if( a != DONE )
    while( t->alarm != DONE )
      steal( get_thief(s) );
  bot--;
}

RTS_steal( Worker *victim ) {
  Task *t = victim->bot;
  State s = t->state;
  if( is_task(s) ) {
    s = test_and_set_to_EMPTY( &(t->state) );
    if( s == EMPTY || victim->bot != t) {
       if( s != EMPTY ) t->state = s;
    } else {
      t->alarm = STOLEN( self_idx );
      victim->bot = t+1;
      get_wrapper(s)(t);
      store_fence();
      t->alarm = DONE;
    }
  }
}
```

Figure 3. The Two Field algorithm

the direct task stack by splitting the function of the `state` field into two fields, `state` and `alarm`. Figure 3 gives the main operations of the Two Field Direct Task Stack.

The main data structure of both versions of the direct task stack is an array of fixed size task descriptors. In contrast to most task stealers, synchronization among thieves and between thief and victim does not use the top and bottom pointers into the stack (indeed, the top pointer is private to the owner of the stack). Instead, a *state* field (in the new version also an `alarm` field) in each task descriptor is used. This means that the `bot` pointers are accessed without explicit synchronization. Instead, the `bot` pointer is logically owned by the worker that owns the task it points to. This can lead to the use of stale `bot` pointers which requires that the algorithm can back out of a steal that has used a stale `bot` pointer [9].

Logically, a task descriptor can be in one of the following states (these are the same as in the original direct task stack):

Empty There is no task stored in the descriptor and it is free to be reused.

Task There is a task in the descriptor that has not been stolen or aquired by the owner.

Busy This is a transient state where the task in the task descriptor has been acquired by a thief, but the thief has not yet commited to stealing it by writing its index into `state` (or `alarm` in the new version). This is represented with the same value of the `state` field as the `EMPTY` state.

Stolen The task is stolen but not yet completed.

Done The task was stolen but the thief has completed execution.

In the two field algorithm, these states are represented using a combination of the `state` and `alarm` fields, with the `state` field used for mutual exclusion when stealing and `alarm` used for communication between a successful thief and its victim.

The `spawn_f()` and `join_f()` functions are *task specific* and generated from the task definition by the `TASK_n` macros. They are similar to the corresponding functions in the original direct task stack algorithm. The `join_f()` function atomically checks that the task is not stolen, setting its state to `EMPTY`. If the task is stolen, it calls a function in the run-time system that handles synchronization with stolen tasks. This function spins until the thief either backs out of the steal or commits by writing its index into the `alarm` field of the task. This index is then used for *leap frogging* [21] until the stolen task is completed.

The `RTS_steal()` function reads the `bot` pointer of the victim, then does a test and set on the indicated task. If the thief succeeds in acquiring the task, it re-reads `bot` to see that the value it used was not stale, either committing to the steal or aborting it (that typically happens in less than 1% of successful steals).

In our port to the Tilera processor, we decided to keep the per worker data structures homed on the worker that owns them. In this way we minimize the overhead of spawns and joins of inlined tasks. The thieves will however use remote[2] stores for stealing.

As noted in [9], many work stealers avoid atomic instructions as a means of coordinating between thief and victim in favour of a Dijkstra style protocol using memory fences. Our port of Wool to the Tilera instead (like the original algorithm) use a test-and-set instruction which is significantly cheaper on this architecture (roughly 8 as compared to 14 cycles).

[2] A *remote* store is an ordinary store to an address that is homed on another tile; it is not a different instruction.

