Research Report 220

PRAM implementation on fine-grained MIMD multicomputers

Francoise Baude

RR220

The more user-friendly tools for expressing data-parallelism are encompassed by the PRAM language mechanisms (mainly, global addressing space, implicit processes synchronization). In this paper, we evaluate the cost for implementing such mechanisms on the only class of parallel architectures that should prove to be really scalable as well as versatile, i.e. MIMD fine-grained multicomputers. Being massively parallel, these architectures are constrained to not exceed a given wire density and also to have fine-grained processing nodes. For these reasons, recent algorithmic and hardware technics for hidding communication latency of PRAM emulations by parallel slackness do not apply.
We consider an abstraction of these machines by reasoning in terms of the actor model of concurrent computation. Given the architectural constraints raised by the chosen class of architectures, the idea for implementing any PRAM program is to transform it into an actor program such that it can be efficiently implemented on the target machine via mapping technics such as for exemple graph embedding. This requires the underlying communication graph of the actor program to be predictable and sparse. This transformation is grounded on probabilistic theoretical simulations of PRAMs on parallel machines based on sparse communication networks.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

May 1992

PRAM implementation on fine-grained MIMD multicomputers

Preliminary Version

Françoise Baude*
Dept of Computer Science
University of Warwick
Coventry CV4 7AL, United Kingdom
e-mail: baude@dcs.warwick.ac.uk

April 29, 1992

Abstract

The more user-friendly tools for expressing data-parallelism are encompassed by the PRAM language mechanisms (mainly, global addressing space, implicit processes synchronization). In this paper, we evaluate the cost for implementing such mechanisms on the only class of parallel architectures that should prove to be really scalable as well as versatile, i.e. MIMD fine-grained multicomputers. Being massively parallel, these architectures are constrained to not exceed a given wire density and also to have fine-grained processing nodes. For these reasons, recent algorithmic and hardware technics for hidding communication latency of PRAM emulations by parallel slackness do not apply.

We consider an abstraction of these machines by reasoning in terms of the actor model of concurrent computation. Given the architectural constraints raised by the chosen class of architectures, the idea for implementing any PRAM program is to transform it into an actor program such that it can be efficiently implemented on the target machine via mapping technics such as for exemple graph embedding. This requires the underlying communication graph of the actor program to be predictable and sparse. This transformation is grounded on probabilistic theoretical simulations of PRAMs on parallel machines based on sparse communication networks.

1 Introduction

The PRAM (Parallel Random Access Machine) has now won fairly widespread acceptance in the theoretical community as an extremely useful vehicle for studying the logical structure of highly parallel (i.e. data-parallel) computation [GR88], [KR90].

*This work was partially funded by EC grant under Esprit Project P440, by Alcatel-Alsthom Research and is now supported by an INRIA post-doctoral grant

There are two major mechanisms that constitue the basis of this attractive way of expressing parallel programs : global synchronization and a global addressing space materialized by a shared global memory to serve as a communication medium between the parallel processes. But they are so powerful that they can not be reasonably considered as hardware primitives and need to be emulated. At the central key to any implementation of PRAM based languages, how to emulate shared memory has been given a lot of attention [KU88], [Ran87], [Val90b]. Even if the current tendency is to reduce the number of calls to these high-level primitives in order to decrease the overall cost of executing a parallel program [Gib89], [ACS89], [Val90a], their implementation is however required.

Here, we are considering the massively parallel architectures such as they are investigated by the J-machine project [DW89] or the MEGA project [GBES90] for exemple. To achieve the connection of a massive number of processing elements, some technological constraints arise.

The physical limit on the wiring density is less penalizing in case of mesh or torus topologies than in case of hypercubes or butterflies (in general in case of logarithmic diameter topologies) [Da89]. So the J-machine [Da89] or the MEGA machine [GBES90] are based on a torus topology.

Hence using such a high diameter routing network as a palliative of the technological impediment raised by large size crossbars, as it would be required for simulating PRAM programs, is not recommended. Nevertheless, some PRAM simulations apply theoretically to these high diameter routing network [Sie89], [Val90b]. But they are not practically attractive; indeed, for the case of a square mesh routing network for exemple, they would use all but one rows of the mesh for simulating a high throughput switching network, while amassing all the PRAM processes on just one row.

