Work Stealing Technique and Scheduling on the Critical Path

Christophe Cérin
Laboratoire d'Informatique de Paris-Nord (LIPN)
UMR CNRS 7030, 99, avenue Jean-Baptiste Clément
93430, Villetaneuse, France
christophe.cerin@lipn.univ-paris13.fr

Michel Koskas
AgroParisTech/INRA UMR 518,
Mathématiques et Informatique Appliquées,
F75005, Paris, France
michel.koskas@agroparistech.fr

Abstract

This paper is about a new framework for high performance thread scheduling based on the work stealing principle when processors may run at different speed. We also take into account memory management problems. We hope that such framework could help multi-threaded or multi-core architects to rely on well founded mathematical results to build appropriate hardware/software schedulers that control the scheduling of threads.

Keywords: *Scheduling algorithms, Work Stealing principle, Memory Management, Heterogeneity of chips.*

1 Introduction and related work

In this paper we propose new insights into the problem of concurrently scheduling threads through mainly the Work Stealing (WS) technique which is one of the technique, with Parallel Depth First (PDF) technique able to efficiently manage fine-grained multithreaded programs. We investigate a nice result of Bender and Rabin about the scheduling of threads in an heterogeneous context, show the limits in terms of (theoretical) performance and we propose to enhance the efficiency of the processor utilisation by keeping the fastest processor working on the critical path of the application. Then we provide a bound on the completion time of our new algorithm.

To effectively exploit available parallelism, chip multiprocessor hardware must address contention for shared resources [6, 5, 3] and also should take care about the scheduling of threads. Our aim is to model the scheduling of processes in order to isolate the main factors that impact the performance of the chips.

Many parallel programming language and run-time systems use greedy[1] thread scheduling principles to maximise processor utilisation. For instance, the Parallel Depth First (PDF) approaches [5, 3] have been proposed for cache sharing as well as for Work Stealing (WS), a popular scheduler technique that takes a more traditional approach. The WS's policy maintains a work queue for each processor and when forking a new thread, this new thread is put on the top of the local thread. When a thread completes, the processor looks for its local queue to execute the next job, and when the queue is empty, the processor tries to steal work from the queue of another processor.

Parallel Depth First (PDF) [4] is another greedy scheduling strategy based on the fact that important sequential programs have already been highly optimised on a single core architecture by maintaining small working sets, by capitalising on good spatial and temporal reuses. In PDF, when a processor completes a task, it is assigned the ready task that the sequential program would have executed the earliest. Note also that [4, 9] indicate how to do this work online without executing the sequential program, we mean online. In a sense, PDF tends to co-schedule work in a way similar to the sequential execution that is to say more favourable for optimisation (cache reuse...). Intuitively, the benefit of using the PDF technique is that the spacial data locality is preserved while WS tends to make 'anarchic' memory accesses by making all the threads working concurrently to ac-

[1] In a greedy schedule, a ready job remains unscheduled only if all processors are already busy

cess the memory without coordination, hence problem with numerous cache misses.

The Critical Path Method, abbreviated CPM, is yet another method for scheduling threads. CPM calculates the longest path of planned threads to the end of the execution, and the earliest and latest that each thread can start and finish without making the execution longer. These algorithms try to shorten the longest path in the application graph by removing communication requirements and mapping the adjacent tasks into a cluster (this is called zeroing an edge). This approach has received a lot of attention and a taxonomy of these techniques can be found in [7]. Our approach is quite similar in the sense that we try to keep a processor busy along a path of the task graph but not for the reason of decreasing the communications but to ensure a fair data locality property: we assimilate arrows on the task graph as data dependencies and not communications.

All of the strategies (WS, PDF, CMP) rely on a model of parallel computation. A program in a model is represented as a directed acyclic graph and the representation is used for the proofs of the bounds on the computational time. Section 2 returns to models of parallelism and the representation of computation. Section 3 reminds the results of Rabin and Bender [2] concerning the WS technique and we introduce in section 4 new insights into the way to refine the bounds. We consider an offline algorithm to schedule threads. Section 5 introduces the problems and solutions to take into account cache effects. Section 6 concludes the paper.

2 Representing a computation and models of parallelism

Our scheduling algorithm is applicable to programming language that supports fork-join constructs. The algorithm assumes a shared memory programming model in which parallel programs can be described in terms of threads. A *thread* is a sequence of actions that are executed serially and taken exactly one time-step (or one clock cycle). A thread may fork any number of child threads and this number need not be be known at compile time. The computation is view as a directed acyclic graph or dag.