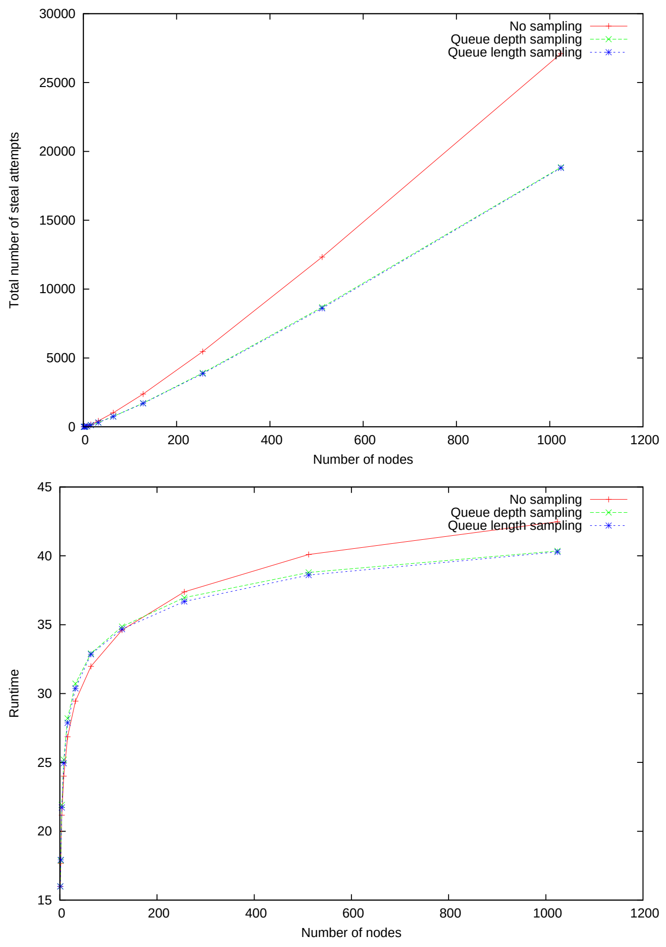

Figure 4. Simulated effects of sampling on time (bottom) and number of steals (top).

4. Simulating sampling

In order to gain insight about the impact of sampling on the overall performance, both in terms of runtime and number of stolen tasks, for larger core counts than we have available, we created a simulation model of the system. This is not a cycle accurate simulator of a particular machine, but an abstract model designed to elucidate the unbalancing phenomenon itself.

As for timing, we have chosen random time distributions that are in line with measurements on the Tilera. While larger machines will typically have larger overhead for stealing, that larger overhead is likely to differ depending on implementation details. Rather than making more or less arbitrary assumptions, we have chosen to model steal and sample costs as independent of machine size. While this might overestimate scaling, we believe that it does not unduly distort the results since the important quantity is the ratio of the cost of stealing to the cost of sampling, and that is likely to stay approximately constant for larger machines.

The model consists of N parallel cores/nodes who either execute tasks if the local task queue is non empty and otherwise tries to steal work with a rate (average interval between steals) λ_{steal}.

We assume that the points in time when different nodes attempt to steal work are independent which enables us to model the steal process as a Poisson process with parameter λ_{steal}. We model the sample process in a similar way with average rate $\lambda_{sample} = 2\lambda_{steal}$ (taking a sample is less costly than stealing since it only involves reads). The benchmark we study in this simulation is `tree` (given in figure 2), which builds a balanced binary tree of tasks. We use a tree depth 2 greater than the logarithm of the number of available nodes (each node will, on average, execute four leaves). Hence runs with larger number of nodes execute proportionally more work.

Comparing with measurements from the TILEPro64, we conclude that the time to complete a leaf task is about the same order of magnitude as the time it takes to complete a stealing cycle, letting us set $\lambda_{work} = \lambda_{steal}$. Each node samples x independent nodes in order to determine which node to steal from, x being the *sample size*.

Figure 4 shows the runtime (top) and number of steal attempts (bottom). First we note that benefit from using sampling in system with moderate number of cores is limited. Under 128 cores, sampling even leads to a slowdown. However, as the system size grows the relative speedup from using sampling does too. The steal plot (top) reveals why; at 1024 processors, sampling eliminates almost a third of the steals.

We have found that 2 is the optimal sample size when $\lambda_{sample} = 2\lambda_{steal}$. Even if more samples results in a better victim it does not compensate for the overhead of taking the extra samples.

5. Measurements on the TILEPro64

All 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.

