Scheduling parallel programs on non-uniform memory architectures

Gerson G. H. Cavalheiro[†] Mathias Doreille François Galilée
Thierry Gautier Jean-Louis Roch

Projet Apache[‡]
Laboratoire de Modélisation et Calcul
100 rue des Mathématiques BP 53
38041 Grenoble Cedex 9, France
http://www-apache.imag.fr

Abstract. This paper presents a hierarchical scheduling runtime to support efficiently the execution of irregular application on a distributed machine. A high level scheduling provides the distribution of load between the nodes of the machine, based on a work-steal mechanism. A low level scheduling implements a multithreading scheme to hide communication latency and exploit the computation power of each node.

1 Introduction

Many irregular applications present a high degree of parallelism, unpredictable at compile time. To obtain efficient executions on a distributed architecture, the scheduling has to bound this parallelism to the one available on the architecture. Two levels of scheduling are usually considered:

- A high level scheduling distributes the load among the nodes of the parallel architecture. One of the most commonly used technique is greedy: when a node becomes idle, it picks some ready tasks if any from a list that may be distributed among nodes. On uniform memory architectures, this technique leads to provable performances for executions that have a small critical path [10, 5]. Various practical implementations on such architectures have given experimentally good performances for a wide range of applications [14, 5].
- A low level scheduling, on any particular node, overlaps part of system overheads by effective computation. Here, the most commonly used technique is multithreading. In theoretical works [13, 11], this technique, which is often referred as "parallel slackness" [13], enables to provide near-optimal emulation of a global memory on a network of identical processors [11]. On a practical point of view [8], various experimentations show that a certain amount of local multithreading enables to hide part of communication delays. On synchronous architectures, the number of threads is related to the hardware characteristics (e.g. TERA machine). Besides, on an asynchronous one, it is also related to the application itself [3].

A scheduling algorithm based on those two levels of scheduling has been proposed and evaluated on a theoretical machine model in [12]. In this paper, we present the implementation of a similar scheduling algorithm but that requires no global synchronization. It takes benefit of those two levels of scheduling on a non-uniform access architecture. The resulting algorithm is general and extends previous works (in particular related to Cilk) in two ways: first, there is no restriction on the synchronization structure in the execution; second, it enables to schedule a program on a hierarchical memory architecture, integrating a multithreading technique to hide part of memory latency.

The description is based on the default scheduling strategy implemented in the Athapascan-1 parallel programming interface [9]. Some experimental results are provided on a simple applications programmed in Athapascan using this strategy.

2 Application modelling and input of the scheduling

At runtime, we assume that the execution of a parallel application generates tasks, submitted to precedence relations. Execution of a task is assumed to be non-blocking but can generate new tasks, recursively for instance. Typically, a precedence between two tasks corresponds to a write-read data dependency. We represent the non completed tasks and their precedences by a graph which is unfolded on the fly at runtime; it evolves depending on tasks creations and completions. Then, at any instant of the execution, this graph represents the state of the parallel program.

[†] CAPES–COFECUB Brazilian scholarship
[‡] Supported by CNRS, INPG, INRIA et UJF

Example. Figure 1 gives an example of a code written in a pseudo-language that enables dynamic task creations (`FORK` statement) and where tasks precedences are deduced from input-output dependencies; an example of a graph describing a possible state for an execution is given on the right side of the figure.

```
void Factorization_LU( matrix<TYPE> A ) : INPUT A; OUTPUT A;
{
  for( int k = 0 ; k < A.row_dim() ; k++ ) {
    FORK Block_Factorization_LU( A(k,k) );

    for( int i = k+1 ; i < A.col_dim() ; i++ )
      FORK Block_Times_Inverse_U( A(i,k), A(k,k) );

    for( int j = k+1 ; j < A.row_dim() ; j++ )
      FORK Block_Times_Inverse_L( A(k,j), A(k,k) );

    for( i = k+1 ; i < A.col_dim() ; i++ )
      for( j = k+1 ; j < A.row_dim() ; j++ )
        FORK Minus_Times( A(i,j), A(i,k), A(k,j) );
  }
}
```

