Title Present!

Gerson G. H. Cavalheiro*
Federal Univeristy Of Pelotas
Gomes Carneiro, 1
Pelotas, Brazil
gerson.cavalheiro@inf.ufpel.edu.br

Alan S. de Araújo†
Federal Univeristy Of Pelotas
Gomes Carneiro, 1
Pelotas, Brazil
asdaraujo@inf.ufpel.edu.br

Cícero A. de S. Camargo
Federal Univeristy Of Pelotas
Gomes Carneiro, 1
Pelotas, Brazil
cadscamargo@inf.ufpel.edu.br

ABSTRACT

Testing tex editing on GitHub! E eu consegui!

Keywords

Concurrent Programming, Multithread Environment, Thread Scheduling

1. INTRODUCTION

The acceptance of multiprocessors in different domains of application and usages are primarily a result of the quest to increase computer system performance. The multicore technology is nowadays a usual solution employed from desktops to machines listed in Top 500 Supercomputer Sites. The main appeal of such architectures is the high potential of performance added to good ratio cost/performance. Another aspect related to the recent popularity of such systems is the simplicity to write programs when compared to distributed memory architecture, since it is unnecessary to deal with addresses of processes to communicate parts of a same program. Thus, multithreading programming is considered a convenient and efficient mechanism to exploit multicore architectures [12, 9]. Nevertheless, development of parallel applications for any multiprocessor architecture requires the programmer ability to both to conceive and write a concurrent/parallel program from an application description and to design a strategy to fully utilize the computing resources.

Several commercial and academic options for programming multiprocessor systems are available. Among the most popular commercial programming tools we can name Pthread [10], OpenMP [3], Cilk+ [8] and TBB [11]. Among the academic/research proposals, we name Athreads [2], Filaments [5], Kaapi [6] and Nabbit [1]. Excepting Pthread, which offers only an API (Application Programming Interface) to develop a parallel program and delegates to the operating system the scheduling of threads over the computing resources, all cited programming tools offer both a multithread API and a runtime supporting a application level ([4]) scheduling strategy. To achieve the scheduling at application level, the runtime must to manage the threads created by the program during the execution, competing with the application program for the system resources and generating overhead at execution time. To reduce the impact of the scheduling overhead, the data structures employed to handle the user level threads (those threads generated by the application program) must to be implemented efficiently.

In this paper we investigate the data structures required to implement, at execution time, a scheduling strategy based on list scheduling, whereas threads are inserted removed taking into account this basic policy: (*i*) a thread can be added to the lists when it has no more dependencies to execute, that is, the thread is ready to execute, and (*ii*) once a processor becomes idle, some criteria is applied to rank all threads in the list in order to chose the one with the grater rank (priority) to be executed at this moment. The goal is to evaluate alternatives to implement efficient support such data structures for a specific multithread runtime.

The runtime scheduler of Athread, Cilk+, Kaapi, Nabbit and TBB take into account the depth of the threads to ranking the list. As far from the first thread generated by the program as deeper the thread is considered. We will consider in this work list scheduling which deal whit this application propriety. Thus, an efficient implementation of such list is essential feature to obtain good performance. Our case study will be conduced in the context of Athread project. We propose two alternatives to implement such list in its runtime kernel. First we evaluate the use of optimized malloc/free internal library then a lock-free strategy. Both strategies were developed considering the requirements of the Athread scheduling kernel.

The remaining of the paper is organized as follows. In Section 2 we introduce the use of list scheduling as basic strategy of scheduling on multithreading environments and we also summarize some techniques employed in some previous listed environments to implement the list of threads. In Section **??**, we detail Athread runtime and in the following, in Section 4, we present the two approaches evaluated in this work. A performance assessment of these two approaches is presented in Section 5. Finally, final remarks and future work are discussed in 6.

2. RELATED WORK

A large number of scheduling heuristics algorithms are based on list scheduling strategy [este, 2, 3, 4, 7]. Con-

*Thanks to?
†Thanks to?

sequently, we found many runtime systems employing dynamic list scheduling techniques at runtime. In this section we briefly present this strategy and data structures required to support the scheduling operations. We also present the employ of list on Cilk, Kaapi and TBB.

2.1 Dynamic list scheduling

A dynamic list scheduling takes as input a DAG (Directed Ciclic Graph) describing task and dependencies among tasks [7]. In this DAG vertices represent the tasks and the edges represent the communication among tasks. A data produced (output) of a given task v_i and necessary (input) to compute another task v_j is represented as an edge $e_{i,j}$ in the DAG and consist in a dependence $v_i \rightarrow v_j$. In other worlds, v_j (called sink task) is not ready to execute before v_i (called source task) ends. The list of immediate successors or predecessors of a task v_i is given by $succ(v_i)$ and $pred(v_i)$, respectively. If a task has not any predecessor it represents an *entry* task, as well as represents a *exit* task when has no successors.The vertices and edges in the DAG can be also annotated at application level with costs required to compute the tasks and/or the communicate the data, respectively.