More generally, the parallel slackness idea currently proposed to hide communication latency of PRAM programs simulations [Val90a] and [ACS89] can not be applied in the present fine-grained context. The design choices of such massively parallel machines is that processing nodes have a small amount of memory. But to hide the communication latency of simulating PRAMs on such architectures, it would require a processing element to simulate $O(\sqrt{N})$ processes of the PRAM program.

For size reasons, the functionality of the routing chips is reduced to the simplest one, mainly the ability to forward a message towards its destination. Thus, it prevents to provide hardware implementation of more complex operations such that combining or duplicating, even if efficient emulation of shared memory could require it [Ran87]. Moreover, bounded size buffers in the routing chips prevent to implement in the hardware priority based routing schemes that do not yield to constant size queues [KU88], [Val90b].

Briefly, machines considered here are MIMD fine-grained multicomputers, based on asynchronous, mesh-like connection networks, each point of the network hosting

one computation processor/routing chip pair (routing chips enabling a virtual total connection between processors) executing a very simple routing algorithm enabling to follow shortest paths. The J-Machine and the MEGA machine illustrate exactly those design choices. By proposing the most efficient use of available chip and board area, such machine model should be the only one really scalable, thus reaching the higher degree of hardware parallelism. But, the now common technics for efficiently (i.e. work preserving) implementing the PRAM model, which basically enable to hide latency and low throughput of the routing network thanks to some parallel slackness, can not be applied given the present architectural choices. So, the present aim is to propose other technics such that it becomes possible to quite efficiently implement the PRAM mechanisms under the chosen architectural constraints.

Outline We will estimate the cost of implementing such PRAM mechanisms on any MIMD fine-grained multicomputer, whatever been the topology of the underlying communication network, to be a logarithmic multiplicative factor increase in time. The independance with respect to the underlying topology comes from the fact that we work with an abstraction of these message-passing based machines by reasoning in terms of a paradigm of such a message-passing computation model, i.e. the **actor model of concurrent computation** [Agh86]. Moreover, this implementation in term of actors will not prevent from exploiting other sources of parallelism (control-parallelism for exemple) at the same time [Agh89].

The implementation of PRAM programs through an automatic translation into an actor program (translation grounded on theoretical simulations of PRAMs on sparse communication network based parallel machines [MV84], [Ran87], [Val90b]) will enable to solve difficult implementation problems while not being too much constrained by the target architecture. Nevertheless, we will seen (section 4) that efficient execution of some type of actor programs can also raise difficult problems. Thus the translation of PRAM programs into actor programs (section 5) will be carefully conducted, in order to obtain only efficiently executable actor programs. After some definitions, we will first estimate the minimum number of calls to actor language primitives required to simulate the PRAM mechanisms (section 3).

2 Definitions

2.1 PRAM

The PRAM [Gol82] language looks like a declarative sequential language, except that instructions can be parallel and in this case, the number of parallel processing units (PPUs) that need to be active at the same time is also specified. The number of PPUs that can be activated is unbounded. These PPUs will execute the same instruction (either each PPU decodes the same program, or a specific one called the CPU, broadcasts the instruction to all PPUs). Thus a PRAM language is SIMD.

Each PPU has some local memory. Moreover, an unbounded size global memory can be accessed by all PPUs. Even if PPUs execute the same instruction, the ability of indirect addressing enables different memory locations (local or global) to be accessed. But an unbounded number of PPUs may try to access the same global memory location at once. The various PRAM languages differ from one another in the way they handle these read or write conflicts. The most powerful ones (CRCW) allow read and write concurrent accesses. A protocol is nevertheless required. For example, if several PPUs $p_1, \ldots, p_k$ attempt to write values $v_1, \ldots, v_k$ into the same variable, only the value v_1 is written because p_1 (the less numbered PPU) has priority (CRCW Priority); or the value $v_1 * \ldots * v_k$ gets written, for some prespecified associative operator $*$ (CRCW-Fetch-&-*).

The time complexity function of a PRAM program depends directly on the maximum number of instructions (sequential and parallel) that must be executed in order to complete the computation for any given input. We note it $T(n)$ where n is the input length. The space complexity function of a PRAM program depends on the maximum number of PPUs that need to be active at once during the computation on any given input (noted $S(n)$).

