Scheduling by Work-Stealing in Hybrid Parallel Architectures*

Vinícius Garcia Pinto, Nicolas Maillard
Parallel and Distributed Processing Group (GPPD)
Institute of Informatics - Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre - RS - Brazil
{vgpinto, nicolas}@inf.ufrgs.br

Abstract

Nowadays, parallel computing systems have been based on multicore CPUs and specialized coprocessors, such as GPUs, due to the limits achieved by traditional architectures. In order to obtain the expected performance in these systems, the workload must be distributed and redistributed in an efficient way through some technique of scheduling, like work-stealing. This work aims to propose, implement and validate a scheduling approach based on work-stealing in parallel systems with CPUs and GPUs simultaneously. The results show that our approach and Cilk have very close performance. Furthermore, the use of both CPUs and GPUs provides clear improvement in performance due to better utilization of processing resources provided by the work-stealing scheduling.

1. Introduction

Until few years ago, the performance improvement of computers architectures was related with the decrease in the size of transistors and the rise of clock frequency. This improvement was based in the *hardware* update without changes in *software* [1, 2]. However, nowadays this model achieved physical limits and computers' performance is improved by addition of more processors or cores in a single chip. Another way used to improve the computers performance is the use of specialized coprocessors or accelerators, like GPUs (Graphic Processing Units) [3].

Today's High Performance Computing systems are also composed in a hybrid model. These systems combine general-purpose homogeneous multicores with specialized coprocessors [4].

Existing programming tools for parallel computing, in special in hybrid architectures, imply in high programming efforts to make efficient use of all processing resources available in the system. These models rely on low level operations such as explicit locks and synchronization points [5, 6].

Task parallelism, implemented by Cilk [7], OpenMP 3 [8] and Intel TBB [9], is considered a generic and high-level model for these new hybrid architectures. However, using this model makes necessary an efficient task scheduler to optimizes the concurrent execution of tasks at runtime [6, 10]. Work-Stealing [13] is an efficient scheduling technique for task parallelism adopted by Cilk and Intel TBB.

In this paper we present and evaluate an approach to scheduling tasks by work-stealing in hybrid architectures that use CPU and GPU resources simultaneously.

The remainder of this paper is organized as follows: Section 2 presents a review about task parallelism concepts. Section 3 presents our approach for scheduling tasks in hybrid systems. Section 4 presents our experimental evaluation. Finally, Section 5 presents the conclusions and future works.

2. Background

In this section, we present a review about task parallelism concepts and work-stealing scheduling.

Tasks: are computation units into which the entire computation is divided by means of decomposition. There may be dependencies among some tasks and tasks may not be all of the same size [11, 12].

Task Parallelism: is the type of parallelism that is naturally expressed by independent tasks in a task-dependency graph. Parallel quicksort, sparse matrix factorization and other algorithms derivated via divide-and-conquer decomposition are examples of algorithms that can be described in this model [12].

Work-stealing: is a distributed algorithm for dynamic scheduling. In work-stealing scheduling, idle processing units, called thieves, steal tasks from busy processing units,

* This work was supported in part by FAPERGS, CNPq and CAPES.

called victims. Each processing unit has its own deque (double-ended queue) of ready tasks [13].

The work-stealing algorithm has been adopted as an efficient technique for scheduling and dynamic loading distribution in several parallel programming tools for multicore CPUs, such as Cilk, Intel TBB and KAAPI [14]. There are several preliminary works that aim to extend the "work-stealing" scheduler for specific scenarios, for example: *adaptive work stealing* [15], *idempotent work stealing*, *scalable work stealing* [16] and *enhanced cilk scheduler* [17]. In the context of GPU processing, recent studies [18, 19, 20] have demonstrated the applicability of this technique for scheduling and load balancing among streaming multiprocessors inside GPU.

3. Our Approach

Our approach to offer work-stealing scheduling in hybrid architectures consists of three main parts: a *Task* component, a set of *TaskProcessors* and a *Manager* component.

- *Manager* component controls the creation of *TaskProcessor* components, the submission of new *Task* components and the submission of *Task* components to the wait queue.
- *TaskProcessor* components are used to execute *Task* components. Each *TaskProcessor* has a deque of *Tasks* to process and a private queue of waiting *Tasks*.
- A *Task* component represents a unit of computation. Each *Task* component has, at least, three methods: a runCPU() method that contains the CPU implementation of the task, a runGPU() method that contains the GPU implementation and a task termination method called runTermination().

A set of TaskProcessors represents the processing resources available in the hybrid architecture. Usually, there is one *TaskProcessor* for each CPU core. When GPU resources are available in the system, one(or more) TaskProcessors are assigned to control these GPUs. When a *TaskProcessor* becomes idle, it tries to steal some task from another *TaskProcessor*.