Let W_1 represents the *total work* that is the total number of nodes in the dag G. Let W_∞ represents the *critical path length* of the graph, that is, the number of nodes in the longest chain in G. Complementary definitions can be found in [4] for instance.

3 WS technique and the algorithm of Rabin and Bender

In [2], Bender and Rabin consider how to execute parallel programs on a collection of heterogeneous processors. They consider p processors labelled $1, \cdots, p$ where processor i has speed π_i steps/time. They assume for the sake of convenience that $\pi_1 \geq \pi_2 \geq \cdots \geq \pi_p$ and that the processor speeds do not change. Let $\pi_{ave} = \frac{\sum_{i=1}^{n} \pi_i}{p}$.

Bender and Rabin consider the *maximum utilisation scheduling policy*. This notion maintains the following invariant. During each time interval in which there are exactly i ready threads, the fastest i processors execute these tasks. If there are $i \geq p$ ready threads, then all processors work. Note that in order to maintain this invariant, the scheduling policy must allow preemption.

The central theorem in [2] is the following:

Theorem:[2] Any maximum utilisation schedule has a completion time

$$\begin{aligned} T_p &\leq \frac{W_1}{p\pi_{ave}} + \left(\frac{\pi_2}{\pi_1} + \frac{\pi_3}{\pi_2} + \cdots + \frac{\pi_p}{\pi_{p-1}}\right)\frac{W_\infty}{p\pi_{ave}} \\ &\leq \frac{W_1}{p\pi_{ave}} + \frac{p-1}{p}\frac{W_\infty}{\pi_{ave}} \end{aligned}$$

4 How to consider the critical path?

4.1 Offline algorithm

In this section we consider that the computation graph is known in advance. So, we introduce an offline scheduling algorithm which is based on the WS principle. Our aim is to replace the π_{ave} terms in the Bender and Rabin result by a more favourable one, especially one which is not related to the mean of speeds. Our main idea is to keep the fastest processor busy on the critical path. For programs with independent tasks, W_∞ is small and the $\frac{p-1}{p}\frac{W_\infty}{\pi_{ave}}$ do not influence the completion time but it remains still a π_{ave} term in the equation of the completion time. Otherwise, if the task graph is like a hackle, the critical path length may become an influencing factor for the completion time. Our approach is twofold: keeping the benefit of Bender and Rabin result when the critical path length is 'small' and keeping busy the most powerful processors with the most heavy tasks as much as possible.

We remains under the heterogeneous case as defined by Bender and Rabin concerning the speed of processors for instance but we need some complementary definitions. We also work with the series-parallel graph model but we would like to mention that our scheduling algorithm is also valid if we consider 'general' DAGs. However, we will see later on that to tackle memory management problems, series-parallel graphs are a requisite.

Nodes are now represented by a tuple $(name, weight, st, et)$ where $name$ is the label of a task, $weight$ is the time duration of the task, st is the time at which the task starts and et the time at which the task

finishes. The Figure 4.1 introduces an example. Note that if we know only the weight of the task at the beginning of the algorithm and assuming that time starts at 0, a DFS (Depth First Search) like algorithms allows us to label the nodes for tuple values in a strait-forward way.

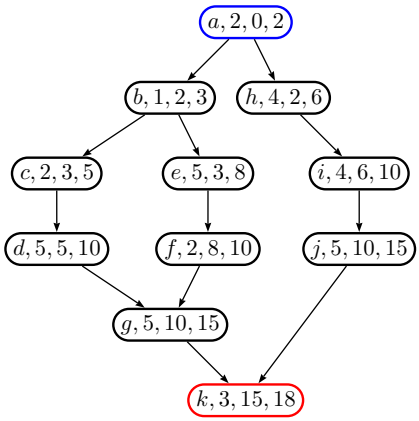

Figure 1. The task graph augmented with information about task durations

4.2 Computation of paths of minimal length

After labelling the nodes, the second step of the algorithm is to compute 'all the critical paths' of the graph. Each (critical) path of maximal weight has a minimal time for starting and a maximal time for finishing. The weight of a path is the sum of weights for tasks that constitute the path. We compute the connected path of maximal weight in the graph in the range $[n = 0, M = \infty]$. This path has an interval of execution of $[m1, M1]$. We continue the process in order to compute the paths of maximal weight in $[m, m1]$ and in $[M1, M]$, recursively. We obtain a set of paths $C_1, C_2, \cdots$ and we store them in an array that give also the processor that should run the tasks on the paths (remind that $\pi_1 \leq \pi_2 \leq \cdots \leq \pi_p$). The array also contains a flag to indicate if a path is completed or not. For the example of Figure 4.1, we get:

Path	Finish?	Proc
C_1	false	1
C_2	false	2
C_3	false	3

with $C_1 = abefgk$, $C_2 = hij$, $C_3 = cd$. Note that the time duration of tasks on the C_2 path is 13; the time duration of tasks on the C_3 path is 7; and the time duration of tasks on the C_1 path is 18;

4.3 Preemptive algorithm

Assuming that the previous array in shared among all the threads, the scheduling algorithms starts by executing the first free path on the fastest processor, the second free path is executed on the 'second' fastest processor and so on.

