Online Mapping of MPI-2 Tasks to Processes and Threads

Joao Vicente F. Lima, Nicolas Maillard
Instituto de Informática – Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{jvlima, nicolas}@inf.ufrgs.br

Abstract

In the last years, distributed platforms become largely used on HPC, and most of this architectures have different levels of parallelism. Hence, one of the key design stages in parallel programming is task mapping, which attempts to maximise processor utilisation and minimise communication cost. However, this depends on a programming environment with efficient mapping scheme. This work implements a library (libSpawn) that adds an efficient mapping scheme for MPI dynamic programs in order to minimise communication and task creation costs. It maps tasks to either processes or threads based on three parameters: available architecture cores, load and resources of the operating system. The results obtained between processes and libSpawn show significant performance improvement on a synthetic benchmark.

1. Introduction

Most of today's high-performance architectures, as clusters of multi-core processors and interconnected clusters, have different levels of parallelism. Programming such machines is not trivial since architecture issues can not be ignored, such as heterogeneity and communication cost. One of the key design stages in parallel programming is task mapping, which attempts to maximise processor utilisation and minimise communication cost. Hence, an appropriate mapping scheme can significantly improve performance and load balancing over processors.

A static mapping scheme can be appropriate for programs with regular computational load known at compile time, but this becomes more difficult for programs whose workload changes over time. This irregular behaviour may cause load imbalances and irregular communication patterns that often makes an efficient parallel implementation difficult. Adaptive and hierarchical algorithms have been designed to unfold parallelism of such applications at runtime through programming techniques for dynamic programs, as master-slave and divide-and-conquer. Many projects have proposed programming interfaces or languages with dynamic adaptation; however, they are restricted to specific architectures. The *Message-Passing Interface* (MPI) is a recognised standard for HPC in distributed memory environment, and version MPI-2 has added new features as remote memory access, parallel I/O, multithreaded programming, and dynamic process creation. Yet, MPI implementations do not provide hybrid task mapping between processes and threads.

This paper presents a library to MPI (libSpawn) that implements a scheme to map tasks in order to minimise communications and task creation costs. It modifies `MPI::Comm::Spawn` to map tasks as either processes or threads based on three runtime parameters: cores of architecture, load and resources of the operating system. In addition, this library implements message passing among threads in same process. We evaluate libSpawn with a dynamic MPI program, Fibonacci. Our experiments demonstrate that the mapping scheme offers significant performance improvement at task creation and communication.

The remainder of this paper is structured as follows. In section 2 we present the context on dynamic parallel programs, and section 3 describes the state of art on tools that support dynamic task creation. In the section 4, our implementation of libSpawn is detailed and section 5 shows the results obtained from our experiments. Finally, section 6 discusses some concluding remarks and plans for the future.

2. Parallel and Dynamic Programs

Foster [5] proposes a methodology for designing parallel programs where machine-specific aspects are delayed in the design process. This methodology can be resumed in three distinct stages:

- **Partitioning**: this stage divides an initial computation into smaller parts, where some or all of which may be executed in parallel.

- **Mapping**: this stage assigns each smaller part to a processor.
- **Communication**: this stage defines the coordination required between processors.

The first stage in parallel program design defines units of computation, which is called *task*. A task is a block of instructions that may contain some communications when a task uses data produced by other tasks. An abstraction to express such dependencies among tasks and their relative order of execution can be represented as a directed acyclic graph (DAG).

The mapping stage attempts to maximise processor utilisation and minimise communication costs. Mapping can be specified statically or at runtime by load-balancing algorithms. Static mapping can be efficient for programs that computational load and total number of tasks are known at compile time. However, this becomes more difficult for programs whose the computational load and total number of tasks dynamically change over time. Irregular algorithms are an example of such programs that performance depends on specific algorithm properties concerning the behaviour of data structures, variations of computational effort, and irregular dependencies between computations. The characteristics of irregular algorithms may cause load imbalances and irregular communication patterns, which often make an efficient parallel implementation difficult.