3.1. Prototype implementation

A prototype of this approach was implemented in C++ using the Boost C++ library [21]. The GPU tasks are implemented using CUDA [22] and Thrust library [23]. Figure 1 presents an example to define a Task using our prototype. Figure 2 presents a code example of a task implementation.

```
class Sort : public TaskCPUGPU {
public:
    Sort(TYPE* input, int sz, int gr);
    virtual void runCPU();
    virtual void runGPU();
    virtual void runTermination();
private:
    void mergesortSeq(TYPE* input, int sz);
    void mergeSeq(TYPE* input, int sz);
    TYPE* input;
    int size, grain;
};
```

Figure 1. Code example to define a Task in our prototype

```
void Sort::runCPU() {
    int mid;
    if (size > grain) {
        mid = size / 2;
        Sort *ms;
        ms = new Sort(input, mid, grain);
        this->taskSub(ms);
        ms = new Sort(input + mid, size - mid,
             grain);
        this->taskSub(ms);
        this->tasksAsyncWaitTerm();
    } else {
        this->mergesortSeq(input, size);
    }
}
void Sort::runGPU() {
   /* CUDA code */
}
void Sort::runTermination() {
    Merge *merge;
    merge = new Merge(input, size);
    this->taskSub(merge);
    this->tasksAsyncWait();
}
```

Figure 2. Code example of Task implementation in our prototype

4. Experimental Evaluation

This section describes the experimental evaluation of the prototype that implements our approach for scheduling tasks by work-stealing in hybrid architectures.

4.1. Experimental Setup

Experiments were conducted on a hybrid platform. This system is composed of a Intel Core i7 930 quad-core CPU running at 2.80GHz with 12 GB of DDR3-1066 MHz RAM. A NVIDIA GTX480 GPU card with 480 CUDA cores running at 1.4GHz with 1.5 GB of GDDR5 RAM is attached through a PCI-E 2.0 bus. This platform runs a Ubuntu 11.10

GNU/Linux operating system, with Linux kernel 3.0.0, NVIDIA driver 4.1 and CUDA 4.1.

4.2. Benchmark Application 1

The first benchmark application consists on a transformation. A transformation is a simple algorithm that applies an operation to each element in an input array and then stores the result in an output array.

This benchmark uses a recursive algorithm derived via divide-and-conquer decomposition to generate many parallel tasks. At each step, this algorithm divides the input array into two smaller sub-arrays and assigns these sub-arrays to new tasks. This recursive division stops when the input array size reaches a threshold value. Our implementation uses different threshold values for CPU and GPU tasks.

4.3. Benchmark Application 2

The second benchmark application consists on a sorting. A sorting is an algorithm that puts elements of an input array in a certain order and then stores the result in an output array.

This application uses a hybrid merge sort algorithm derived via divide-and-conquer decomposition to generate many parallel tasks. This algorithm divides the input array into two smaller sub-arrays and assigns these sub-arrays to new tasks. When these two new tasks complete their work, a new task is created to merge the two sorted sub-arrays. This recursive division stops when the input array size reaches a threshold value. If sub-array size is smaller than the threshold value then another sort algorithm is used (e.g., insertion sort, introsort sort, radix sort). For this reason, we call this algorithm as hybrid merge sort. Our implementation of hybrid merge sort algorithm also uses different threshold values for CPU and GPU tasks.

4.4. Results

Results presented in this Section were obtained considering an average of one hundred executions. The graphs presented in Figures 3 and 4 show a performance comparison between our prototype and Cilk with benchmark applications 1 (4.2) and 2 (4.3) using only CPUs. The input sizes used were 268435456 elements for application 1 and 134217728 elements for application 2. The graph of Figure 5 shows a performance comparison of our prototype with benchmark application 2 using both CPUs and GPU and using only CPUs. The input size used was 16777216 elements.