After this initialisation step, the scheduler enter to a permanent regime and a processor:

α**:** becomes idle because it has finished its 'job' on a path;

β**:** must interrupt its work because it needs to wait for a task to complete.

The protocol is then as follows for the α, β cases:

α**:** the processor looks at the array, marks the 'finish?' field with true and considers the first path which is either not yet started or, if it has been started by a slower processor, it steals the work. A processor which has been stolen by another one has the behaviour of a processor that is idle (its job is just terminated);

β**:** the processor interrupts its work on its path. Let N be the node requiring such synchronisation point. We consider the paths belonging to the parents of N and not yet terminated.

If the paths have not yet been started, we start one of them on the interrupted processor;

If these paths are currently executed on slowest processors, we steal the work of the slowest until we reach the node N, we stop the computation on the stolen path by marking it as executed by an infinitely slow processor and we restart the computation after node N;

If these paths are executed on fastest processors and that there is no more slowest processors, we ask to be notified when node N completes. During the waiting step, the behaviour of the processor is as it has finished its work. When the processor receives the signal, it restarts its initial work and freezes its current work by marking it as executed by an infinitely slow processor.

Theorem: The above algorithm has a completion time :

$$T_p \leq \frac{\sum_{i=1}^{n} W_i + S_i}{\sum_{i=1}^{p} \pi_i}, \text{for } n > p;$$

$$T_p \leq \frac{\sum_{i=1}^{n} W_i + S_i}{\sum_{i=1}^{n} \pi_i}, \text{for } n \leq p$$

where S_i is the starting time of W_i divided by the minimum speed of the processors dealing with the parents of the first node of W_i.

Proof. By construction, the processor working at π_i steps per second is working on the i-th critical path as defined above and when processor i must wait for one parent it makes a progress in the computation and it contributes for a fraction of $W_j, (j > i)$ at speed π_i. □

5 Offline Algorithm and cache effects

In this section we investigate the problems of scheduling threads efficiently whereas controlling the cache misses. Two orthogonal facts characterise the WS principle: theoretical bounds on the completion time can be derived and these bounds show the potential of the method including for practical cases (fork-join programming) but since the method tends to activate concurrently all the threads with no control of what they are doing concerning the memory, there is no chance for regular memory access patterns (we mean that a memory access is accomplished in a random way provided by the memory controller) and thus we tend to multiply the cache misses.

For that reason, the PDF technique has been studied in the past in order to have concurrently executing threads share a largely overlapping memory portion. The aim of the PDF technique is to get good spatial and temporal reuse as follows: when a core completes a task, it is assigned the ready-to-execute task that the equivalent sequential program would have executed the earliest. The underlying idea is that we can count on sequential codes because they are now optimised by compilers: code reordering (unfolding loop), memory prefetching inside a loop, 128 bits register use and many other techniques are automatically done and increase the performance. However, there is no result, to our knowledge, about how to adpat PDF to the heterogeneous case, hence our work to guaranty good results faced to cache misses.

Paper [3] from Blelloch and Gibbons shows important properties of PDF schedules regarding cache misses. In that paper, authors compare the number of cache misses M_1 for running a computation on a single processor equipped of a cache of size C_1 to the total number of misses M_p for the same computation when using p processors and a shared cache of size C_p. They demonstrate that for any computation, and with an appropriate greedy parallel schedule, if $C_p \leq C_1 + p.W_\infty$ then $M_p \leq M_1$.

As quoted in the paper, "this gives the perhaps surprising result that for sufficiently parallel computations the shared cache need only be an additive size larger than the single processor cache" in order to guarantee no additional misses for any computation. The key point to explain the result is the choice of the schedule and we could add that it determines the result. Note that the $p.W_\infty$ product is small in practice because p is small and W_∞ is not of the same magnitude of the input size but much smaller. Note also that the result considers an ideal cache model which is a fully associative cache and it uses an ideal replacement policy meaning that the evicted memory bloc is the candidate memory block whose next access is infrequent in the future, which is optimal as proved by Belardy[2].