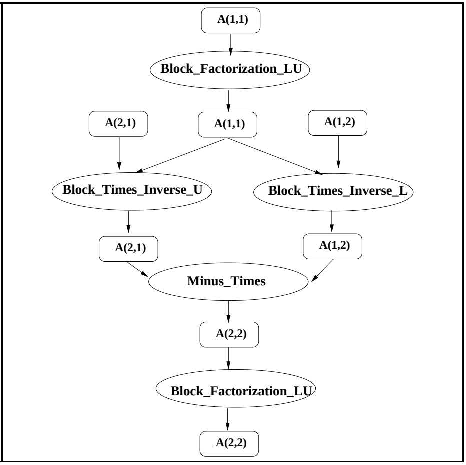

Fig. 1. A pseudo-parallel code for a Gauss LU-factorization. The graph on the right corresponds to the state of an execution for a 2×2 input block-matrix, after execution of the task `Factorization_LU`. In this graph, ellipses represent tasks that are not yet completed and boxes data. Arrows correspond to data dependencies and define synchronizations between tasks. Note that tasks computing blocks (e.g. `Block_Factorization_LU`) may generate other tasks (using `FORK` statements in their body).

More generally, a task is submitted to synchronization conditions (precedences), called *transitions*. A task is said *waiting* while all its input transitions are not completed; then it is said *ready*. In order to respect synchronization conditions, we assume that a task cannot be scheduled (on a processor) before being ready. Thus, the scheduling can only execute ready tasks on the processors of the architecture.

We assume that a distributed runtime manages tasks creation and state modifications with a constant bounded overhead [9]. When such a modification is performed on a processor (for instance a FORK statement or a task completion), the runtime calls a function implemented by the scheduling algorithm, called the scheduler. This scheduler is itself a distributed algorithm [7]. Taking benefit of the availability of several processors in the architecture, its objective is to optimize the completion time of the whole execution. Furthermore, due to the non-uniformity of the distributed architecture, it has to hide latency of remote memory access.

3 A two levels scheduling algorithm

We first present the scheduling of tasks on the various processors on the architecture, which is greedy and based on work-stealing. Then, at a second level, the use of threads on each node of the architecture enables to hide communication latency related to non-local access. An overview of this hierarchical scheduling with two levels is presented in figure 2; its components are described in the next paragraphs.

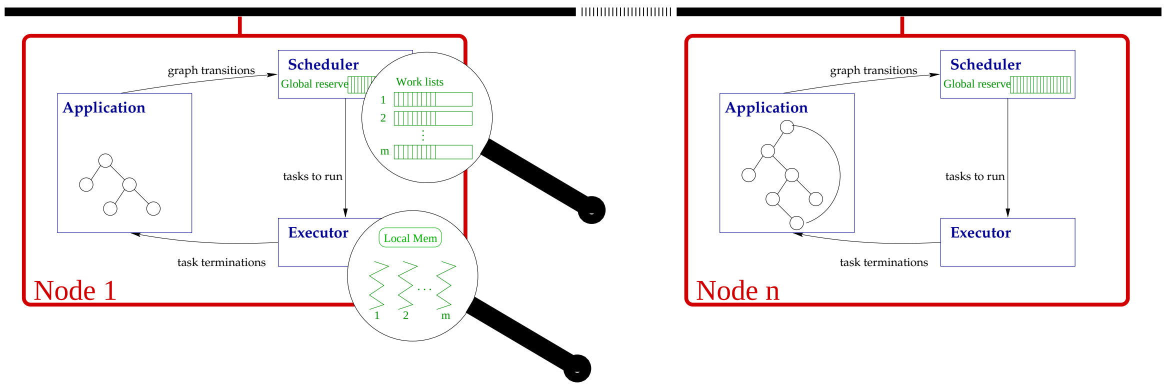

Fig. 2. Runtime scheduling has two levels: at a high level, a global scheduler manages the load distribution between nodes while, at a lower level, a local multithreaded kernel optimizes the use of each processor. The zoomed regions show details of the lower level scheduling.

