Overview of Task Parallelism Targeting Distributed Memory Architectures

Bruno Gallina Apel, Nicolas Maillard
Instituto de Informtica - UFRGS
Grupo de Processamento Paralelo e Distribudo
91501-970 Porto Alegre - RS - Brasil
{bgapel,nicolas}@inf.ufrgs.br

Abstract

This work deals with the use of the explicit task model of parallelism at distributed memory environments. Task parallelism is largely used in shared memory hardware, but communication overheads bound it at the distributed memory scenario. The paper lists current, state-of-the-art challenges in transposing the current shared memory, task-based model to the distributed memory paradigm, as well as ideas in overcoming these challenges.

1. Introduction

One of the reasons that difficults the easy usage and massive propagation of parallel programming is the extensive variety of computer architectures, like multicores, GPGPUs, shared memory, distributed memory and more, without even citing the different programming paradigms like message passing, data, task, and instruction parallelism. The existence of an unified programming model would help to overcome this scenario. However, to create this model is not that simple, because every architecture/paradigm has its own peculiarities and characteristics.

It is desirable to write code lines without thinking in which type of hardware they will execute. So, the use of system threads is an alternative, but it's a non-portable and low-level solution. By contrast, the use of libraries to express parallelism is portable, but in most cases is necessary to re-write the algorithms and data structures due to particularities of the library [17].

Nowadays, the use of *tasks*, as units of work, is encouraged due to its facility to express non-structured parallelism and his possibility to dynamically create new tasks at runtime[1]. According to literature, OpenMP defines logical tasks as "a specific instance of executable code and it's data environment", whereas Intel Threading Building Blocks describes tasks as "*quanta* of computation mapped into physical threads." To this work, one task will be defined as a sequence of instructions that can execute in parallel with other tasks.

Trying to achieve an unified model of programming, this work looks for a way to migrate the task parallelism paradigm, already stablished and efficient in shared memory architectures, to distributed memory. This work relates to *Supporting Task Parallelism in MPI*, also developed at GPPD.

The rest of this paper is organized as follows. Section 2 discusses some definitions and challenges to effectively migrate the task parallelism paradigm to distributed memory environment. Section 3 presents tools which support task parallelism, or have similarities with it. Section 4 does an analysis of what tools or characteristics exposed in Section 3 could be used to achieve the objective of this work and summarizes our conclusions.

2. Concepts and Challenges

The use of tasks brings some implicit efficiency, because avoids CPU's under- or oversubscription [17]. It occurs when the software creates respectively less and more threads than available cores, creating an management overhead. This fact does not happen with the use of tasks because the huge ammount of tasks are scheduled inside just some threads.

Many types of schedulers can be used to manage tasks, but those which are based in *work stealing* technique have provably efficiency in terms of time and space over shared memory[3]. The work stealing algorithm defines that every idle processing node tries to "steal" work from other nodes.

Despite the already established use of tasks in shared memory architectures, when the paradigm changes to distributed memory some challenges must be overcome. There is an extra communication overhead to treat when using randomize work stealing with distributed memory. It occurs at unsuccessful tries to steal work, i.e. when a "victim" node does not have work.

Another problem to be solved is how to migrate tasks between nodes. They need to be serialized due to be transported by network.

Besides, the determinant point associated with communication overhead is the granularity of the tasks. If they are too small, there is an unnecessary overhead to communicate all of them [15]. Fact avoided by using large tasks, or various small tasks grouped as a bigger one.

To summarize, the challenges to overcome are:

- Which scheduler is better to use with tasks, on distributed memory architectures?
- How to pack and migrate task's data?
- How to determine the best grain size of a task?

3. Support to Task Parallelism

This Section will list some tools which support some level of task parallelism or could be used to obtain it on distributed memory.

Beginning with the the analysis of tools that work with shared memory, there are Cilk, OpenMP, Intel Threading Building Blocks and Google Go.

Cilk [4] is a language extension to C designed to simplify programming shared memory multiprocessor systems. It uses work stealing and concentrates on minimizing overheads that contribute to the work, even at the expense of overheads that contributes to the critical path. There is also a pre-compiler Cilk to C++ object-oriented language, named Cilk++ [12]. Both work properly at environments with heterogeneous processors [2], but they were not designed for environments that use distributed memory.

OpenMP [5] is one of the most widespread interfaces for communication on shared memory. It is based on the inclusion of compilation directives in order to parallelize existing code [16]. The scheduling in OpenMP could be static or dynamic, moreover the granularity could be controled by defining sizes of chunks to iterate. Initially focused on structured parallelism, i.e. loops, the OpenMP v3.0 includes the possibility to manipulate nested parallelism and dynamic creation of tasks.

