MISTA 2015

On Task Scheduling Policies for Work-Stealing Schedulers

Steven Adriaensen · Yasmin Fathy · Ann Nowé

1 Background

Parallel computing architectures are becoming more and more mainstream. To take advantage of their parallel processing capabilities, a computation must be divided in a set of interdependent tasks that can be executed in parallel [7]. Such computation can be represented by a Directed Acyclic Graph (DAG), where vertices are the instructions and edges represent execution order dependencies. Instructions that do not depend on each other can be executed in parallel. In this article we consider the fork-join model of computation. Here, fork instructions generate a new (sub)task that can be processed independently (out-degree of 2). Join instructions cause a task to wait for the completion of another (in-degree of 2). All other instructions have an in/out-degree of at most 1 and make up the task. Typically a computation can be divided in many more tasks than there are processing units. This gives rise to the task scheduling problem: *Which tasks are to be executed by which processor, and in which order?*

Work-stealing [2] is a state-of-the-art dynamic, distributed scheduling algorithm. Here, each worker maintains its own local work-pool. One of the workers starts processing the root task. When a fork is encountered, it places one of the tasks in its pool and continues to process the other. If a task stalls (join) or is completed (out-degree 0) the worker will start working on another task from its own pool, or, if it is empty, it will steal work from another worker's pool. Many work-stealing implementations exist [1,5,4], ranging from libraries to runtimes of parallel programming languages. These systems make different design decisions that impact their performance in subtle ways, causing them to perform well in some settings, and poorly in others.

Steven Adriaensen
Vrije Universiteit Brussel
E-mail: steven.adriaensen@vub.ac.be

Yasmin Fathy
University of Surrey
E-mail: Y.Fathy@surrey.ac.uk

Ann Nowé
Vrije Universiteit Brussel
E-mail: ann.nowe@vub.ac.be

In this abstract we'll discuss one of these design decisions, i.e. the scheduling policy used. In Section 2 we discuss the choice of scheduling policy, in particular the impact of the structure of computation thereon. Section 3 describes an existing adaptive scheduling policy and its weaknesses. Finally, Section 4 reports our ongoing research attempts towards more general scheduling policies.

2 Choice of Scheduling Policy

When executing a fork the system is faced with a choice, i.e. continue the current or execute the spawned task? The choice a system makes is determined by its scheduling policy. Here, most systems either always continue the current (help-first [5]) or always execute the spawned task (work-first [1]), i.e. use a pure policy.[1] Some systems implement a mixed policy known as SLAW [4] (see Section 3).

Using help-first, a fork is implemented as a call to the scheduler, which creates and stores an object for the spawned task in the work-pool. Using work-first, a fork is implemented as an ordinary function call, which only returns after the subtask is completed. To steal a continuation, the thief modifies the runtime stack (which holds the continuation) of its victim. Using help-first, steals are more efficient, but the overhead is higher than using work-first. Usually only a fraction of the tasks is stolen and therefore work-first implementations tend to be more efficient on average. Furthermore, minimizing overhead is essential to allow fine task granularity [1]. Work-first also has desirable theoretical properties: Let S_1 be the space required by a serial execution, then a parallel execution on P processors using work-first requires at most $S_1 P$ space, which is existentially optimal to within a constant factor [2].

One might wonder, if work-first is more efficient on average and has attractive theoretical properties, why do (the majority of the) systems use help-first? An important reason is that it is easier to implement as a library (without compiler support). Also in some systems ordinary function calls are expensive [6]. In addition, using work-first, recursive forks can cause the runtime stack to overflow and for particular computation structures it fails to distribute work efficiently if the residual parallelism[2] R is low [3]. Consider the extreme, yet common, example of an iterative parallel loop which forks P sequential body computations ($R \approx 1$). To exploit the parallelism of this computation, each worker should process a single body. Using help-first, the first worker executes the loop task, generating P body tasks and each worker steals one of them in *parallel*. Using work-first, the first worker starts processing the first body and the loop task is handed from worker to worker *sequentially*. Here, help-first clearly distributes work more efficiently. Also, as mentioned before, help-first induces a lower cost on stealing. It is therefore tempting to conclude that help-first performs better when R is low, while work-first performs better when R is high (as in [3]). However, we'll argue this not to be true in general. Consider a recursive parallel loop, reversing the argumentation above, work-first will distribute the body tasks much more quickly than help-first. In general, you can *mirror* any fork-join computation where help-first generates work more quickly[3] than work-first. Rather, if R is low and peer workers are idle, we want to execute the sub-computation with the highest parallelism first.

---

[1] In literature, the help/work-first policies are also known as child/continuation-stealing.