5.1 Benchmarks

For our experimental evaluation, we use relatively simple programs taken from the BOTS benchmark suite [8] as well as some of the programs used in earlier work on Wool [9]. The latter programs have small kernels that are repeated to get at least a second of parallel run time. We use them to 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 of these programs.

Table 1 gives some statistics of the programs. We see that the programs are unusually fine grained to be run on this number of cores. The **RepSz** column gives the run time in thousands of clock cycles for one repetition of the benchmark when executed on a single processor. For each such repetion, work is distributed over the machine followed in the end by joining together all of the work (much as in a tree barrier).

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 (four per row, which gives numerical stability and increasing sparseness for larger workloads).

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.

stress A micro benchmark written to have a precisely controllable parallelism and granularity (the code is given in figure 2). The program creates a balanced binary tree of tasks with each leaf executing a simple loop making no memory references. A stress workload labelled `n d r` makes `r` sequential task trees `CALL( tree, n, d )`. The granularity of the leaf tasks is $2 \times$ `n` cycles and the granularity of the parallel regions is $2 \times$ `n` $\times 2^{d}$ cycles. We have `n` as 256 and we use `d` as scaling parameter.

fib Computes the Fibonnaci function using the naive algorithm, taken from the BOTS suite. We've included it as an example of a program with extremely fine grained tasks which exposes the behavior of inlined tasks. The input is 43.

uts The Unbalanced Tree Search program has been used to benchmark OpenMP task implementations [17] and as a challenge for a distributed memory work stealer [7]. It is also included in BOTS. 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 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).

multisort The program is originally from the Cilk-5 distribution, and is also in BOTS. It sorts an array of integers using a combination of sorting algorithms. The outermost is a parallel merge sort which is replaced by a serial sort for sub arrays under a threshold size.

nqueens Also a BOTS program originating with Cilk-5. Solves the NQueens problem using a straight forward depth first search. N=13.

Looking at `stress` with a tree depth of 10, we see that the sequential run time of a repetition is just 601k cycles, or less than 10k cycles per core for the 63 core case. Looking at the rightmost column, we see that we have about 3k cycles of sequential execution between steals, making load balancing a very frequent operation.

5.2 Speedup

Figure 5 shows relative speedups[3] for the benchmarks. We choose relative speedup since our main focus is with the parallel scaling behavior. With one exception, the numbers are very close to absolute speedup since the overheads of Wool when running on a single tile is only around 30 cycles per task spawned. `fib`, however, has small enough tasks (table 1 gives an S_T of 63 cycles on average, about half of which is consequently overhead) that there is a significant difference, with absolute speedup being about half of the relative speedup that we report.

For the scaling workloads we show increasing granularities in the plots from left to right. Each graph shows one victim selection strategy:

RAND A random victim is selected for each steal attempt. In order to compare fairly to the **SET** strategy, we do not compute a random number for each steal as that is slow enough to lead to performance degradation. Instead we use the same precomputed permutation as **SET**, but we have $W = 63$, so that all potential victims are visited.

SAMP For p workers, up to about the square root of p workers are sampled. Sampling is not performed after failed steal attempts, and sampling is aborted (and a victim is selected among the already sampled workers) if a worker with no stealable work is sampled.

SET The victim set algorithm described in section 2.2.

SAMP+SET A combination of **SAMP** and **SET**. Sampling is abandoned when the set wraps around; that is, the same worker is never sampled twice in the same collection.

[3] Relative speedup is in comparison to the parallel program running on a single processor, while absolute speedup compares to a purely sequential program.

