Towards a Work Stealing Scheduler for Divide & Conquer Task Spawning in MPI-2

Stéfano D. K. Mór, Nicolas Maillard
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Bento Gonçalves, 9500 Porto Alegre, RS, Brazil
{sdkmor, nicolas}@inf.ufrgs.br

Abstract

MPI-1 is the de facto *standard on message-passing parallel computation. MPI-2 has extended the first one, offering, among others, process creation at execution time. A MPI-2 task is an executable and is always ran as a process. MPI-2 lacks on specifying a canonical scheduling policy for just-spawned processes; each MPI-2 distribution should implement its own assign algorithm. Major MPI distributions apply non-efficient scheduling when referencing Divide & Conquer algorithms. One efficient solution is to use a shared memory randomized work-stealing scheduler when allocating just-spawned tasks. Our main contribution is to address four main problems when one attempts to build an MPI-2 task spawning scheduler that uses randomized work stealing algorithm on a distributed memory scenario: spawning, synchronization, concurrent memory access and checkpoiting. We present solutions that seems to work on a proper manner in solving this problems and, at conclusion, we point our mainstream project for providing an parallel interface with multi-level scheduling. At the end, we show some still open issues that derivate from our solutions and that must be solved as soon as possible.*

1. Introduction

MPI-1, the *de facto* standard on message-passing parallel computation, had its main focus on message-passing interprocess communication [3]. MPI-2 [4] has extended the previous, offering, among others, process creation at execution time. This advent was very praised, because, on High Performance Computing (HPC), one natural way to introduce parallelism on D&C is "*task spawning*"; one names a recursive call a task and spawns it in parallel, blocking until all spawned tasks return [2]. Moreover, a task spawning algorithm is *processor-oblivious*, *i.e.*, the number of available processors is not a parameter for the algorithm [9]. A MPI-2 task is an executable and is always ran as a process. MPI-2 lacks on specifying a canonical scheduling policy for just-spawned processes; each MPI-2 distribution should implement its own assign algorithm. Because major MPI-2 distributions (*e.g.*, LAM-MPI, OpenMPI, MPICH2) offer a simple round-robin scheduling, task spawning is very inefficient on MPI.

On shared memory processors one classical and efficient algorithm for efficient D&C scheduling is randomized work-stealing [2], which is proven to be optimal. On a distributed memory scenario, however, some key elements of the algorithm are largely modified and some adaptations must be provided. The D&C programming model on MPI direct implies on multiple MPI process spawns and depends on processor-oblivion, because a programmer cannot compromise it's D&C execution tree with extra scheduling processing time that is likely to do more work than each one of its leaves. We have no knowledge of any MPI distribution providing an efficient distributed D&C scheduler for dynamic-created processes.

The main challenge to be overcame is to implement a message-passing distributed-memory equivalent version of (shared memory) proven efficient Randomized Work Stealing Algorithm (RWSA). We believe to provide important considerations on this topic throughout this article. This paper focuses on showing four main problems arriving on this work:

1. protocols for spawning tasks at MPI scenario,
2. synchronization between program flows,
3. concurrent memory access, and
4. scheduler's checkpointing.

To each issue we propose a solution and to each solution we show limitations and still open questions. At the end, we give some highlight on our mainstream project for providing an parallel D&C interface with two-level scheduling (thread level and process level) and trace some conclusions, based on the remaining open questions.

The remainder of this paper is organized as follows: Section 2 describes RWSA on thinner grain; Section 3 details the four previously addressed problems that arrives on the distributed memory scenario; and Section 4 brings our conclusion and future work on the topic.

2. The Randomized Work Stealing Algorithm

RWSA [2] is a distributed on-line thread scheduling algorithm designed for shared memory multiprocessors. The algorithm is proven efficient whenever it runs over a *fully-strict computation* [2] model. Figure 1 [2] shows a Directed Acyclic Graph (DAG) for fully-strict computations model.