3.1 High level scheduling: work-stealing

Classical scheduling algorithms to minimize completion time on a given number of processors are based on a greedy strategy [10]: when a processor becomes idle, it steals a ready task available on a non-idle processor in order to execute it. Such a strategy achieves provable performances but, in order to prevent for memory exhaustion, some precautions have to be taken [5].

In the following, we assume that a total order on the tasks is known by the scheduler that would result in a correct sequential execution (lexicographic order for instance [9]). Then, scheduling tasks with respect to this order allows to bound the memory space with respect to the one required by such a sequential execution [4]. Such a strategy has been implemented in the Cilk parallel language for dynamic series-parallel graphs; it has lead to very good experimental performances [6] on parallel architectures with uniform memory access.

Similarly to Cilk, the high level scheduling implements a work-stealing algorithm. On each node of the architecture, a list of ready tasks is managed. Each node uses it to store tasks that become ready. When a node completes a task, it picks another ready task from its local list to execute it. When the size of its list reaches a minimal threshold, a thief procedure is trigged: a *victim* node is randomly selected and a task is *stolen* from its reserve list. Moreover, if the size of the victim reserve list is null, the stolen message is forwarded to another victim.

In order to optimize the memory space, the management of the reserve on each node is not symmetric. Each node adds new ready tasks on the head of its reserve. Then, when it completes a task, it takes a new one also from the head, privileging a depth traversal of the task graph. However, tasks are stolen by other processors from the tail of the list. Especially for recursive programs, this strategy enables to bound the memory space required on each processor [5] while not increasing the critical path.

Moreover, when a task t is stolen by an idle node P_I on a non-idle one P_S, all successors of t located on P_S will wait for the completion of t on P_I to become ready. In order to avoid extra communications related to t completion, it thus can be worth fully to export not only t to P_I but also some of its successors (at least all successors of t that have no other predecessor on P_S except t). This strategy tries to reduce the number of three kinds of communications:

- Tasks state management: since the graph is distributed, immediate successors of t that are waiting for its completion will be locally updated.
- Data transfers: data updated by t that are required by its successors already in P_I can be accessed directly on the local memory cache of P_I with no communication.
- Work-steal messages: it can be expected that when t will be completed on P_I, one of its successor will be ready on P_I.

3.2 Low level scheduling: multithreading

In order to overlap communication latencies and other systems overheads (I/O, swap, etc.) by the effective computation of the application, a pool of m threads is used on each node. Each of those threads executes an infinite loop: at each iteration, it gets a task from the reserve and executes it until completion.

On each node, this reserve list is seen by the high level scheduling. However, in order to avoid contentions due to unnecessary mutex [1, 2], this reserve is splited in m sub-lists, one for each thread of the node (see details in the figure 2). Preserving the depth first strategy of the high level work-stealing scheduling, each of the m threads handles its sub-list in a LIFO manner. When a thread tries to get a task from an empty list, it first looks for work in the tail of the sub-lists of the other threads running on the same node. The work-stealing level may be activated to thief tasks from another node if all sub-lists are empty.

Multithreading on a SMP architecture. In the case of a symmetric multiprocessors node (SMP), the use of several local threads also attempts to exploit the various processors availables on the node.

4 Experimentation

In this section we present briefly some performances results obtained for a Mandelbrot computation. The Mandelbrot algorithm implemented is recursive: it takes in input a start region and a threshold. While the threshold is not reached, the region is splitted in four sub-regions; otherwise, the sub-region is sequentially computed. For each experiment, 314 tasks were generated: among those, 256 tasks correspond to the sequential computation of a sub-region (with size lesser than the threshold) each of unknown computational cost; the others compute the recursive splitting.