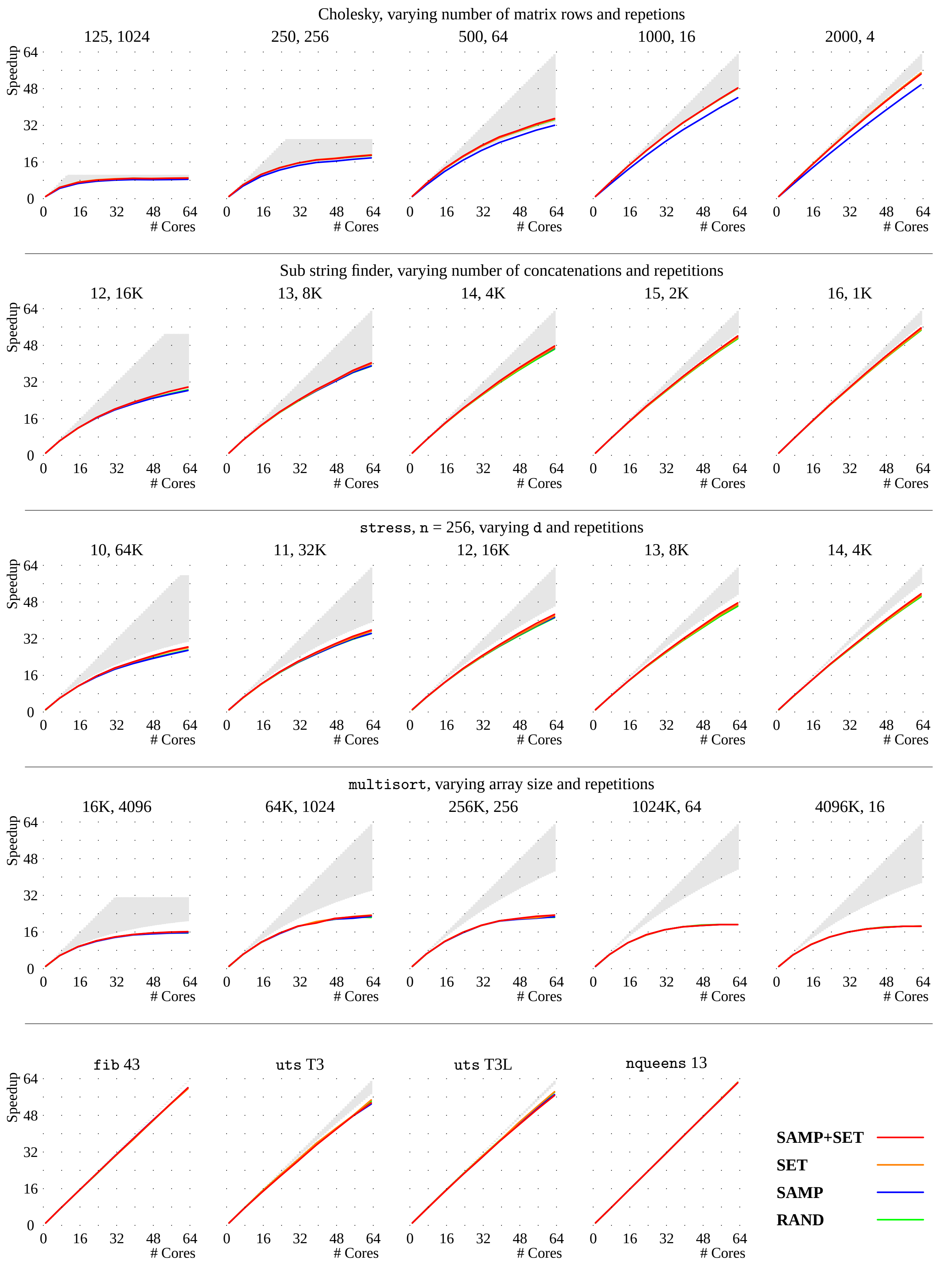

Figure 5. Speedup on a TILEPro64

Params	Reps	Avg. Parallelism		RepSz	Granularities								
		0	1000		S_T	I_S^7	I_S^{15}	I_S^{23}	I_S^{31}	I_S^{39}	I_S^{47}	I_S^{55}	I_S^{63}
`cholesky`, parameters: number of rows													
125	1K	11.7	10.4	42841	7008	37	21	16	14	13	12	12	11
250	256	29.6	26.0	280848	6824	85	41	29	24	21	19	18	16
500	64	81.8	71.0	2073750	6732	233	101	67	52	44	38	34	31
1k	16	226.5	200.0	17013938	6950	761	307	196	148	120	102	89	80
2k	4	546.3	503.5	157477250	7132	2857	1121	688	510	405	338	292	257
`ssf`, parameters: number of concatenations													
12	16K	92.8	52.9	928	6489	31	17	13	11	10	10	9	9
13	8K	149.4	104.1	2661	11469	78	41	30	25	22	20	19	18
14	4K	245.3	192.1	7591	20190	177	87	64	53	47	42	39	36
15	2K	403.6	344.0	21557	35398	396	190	139	114	99	89	82	77
16	1K	645.8	584.8	60908	61773	1017	474	339	275	237	213	194	179
`stress`, leaf size 256 iterations, parameters: tree height													
10	64K	278.4	59.7	601	587	13	6	5	4	3	3	3	3
11	32K	404.8	102.7	1201	587	21	9	7	6	5	4	4	4
12	16K	534.3	170.6	2402	587	32	14	10	8	7	6	5	5
13	8K	631.3	269.5	4805	587	50	22	15	12	10	9	8	7
14	4K	1099.6	480.5	9610	587	80	33	23	18	15	13	12	11
`multisort`, parameters: array size													
16K	4K	68.3	31.0	2707	1142	16	8	6	4	4	3	3	3
64K	1K	108.8	73.8	11512	4645	58	27	19	15	13	12	11	10
256K	256	148.9	129.7	52500	18130	233	108	75	60	51	45	41	38
1M	64	139.9	137.4	279891	61626	1062	480	335	266	226	199	179	166
4M	16	91.5	91.7	1442000	122623	4510	1950	1343	1032	867	759	684	624
`fib`, parameters: n													
43	1	116802.5	110982.2	44121000	63	27042	1216	1045	1264	817	859	753	616
`uts`, parameters: sample tree													
T3	1	1087.6	682.4	8491000	2359	240	93	55	40	33	29	29	27
T3L	1	4457.1	2421.6	228676000	2567	316	79	46	35	35	32	29	31
`nqueens`, parameters: n													
13	1	65311.8	27140.4	31171000	521	13947	10399	6066	3876	3241	2600	2110	2072