2.2 Actors

The actor language [Agh86] enables to define an unbounded number of asynchronous, autonomous computing entities, called actors, which can only be activated by the mean of asynchronous message sending. More precisely, an actor has some bounded size local memory, a script which describes how the actor must deal with the next message it will receive and a mailbox to store incoming messages. In response to one message, an actor can only execute a bounded number of some of these following actions :

- it can create a bounded number of new actors (in this case, their initial script and the initial values of their memories have to be noticed); for each new created actor, a unique address is automatically allocated and can be memorized by the creator actor;
- it can send a bounded number of messages to actors whose address is known; all of the actors with wich an actor can communicate are called its acquaintances;
- it can modify its memory or/and change its script. There are two types of actors : serialized actors that can modify their state, unserialized actors that are unable to modify one thing in their state and that are also qualified as being insensible to the history of the ongoing computation. We consider that this last type of actor has an additional property compared with the orthodox actor language [Agh86], i.e. that unserialized actors can treat an unbounded number of messages at once (because this assumption has no effect on the thread of the computation).

To understand how an actor program computes, we refer us to an ideal actor machine onto which actor programs proceed. On this machine, each one of the computation entities is an actor. Computation entities can be created dynamically. Those entities are placed around a communication system, whose task is to forward messages to their destination. This communication system works in an unspecified way. Its only specified property is that every message will be eventually delivered at destination. Thus the order in which messages arrive at destination is undeterministic. Each time a serialized actor has terminated with the treatment of one message, it checks in its mailbox to see if a message is present and can be taken from it. If not, the arrival of a message will wake it up. Each time an unserialized actor has terminated with the treatment of one or more messages, it checks its mailbox. If there are messages, they are all taken from the mailbox at once, and their simultaneous treatment begins.

This informal description is best represented by a partial order of the events that arise during the computation [Cli81]. One event is the treatment of one message (or more messages for the case of unserialized actors). Essentially, events are ordered according to the trivial rule that a message can not be received before it has been sent.

This abstraction of the execution of actor programs is the basis of the definition of complexity measures for actor programs [BVN91]. To do this, it is assumed that the treatment of message(s), i.e. event, costs one unit of time, and that the reception of one message occurs at the best one time unit after the message has been built. Taking into account that non-determinism of messages arrival can yield to many possible partial orders of events from the execution of the same actor program on the same input, the two major complexity measures for actor programs are :

Definition 1 (Actor Time Complexity [BVN91]) *The function of time complexity of an actor program $\mathcal{A}$ is defined by the maximum of the* actor time *of the program $\mathcal{A}$ on all possible inputs of length n. The actor time is the maximum depth of all partial orders that can be generated by $\mathcal{A}$ on one given input.*

Definition 2 (Actor Space Complexity [BVN91]) *The function of space complexity of an actor program $\mathcal{A}$ is defined by the maximum of the* actor space *of the program $\mathcal{A}$ on all possible inputs of length n. The actor space is the maximum of the* required space *considering all partial orders that can be generated by $\mathcal{A}$ on one given input. The required space of a computation is defined to be the sum of the local memory length over all actors created by this computation.*

Now that the actor language is defined, it is clear that it is more close to the chosen target machine than the PRAM language is, but also that it is independant of one specific machine (thus giving generality to our solution). Indeed, like the target machine type, the actor language is message-passing based, computation is asynchronous, all the information is distributed (thus requiring that to retrieve an information, it is necessary to have been *explicitely* informed of its physical

localisation); unlike in the case of a specific machine, message forwarding is left unconstrained and actors can be placed anywhere around the network whatever been the topology of the underlying architecture.

