Reducing the impact of communication times on heterogeneous applications scheduling using KAAPI

David Beniamine, Guillaume Huard, Denis Trystramn
Grenoble Informatics Laboratory (LIG)
Inria
Grenoble, France
david.beniamine@imag.fr, guillaume.huard@imag.fr, denis.trystramn@imag.fr

***Abstract*—High Performance Computing machines use more and more Graphical Processing Units as they are very efficient for homogeneous computation such as matrix operations. However before using these accelerators, one has to transfer data from the processor to them. Such a transfer can be slow.**

In this study, our aim is to reduce the impact of communication times on the makespan of a scheduling. Indeed, with a better anticipation of these communications, we could use the GPUs even more efficiently. More precisely, we will focus on machines with many GPUs and on applications with a low ratio of computations over communications.

During this study, we have implemented two offline scheduling algorithms within XKAAPI's runtime. Then we have led an experimental study, combining these algorithms to highlight the impact of communication times.

Finally our study has shown that, by using communication aware scheduling algorithms, we can reduce substantially the makespan of an application. Our experiments have shown a significant reduction of this makespan on a machine with several GPUs executing homogeneous computations.

***Keywords*-GPU; Clustering; Scheduling; KAAPI; Commuications;**

I. Introduction

II. State of the Art

Numerous programing environment have been designed to help programmers writing efficient parallel code. Although these middlewares are conceived for various architectures and have different programing models, most of them need to find a suitable scheduling.

To present the related work, we are going to explain first the various ideas carried out by the existing parallel programing environments in section II-A. Then in section II-B we will present several scheduling algorithms more or less suited to our problem and discuss their pros and cons.

A. Writing parallel applications

When we need to write parallel code, before choosing the middleware or the API we will use, we have to think about the architecture that we target. For example, if we want to run some computation on a GPU, we can use a vendor's interface such as CUDA [1] or an open one like OpenCL [2]. We can combine this code with a classical API for parallelization on Symetric MultiProcessing such as OpenMP [3]. And if we need to work on a cluster of machines and send messages through a network, we can add some MPI [4] code.

However each of these environments are designed for a specific kind of architecture, if we just combine them, we still have to manage all data transfers and synchronizations. Other runtimes such as KAAPI [5], StarPU [6] and OmpSS [7] have been designed to provide a higher level of abstraction to the programmer. Therefore he does not have to manage the data transfers and synchronizations which can be deduced by the middleware, from the dataflow graph of the application. The user only need to specify for each task the access mode of the data (read and/or write).

However this classification is not the only one possible, indeed all these softwares have evolved since their first release, hence it is interesting to discuss another difference between them such as the policy used for the scheduling of the tasks onto the resources.

The simplest policy consist in dividing the work into equally sized chunks (i.e. cut a loop into as much chunks as many resources we have) and give the same number of chunks to each processors. This policy is used by default in OpenMP, however when either the resources or the work are heterogeneous, it leads to workload imbalances. To avoid that, and following the example of Cilk [8], several middlewares implement a dynamic algorithm such as workstealing [9]. Among them we can find the latest revision of OpenMP [7], XKAAPI [10], its former release KAAPI [5] and StarPU [6].

Although the workstealing produce a good work balancing, it does not take into account the outgoing communications of the stolen tasks, hence it can take bad decisions regarding communication times and contention. Therefore some runtimes such as XKAAPI [10] and StarPU [6] embed more sophisticated scheduling algorithms such as HEFT which take both heterogeneity and communication times into account. These middlewares also have an API to develop other schedulers.

It's important to note that formalisms which have been designed initially for different purposes are converging to a dataflow model with a very sophisticated runtime. For example OmpSS which is aimed at providing OpenMP's model for heterogeneous platforms allow the user to express the dependencies between tasks. As these middlewares manages heterogeneous resources, they are all concerned by the limitation of the workstealing algorithm explained previously.