Params are the parameters of each iteration (see section 5), **Reps** is the number of repetitions, **Avg. Parallelism** is T_1/T_∞ assuming stealing overhead zero or 1000 cycles, **RepSz** is the size in 1000s of cycles of each repetition, **Granularities** are the average task size S_T in cycles and the average steal interval I_S in 1000s of cycles for 7 to 63 processors. Throughout, k is 1000 and K is 1024.

Table 1. Characteristics of the benchmarks

The grey area shows the expected speedup in a simple performance model based on the critical path length, or *span* of the computation [10].

Given a span t_∞ and a sequential execution time t_1 (not including stealing overhead), the low end of the speedup region for p processors in the figure is given by $\frac{t_1}{t_p}$ where t_p is the standard upper bound on the execution time of a work stealing computation:

$$t_p = t_\infty + \frac{t_1}{p}$$

The upper end is given by $\frac{t_1}{T_p}$ where T_p is the lower bound on execution time given as the maximum of t_∞ and $\frac{t_1}{p}$.

The difference between the upper and lower ends reflects the different shapes that a dependence graph with the same span and work can have. The upper end reflects a situation where the parallelism varies little over time; in particular, with no significant sequential periods, while the lower end reflects massive parallelism alternating with sequential execution. Our granularity scaling benchmarks, with their repeated parallel kernels, clearly belong in the latter category.

Overall, the programs scale close to the theoretical prediction which indicates that the Tilera processor has adequate resources and no obvious bottlenecks. It can support all the cores being active. There is a small tendency in several runs that the speedup falls away from the theoretical model at high core counts since the search for work is not a constant time operation; in a larger machine, a thief generates more failed steal attempts.

Only `multisort` deviates significantly from the theoretical prediction. To understand why, we next turn to a breakdown of execution time into application time, idle time and overhead time.