3 Comparison of the expressiveness of PRAM and actor languages

As proved in [BVN91], the actor language is a parallel computation model in the sense that if a problem is known to belong to the $\mathcal{NC}$ complexity class (i.e can be solved in time $\log^k(n)$ and space $poly(n)$ on any given parallel computation model such that for example PRAM or boolean circuits) then it can also be solved in time $\log^k(n)$ and space $poly(n)$ using the actor language. The proof was conducted by comparing actors with PRAM. This result implies that the actor language expressiveness is not too much distant from this of the PRAM language. We will now exactly evaluate how much they differ. More precisely, we will establish the following :

Theorem 1 *There exists at least a PRAM program having time complexity $T(n)$ and space complexity $poly(n)$ that can not be simulated by an actor program having less than $O(T(n) \cdot \log n)$ time complexity and $poly(n)$ space complexity.*

Proof The CPU and each PPU is simulated by a serialized actor whose behaviour corresponds to the PRAM program translated as follow : the length S sequence of instructions is compiled into S distinct behaviours. The transition from one behaviour to the next is done by a 'become' instruction. For example, if the nth PRAM instruction is "goto M if $y_j > 0$" then the corresponding behaviour named "Instruction_n $(\ldots)$" ends by the statement "if $y_j > 0$ then become Instruction_M else become Instruction_$n + 1$".

In the PRAM, the PPUs and the CPU are synchronized. In actor, treating a message means to execute at most a bounded number of instructions including messages sending. Thus, during the treatment of one message, an actor can broadcast the synchronization signal to only a bounded number of its acquaintances. Hence, the actor time required to activate an actor is $\Omega(\log S(n))$ if $S(n)$ is the number of actors simulating PPUs, as established by [AFL83].

Each of the memory cells must be hosted by a serialized actor because it can then record an eventual change of the cell value. According to the PRAM language definition, a PPU can have access to any cell of the global memory, simply by giving its rank from the first cell in the memory. It is then necessary to provide a mean for an actor simulating a PPU to obtain the address of any actor hosting the required cell. Obviously, an actor has not all actors of the program as acquaintances. In fact searching for the wanted actor address can be formulated as progressing on a bounded degree (number of acquaintances) actors network, each "crossed" actor

orienting the message search in the right direction. Indeed, it is the same as to realize a communication between any pair of actors that are not connected in this network. Because of the bounded degree, the lower bound of $\Omega(\log N)$ proved in [Mey86], theorem 10, applies, where N is the number of actors around this network.

Once the address is known, the access is simply realized by the mean of one message sending. Given the fact that the access request message contains the address of the actor issuing the request, the answer to the access is simply realized by one message sending from the actor hosting the cell to the actor simulating the PPU.□

This lower bound gives indication on minimal cost for implementing PRAM programs by their translation into actor programs, but its proof does not provide an implementation scheme. Although it clearly appears that the general mechanism of global memory access has to be implemented by actor message routing, many points remain unsolved : how many actors should be used, how is the global memory distributed, how to deal with concurrent memory address searches, how to deal with concurrent accesses to a same memory cell, etc ...

Implementation with sublogarithmic time-loss Before solving these specific points, let us observe that the lower bound does not apply for some specific PRAM programs. In fact, if one has not to search for the actor address hosting a given memory cell because this actor is already an acquaintance, and if the actors simulating the PPUs can be locally synchronized then the simulation can be achieved with only a constant time-loss.

Definition 3 (Problems with predictable communication) *A parallel algorithm has predictable communication if each process using only its local state, can infer in constant time, the address of the next process from which it will receive a communication.*

For a process can infer the process from which it will receive the next communication, this means that the computation does not depend on the value of the input data but rather on its length. Thus, it becomes possible to build an actor network before the beginning of the computation, such that every actor simulating a PPU is connected with actors hosting the memory cells the PPU will access. Nevertheless, this is only reasonable if the number of acquaintances is not too high, otherwise contradicting the assumption that actors have bounded memory. So we will adopt this way of implementing PRAM programs, for a $f(n)$ number of different memory cells a PPU will access to (obviously, $f(n) \leq T(n)$), such that $f(n)$ is at most a slowly growing function of n, i.e. $\log(n)$ or less (for $f(n) = \log(n)$, the simulation in [PV81] enables to implement this with only constant time-loss using bounded degree processes network). Moreover such a class of problems is far from empty and contains famous examples such that bitonic sort, prefix computations, and more generally problems built on the ascend-descend scheme [PV81].