Each thread Γ_i has an activation frame, represented by a rounded-corner gray rectangle, that is the amount of memory that it uses, and a set of serial unitary time single clock-cycle instructions (individually labeled v). This instructions are intuitively defined as "single time-step computations". Inside an activation frame, arrows represents serial execution of instructions. Outside the activation frames, shadowed arrows represent *spawn* instructions, responsible for creating another thread at runtime. Curved arrows are *sync* primitives, that communicate to a parent that a son has ended its work, its results can be used from now on and the thread is dead (*sync* must be the last instruction on the thread). One thread cannot execute next instruction (departing arrow) until all dependencies (arriving arrows) are satisfied. It is stated that all processors have exactly same speed (global clock synchronization).

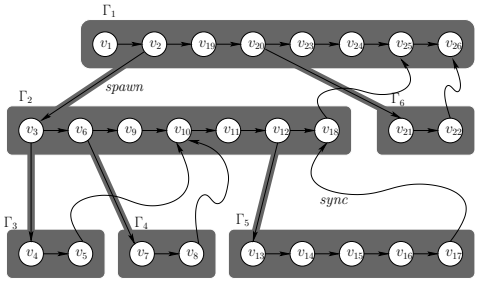

Figure 1. DAG for fully-strict computation [2].

RWSA is a greedy scheduling algorithm that acts distributed, as follows. Each processor has a *deque* of ready threads. Threads are inserted at the deque's bottom and are removed by either deque's bottom or deque's top. A processor obtains work removing a thread from the bottom of its deque. It pops and starts working on thread Γ_a until it spawns, stalls, dies or enables a stalled thread. In each case, it does:

Spawns. If Γ_a spawns Γ_b, then Γ_a is placed at the bottom of the ready queue and and processors begins to work on Γ_b.

Stalls/Dies. If Γ_a stall or die, then processor begins to work on bottommost thread of the ready deque. If there are no threads at the ready deque, it begins work stealing; it randomly choses a processor (the "*victim*") and steal the topmost thread of it. If the victim have no threads, it randomly choses another victim.

Enables. If thread Γ_a enables stalled thread Γ_b, the now-ready thread Γ_b is placed at the bottom of the ready deque of Γ_a's processor.

It has been proven on [2] that RWSA has an expected execution time of $T_1/P + \mathcal{O}(T_\infty)$ on a fully-strict computation, where T_1 is execution time with one processor, P is the number of processors and T_∞ (*critical path lenght*) is the minimum execution time with an unbounded number of processors. Similarly, the space required for computation is $\mathcal{O}(S_1 P)$ where S_1 is the minimum serial space requirement and any communication latency is proportional just to the number of messages and not to any singular snapshot constraint.

3. The Distributed Memory Scenario

In order to provide an efficient distributed D&C process scheduler for MPI, we adapt RWSA into a message-passing distributed memory environment. By doing so, at least four problems are addressed:

1. the spawning problem;
2. the synchronization problem;
3. the concurrent memory access problem;
4. the checkpointing problem.

Our objective is to present some considerations about these problems and to propose solutions that preserve RSWA's efficiency.

To model our distributed memory structure, let each process Φ_i be composed of an arbitrary number of threads, each one labeled as Γ_i and exactly as described in Section 2. That way,

1. each Φ_i has n_i threads Γ_j $(1 \leq j \leq n_i)$; and
2. the activation frame of the process Φ_i is equal to the sum of all it's threads activation frames.

Lets consider, initially, that each process have only one thread at a time. This will be modified later. So now, lets approach the problem of modeling communication between any pair of processes.

3.1. The Spawning Problem

At the previous shared memory scenario one didn't have to worry about sending and receiving data, because every variable is immediately accessible. This is not the case here. Even so, an D&C model only needs to perform one pair of sending/receiving primitives per son; the initial data on which the son will begin work and the already computed data by it. In that way, we implement *send*(μ, i), which sends data μ to the process Φ_i and is non-blocking.

Since fully-strict computation model allows communication only through sons and parents, it's desirable that the i above has some restrictions. Rather than being a global process identifier, i should refer only to previously spawned processes and that should be all the knowledge a process must have about his environment. We can achieve this by replacing *spawn* with *cspawn*, whose only difference from the normal one is that when is called, it returns a *communicator* on both the spawning process and the spawned one. A communicator is an abstract data structure that identifies a given process. So i must be necessarily a communicator in order to make *send* works.

