Fine Grain Distributed Implementation of a Dataflow Language with Provable Performances

Thierry Gautier, Jean-Louis Roch, Frédéric Wagner

MOAIS Project, LIG Lab., INRIA-CNRS, Universités de Grenoble, France
{thierry.gautier,jean-louis.roch,frederic.wagner}@imag.fr

Abstract. Efficient execution of multithreaded iterative numerical computations requires to carefully take into account data dependencies. This paper presents an original way to express and schedule general dataflow multithreaded computations. We propose a distributed dataflow stack implementation which efficiently supports work stealing and achieves provable performances on heterogeneous grids. It exhibits properties such as non-blocking local stack accesses and generation at runtime of optimized one-sided data communications.
Keywords: *dataflow, distributed stack, work-stealing, work depth model.*

1 Introduction

Multithreaded languages have been proposed as a general approach to model dynamic, unstructured parallelism. They include data parallel ones – e.g. NESL [5] – , data flow – ID [7] –, macro dataflow – Athapascan [10] , Jade [15] – languages with fork-join based constructs – Cilk [6] – or with additional synchronization primitives – Hood [2], EARTH [11] –. Efficient execution of a multithreaded computation on a parallel computer relies on the schedule of the threads among the processors. In the work stealing scheduling [2,1], when becoming idle, a processor steals a ready task (the oldest one) on a randomly chosen victim processor. Usual implementations of work stealing are based on stacks to store, locally on each processor, the tasks still to complete.

Such scheduling has been proven to be efficient for *fully-strict* multithreaded computations [6,8] while requiring a bounded memory space with respect to a depth first sequential execution [14]. However, some numerical simulations generate non serie-parallel data dependencies between tasks; for instance, iterative finite differences computations have a diamond dag dependency structure. Such a structure cannot be efficiently expressed in term of neither fully-strict nor strict multithreaded computation without adding artificial synchronizations which may limit drastically the effective degree of parallelism. The Athapascan [10] parallel language enables to describe such recursive multithreaded computations with non serie-parallel data dependencies as described in Section 2.

In this paper, we propose an original extension named DDS (Section 3) of the stack management in order to handle programs which data dependencies do not fit the class of strict multithreaded computations. The key point consists in linking one-sided write-read data dependencies in the stack to ensure constant time non-blocking stack operations. Moreover, on distributed architectures, data links between stacks are used to implement write-read dependencies as one-sided efficient communications. Those properties enable DDS to implement macrodataflow languages such as Athapascan with provable performances (Section 4). Section 5 reports experimental performances on classical benchmarks on both cluster and grid architectures up to a thousand processors confirming the theoretical performances.

2 Model for recursive dataflow computations

This section describes the basic set of instructions (abstract machine) used to express parallel execution as a dynamic data flow graph. It is based on Athapascan which models a parallel computation from three concepts: tasks, shared objects and access specifications [10]. Following Jade [15], Athapascan extends Cilk [9] to take into account data dependencies; however, while Jade is restricted to iterative computations, Athapascan includes nested recursive parallelism to take benefit from the work stealing.

The programming model. A task represents a non-blocking sequence of instructions: Like in ordinary functional programming languages, a task is the execution of a function that is strict in all arguments (no side effect) and makes all result values available upon termination. Tasks may declare new tasks. Synchronization between tasks is performed through the use of write-once shared objects denoted `Data`. Each task has an access specification that declares how it (and its child tasks) will read and write individual shared objects: the type `Data::Read` (resp. `Data::Write`) specifies a read (resp. write) access to the effective parameter. To create a task, a block of memory called a closure is first allocated using `AllocateClosure` (Figure 1). Then the effective parameters of the task are pushed to the closure, either immediate values or shared objects. For each shared parameter, the access specification is given: either *read* (`push::Read`) or *write* (`push::Write`). An immediate value parameter is copied using `push::Value`. Finally, the `commit` instruction completes the description of the task.

Synchronization between tasks is only related to access specification. The semantic is lexicographic: statements are lexicographically ordered by ';'. In other words, any read of a parameter with a `Read` access specification sees the last write according to a lexicographic order called *reference order*. Figure 1 is an example of code using Athapascan for the folk recursive computation of Fibonacci numbers: the tasks `Sum` reads `a`, `b` and writes `r`.