Based on logical task concept, the Intel Threading Building Blocks (TBB) [13] maps these tasks in C++ objects. They also are more lighter than threads, due to have less aggregate data, like stacks and state of registers. During the initialization is created one task pool to each thread and these pools are managed by a work stealing scheduler. Also provides support to data decomposition, automatic granularity control and nested parallelism [14].

Google Go [9] is a programming language based in data communication to obtain parallelism on shared memory environments. It shares data through *channels*, an adopted concept to express communication. The channels are also applied to synchronize two or more goroutines. Goroutines are Google Go's equivalent to tasks, also lighter and cheaper. Presently, the goroutines scheduler is not as efficient as it could be, being this a future work to its developers.

Moving to the tools that run on both shared and distributed memory, there are MPI, BoostMPI, KAAPI and Charm++.

MPI [6] is the de facto standard on communication over message passing on distributed memory. Its extension, MPI-2, allows dynamic creation of process besides the use of intercommunicators to facilitate its grouping and management [7].

The Boost [10] libraries were created to extend and optimize the C++ standard libraries. Relevant to this work there are Boost MPI and Boost Serialization, among several options supported by Boost. The first one brings the standard MPI to an object oriented approach, facilitating its integration with existing code. And, the second one provides ways to desconstruct an arbitrary set of data, e.g. a C++ object, in one byte sequence. This operation are important because allows the data exchange between processor nodes.

The *Kernel for Adaptative, Asynchronous Parallel and Interactive Programming*, or just KAAPI [8] is a C++ library which enables the fine/medium grain multithread computation. Similar to Cilk, it also support dynamic data flow synchronization and has a work stealing based scheduler.

And finally, Charm++ [11] is an object oriented parallel language with asynchronous message passing. It defines concurrent objects called *chares*, and entry methods to these chares. An entry method is a special method which can be invoked by another chares, besides to be the responsible to define what to do with any message received. These methods returns *void* immediately after its invocation and guarantees that they will execute at some time.

Charm++ has a framework called Pack and UnPack (PUP), responsible to do the packaging and data serialization, for future transfer. It also offers the Runtime System (RTS) with the objective of remove from the developer the need to know any characteristic of the hardware, from the number of processor up to how they interact. The RTS is also responsible for the load balancing, migrating chares between processors, and dynamic relocation of resources.

3.1. Conclusion

After the study of the tools above described, the following can be concluded:

- Cilk, TBB and KAAPI use the algorithm of work stealing to schedule. Which yet is considered one of the most efficient schedulers to manage tasks.

- OpenMP and TBB have its own concept of explicit task. With the possibility of generic task definition, the model proposed by TBB is close to the ideal. However, both tools can only execute on shared memory environment.
- Google Go and Charm++ creates concepts similar to explicit task, which are goroutine and entry methods, respectively. Go runs only over shared memory, Charm++ can be executed in both memory models.
- KAAPI and Charm++ have similarities, but the distributed memory version of KAAPI is not stable. Besides that Charm++ provides ways to serialize and migrate objects.
- MPI and Boost libraries do not provides ways of scheduling or managing the granularity, making these actions an user's responsibilitie.

Therefore, as a future work is proposed the creation and addition of work stealing scheduler and granularity control to Charm++ to achieve similar performance that were obtained in TBB. The development of these mechanisms will be the objective of the author's master thesis.

References

[1] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404 –418, mar. 2009.

[2] 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, pages 13–21, New York, NY, USA, 2000. ACM.

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

[4] 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, 1995. ACM.

[5] O. A. R. Board. Openmp application program interface - version 3.0. Technical report, 5 2008.

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

[7] M. P. I. Forum. MPI: A Message-Passing Interface Standard ver. 2.1. Technical report, Message Passing Interface Forum, Junho 2008.

[8] 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.

[9] Google Go Project Team. The go programming language specification. Technical report, Google, 2011.

[10] A. Gurtovoy and D. Abrahams. The boost c++ metaprogramming library. Technical report, MetaCommunications, 2002.

[11] L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[12] C. E. Leiserson. The cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

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

[14] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[15] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011. ACM.

[16] M. Sato. Openmp: parallel programming api for shared memory multiprocessors and on-chip multiprocessors. In *System Synthesis, 2002. 15th International Symposium on*, pages 109 – 111, 2002.

[17] T. Willhalm and N. Popovici. Putting intel®threading building blocks to work. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 3–4, New York, NY, USA, 2008. ACM.