At the proposed scenario, when a son has done its computation, it must send the data back to its parent. To get this data a parent must call *receive*(μ, i), meaning that it receives μ (data) from process identified by communicator i. *Receive* is blocking and replaces the use of *sync*; it was previously used for indicating that some shared return variable was at an proper state to be read. Now the parent process can block on *receive* just as equal it did with *sync*, waiting

the desired data to arrive. Since a *receive* must be aware of the sending process, all *cspawn* calls must send, along with the data needed to start, the identifier of the spawning process, to be used at the later *send*. Figure 2 shows an modified DAG with only two processes that uses the method we described above. The now introduced dashed lines are inserted modifications on the original DAG, since new instructions should be add in order to make it compatible with message-passing model.

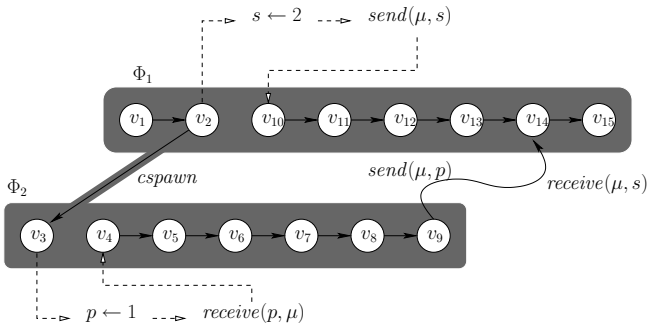

Figure 2. Modified DAG for fully-strict computation on distributed memory (simplified version).

3.2. The Synchronization Problem

While a multiprocessor environment allows passive data sharing, a message-passing multicomputer one does not; an explicit *receive* must be used every time some process needs data belonging to another process. It implies that any steal attempt against a process would have to either interrupt its processing or wait it to finish. Both cases induce an unacceptable synchronization need from the RWSA point of view, because the data would not be "stealed" anymore.

As solution we extend the previous model to admit exactly two process per processor executing in parallel at a single time; one process executes the normal work flow, while another one acts as a "process manager". Each process manager communicate with another through the use of special *send*/*receive* calls to handle work stealing. Although *send* remains the same, *receive* becomes universal, which means that it accept messages from *any* source. To make difference between normal *receive* and this one we will call the later *ureceive*. We highlight that *ureceive* retains the information of who was the request sender and holds it on τ, because the protocol demands respond the work stealing request to the sender. The process manager holds the process deque and continually executes distributed RWSA. Process manager has one "watcher" thread that treats process dieing/stalling, spawning and enabling and another thread that manages work stealing attempts. Since both share the deque in shared memory, there is no need to interrupt the handler thread and thus work stealing acts just as on multi-thread scenario.

We highlight that making it work properly requires an fully-parallel SMT processor. Figure 3 represents this solution.

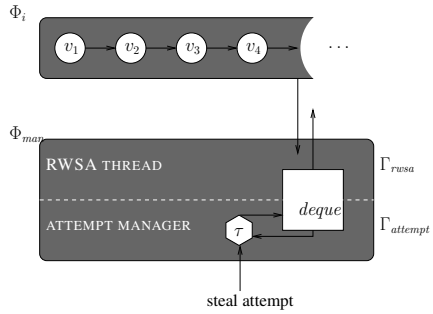

Figure 3. Solution for the synchronization problem with two processes, one multi-threaded. The white octagon acts as *ureceive*/ *send*.

3.3. The Concurrent Memory Access Problem

The work on Cilk language [1] raises an interesting question about work stealing attempts; for RWSA's performance-sake each request must be solved in constant time, being not relevant whether the request is successful or not. With classical data synchronization (*e.g.,* mutex, semaphore, monitor, *etc.*), however, there is no condition to estimate such constant time.

As a solution, [1] proposes a protocol named THE, which, summarizing, doesn't block on accessing locked variables; in presence of a lock it returns immediately saying that the request has failed. THE protocol can be emulated on our distributed memory scenario. Because, as in previous subsection, we proposed a manager process that has two threads that access the deque synchronously. So, attempt manager thread should implement THE protocol in order to share the deque with RWSA thread, because it has always the preference. In fact, this way is very similar to the one found on [1].