Behaviour of the high level scheduling The figure 3 shows a distribution of work obtained after the execution of a Mandelbrot computation. The threshold for halting the recursive splitting is fixed: it corresponds to the size of the smallest squares in the image on the right. Note that in this experiment, 256 additional tasks are executed by the node 0 (the dark-grey one in figure 3); it can be noted that this processor among the five is the one that computes the smallest surface. Besides, as it can be seen on the right image, only few fine grain tasks are exported to other processors (except at the end of the execution). This is related to the previously described implementation of the high level scheduling; it privileges local depth first execution. As it can be seen on the right image, this enables to bound the space of the distributed execution and to avoid unnecessary steal messages.

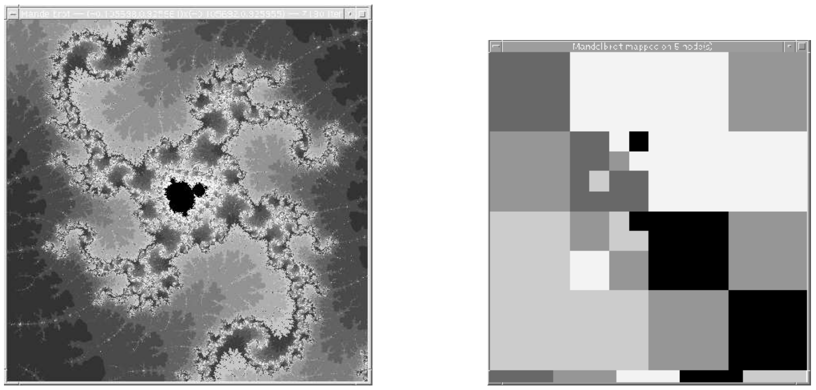

Fig. 3. A Mandelbrot plotting and the related distribution of the work on a 5 node parallel machine. The algorithm is recursive: it cuts the original region in four sub-regions until a fixed threshold (which corresponds to the size of the smallest squares on the right). On the right image, each color corresponds to the specific node that has computed the correspondig area in the Mandelbrot set on the left; the line on the bottom of this image depicts the five colors, each one corresponding to a node. The processor that starts the computation and performs the display of the Mandelbrot set is the right-most one (indexed 0 and coloured dark-grey).

Performances of the two levels scheduling To evaluate the speed-up, plotting tasks have been suppressed; only the 314 tasks corresponding to recursive splitting and computation are executed. Experiments have been performed on two different architectures: a SMP machine with 4 Pentium Pro (Solaris 2.6, POSIX) (performances are presented in the table on the left of figure 4) and a IBM-SP (IBM RS-6000/370, AIX 4.2, IBM-MPI, kernel threads) using from 1 to 6 monoprocessor nodes (performances are presented in the graphic on the right of figure 4). On both machines, speed-up (*sp*) has been computed by: $sp = T_1^1 / T_m^p$, where T_m^p is the time obtained for the parallel execution on a p node parallel machine using m threads by node; then T_1^1 is the time for the execution on an one node machine with only one execution thread.

Observing the given measures on the SMP, we see that local multithreading enables to overlap the system overhead: a speed-up 3.6 is obtained which is near to the number of available processors (4). Moreover, the basic time T_1^1 is very close to the cost of a pure sequential execution (with no parallelism overhead) on this architecture.

# threads	T_m^1	*sp*
1	23.80	1.0
2	12.04	1.9
4	7.71	3.0
6	7.22	3.3
8	7.02	3.3
10	6.68	3.6
12	6.74	3.5
14	6.73	3.5
16	6.74	3.5