5.3 Breakdown of execution time

Figure 6 shows a breakdown of the CPU time spent in the benchmarks into application time (useful work), stealing overhead (cost of succesful steals) and search time (cost of unsuccessful steals), with application time at the bottom and search time at the top. The time unit is the application time of the single processor execution (which equals the entire run time for a single processor execution); thus a total time of two would indicate a total CPU time of twice the sequential CPU time and consequently half of linear speedup. Each category is further refined into time spent while blocked in join operations (the vertical bar in the upper part) or not (below the bar). Application time in a join operation denotes leap frogging, while search time in join indicates waiting. A large amount of join related search time indicates a failure of leap frogging to keep the machine busy (or simply insufficient parallelism).

In all cases, the overhead of succesful steals form a minor part of the execution time and large search times are correlated with insufficient parallelism; it is a symptom of idle workers. It may also originate from the additional steal attempts that a thief needs to on a bigger machine. Note that our version of Wool does not back off; thieves are always aggressively looking for work.

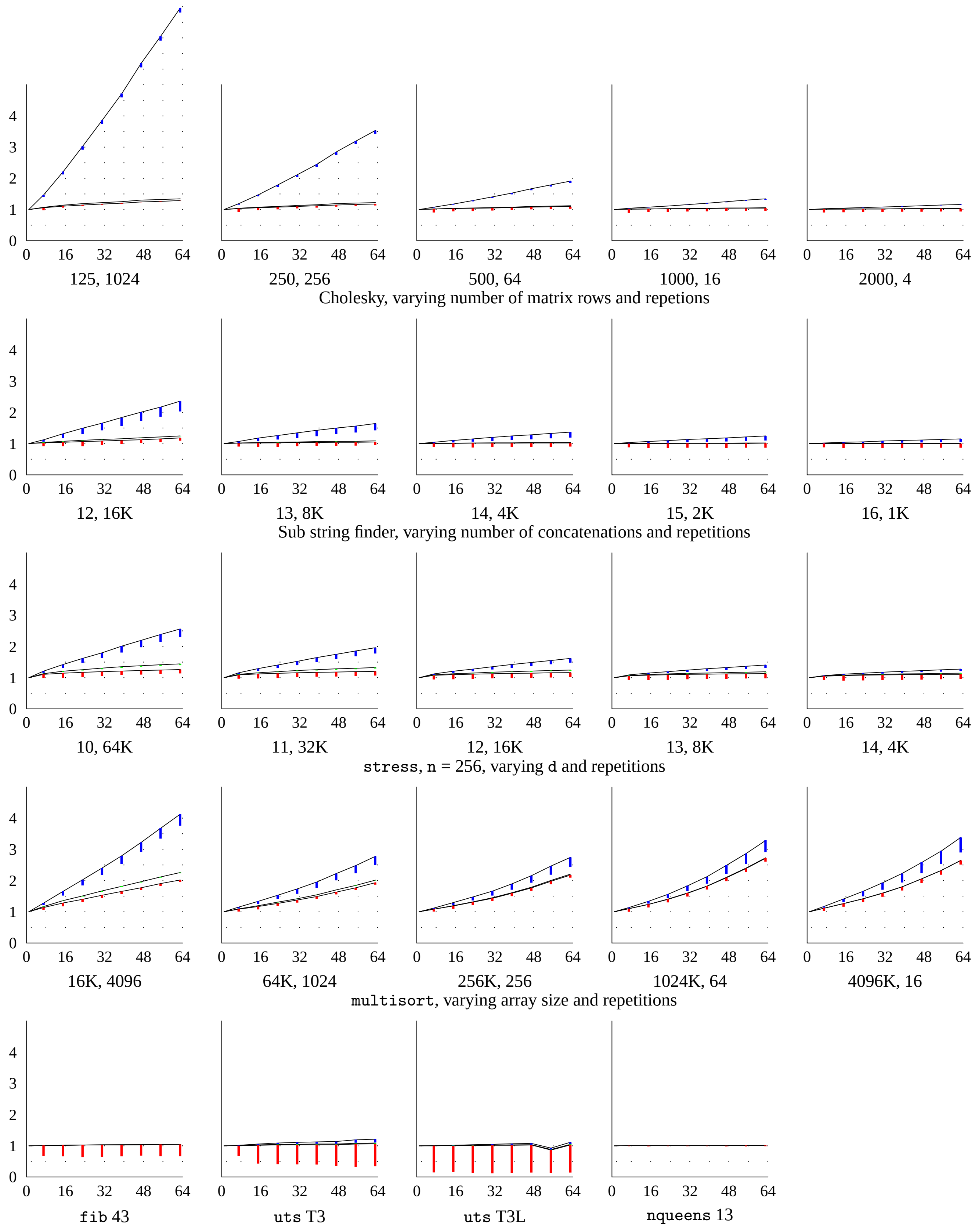

Figure 6. Breakdown of execution time into application time, overhead and idle time.