Adaptive and hierarchical algorithms have been designed to reduce computation time and to unfold parallelism at runtime, which are also called dynamic programs. Dynamic programs can adapt to the underlying architecture through dynamic task creation and task mapping on demand. Two programming techniques are commonly used on dynamic programs to unfold parallelism at runtime: master-slave and divide-and-conquer.

There are many parallel programming environments that offer support for dynamic programs, as presented in the next section.

3. Related Work: Parallel Programming Environments

Many projects have proposed other programming interfaces or languages with dynamic creation support. Cilk [6] is a language for multithreaded parallel programs on SMP architectures. It extends C language with addition of three keywords to indicate parallelism and synchronisation: `cilk`, `spawn`, and `sync`. Cilk parallelism is created when the keyword `spawn` precedes the invocation of a function; besides, the semantics of `spawn` differs from a standard C function call. Through Cilk `spawn` keyword, the parent can continue to execute in parallel with the child as a new task. Intel©*Threading Building Blocks* (TBB) [12] is a C++ runtime library without compiler support or language extensions. The parallelism can be expressed in terms of tasks, which are represented as instances of a task class. TBB also provides concurrent container classes through generic interfaces.

A number of other programming environments exist that support distributed architectures. Satin [14] is a divide and conquer system based on Java for grid platforms. It implements a efficient load balancing algorithm for clustered, wide-area platforms. KAAPI [7] is a runtime system for scheduling irregular macro data flow programs on clusters of multi-processors. Its programming model includes a global address space called *global memory* and allows express data dependencies between tasks that access objects in the global memory. *Adaptive MPI* (AMPI) [9] is an adaptive implementation of MPI standard with load balancing that introduces the concept of *virtual processors* (VPs). But AMPI does not support all MPI features, like dynamic process creation.

The original MPI-1 standard [3] required the number of processes to be specified at startup time as a parameter. The more recent MPI-2 standard supports dynamic task creation, through its dynamic process management, and multithreaded programming [4]. MPI-2 specifies two sets of functions for dynamic process management: `MPI_Comm_spawn` and related, and client/server functions. These sets provide a mechanism to establish communication between the newly created processes and the existing MPI application. Other approaches have been proposed in order to improve the dynamic process creation in MPI. Cera et al. [2] proposed a centralised on-line scheduler for load balancing among the available resources, and Pezzi et al. [11] proposed a scheduling algorithm, called *Hierarchical Work Stealing*, suited to a dynamic N-Queens implementation.

Furthermore, the standard has specified the interaction between MPI and user-created threads. A MPI program may initialise with `MPI_Init_thread` function in order to specify the desired level of thread support. A fully thread-compliant MPI implementation will support `MPI_THREAD_MULTIPLE` level of thread safety, where multiple threads may make MPI calls at any time. However, MPI do not identify threads as members of groups or communicators. The standard also says that is user's responsibility to prevent races when threads in the same program post conflicting MPI calls. Other projects have evaluated thread-safety issues in MPI implementations [1, 8, 13].

A dynamic MPI program can use both approaches, as processes and threads, for performance improvements at task creation and communication. It can spawn tasks as POSIX threads, for example, in order to increase local resource usage. When local system becomes overloaded, new processes should be spawn to remote processors that may

be idle. [10] evaluated a similar approach through a modified master/slave scheme, described early, whose slaves deploy either OpenMP threads or MPI processes according to available processors. However, the design of such applications is not trivial because of the various aspects that must be considered as task mapping and safe communication.

4. libSpawn: Task Mapping on MPI-2

libSpawn is a C++ library that implements a scheme to map tasks between processes or threads in order to minimise communications and task creation costs. The `MPI::Comm::Spawn` method is changed to map tasks based on three runtime parameters: cores of architecture, load and resources of the operating system. Furthermore, libSpawn implements message passing among threads in same process because MPI lacks to identify threads as members of groups or communicators, as explained early.