Lastly, synchronization between PPUs can be simply achieved by forcing actors to have a cadanced behaviour : an actor simulating a given PPU must treat messages not in the order it receive them (this being in fact undeterministic in the actor language context) but according to what the algorithm requires. To describe this order more simply, a high-level construct introduced in the actor language, called "select-accept" [BVN91] can be used.

Once we have checked that the lower bound to implement PRAM programs by mean of their translation in the actor language is affordable, we will check how actor programs can now be implemented on the chosen machines.

4 Implementation of massively parallel actor programs

As shown in [DW89], the actor language should be easily implementable on the chosen class of machines, having the J-Machine as a buildable exemple. Indeed, even if communicating actors can be physically distant, [Da89] say that "the latency of sending a message from node i to *any* other node j is sufficiently low that the programmer sees the network as a complete connection". But, when asserting such attractive performances, it is assumed that the network is not at all loaded.

This unloaded network assumption does not apply here, because programs of interest are massively parallel and fine-grained : such programs are based on a huge number of concurrently active processes and moreover, each process uses to initiate a communication after executing only a small number of local operations. Thus, without care, contention on the underlying network can be very high, even in the case where messages would come all from distinct nodes and would all be destined to distinct nodes. This case of permutation routing of N messages has a worst case contention (number of messages crossing the same router) of $\sqrt{N}$ when using an oblivious routing algorithm, whatever been the diameter of the network [BH85].

But clearly, this worst case does not apply if communication tasks are restricted to those that send messages between nodes such that routing paths do not overlap. This arises if communicating processes are physically close to one another, but also, given the performances of the underlying network without contention, we can accept programs that yield to a low contention on the network even by not exploiting physical locality.

So, if all possible communicating pairs of actors are known before communications themselves arise, then the initial placement of any actor can be achieved such as to guarantee the physical locality of its future communications, or even only a low contention on the network. In order to realize such mappings, many technics are currently used such that graph embedding [Ros81], simulated annealing [Dah90]. Thus, it is possible to efficiently deal with communication predictable PRAM programs by first transforming them into communication predictable actor programs and then, at initial time, do the best possible mapping of the actor program communication graph on the target network connection graph.

The problem that remains concerns communication unpredictable PRAM programs. One technic to deal with programs that behave dynamically is process migration whose aim is to put a process more close of the process it has just planed to communicate with. Because it applies during execution, the migration cost shows itself tolerable only if the amount of the communications between the two processes is really high. In other respects, we want not to afford the cost of the simultaneaous routing of $O(N)$ messages having any possible destination.

The proposed solution is to manage so that a PRAM program even if it has unpredictable communications will be simulated by a communication predictable actor program, even if the time-loss is not constant. In other respects, we have seen in §3 that simulation of communication unpredictable PRAM programs using the actor language can not have a constant time-losss but should raise at least a logarithmic time-loss. The next part will present a way of transforming communication unpredictable PRAM programs into communication predictable actor programs that yields to a time-loss very close to the lower bound. Then given the bounded number of predictable communication tasks that those actor programs would generate, these should be efficiently implementable on the target machines using mapping technics.

5 Simulation of PRAM programs by efficiently implementable actor programs

As suggested in [Mey86], a solution to deal with unpredictable communications is to constrain all message sending between any two processes (even if they know their respective addresses) to pass through a bounded degree network. The new communication pattern of the program can then be more easily embedded giving the physical netwotk topology. But, the distance between any two communicating processes can raise the diameter of the network. With a bounded degree, we have seen that the best lower diameter for the case of connecting N nodes is $\Omega(\log N)$.

Of course, it requires that the bounded degree network be able to route messages to their respective destination. For scalability purposes, the routing must be distributed (i.e., any node takes the decision where to forward a message only on the basis of local informations). Until now, the best known strategies to route a $O(N)$ set of messages on a bounded degree network built with N nodes achieve a $O(\log N)$ with high probability delay (taking into account the path length as well as the total time a message waits in queues) [Upf84], [Ran87], [Val90b].

The simulation, whose following theorem gives the performances, will thus be based on such a network of actors programmed by one of those distributed routing strategies.