Spawn tree and Reference order. Recursive description of tasks is represented by a tree $\mathcal{T}$, called *spawn tree*, whose root is the main task. A node n in $\mathcal{T}$ corresponds to a task t and the successor nodes of n to the child tasks of t. Due to the semantics of Athapascan, the non-preemptive sequential schedule of tasks that follows the depth-first ordering of $\mathcal{T}$ is a valid schedule. This ordering is called *reference order* and denoted by R. According to R, the closures consecutively committed by a task t are executed after completion of t in the same order, while in a depth-first sequential schedule, a closure is executed just after committing.

```
void Sum (Data a, Data b, Data r) {
   r.write(a.read() + b.read());
}

void Fibo(int n,Data r) {
  if (n <2) r.write( n );
  else {
    int r1, r2;
    Task f1 = AllocateClosure( Fibo );
      f1.push( ReadAccess,  n-1 );
      f1.push( WriteAccess, r1 );
      f1.commit();
    Task f2 = AllocateClosure ( Fibo );
      f2.push( ReadAccess,  n-2);
      f2.push( WriteAccess, r2);
      f2.commit();
    Task sum = AllocateClosure ( Sum );
      sum.push( WriteAccess, r);
      sum.push( ReadAccess,  r1);
      sum.push( ReadAccess,  r2);
      sum.commit();
  }
}
```
Fig. 1. Fibonacci program with abstract machine instructions (it corresponds to the folk original Athapascan code for Fibonacci in [10], fig. 3.).

Work-stealing scheduling based on reference order. The nested structure of the spawn tree enables a depth-first work-stealing scheduling, similar to *DFDeques* scheduling proposed in [14] but here based on the reference order R instead of the standard sequential depth first order. All tasks in the systems are ordered according to R in a distributed way. Locally, each processor manages its own deque in which tasks are ordered according to R. When a closure is allocated on a processor, it is pushed on the top of the local deque but, following R, execution of current closure pursues. When the current closure completes, a new one is popped from the local deque. If this deque is empty, a new closure is stolen from the bottom of the deque of another randomly chosen processor.

3 Distributed implementation: DDS

This section presents the distributed data-flow stack implementation, named DDS, of the abstract machine model presented in section 2. DDS implements local stacks by allocating contiguous blocks of memory that can store several frames. A frame is related to the execution of a task; it is used to store all closures created by the task with direct links describing `Read` or `Write` data accesses. A new frame is pushed on the stack when a task begins its execution. Tasks are executed according to the reference order R.

Figure 2.a, shows the state of the stack during the execution of the recursive computation of the program of Figure 1. Starting from the base stack pointer, the frame related to the task `fibo(n,r)` is first pushed on the stack. During its execution `fibo(n,r)` creates three new tasks: `fibo(n-1,r1)`, `fibo(n-2,r2)` and a `sum` task to compute `r:=r1+r2`. The associated closures including their arguments are then allocated in the frame. When `fibo(n,r)` completes, the task `fibo(n-1,r1)` is popped from the top of the frame and a new frame is allocated for its execution. This new frame is pushed on the stack. When all closures allocated by a task are completed or stolen, its associated frame is popped and the execution of its successor according to R can start. In order to manage data dependencies, `Read` or `Write` data accesses are pushed into the closure and linked between closures according to the reference order (Figure 2.b).

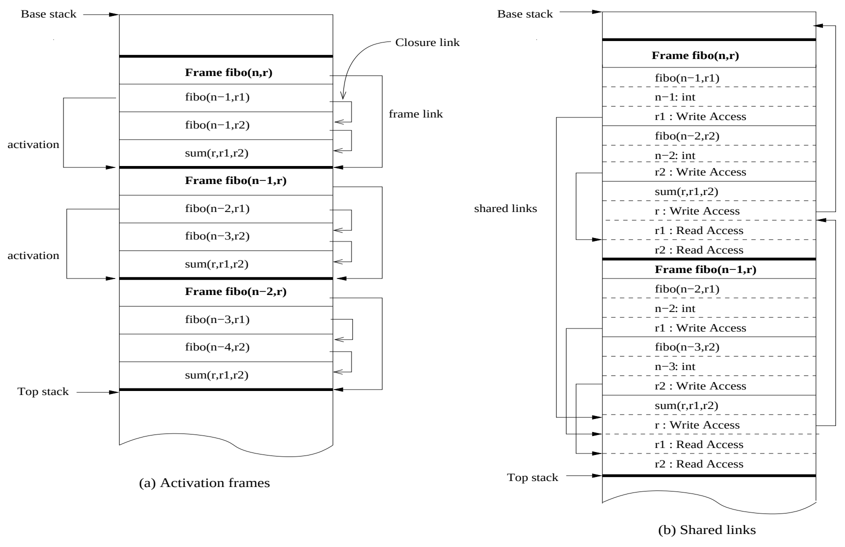


Fig. 2. (a) Stack structure with activation frames. (b) Data flow link.