3.4. The Checkpointing Problem

RWSA's first clause says that whenever a new task is spawned, current working thread Γ_a goes on the top of the deque and the newly spawned thread Γ_b begins to execute on the processor. Because we are running a process (with the help of a manager), checkpointing of Γ_a for later resume is very complex and demands heavy non problem related work.

To overcome it, we adopt an approach next to the one seen on [8]. A distributed memory RWSA program structure would be as follows.

According to RWSA, v_1 and v_2 executes and after, in line 3, RWSADISTMEN goes to deque's top and t_1 begins to execute. When RWSADISTMEN begins to execute again (on local or remote processor), it must begin executing line 4.

Algorithm 1 RWSADISTMEN()

1: v_1
2: v_2
3: $a \leftarrow cspawn\ t_1$
4: $b \leftarrow cspawn\ t_2$
5: $c \leftarrow cspawn\ t_3$
6: **for** $i \leftarrow a$ to c **do**
7: $\quad receive(\mu_i, i)$
8: PROCESSRESULTS(μ_a, μ_b, μ_c)

We adopt the following strategy for queuing the current active process:

1. Everything starting from the *receive* is treated as a single task.
2. After line 3 we spawn the rest of the code as a single task, that will be only the serial execution of the remaining tasks.
3. It works recursively for the remaining $cspawn$.

The last, artificially created task (line 8) will be called a "continuation task" [8]. It is different from the others because it cannot execute until all other tasks have executed, so whenever it is executed, it blocks, and waits until it is enabled to regain execution. It cannot be subject of stealing because of its local dependencies.

4. Conclusions, Related & Future Work

Our practical purpose on presenting this kind of problems is to provide full compiler support for parallelism, based on code comments and multicore oblivious processing. In short, we are going to take serial code and insert on it some special comments that will tell a compiler to add code for both levels of scheduling,

1. thread-level, using *Intel Thread Building Blocks*; and
2. process-level, using a modified version of MPICH that implements our randomized work stealing scheduler for distributed memory.

We have chosen MPICH for implementing our distributed memory work stealing scheduler for MPI. Asides being one of the major MPI distribution, MPICH also

- fully supports thread programming and communication, allowing our TBB layer to make thread-safe message-passing calls;
- its process manager, MPD, is written in Python, making it more easily modifiable; and
- the two other major distributions, LAM-MPI and OpenMPI, are not at good situation by now; the first is discontinued and the later is still very instable.

There is some work to provide adaptive and efficient scheduling on MPI [6, 7], which presented a very efficient and versatile architecture of virtual resources that does migration between nodes. Yet, remains the fact that it doesn't handle dynamic process creation till present date.

It is very important to highlight that very mature work exist outside MPI; to name a few, there is KAAPI program interface [5] and Satin program interface [10, 11].

Before implementing any work stealing scheduler for MPI-2, we have to face the open questions that still remains:

1. Blocking *send*/*receive* does affect RWSA's efficiency?
2. How to modify MPD in order to implement RWSA? Should one allow all-to- all communication (and be susceptible to hardware constraints) or simulate it through ring topology (which may introduce undesired variable overhead)?

Until the end of the year we must have a working compiler and results that shows how good our solution is. We are pretty confident that we will achieve considerable performance gain.

References

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, July 1995. ACM Press.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS*, pages 356–368, 1994.

[3] M. P. I. Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, University of Tennessee, Knoxville, Tennessee, 1994.

[4] M. P. I. Forum. Mpi: A message-passing interface standard ver. 2.1. Technical report, Message Passing Interface Forum, Junho 2008.

[5] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 2007.

[6] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, volume 2958/2004 of *Lecture Notes in Computer Science*, pages 306–322, College Station, Texas, Oct. 2004. Springer Berlin / Heidelberg.

[7] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance Evaluation of Adaptive MPI. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 12–21, New York, NY, USA, Mar. 2006. ACM Press.

[8] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT, Junho 1998.

[9] D. Traoré, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard. Deque-free work-optimal parallel stl algorithms. In E. Luque, T. Margalef, and D. Benitez, editors, *Euro-Par*, volume 5168 of *Lecture Notes in Computer Science*, pages 887–897. Springer, 2008.

[10] R. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient parallel divide-and-conquer in java. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 690–699, London, UK, 2000. Springer-Verlag.

[11] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, New York, NY, USA, 2001. ACM.