Theorem 2 *For every CRCW-PRAM program $\mathcal{P}$ having time complexity $T(n)$ and using $S(n)$ PPUs, it is possible to automatically generate a communication predictable actor program that interprets $\mathcal{P}$ whose actor time complexity is $O(T(n)\cdot$*

$\log n$) *with high probability using* $O(S(n))$ *actors.*

Proof We have chosen the routing strategy developped in [Ran87] because it not only efficiently, and in a very practical way, solves the routing of N messages but it also serves other purposes such as combining memory access requests as a solution for managing concurrent access to identical memory cells. Moreover, when used in conjunction with universal hashing as a global memory distribution strategy [MV84], the total needed amount of randomness is reduced compared with others models such that [Val90b] for exemple.

The network of actors is connected following a butterfly. The number of nodes in a butterfly with n levels is $N = 2^n \cdot n$, and we assume that levels 0 and n are identified, so that the butterfly is wrapped around. Each node in a butterfly is assigned a unique number $< c, r >$ where $0 \leq c < n$, $0 \leq r < 2^n - 1$. Node $< c, r >$ is connected to nodes $< c+1 \ mod \ n, r >$ and to node $< c+1 \ mod \ n, r \oplus 2^c >$, where $\oplus$ denotes bitwise exclusive or.

Each node $i = r \cdot n + c$ of the $S(n)$ nodes butterfly network is in fact an actor dedicated for routing messages, which, in addition to its neighbours in the network, has some others actors as acquaintance : one actor is dedicated for simulating the *ith* PPU of the PRAM program, one is dedicated for hosting the cells of global memory allocated by the hash function, one is dedicated to apply the hash function to any PRAM memory address. In order to host memory cells and hash function parameters while actors have fixed-size memories, the solution described in the Appendix of [Ran89] is relevant.

The amount of randomness that is introduced by allocating memory by universal hashing (picking at random one function in a class of universal hash function) is sufficient to ensure a deterministic routing of N messages in $O(\log N)$ *synchronous* steps with high probability.

Indeed, this strategy as well as all other quoted routing strategies are described and evaluated by assuming that the underlying network has a synchronous behaviour. Clearly it is not the case of an actor network because message sending between actors is assumed to be asynchronous and arrival order of messages is undeterministic. But, the synchronous routing strategy of [Ran87] is original because it is based on a communication predictable algorithm. This is due to the fact that each node invariably forwards a message to each of its successors in the network. Because of the 4 degree underlying support for routing, the programming of this algorithm using the actor language has only a constant-time loss compared with the synchronous version (§[BVN91] or Appendix for details). Moreover, it uses fixed-size FIFO buffers (while remaining deadlock-free) that can be managed with a fixed number of actor computation steps.

Finally, this routing strategy supports, *without any further time-loss*, any protocol for managing concurrent accesses to common memory cells.

Transition to the next PRAM instruction Because all actors simulating PPUs have a copy of the same code, they can proceed independently in case of instructions that are local to the PPUs. In case of a global memory access instruction, each actor simulating a PPU must send a message to the associated actor that routes, even if the PPU is not concerned. Or, we can use a more flexible model of PRAM [Gib89] that enables PPUs to not issue the same number of memory accesses, and that introduces a barrier synchronization instruction. But, such a synchronization can be easily implemented on top of the routing algorithm using the End of Stream messages (see [Bau91] for details). After the routing of requests, access, and routing of replies, termination is automatically detected by each actor that routes, as soon as it has received an End Of Stream message from each of its predecessors on the butterfly. □

Remarks A similar automatic transformation enables to execute massively parallel and communication unpredictable actor programs, where actors are dealt in the same way as the global memory of a PRAM program. For $S(n) > p$, p being the number of processors of the machine, the resulting actor program is built around a $O(p)$-nodes butterfly and each node hosts [1] $S(n)/p$ actors simulating PPUs. To simulate one memory access step of the PRAM program, the preceeding simulation has to be applied $O(S(n)/p)$ times. But we can hope to take advantage of a pipeline effect like when applying instruction lookahead (see [Ran89] for implementation details).

6 Conclusions and further work

The results that have been presented establish a bridge between parallel programming in $\mathcal{NC}$ on one side (where in fact the way of programming in $\mathcal{NC}$ were for long amalgalated with SIMD architectures) and MIMD concurrent multicomputers on the other side. Indeed, we have shown that the execution of highly parallel PRAM-like programs on these architectures can in fact be expressed as a problem that is very common in this domain, i.e. a problem of mapping one set of communicating processes on a given physical topology [Ath87].