Distributed work stealing and extended stack management A new stack is created when a processor becomes idle and steals work from another processor. When the current stack of a thread becomes empty or the current task is not ready, a steal request occurs. In this case, the thief thread first tries to steal a ready closure in another stack: first locally on SMP machines or, when no closures are found, a steal request is sent to a randomly chosen distant processor.

The stolen closure is ready to execute, i.e. all its input parameters are produced. For instance, in figure 2 a), in the top frame, the closure `fibo(n-1,r1)` is already completed, the closure `fibo(n-2, r2)` is ready while the closure `sum(r, r1, r2)` is not ready since its input parameter `r2` has not been produced. Using access links, the computation of ready closures is only performed on steal requests. Indeed, since the reference order is a valid sequential schedule, local tasks in a stack are executed without computing the readiness of closures. Following *work first principle* [9], this enables to minimize scheduling overhead by transferring the cost overruns from local computations to steal operations. In particular, atomicity of local accesses is ensured by non-blocking locks (*compare-and-swap* instruction).

Once the choice of a victim has been made, a copy of the chosen closure is pushed in a new stack owned by the thief processor. The original closure is marked as stolen. If the thief is a remote processor, input parameters of the task are copied and sent with the closure. In order to manage the data-flow for output parameters, a signalization task is pushed after the closure copy. This task plays the role of signaling that output accesses of the stolen task are produced – in order to compute readiness of successors – and sending the produced data to the victim processor.

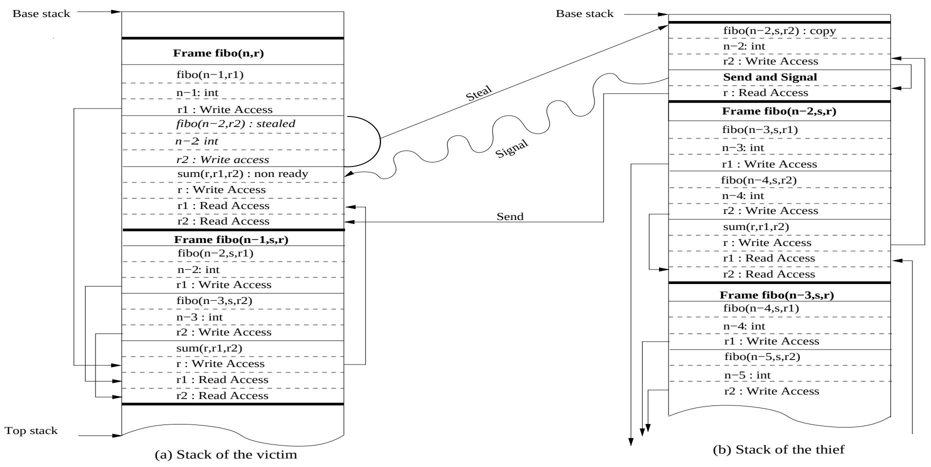


Fig. 3. Structure of both victim (a) and thief (b) stacks. A new task (Send Signal) is forked into the thief stack. Its role is to send back the result and signal the tasks marked as non ready that depend on the stolen task.

Remark Since DDS describes all tasks and their dependencies, it stores a consistent global state; this is used in [12] to implement fault tolerant checkpoint/restart protocols.

4 Theoretical analysis

This section provides a theoretical analysis of the DDS implementation, resulting in a language-based performance model for Athapascan macrodataflow parallel programs on heterogeneous grids. To model such an architecture, we adopt the model proposed in [3]. Given p processors, let $\Pi_i(t)$ be the instantaneous speed of processor i at time t, measured as the number of elementary operations per unit of time; let $\Pi_{ave} = \frac{\sum_{t=1}^{T}\sum_{i=1}^{p}\Pi_i(t)}{T}$ be the average speed of the grid for a computation with duration T.