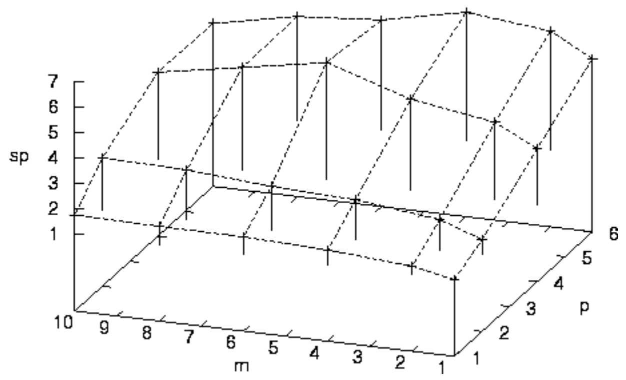

Fig. 4. Performance measurements for Mandelbrot on 2 architectures. In left side, a table presents the time (in seconds) and the speed-ups obtained on a 4-processors SMP architecture. The graph in the right side presents the speed-ups on a IBM-SP, using from 1 to 6 nodes.

On the IBM-SP also, the system overheads (including the one related to the high level work-stealing when $p \geq 2$) is partially overlapped by using local multithreading. In this experiment, it can be seen that the best value for m depends on p. Besides, a good speed-up (about p with the best value for m) is obtained related to T_1^1. However, a large overhead is introduced here with respect to a pure sequential computation on one node (a factor of about 3), mainly due to the polling of communications; this overhead is recovered when using several computational ressources ($p \geq 4$). We are now working to decrease this basic overhead on this machine.

5 Conclusion

In this paper we have presented a hierarchical scheduling runtime for irregular applications on a distributed non-uniform architecture. The higher level implements a greedy scheduling algorithm to distribute the load between the nodes of the architecture. The lower level uses multithreading to overlap systems overheads by effective computation and to exploit SMP architectures.

Experimentations were obtained by the execution of an application implemented in Athapascan-1 [9], a parallel programming interface that provides an implicit scheduling. Athapascan-1 is implemented as a C++ library on top of a multithreaded runtime, named Athapascan-0, available on various distributed architectures (`http://www-apache.imag.fr`). In order to deal with data and control flow at a grain defined by the user (macro-data flow), parallelism is expressed through asynchronous remote procedure calls, denoted *tasks*, that communicate and are synchronized only via access to a shared memory by read and write operations.

References

1. S. Aditya, Arvind, L. Augustsson, J-W. Maessen, and R. Nikhil. Semantics of ph: A parallel dialect of haskell. Computation Structures Group Memo 377-1, June 1995.
2. Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxon. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of X SPAA*, Puerto Vallarta, Mexico, June 1998.
3. P.-E. Bernard, B. Plateau, and D. Trystram. Using threads for developing applications: Molecular dynamics as a case study. In Trobec, editor, *Parallel Numerics 96*, pages 3–16, Gozd Martuljek, Slovenia, Septembre 1996.
4. Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space efficient scheduling of Parallelism with Synchronization Variables. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures.* ACM Press, June 1997.
5. Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.
6. Robert D. Blumofe and Dionisios Papadopoulos. Semantics of ph: A parallel dialect of haskell. TR-98-13, May. 1998.
7. Gerson G. H. Cavalheiro, Yves Denneulin, and Jean-Louis Roch. A General Modular Specification for Distributed Schedulers. In LNCS 980 Springer Verlag, editor, *Proceedings of Europar'98*, Southampton, England, Sept. 1998.
8. I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *IEEE Journal of Parallel and Distributed Computing*, 1997.
9. François Galilée, Gerson G. H. Cavalheiro, Jean-Louis Roch, and Mathias Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *PACT*, Paris, France, October 1998.
10. R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–426, 1969.
11. R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. *Algorithmica*, 16:517–542, 1996.
12. J.-C. Konig and J.-L. Roch. Machines virtuelles et techniques d'ordonnancement. In *Proceedings of ICARE'97*, Aussois, France, Dec. 1997. école du CNRS.
13. L. Valiant. A bridging model for parallel computation. *Communication ACM*, 33:103–111, 1990.
14. M. H. Willebeek-Le-Mair and P. Reeves. Strategies for dynamic load-balancing on higly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.