We would be interested by works to develop mapping technics adapted to the specific case of a butterfly communication network on one side and different possible physical topologies, including meshes, torus, on the other side.

In the context of these MIMD multicomputers, we have shown that the distinction between communication predictable and communication unpredictable programs is crucial. Communication predictable programs have the property that they can be directly mapped although communication unpredictable ones have to be first reorganized and suffer at least a logarithmic increase in the total number of operations for their execution.

[1] or is connected with, for the case where local memory is too small

This puts once more in light the crucial importance of works whose aims are to decrease the total cost of the communications, for exemple by taking advantage of the communication latency [ACS89]. But, given the present architectural constraints whose essentially prevent to use parallel slackness, we have achieved the best possible implementation. Thus, the only way to reduce PRAM programming costs would be to use as intensively as possible communication predictable PRAM primitives. In this context, the redefinition of PRAM programs by using communication predictable primitives instead of unpredictable communication ones could be a way of achieving better performances. For example, we should remplace whenever possible the list-ranking procedure with the prefix sum procedure [GMT88]. Also, using the powerful CRCW-Fetch-&-* protocol could save global memory accesses while its implementation does not cost more than the EREW protocol.

References

[ACS89] A. Aggarwal, A.K. Chandra, and M. Snir. Communication Latency in PRAM Computations. In *Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[AFL83] E. Arjomandi, M.J. Fischer, and N.A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *J. ACM*, 30:449–456, 1983.

[Agh86] G. Agha. *ACTORS : A model of concurrent computation in distributed systems.* MIT Press, Cambridge Massachusetts, 1986.

[Agh89] G. Agha. Supporting Multiparadigm Programming on Actor Architectures. In E. Odjyk, M.Rem, and J.C. Syre, editors, *Proceedings of PARLE 89*, pages 1–19, Eindhoven, 1989. Springer Verlag.

[Ath87] W.C. Athas. Fine grain concurrent computation. CALTECH Technical Report 5242:TR:87, 1987.

[Bau91] F. Baude. *Utilisation du paradigme acteur pour le calcul parallèle.* PhD thesis, Université Paris-Sud, Orsay, 1991.

[BH85] A. Borodin and J. Hopcroft. Routing merging and sorting on parallel models of computation. *Journal of Comp. and System Sciences*, 30:130–145, 1985.

[BVN91] F. Baude and G. Vidal-Naquet. Actors as a parallel programming model. In *Proc. of the 8th Symposium of Theoretical Aspects of Computer Science*, volume 480, pages 184–195. LNCS, 1991.

[Cli81] W.D.C. Clinger. *Foundation of Actor Semantics.* PhD thesis, MIT, Cambridge, Mass, 1981.

[Da89] W.J. Dally and al. The J-Machine : a fine-grain concurrent computer. In *IFIP Congress*, 1989.

[Dah90] E. Denning Dahl. Mapping and compiled communication on the connection machine system. In *Proceedings of the 5th Distributed Memory Computing Conference*, 1990.

[DW89] W.J. Dally and D.S. Wills. Universal Mechanisms for Concurrency. In E. Odjyk, M.Rem, and J.C. Syre, editors, *Proceedings of PARLE 89*, pages 19–33, Eindhoven, 1989. Springer Verlag.

[GBES90] C. Germain, J.L Béchennec, D. Etiemble, and J.P Sansonnet. An interconnection network and a routing scheme for a massively parallel message-passing multicomputer. In *Proc. of the 3rd Symposium on Frontiers of Massively Parallel Computation*, College Park, MD, 1990.

[Gib89] P.B. Gibbons. A More Practical PRAM Model. In *Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[GMT88] H. Gazit, G.L. Miller, and S.H. Teng. *Concurrent computations*, chapter Optimal tree contraction in the EREW PRAM model, pages 139–155. Plenum Publishing Corporation, 1988.

[Gol82] L. M. Goldschlager. A Universal Interconnection Pattern for Parallel Computers. *J. ACM*, 29:1073–1086, 1982.