The `Spawn` function evaluates three runtime parameters:

1. **architecture cores available** - this value is configured at process startup;
2. **local system load** - one dedicated thread monitors system average load at runtime;
3. **local system resources** - threads creation depends on available system resources; indeed, a program should be aware of its resource usage to avoid runtime errors.

libSpawn takes some steps in task mapping, as illustrated in Figure 1. Each process creates a monitoring thread that reads system average load periodically. A call to `Spawn` function begins the evaluation of the runtime parameters. First, the number of running threads is compared against the architecture cores available **(2)**. If the number of threads is smaller than the number of cores, the new tasks are mapped to threads. On the other hand, when this test fails, the other parameters are evaluated. This approach guarantees efficient mapping in first tasks by skipping other parameters.

The second parameter **(3)** considers the local system load at runtime. When the number of running threads are equal to cores of architecture, some threads can be blocked on a system call or a communicating operation, for example. The system average load describes how many tasks are ready for execution [2]. Notice that this value includes tasks from either operating system or another user, which offers a metric of architecture cores usage. When the library detects overload on the cores, libSpawn maps the new tasks to processes through original MPI-2 `Spawn` method. Thus, the library `Spawn` function assigns the work of these tasks to remote nodes, which may be idle.

Finally, the library evaluates the system resources available for processes on the current node **(4)**. The operating system can set limits on the resources obtained by users.

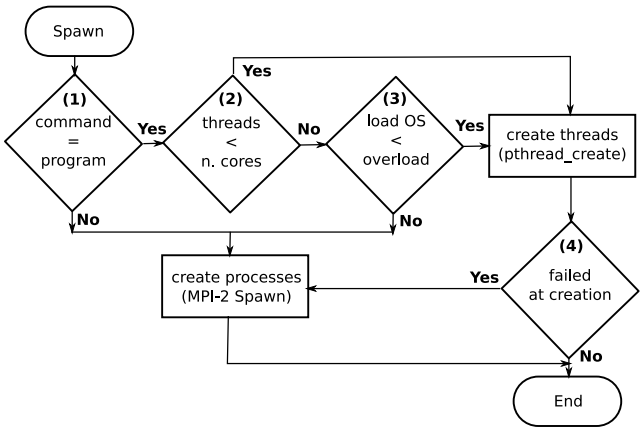

Figure 1. The flowchart of `Spawn` function with each step of libSpawn decision.

Theses resources can be either hardware, like memory and storage usage, or software. The software resources are specified at operating system implementation and depends on specific system politics and configurations. For example, the number of threads and open files by process are software constraints on UNIX systems. In libSpawn, an resource allocation is made for each thread or memory allocation. For that reason, the library must tests each allocation in order to detect resource failure. If some error occurs, the tasks are mapped to processes on a remote node.

5. Experiments

The experiments compare timing results of a dynamic MPI program linked to libSpawn with standard MPI-2 process creation. These experiments were performed in 30 nodes from the French Grid'5000 testbed with two Dual Core processors (four cores per node) and Gigabit Ethernet network. The MPI-2 implementation used was MPICH2 version 1.0.8p1 that has support for dynamic process creation and multithreaded MPI calls. Each measure is the average of 20 executions with its standard deviation.

The input of Fibonacci sequence was 20, which generated 13.530 tasks. Figure 2 compares running times with processes and libSpawn. The library shows significant gains in all configurations compared with MPI default process creation This performance improvement has achived from two main reasons:

- **Dynamic task creation** - mapping tasks to threads provides significant performance gain as local thread creation is very fast when compared with either local or remote process creation;

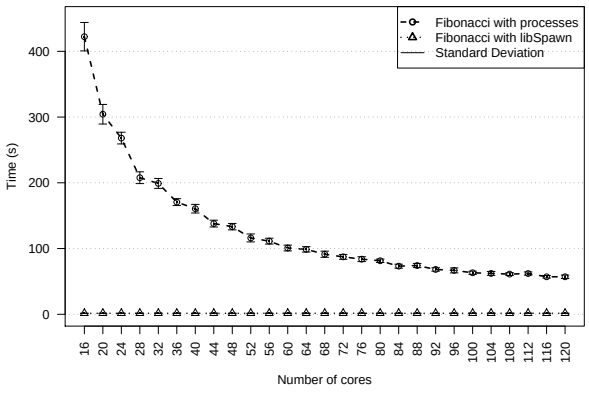

Figure 2. Results obtained from Fibonacci of 20.