B. Scheduling algorithms

As we mentioned earlier, the task scheduling is a NP complete problem [11]. The most widely used scheduling heuristic is the list scheduling [12], [13]. The original idea is to forbid any processor to be idle when at least one task is ready (i.e. all its predecessors are finished) to be executed. When there are more than one ready tasks, they are sorted according to some priorities, and the first task of the list is executed as soon as possible. On a model with identical processors and without communication, this algorithm give us a 2-approximation. The workstealing algorithm is based on the same idea and is indeed quite efficient for SMP architecture.

However when we work on heterogeneous machines (without communication) the list scheduling can give bad results as the bound becomes $\frac{\omega}{\omega^*} \leq \min(s+2-\frac{2s+1}{n}, \frac{n+1}{2})$ [14] where s is the number of type of resources and n the number of processors. Thus each times we add a different resource, we add one time ω^* to the worst case of the algorithm. Furthermore with identical processors and communication times we have a bound of $\omega \leq (2-\frac{1}{n})\omega^*+C$ [15] where C is the maximum chain of communications through the graph. Although in this bound C is an additive constant, if there are some communication all along the critical path, $C \leq c\omega^*$ where c is the maximal cost of one communication. Hence the worst case is proportional to the cost of a communication. Consequently the list scheduling does not scale for heterogeneous machines and we need to take into account the communications.

In this section we will first describe some adaptations of the list scheduling to heterogeneous model, then we are going to present some clustering algorithms which works on the structure of the graph and group the tasks together, zeroing the communication cost inside a cluster.

1) List scheduling: Many searchers have revisited the list scheduling, adapting it to various architectures and testing several heuristics. Nevertheless, as our aim is to study the impact of data transfers, we will only present works related to scheduling with communication times.

Earliest Tasks First [15] is designed for a finite number of identical processors with a given unitary cost of communication for each couples of processors and without communication contention.

For a tasks T we defined the earliest starting time $e_s(T) = max(CM, min_{p\in I}(r(T,P)))$ where CM is the Current Moment, I is the set of free processors, and $r(T,P)$ is the minimum time when all message needed by T are received by P. ETF always execute the task with the smallest earliest starting time.

Although this algorithm find a scheduling with a makespan $\omega_{ETF} \leq (2-\frac{1}{n})\omega^*+C$, it is designed for homogeneous processors while this study target heterogeneous machines.

The algorithms Heterogeneous Earliest Finish Time and Critical Path On a Processor [16] target heterogeneous clusters where both communication times and execution times are not the same on all processors.

The HEFT algorithm sort the ready list by non increasing upward rank, which is defined recursively for a task t by the following:
$rank_u(t) = \overline{\omega_t} + \max_{t'\in succ(t)}(\overline{C_{t,t'}} + rank_u(t'))$ where $\overline{\omega_t}$ is the mean execution time of t and $\overline{C_{t,t'}}$ is the mean communication time from t to t'. Then, for each task of the ready list, HEFT assign it to the processors which minimize its finish time.

CPOP is a variant of HEFT where the rank is the sum of upward and downward rank. Moreover CPOP compute the critical path and its execution time for each processor. The best processor is dedicated to the tasks of this path. All the other tasks are assigned the processor which minimize their finish time among the remaining processors.

As the HEFT algorithm assumptions correspond to our case and as it is already implemented in XKAAPI [10] and StarPU [6], we will use this algorithm in our study. However HEFT assigns tasks without considering the incoming communications of their children which can be costly while the clustering algorithms focus on the structure of the graph, and can avoid this kind of mistakes.

2) Clustering algorithms: Clustering algorithms are designed for an unbounded number of processors, they consider the structure of the DAG and group tasks into clusters. All the tasks which belong to one cluster are assigned to the same processor (i.e. there are no communication between them).

One of the most known clustering heuristic is the Dominant Sequence Clustering [17], it assigns to each task a priority corresponding to the sum of the longest path from a root node to this task and the longest path from this task to a sink node.Then the algorithm sort the tasks by decreasing priority and try to affect each task to the cluster which minimize its start time among the cluster containing its predecessors. However if a task can start earlier when we assign it to a new cluster instead of assigning it to an existing one, we create a new cluster for this task.