To predict the execution time T on the grid, following [4], we adopt a language-based performance model using work and depth. The work W is the total number of elementary (unit) operations performed; the depth D is the critical-path, i.e. the number of (unit) operations for an execution on an unbounded number of processors. Note that D accounts not only for data-dependencies among tasks but also for recursive task creations, i.e. the depth of the spawn tree.

The work (and depth) of an Athapascan parallel program includes both the sequential work (W_S) and the cost of task creations but without considering the scheduling overhead; similarly to a sequential function call, the cost of a task creation with n unit arguments is $\tau_{fork} + n.\tau_{arg}$. If the cost of those task creations is negligible in front of W_S, then $W \simeq W_S$.

Theorem 1. *In the DDS implementation, when no steal operation occurs, any local access or modification in any stack is performed in a constant number of operations. Then, τ_{fork} and τ_{arg} are constants.*

The proof is direct: when no steal operation occurs, each process only accesses its own local stack. Due to the links in the stack and non-blocking locks, each access is performed in (small) constant time.

Since DDS implements a distributed work-stealing scheduling, a steal operation only occurs when a stack becomes empty or when the current task is not ready. In this case, the process becomes a thief and randomly scans the stack of the other processes (from their top) to find a ready closure; the resulting overhead is amortized by the work W when $D \ll W$. Indeed steal operations are very rare events as stated in [2,3] on a grid with processors speeds ratios may vary only within a bounded interval.

Theorem 2. *With high probability, the number of steal operations is $O(p.D)$ and the execution time T is bounded by $T \leq \frac{W}{\Pi_{ave}} + O\left(p\frac{D}{\Pi_{ave}}\right)$.*

The proof (not included) is derived from theorems 6,8 in [3]. Then, when $D \ll W$, the resulting time is close to the expected optimal one $\frac{W}{\Pi_{ave}}$.

5 Experiments

A portable implementation of DDS supporting Athapascan has been realized within the Kaapi C++ library [13].

Results on a cluster A first set of experiments has been executed on a Linux cluster of 100 PC (100 Pentium III, 733Mhz, 256MBytes of main memory) interconnected by fast Ethernet (100MBits/s). On this implementation, $\tau_{fork} = 0.23\mu s$ and $\tau_{arg} = 0.16\mu s$ are observed constant in accordance to theorem 1.
In the timing results (Figure 1): T_1 denotes the time, corresponding to W, to execute the benchmark on one processor; T_p the time on p processors; T_S the time of the pure C++ sequential version of the benchmark, it corresponds to W_S. Recursive subtasks creation is stopped under a threshold th where further recursive calls are performed with a sequential C++ recursive function call; the timing of a leaf task with $th = 15$ (resp. $th = 20$) is 0.1 ms (resp. 1 ms). Left and right tables report times respectively for the Fibonacci benchmark with up to 32 processors and for the Knary benchmark up to 100 processors. Both show scalable performances up to 100 processors, conformally to theorem 2.

Results of grid experiments We present here experimental results computed on the french heterogeneous GRID5000 platform during the *plugtest*[1] international contest held in november 2006. On the NQueens challenge (Takkaken

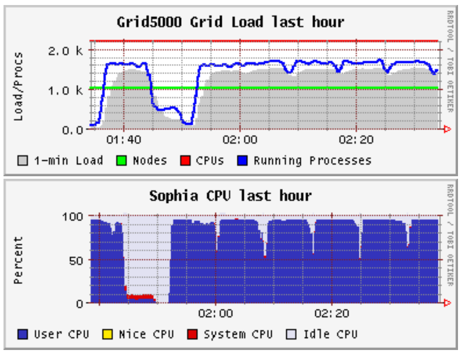
[1] `http://www.etsi.org/plugtests/Upcoming/GRID2006/GRID2006.htm`

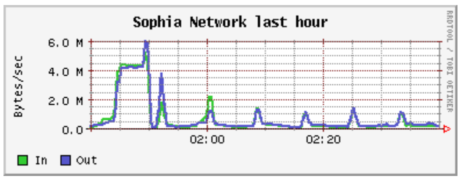
	fib(40) ; $th = 15$			fib(45) ; $th = 20$		
p	T_p	T_1/T_p	T_S/T_p	T_p	T_1/T_p	T_S/T_p
1	9.1	1	0.846	88.2	1	0.981
4	2.75	3.3	2.8	22.5	3.92	3.84
8	1.66	5.48	4.6	12.35	7.14	7
16	1.01	9	7.62	6.4	13.78	13.52
32	.99	9.19	7.78	3.7	23.83	23.37