In [1] Acar, Blelloch and Blumofe show that the bound compares favourably to WS where cache size must be at least $C_1.p.W_\infty$ to guarantee M_1 misses.

A standard sequential execution corresponds to a particular depth-first schedule of the dependence graph; Blelloch and Gibbons call it a 1DF-schedule. Then they consider parallel schedules that are prioritised based on a given sequential schedule i.e. if there are multiple tasks ready to execute at a given time step, the "schedule will preferentially pick the tasks that are earliest in the sequential schedule". A parallel schedule based on a 1DF-schedule is called a PDF-schedule [4].

In [4] authors show how to maintain a PDF schedule online for various types of computations, in particular for computations with parallel loops or fork-join construct. The main fact to succeed is based on properties of DAG that we use that must be planar graphs. Series-parallel graphs are planar graphs and this fact may guaranty to get theoretical results. Another property of PDF schedule to transform it such that it behaves well online is based on the fact that with series-parallel graphs we are able to maintain priorities on the ready nodes without knowing all the graph: "the children of any node v have the same (1DF-schedule) priority as v relative to other ready nodes; thus, they can be substituted in for v in any sequance of ready nodes ordered by their 1DF-numbers and the sequance remains ordered by 1DF-numbers" [4].

All these reasons explain why we concentrate our attention on series-parallel graphs.

An intuitive idea for achieving good data locality with WS is to execute on the same processors nodes that are close in the computation graph. Following this idea, each thread can be given an affinity for a process and when a process obtains work, it gives priority to threads with affinity for it. To enable this, in addition to the (Path, Finish, Proc) information of our scheduler, we add another array containing for each task a first-in-first-out (FIFO) queue of tasks that have affinity for. We also add a field, protected by a mutex, specifying if the task is completed or not. Let us call this data structure a *mailbox*. We are now faced to the following problems:

How to choose the affinity threads of a given thread?

How to schedule threads equipped with this new information?

What about the completion time?

[2] Refer to http://www.research.ibm.com/journal/sj/052/belady.pdf

What about the cache misses?

5.1 How to choose affinity threads

The computation of the affinity list of a task is simply the ancestors of the task in the task graph until the root. We assume that we deal with series-parallel graphs such that we have only a unique common ancestor for each task which is the root of the graph.

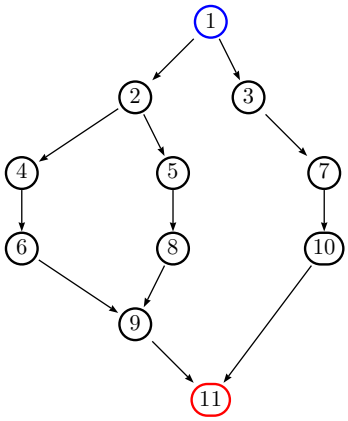

Figure 2. The task graph where nodes are numbered in order of a 1DF-schedule

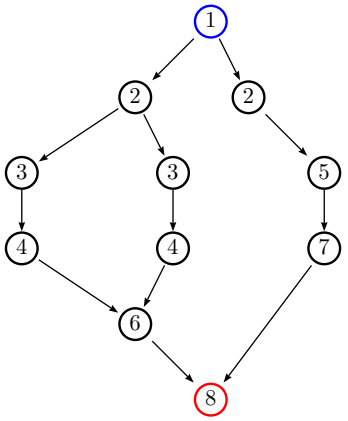

Figure 3. The task graph labelled as a PDF-schedule

5.2 How to schedule threads

In our scheduling algorithm taking into account the critical paths, we have distinguished two cases when a processor interrupts its work because it needs to wait for a task to complete. Only the α case is under concern here because for the β case we have already a criteria that requires to examine a parent of the 'faulty' node. A parent is necessary in the affinity list, so we have nothing to do.

In the α case, we modify the protocol as follows. The idle processor looks at the main array, marks the 'finish?' field with true and begins by considering the first path which is not yet started, and second, if there is no free path, it considers a slower processor: *it steals the work of the processor with more tasks that remains to accomplish in the affinity list of all tasks on active paths. In case of tie, it behaves like the initial algorithm: it steals on the processor with the speed that is immediately inferior to itself.* A processor which has been stolen by another one has the behaviour of a processor that is idle (its job is just terminated);

5.3 Completion time of the new framework

The overhead introduced by this new technique is linked with the management of the affinity lists. Since the graph is known in advance, we can pre-compute the affinity lists and store them in our data structure. The sink node of the task graph has no affinity list since when it terminates we have nothing to do. In the worst case, the parent(s) of the sink node have all the critical paths to consider in the affinity list.