Figure 1. Different type of CLANS.

The idea of CLANS [18]–[20] is to decompose the DAG into a tree of clans representing a recursive expression of the relationship between tasks. For a graph $G = (V, E)$, a set of nodes $X \subset V$ is a clan of the graph G if and only if $\forall (x, y) \in X, \forall z \in V - X$, z is an ancestor (or successor) of x if and only if z an ancestor (resp. successor) of y. There are three types of clans: Independent, linear and primitive (see figure 1). Once the graph is decomposed into clans, the algorithm traverse the tree bottom up using the communication and execution times to determine if two clans have to be merged or not.

A theoretical comparison [21] between the algorithms DSC, MCP (Modified Critical Path, similar to DSC), MH, HU (lists algorithms) and CLANS have shown that CLANS is robust and efficient for a large and diverse set of graphs. All the others algorithms give very bad performances for graphs with a low ratio computation over communications (also called grain).

Figure 2. Convex and non convex group of tasks.

The convex clustering algorithm [22] produces an acyclic graph of clusters. Indeed a cluster C is convex if and only if $\forall$ $(x, z) \in C$ all task y, such that $x \prec y \prec z$, is in C (see figure 2). By this property, we know that there are no two way communications between two clusters. Moreover as the graph obtained by merging the tasks inside a cluster is a Direct Acyclic Graph, it allow us to run any scheduling algorithm on it.

To study the impact of communication times, we want to reduce the grain of the DAG by managing clusters as if they were large tasks. As the dataflow model implemented by XKAAPI doesn't allow cyclic dependencies between tasks, the only algorithm which fits our needs is the convex clustering. Hence we will implement this algorithm inside XKAAPI, and we are going to combine it with the HEFT algorithm to understand the impact of communication times on a scheduling.

III. Implementation of the convex cluster algorithm

IV. Analysis and reduction of the communication times

V. Conclusions and future work

References

[1] C. Nvidia, "C programming guide 3.2," *NVIDIA Corporation*, 2010.

[2] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[3] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[4] E. Lusk, S. Huss, B. Saphir, and M. Snir, "Mpi: A message-passing interface standard," *The International Journal of Supercomputer Applications and High performance Computing*, vol. 8, no. 3, pp. 159–416, 1994.

[5] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 2007, pp. 15–23.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[7] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *Euro-Par Parallel Processing*. Springer, 2011, pp. 555–566.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM Sigplan Notices*, vol. 33, no. 5, pp. 212–223, 1998.

[9] 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*, vol. 5. ACM, 1995, pp. 207–216.

[10] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2013*, 2013.

[11] J. D. Ullman, "*NP*-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[12] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[13] ——, "Bounds on Multiprocessing Timing Anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[14] M. R. Garey and R. L. Graham, "Bounds for multiprocessor scheduling with resource constraints," *SIAM Journal on Computing*, vol. 4, no. 2, pp. 187–200, 1975.

[15] J. Hwang, Y. Chow, F. Anger, and C. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal on Computing*, vol. 18, no. 2, pp. 244–257, 1989.

[16] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[17] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, 1994.

[18] A. Aubum and V. McLean, "Efficient exploitation of concurrency using graph decomposition," *Department of Computer Science and Engineering CSE-90-03, Auburn University*, 1990.

[19] C. McCreary, J. Thompson, H. Gill, T. Smith, and Y. Zhu, "Partitioning and scheduling using graph decomposition," *Department of Computer Science and Engineering CSE-93-06, Auburn University*, 1993.

[20] C. McCreary and A. Reed, "A graph parsing algorithm and implementation," *Department of Computer Science and Engineering CSE-93-04, Auburn University*, 1993.

[21] A. Khan, C. McCreary, and M. Jones, "A comparison of multiprocessor scheduling heuristics," in *ICPP International Conference on Parallel Processing*, vol. 2. IEEE, 1994, pp. 243–250.

[22] R. Lepère and D. Trystram, "A new clustering algorithm for large communication delays," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002*. IEEE, 2002, pp. 68–73.