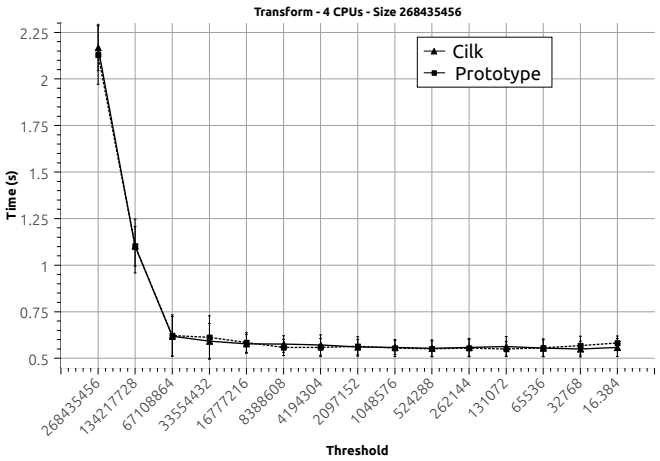

Figure 3. Performance Cilk vs Prototype - Benchmark Application 1

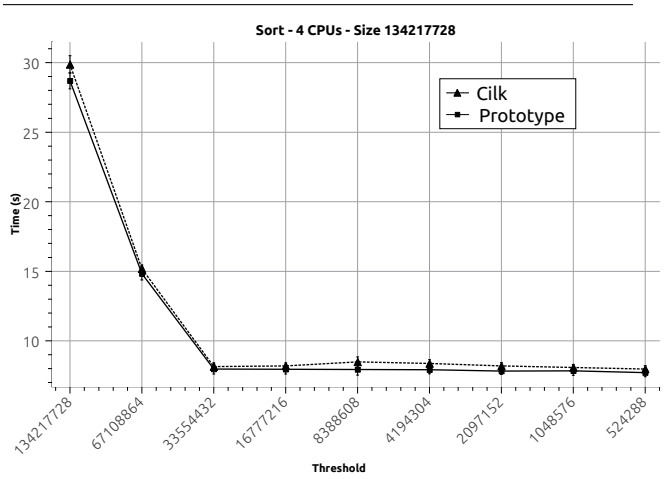

Figure 4. Performance Cilk vs Prototype - Benchmark Application 2

5. Conclusions and Future Works

This work presented and evaluated an approach for scheduling tasks by work-stealing in hybrid architectures. Results showed that the performance of our prototype is very close to the performance of Cilk when we only use CPUs for computations. Furthermore, these results show that the use of both CPUs and GPUs simultaneously provides a clear improvement in the performance.

For future work, we plan to evaluate the performance of

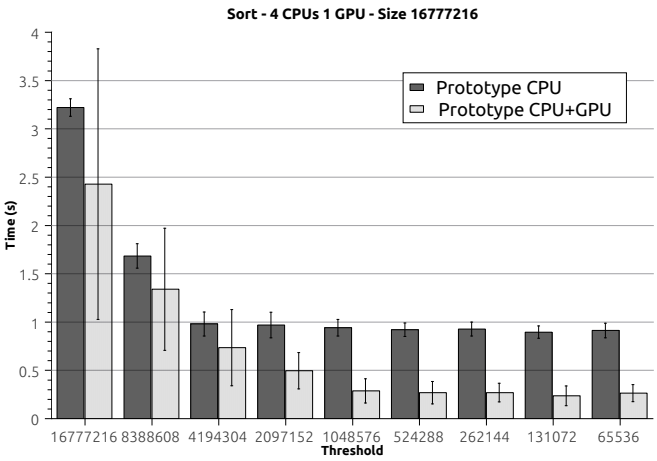

Figure 5. Performance Prototype with CPUs only vs Prototype with CPUs + GPU - Benchmark Application 2

our prototype with other benchmarks and compare the performance with other implementations that use only GPUs.

References

[1] D. Callahan, "Design Considerations For Parallel Programming," *MSDN magazine*, vol. 23, pp. 74 – 85, Oct. 2008.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, pp. 56–67, Oct. 2009.

[3] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Comput.*, vol. 36, pp. 232–240, June 2010.

[4] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Petascale Computing with Accelerators," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, (New York, NY, USA), pp. 241–250, ACM, 2009.

[5] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, May 2006.

[6] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "Parallel programming of general-purpose programs using task-based programming models," in *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, (Berkeley, CA, USA), p. 13, USENIX Association, 2011.

[7] 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 *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, (New York, NY, USA), pp. 207–216, ACM, 1995.

[8] O. Architecture, "OpenMP Application Program Interface v3.0," tech. rep., 2008.

[9] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Series, O'Reilly, 2007.

[10] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine, "PFunc: modern task parallelism for modern high performance computing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, (New York, New York, USA), p. 1, ACM Press, Nov. 2009.

[11] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Software patterns series, Addison-Wesley, 2005.

[12] A. Grama, *Introduction to Parallel Computing*. Pearson Education, Addison-Wesley, 2003.

[13] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, (Washington, DC, USA), pp. 356–368, IEEE Computer Society, 1994.

[14] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proc. of The 2007 international workshop on Parallel symbolic computation, PASCO'07*, (London, CAN), ACM, 2007.

[15] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu, "Adaptive work-stealing with parallelism feedback," *ACM Transactions on Computer Systems*, vol. 26, pp. 1–32, Sept. 2008.

[16] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis - SC '09*, (New York, New York, USA), p. 1, ACM Press, Nov. 2009.

[17] M. A. Bender and M. O. Rabin, "Scheduling Cilk multithreaded parallel programs on processors of different speeds," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, (New York, NY, USA), pp. 13–21, ACM, 2000.

[18] J. Toss and T. Gautier, "A New Programming Paradigm for GPGPU." To be published in the Proceedings of the 18th International Euro-Par Conference on Parallel Processing (Euro-Par 2012).

[19] D. Cederman and P. Tsigas, "On Dynamic Load Balancing on Graphics Processors," *Technology*, pp. 57–64, 2008.

[20] D. Cederman and P. Tsigas, "Dynamic Load Balancing Using Work-Stealing," in *GPU Computing Gems: Jade Edition* (W.-M. W. Hwu, ed.), pp. 485–499, Elsevier, 2011.

[21] B. Ling, *The Boost C++ Libraries*. XML Press, 2011.

[22] NVIDIA, "CUDA C Programming Guide," tech. rep., 2011.

[23] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. Version 1.3.0.