Looking at the graphs for `sort`, we see that the limitation for speedup is that the application time component increases. For the larger workload, the cause is clear since in the largest workload, each of the three arrays used are 16MByte in size, against a total of 4MBytes of combined L2 cache. The problem is exacerbated by the low reuse factor for data. For the smaller workloads (especially the smallest), we believe that the cause is related to migrating the working sets between the cores on steals in combination with not very ample parallelism (refer to figure 5). Since Wool is a classical work stealer rather than a *parallel depth first* stealer [5], it is not tuned for constructive cache sharing. Different processors work on as distant subtrees of the main computation as possible, which means that they in general work on subtrees that share little data. This reduces the impact of scheduler overheads but gives little reuse between parallel activities.

5.4 A closer look at sampling

Figure 7 shows the number of steals (bottom) and the average time for a steal (top) for **SAMP+SET** relative to **SET** (a value greater than one means that the sampling version has the larger count or time, respectively). The data hints at why sampling has such a small effect on execution time; while it manages to reduce the *number of steals*, their average *cost* increases, balancing the gain.

Since the cost increase is caused by the sampling, which is substantial in comparison to the steal overhead, it is interesting to look at what would happen if steals were more expensive. One would expect that sampling would be more beneficial in that case. To test this hypothesis, we have added an artificial delay of 4000 cycles to the steal operation, placed right before invoking the wrapper function, and just after committing to the steal. Figure 8 gives the results.

While `stress` and `ssf` get minor speedups, it is clear that in general, a more expensive steal operation does not make sampling that much more attractive. Our tentative conclusion is that the benefit of sampling is closely linked to the number of cores in the system.

6. Conclusions and future work

We have studied the performance of the Wool work stealing task scheduler on a 64 core Tilera processor. We found that most programs scale well, being primarily limited by available parallelism. We have demonstrated that it is possible to achieve stealing oveheads counted in hundreds of cycles rather than thousands.

We have also studied the unbalancing effects of random work stealing and applied sampling to mitigate the problem, finding that the reduction in the number of steals does not pay for the sampling overhead for 63 cores. Simulation results indicate that the imbalance grows with system size and that sampling will be more attractive when the core counts pass a hundred.

We have also presented two algorithms which adapt Wool to the Tilera processor; a task stack management algorithm which only needs a test-and-set primitive and a victim selection algorithm that improves cache and TLB locality, something that is important for a processor with many rather simple cores.

For the future we intend to investigate the use of polling and other techniqes for mitigating stealing imbalance, especially with a view to the trade-off between the information obtained by polling and the cost of obtaining it.

References

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

[2] 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.

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

[4] 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.

[5] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] H. Peter Hofstee. Power efficient processor architecture and the cell processor. *High-Performance Computer Architecture, International Symposium on*, 0:258–262, 2005.

[13] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49:589–604, July 2005.

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

[15] 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.

[16] Michael David Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), October 2001.

[17] 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.