[2] $R = \frac{T_1}{P * T_\infty}$, where T_n is the minimal execution time on n processors

[3] The same holds for memory consumption

3 SLAW

As discussed in previous section, what scheduling policy performs best depends on factors unknown before execution. SLAW is to date, the only policy that dynamically adapts its scheduling policy to avoid stack-overflows (help-first at threshold depth), keep memory consumption within theoretical bounds (work-first when # active tasks exceeds threshold) and efficiently distribute tasks. The latter is achieved by switching policy periodically from help-first to work-first if the number of times the worker was victimized (stolen from) is smaller than the number of tasks generated during last period (i.e. enough work is available). Here, [4] makes the overgeneralizing assumption that work-first is more time/memory efficient and help-first generates work more quickly. When this assumption does not hold, SLAW can be shown to perform poorly, consistently making the wrong choice. Another downside of SLAW is that the choice of the period introduces a tradeoff between the increase in overhead due to frequent policy switching on the one hand, and the adaptiveness of the system on the other.

4 Ongoing Research

We're currently looking into alternative scheduling policies to overcome the weaknesses of SLAW (see Section 3). One mixed policy that shows promise is Anti-Imitation (AI). Using AI the first worker starts using a random pure policy, when stealing a task, the stealer anti-imitates its victim, using the opposite policy. Independently of the random choice of the initial worker, AI manages to distribute work quickly for a wider range of computations than SLAW. Note that any (iterative or recursive) parallel loop task is stolen at most once before generating all body tasks. As policy switching occurs only when stealing a task, it doesn't increase the overhead (unlike SLAW). AI, however, has weaknesses of its own. When one of the pure policies is more time efficient, on average half of the active workers will be using the slower policy (i.e. be slower). Also, memory efficiency and potential stack-overflows are still concerns that need to be addressed.

Acknowledgements
Steven Adriaensen is funded by a Ph.D grant of the Research Foundation Flanders (FWO).

References

1. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system, vol. 30. ACM (1995)
2. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM) **46**(5), 720–748 (1999)
3. Guo, Y., Barik, R., Raman, R., Sarkar, V.: Work-first and help-first scheduling policies for async-finish task parallelism. In: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, pp. 1–12. IEEE (2009)
4. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: Slaw: A scalable locality-aware adaptive work-stealing scheduler. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp. 1–12. IEEE (2010)
5. Lea, D.: A java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande, pp. 36–43. ACM (2000)
6. Robison, A.: A primer on scheduling fork-join parallelism with work stealing. Tech. Rep. ISO/IEC JTC 1/SC 22/WG 21, The C++ Standards Committee (2014)
7. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobbs journal **30**(3), 202–210 (2005)

Appendix

In this section we present and discuss some motivating, preliminary results. All results were obtained in simulation, using a cost model valided by using it to accurately reproduce prior experiments (more specifically those in [3,4]).

Generalized Loop Benchmark

In this experiment we consider the Generalized Loop Benchmark (GLB), the pseudo-code of which is given in Figure 1. This computation consists of 2 types of tasks:

LOOP TASK: Performs no work, but splits itself up into a Loop and Body task.
BODY TASK: Performs the actual work,[4] but spawns no further tasks.

A parameter p_{left} determines the probability that the loop task is computed in the current thread, rather than the spawned thread. For p_{left} values 0 and 1, GLB reduces to a recursive and iterative parallel loop respectively. In our experiments the # body tasks (n) is taken equal to the number of processing units ($P = 64$), such that $R = 1$.

While rather artificial, this benchmark was chosen as it clearly illustrates the properties of help and work-first w.r.t. work-distribution, discussed in Section 2.

```
procedure LOOP(i,n)
    if i < n then
        with probability p_left do
            FORK BODY(i)
            LOOP(i+1, n)
        otherwise
            FORK LOOP(i+1, n)
            BODY(i)
        JOIN
    else
        BODY(i)
    end if
end procedure
```

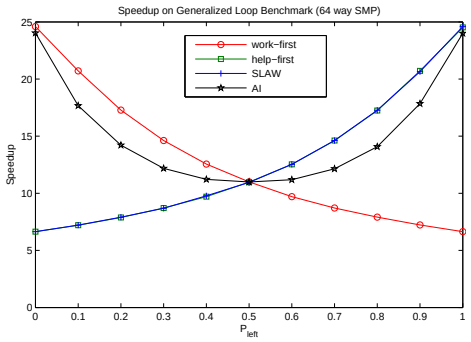

Fig. 1 Code and results for the Generalized Loop Benchmark (GLB)

Observations

Figure 1 shows the Speedup $\frac{T_1}{T_P}$ obtained for GLB, on a 64 way SMP machine, averaged over 1000 runs, using the work-first, help-first, SLAW and AI policies.

We observe that help-first outperforms work-first if and only if $p_{left} > 0.5$, which corresponds exacty to the case where the expected parallelism of the current thread is higher than that of the spawned thread. As R is low, SLAW will always use help-first on this benchmark, failing to distribute work efficiently when $p_{left} < 0.5$. AI on the other hand manages to distribute work reasonably efficiently for all p_{left}, with near oracle performance for $p_{left} \to 0, 1$. Its speedup w.r.t. SLAW ranges from 0.8 to 3.6.

[4] In our experiments, as dummy work, a body task computes a single iteration of the successive over-relaxation (SOR) benchmark on $\frac{1}{P}^{th}$ of a 2000x2000 matrix.