For instance, on Figure 4.1 the nearest common ancestor for node g is the root (because when node g completes, d, c, f, e, b has been completed) and only the citical path on the right of the Figure may not yet finished. For node j, the nearest common ancestor is also the root, so the affinity list for node j is the critical paths starting from the root (here the two paths at the left of the Figure). The number of entries in the affinity list of each node is bounded by the number of critical paths in the graph.

5.4 Number of cache misses for the new framework

Let us remind some key points about schedules according to Blelloch [3, 1]. First, remind that we use series-parallel graphs. Second, such graphs have interesting structural properties. Let's start by terminology. We call a node in the task graph with out-degree of 2 a *fork node*. We call a node that has an in-degree of 2 a *join node* and we partition all the nodes that have in-degree 1 into two categories: a *nomadic node* has a parent that is a fork node and a *stable node* has a parent that has out-degree 1. the root node do not belong to any of these categories but it never minds. The series parallel graphs have the following properties in the context of our work stealing algorithm:

1. The least common ancestor of any two nodes is unique;

2. The greatest common descendant of any two nodes is unique;

3. Let s be a fork node, then no child of s is a join node;

Let us consider one critical path C_i. Assume that during its execution there is no stolen node and that the number of cache misses is M_i for a cache size of CS blocks. Therefore, the number of extra cache misses is:

$$CM = A - \sum_{i=1}^{n}(W_i) + St$$

where A is the number of edges in the task graph and St is the number of steals.

6 Conclusion

This paper introduces new insights into the problem of scheduling tasks according to the Work Stealing (WS) principle for heterogeneous multi-core architectures. We have shown that compromises have to be done if we want also good performances for memory management. WS do not preserve data locality by nature so we have proposed to schedule tasks according to the different critical paths (ranked by their time durations) and also, to force active processors to execute tasks that are close in the task graph by the mean of the affinity lists.

The experimental evaluation of the indroduced framework is difficult due to the lack of workload models or multiple execution traces for fork-join graphs which is our privileged model for computation. We could use the work of Lubin and Feitelson [8] to generate task graphs but this paper considers coarse grain tasks, typically tasks submitted to job schedulers of any production system. In our case, the target architecture is multi-core processors and not distributed multi-processors systems or grid systems. Another idea would have been to take random tasks generated according to a fork-join principle but in this case we are not sure that such graphs represent the realm of programs that we effectively run on current technologies.

Gibbons and all in [5] use 3 benchmarks for evaluating the Work Stealing and the Parallel Depth First frameworks: LU factorisation, hash join and mergesort. They count on online algorithms and they do not consider static graphs that are central in our work. These benchmarks are not yet appropriate for evaluating our work.

The Kaapi[3] framework relies on a variant of series-parallel graphs and it offers examples which are programmed in C++ augmented by special primitives for creating, synchronising threads. Kaapi means Kernel for Adaptative, Asynchronous Parallel and Interactive programming. It allows to execute multithreaded computation with data flow synchronization between threads. The library is able to schedule fine/medium size grain program on distributed machine. The data flow graph is dynamic (unfold at runtime). Target architectures are clusters of SMP machines and more interesting for our purpose: it is based on work-stealing algorithms. Even if Kaapi examples can serve as workloads, it remains that we also have to observe the memory performance. It can be accomplished with the Papi[4] tool that can observe the performance counters of modern chips but we are not sure that the current version of Papi supports multi-core or SMP architectures in a coherent way.

So, future work will consist in extending our framework towards the online paradigm and also to find workload generators in order to conduct simulation, for instance to estimate the size of affinity lists in real applications.

[3] See: http://kaapi.gforge.inria.fr/

[4] http://icl.cs.utk.edu/papi/

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, 2000.

[2] M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems Special Issue on SPAA00*, 35:289–304, 2002.

[3] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In P. B. Gibbons and M. Adler, editors, *SPAA*, pages 235–244. ACM, 2004.

[4] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95*, pages 1–12, Santa Barbara, California, 1995.

[5] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. 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 Press.

[6] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *ISCA*, pages 357–368. IEEE Computer Society, 2005.

[7] A. Gerasoulis and T. Yang. Acomparisonof clustering heuristics for scheduling dags on multiprocessors. *Journal of Parallel and Distributed Computing, 16, 276-291*, 1992.

[8] U. Lublin and D. G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. Technical Report 2001-12, 2001.

[9] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, 1999.