	Knary(35,10) ; $th = 15$		
p	T_p	T_1/T_p	T_S/T_p
1	2435.28	1	0.984
8	306.17	7.95	7.83
16	153.52	15.86	15.61
32	77.68	31.35	30.86
64	40.51	60.12	59.18
100	26.60	91.55	90.13

Table 1. T_1, T_p and T_S (in second) for Fibonacci (a) and KNary (b) benchmarks.

sequential code), our implementation in Athapascan on DDS/Kaapi showed the best performances, honored by a *special prize*: On instance 23 solved in $T = 4434.9s$, an idle time of $22.72s$ was measured on the 1422 processors; this experimentally verifies theorem 2 with a maximal relative error $\frac{22.72}{4434.9} = 0.63\%$. Figure 4 shows the global grid load together with CPU and network load on one of the clusters composing the grid (cluster from the sophia site). These results have been obtained using GRID5000 monitoring tools during the last hour of execution. Our computations start approximately at 01:50. Different instances of nqueens problems are executed sequentially. The different graphs show a very good use of CPU ressources. At the end of each execution work stealing occurs, increasing briefly network load while enabling to maintain efficient CPU usage.





NQueens	p	T
21	1000	$78s$
22	1458	$502.9s$
23	1422	$4434.9s$

Fig. 4. CPU/network loads and timing reports.

6 Conclusions

Multithreaded computations may take benefit of the description of non strict data dependencies. In this paper we present a novel approach, DDS, to implement efficient work stealing for multithreaded computations with data flow dependencies. Local stack operations are guaranteed in small and constant time, while most of the overhead is postponed onto unfrequent steal operations. This important property enables us to predict accurately the time of a (fine grain) parallel program on an heterogeneous platform where processors speeds vary in a bounded ratio (theorem 2). Experiments reported on a cluster and a grid infrastructure with 1400 processors showed scalable performances.

Besides, by providing a consistent global state, DDS implementation enables to support fault tolerance. A perspective of this work is to use fault-tolerance to extend theorem 2) to dynamic grid platforms where speed ratios cannot be considered bounded anymore, e.g. when a processor leaves (resp. enrolls) its speed becomes zero (resp. non zero). Under a given speed threshold, considering a processor as faulty might be a practical way to ensure the bounded ratio property.

Acknowledgments. The authors gratefully acknowledge Serge Guelton, Samir Jafar and Rémi Revire for useful discussions and participation to implementation and experimentations.

References

1. U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory Comput. Syst.*, 35(3):321–347, 2002.
2. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.
3. M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Th. Comp. Sys.*, 35(3):289–304, 2002.
4. G. E. Blelloch. Programming parallel algorithms. *Com. ACM*, 39(3):85–97, 1996.
5. G.E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, April 1993.
6. R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.
7. D.E Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the* 15*th Annual International Symposium on Computer Architecture*, pages 141–150, Honolulu, Hawai, 1989.
8. P. Fatourou and P.G. Spirakis. Efficient scheduling of strict multithreaded computations. *Theory of Computing Systems*, 33(3):173–232, 2000.
9. M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. In *Sigplan'98*, pages 212–223, 1998.
10. F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, editor, *Pact'98*, pages 88–95, Paris, France, October 1998.
11. L. J. Hendren, G. R. Gao, X. Tang, Y Zhu, X. Xue, H. Cai, and P. Ouellet. Compiling c for the earth multithreaded architecture. In IEEE, editor, *Pact'96*, pages 12–23, Boston, USA, 1996.
12. S. Jafar, T. Gautier, A. W. Krings, and J.-L. Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In LNCS Springer-Verlag, editor, *EUROPAR'2005*, Lisboa, Portogal, August 2005.
13. MOAIS Team. KAAPI. `http://gforge.inria.fr/projects/kaapi/`, 2005.
14. G.J. Narlikar. Scheduling threads for low space requirement and good locality. Number TR CMU-CS-99-121, may 1999. Extended version of Spaa'99 paper.
15. M.C. Rinard and M.S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Programming Languages and Systems*, 20(3):483–545, 1998.