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

[19] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[20] Jim Sukha. Brief announcement: a lower bound for depth-restricted

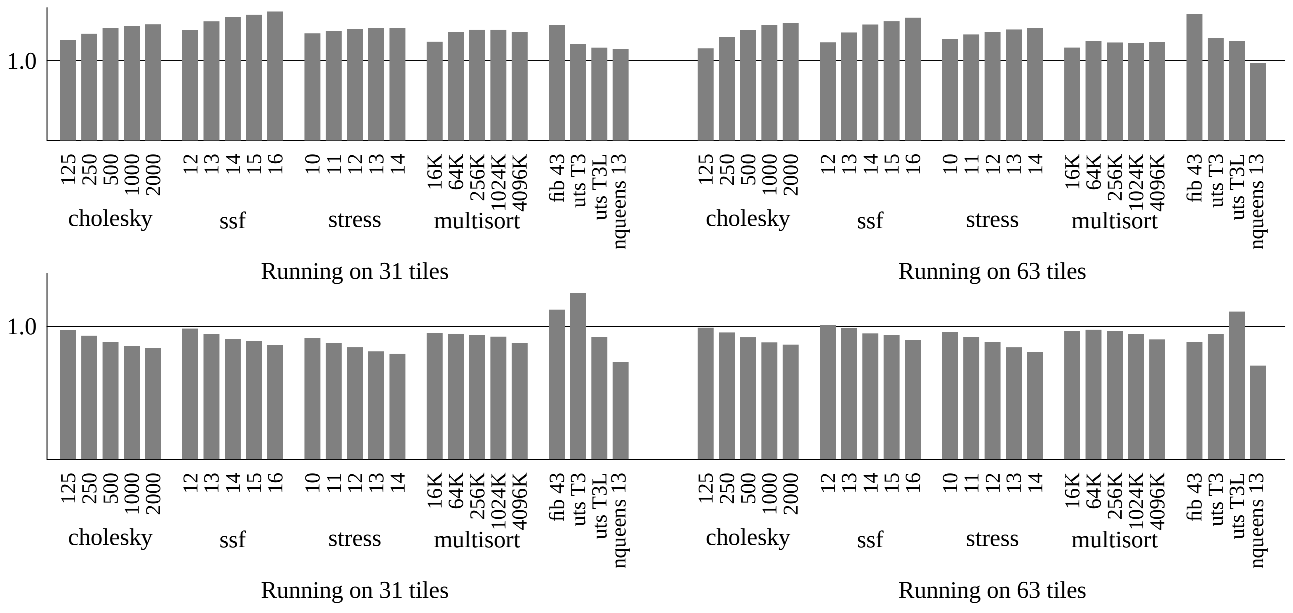

Figure 7. Effect of sampling (**SAMP+SET**) on the number of steals (bottom) and their average cost (top), relative to no sampling (**SET**).

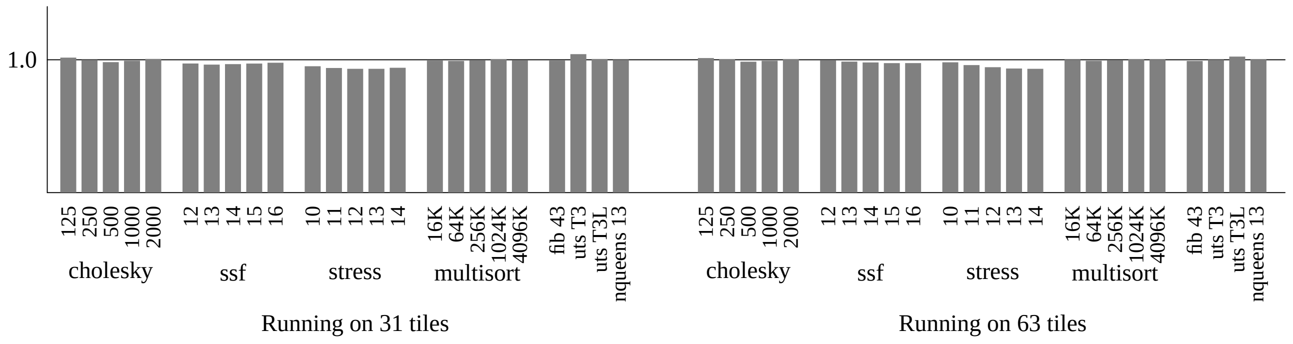

Figure 8. Effect of sampling (**SAMP+SET**) on execution time when steals are artificially delayed, relative to no sampling (**SET**).

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.

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