DEFINITION 1 (TASK). *A task v_i of a given DAG G is ready to execute when all tasks in $pred(v_i)$ were finished; once a task is started, it must to be executed until completion. A running task can not be neither preempted nor migrated.*

This task model implies that a processor is able to execute a task at once. As result, if the number n of ready tasks is larger than the number m of processors, ready tasks must to be maintained in data structure waiting for its launching. Regardless specific implementation, we assume this data structure is a *list* globally accessible to all processors. For sake of simplicity we assume also a multiprocessor machine with no costs (latencies) associated to a ask read/write input/output data.

The basic algorithm for a dynamic list scheduling is given in Algorithm 1. In this algorithm, a ready task is a task having all its predecessors scheduled. The output of the scheduler is a matrix representing the mapping of tasks onto the processors. Each matrix line represents one of the m processor of a given parallel architecture and each column a task in the graph G to be scheduled.

Input: G: a DAG with n vertices ; m: the number of processors.
Output: $M[m,n]$: each line in M represents a processor and each column a vertex of G. For a given position $M(i,j)$, a non *null* value represent the data d to start task.

$V \longleftarrow \emptyset$; {The list of ready tasks}
$M(i,j) \longleftarrow null$; {For all $0 \leq i \leq m, 0 \leq j \leq n$}
while $G \neq \emptyset$ **do**
  $V \longleftarrow V \cup G.readyTasks$; {Read and remove...}
  $G \longleftarrow G - G.readyTasks$; {...ready tasks}
  $V.sort$; {Apply a ranking policy}
  $v \longleftarrow V.first$;
  $p \longleftarrow M.nextIdleProcessor$
  $M(p,v) \longleftarrow dataStart(v)$ {Maps the task and...}
  {...sets all $succ(v)$ as ready}
end

Algorithm 1: A general Dynamic List Algorithm scheduling algorithm.

2.2 Work stealing

Work stealing has proven [**?**] to be an effective method for scheduling parallel programs on multiprocessors. One of the reasons is that this approach limits the number of scheduling operations involving more than one processor to the situation were a processor has no more local task to process. When this situation happens, a steal is initiated by the idle processor to obtain work from another (randomly chosen) processor. Thus, a local scheduler operates a local partition of the list of ready tasks. a processor accesses the partition belonging to another processor only when it's own partition becomes empty.

Probably one of most know work steal implementation is provided by Cilk. The local lists are implemented with a double ended queue (*deque*). The local scheduler accesses the *bottom* of the queue to insert and remove tasks. In a steal, the thief accesses the *top* of a remote partition. This strategy allows to attain remarkable performance indexes when associated to an efficient heuristic to detect the critical path (that is, the long chain of dependencies of tasks [**?**]) of the running program. In Cilk the critical path can be inferred since its API limits writing parallel programs in a nested fork-join style. In such way, closer a task is from the top of a partition, more amount of work (new tasks) it is able to create. A steal in this case potentialize the success of steal, providing a task which will makes grow the thief deque, while exploit at local scheduler the locality of tasks limiting synchronization among processors.

2.3 Task stealing

cicero 2 paragrafos

2.4 Private queues

Gerson

2.5 Distributed list

Gerson

3. ATHREAD

4. IMPLEMENTATION

5. EVALUATION

6. CONCLUSION

7. REFERENCES

[1] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, pages 1–12, 2010.

[2] G. G. H. Cavalheiro, L. P. Gaspary, M. A. Cardozo, and O. C. Cordeiro. Anahy: A programming environment for cluster computing. In *VII High Performance Computing for Computational Science*, Berlin, 2007. Springer-Verlag. (LNCS 4395).

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, 2001.

[4] D. Feitelson. Job scheduling in multiprogrammed parallel systems. *IBM Research Report*, 19790, 1997.

[5] V. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. In *In First Symposium on Operating Systems Design and Implementation*, pages 201–213, 1994.

[6] 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*, pages 15–23, New York, NY, USA, 2007. ACM.

[7] R. L. Graham. Bounds for certain multiprocessor anomalies. *The Bell Syst. Tech. J.*, CLV(9), 1966.

[8] C. Intel. Using intel(r) cilk(tm) plus. http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/mac/cref_cls/common/cilk_bk_using_cilk.htm, 2012. Acessado em Janeiro/2012.

[9] C. Mei, G. Zheng, F. Gioachin, and L. V. Kalé. Optimizing a parallel runtime system for multicore clusters: a case study. In *Proceedings of the 2010 TeraGrid Conference*, TG '10, pages 12:1–12:8, New York, NY, USA, 2010. ACM.

[10] B. Nichols, D. Buttar, and J. P. Farrell. *Pthreads Programming*. O'Reilly, Cambridge, 2 edition, 1998.

[11] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[12] A. C. Sodan. Message-passing and shared-data programming models - wish vs. reality. In *HPCS*, pages 131–148, 2005.