- **Communication** - communications among tasks as threads are performed through shared-memory operations, which are more efficient than remote references with sockets.

6. Conclusion and Future Work

This paper presented a library to MPI (libSpawn) that implements a scheme mapping to map tasks between processes and threads in order to minimise communications and task creation costs. It modified `MPI::Comm::Spawn` call to map tasks as either processes or threads based on on three runtime parameters: available architecture cores, local system load and local system resources. This library also controls message communications among threads in same process.

We evaluated libSpawn and the standard process creation of MPI-2 through a synthetic benchmarks, Fibonacci. The experiments exhibited gains in almost all configurations compared with process. This performance improvement is due to the mapping scheme proposed that offers significant reduction time at task creation and communication. The solution presented also provides a scheme mapping transparent to programmer. Besides, it allows safe message communications among threads, which was not possible with standard MPI calls.

Future work in libSpawn would achive more MPI features like collective communication and group management. In addition, we plan improve our experiments through measures on different architectures, as cluster of clusters, and more dynamic programs.

References

[1] P. Balaji, D. Buntinas, D. Goodwell, W. Gropp, and R. Thakur. Toward efficient support for multithreaded mpi communication. In *Proc. of The 15th EuroPVM/MPI 2008*, pages 120–129, Dublin, IRL, 2008. Springer-Verlag.

[2] M. C. Cera, G. P. Pezzi, E. N. Mathias, N. Maillard, and P. O. A. Navaux. Improving the Dynamic Creation of Processes in MPI-2. In *Proc. of The 13h EuroPVM/MPI 2006*, volume 4192/2006, pages 247–255, Bonn, DEU, Sept. 2006. Springer Berlin / Heidelberg.

[3] M. P. I. Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, University of Tennessee, Knoxville, USA, April 1995.

[4] M. P. I. Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report CDA-9115428, University of Tennessee, Knoxville, USA, July 1997.

[5] I. Foster. *Designing and Building Paralllel Programs*. Addison-Wesley, 1995. Disponível em: http://www.mcs.anl.gov/dbpp. Acesso em: fev. 2009.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, May 1998.

[7] 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 PASCO 2007*, London, CAN, 2007. ACM.

[8] W. Gropp and R. Thakur. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, Sept. 2007.

[9] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proc. of The 16th LCPC 2003*, volume 2958/2004, pages 306–322, College Station, USA, Oct. 2004. Springer Berlin / Heidelberg.

[10] C. Leopold and M. Süß. Observations on mpi-2 support for hybrid master/slave applications in dynamic and heterogeneous environments. In *Proc. of The 13th EuroPVM/MPI 2006*, volume 4192, pages 285–292, Bonn, DEU, 2006. Springer.

[11] G. P. Pezzi, M. C. Cera, E. Mathias, N. Maillard, and P. O. A. Navaux. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. In *Proc. of The 19th SBAC-PAD 2007*, pages 247–254, Gramado, BRA, 2007. SBC.

[12] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly & Associates, Inc., Sebastopol, USA, 2007.

[13] R. Thakur and W. Gropp. Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. In *Proc. of The 14h EuroPVM/MPI 2007*, volume 4757, pages 46–55, Paris, FRA, Sept. 2007. Springer Berlin / Heidelberg.

[14] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Proc. of The 6th International Euro-Par Conference*, volume 1900, pages 690–699, Munich, DEU, August 2000. Springer.