[GR88] A. Gibbons and W. Rytter. *Efficient parallel algorithms.* Cambridge University Press, Cambridge, 1988.

[KR90] R.M. Karp and V. Ramachandran. *Handbook of theoretical computer science*, volume B, chapter Parallel algorithms for shared-memory machines, pages 869–942. Elsevier, 1990.

[KU88] A.R. Karlin and E. Upfal. Parallel hashing : An efficient implementation of shared memory. *J. of ACM*, 35:876–892, 1988.

[Mey86] F. Meyer:auf der Heide. Efficient simulations among several models of parallel computers. *SIAM J. Comput*, 15:106–119, 1986.

[MV84] K. Melhorn and U. Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, 21:339–374, 1984.

[PV81] F.P Preparata and J. Vuillemin. The cube-connected cycles : A versatile network for parallel computation. *Comm. of the ACM*, 24:300–309, 1981.

[Ran87] A.G. Ranade. How to emulate shared memory. In *Proc. of the 28th IEEE symp. on FOCS.*, pages 185–194, 1987.

[Ran89] A.G. Ranade. *Fluent parallel computation.* PhD thesis, Yale Univ., Yale, US, 1989.

[Ros81] A.L. Rosenberg. Issues in the study of graph embeddings. In *Proc. of the International Workshop WG80*, volume 100, pages 150–176. LNCS, 1981.

[Sie89] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proc. of the 30th IEEE symp. on FOCS.*, pages 20–25, 1989.

[Upf84] E. Upfal. Efficient schemes for parallel communication. *J. ACM*, 31:507–517, 1984.

[Val90a] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[Val90b] L.G. Valiant. *Handbook of theoretical computer science*, chapter General purpose parallel architectures, pages 943–972. Elsevier, 1990.

Appendix : Actor programming of [Ran87] routing strategy

In [Ran87], thanks to the forward of "ghost" messages, every node is continuously informed about messages forwarded by its predecessors (the nodes which can send it messages). In case of an asynchronous and non-deterministic context, it is useful that each node sends back an acknowledgement each time it has received a data message. In other words, the following rule has to be coded as the behaviour of actors dedicated for routing :

Let succ (resp. pred) be the number of neighbours to which messages are forwarded (resp. from which messages are received).

If (nb.mess.received = pred) and (nb.ack.received = succ)
then nb.mess.received:=0; nb.ack.received:=0
    selection of messages from the buffers
    construction and forwarding of *succ* messages with timestamp=T
    construction and forwarding of *pred* acks with timestamp=T
    T := T+1

At minimal cost, this rule keeps the execution of the whole routing task cadenced, without requiring any global synchronization. The cadenced behaviour of the neighbours of any node is described below.

Property 1 *For every actor A dedicated for routing on the network, at each step T of the routing, A forwards data messages to its successors which will receive them during their T^{th} routing step, and A acknowledges the data messages received from its predecessors during this T^{th} routing step, and these acknowledgements will be received by these predecessors during their $(T+1)^{th}$ routing step.*

Application to the analysis of Ranade's strategy implemented on the actors network A classical technic used to evaluate performances of routing strategies is based on the buiding of a *delay sequence* by looking for messages which have caused delays, starting from the most delayed one. Here instead of tracing back using a global time, the backward is done according to the partial order $\mathcal{C}$ of actor events. Although the global time concept is not used for our case, the transposition in actor terms of the definitions of the two primitive delays ($b-delay$ and $m-delay$) enables to observe that messages in the delay sequence are in fact the same as those that would have been extracted by the analysis of the synchronous execution of the routing for the same input. Then, we can adapt the theorem 2 of [Ran87], where d is the diameter of the butterfly network constituted by $S(n)$ actor executing the routing :

Theorem 3 *Suppose that the routing of a memory requests set of size p takes at most an actor computation time equal to $(7 \cdot d + \delta) \cdot 4$. If each buffer is of size b (b*

being a constant independant of the network size), then there exists an input-output path S of length $8d$ and a sequence of $min(\delta\ ,\ b \cdot d)$ messages which is polarized along S.

Theorem 3 of [Ran87] that shows why "bad" sequences of messages occur with very low probability can be applied, thus the routing is terminated after $\log p$ routing steps with overwhelming probability.