OAL: an Implementation of an Actor Language on a Massively Parallel Message-Passing Architecture

Jean-Louis Giavitto, Cécile Germain
LRI - Architecture et Conception des Circuits Intégrés
Bât 490 Université de Paris XI - 91405 Orsay cedex France
email: giavitto@lri.lri.fr
Julian Fowler
LFCS - University of Edinburgh

ABSTRACT

We study the implementation of an actor language, OAL, *on a massively parallel message-passing architecture:* MEGA. *Motivations and implementation constraints are exposed. A simulator has been built to investigate resource consumption. First results show the feasibility of the implementation but indicates serious problems in memory usage. Load-balancing strategies are presented which partially solve the memory problem. Actors adequacy as a model for exploiting massive parallelism is discussed in conclusion.*

I. Introduction

MEGA (Machines to Explore Giant Architectures) is a family of architectures dedicated to the exploration of message passing on massively parallel MIMD machines for Artificial Intelligence applications. Such computers attempt to achieve massive parallelism (more than 10^9 ips) using a very large number of processing elements and intensive communication between them [DAL88]. That implies the expression and the exploitation of a very fine-grained parallelism.

The Computer Architecture group of LRI has worked on several studies to investigate design key-points: the *processing element* [CAP90], the *network architecture* and the *packaging* [GER89] [BEC89], the *routing strategy* [GER90a] [GER90b] and the *execution model* [CAP91].

The last point is devoted to language implementation issues requested to exploit fine-grained parallelism and their repercussions on the architectural level. Actors languages have earned a reputation of being able to express the full parallelism of an application. The aim of this paper is to present motivations and considerations arising in the implementation of an actor language on MEGA.

The following section describes the MEGA underlying architecture, CPU and network. The second section presents briefly the actors concepts and our motivations in implementing them. Next we present the canonical actor language used in this study, **OAL**. The fourth section

exposes the implementation of this language on MEGA and the simulator used to evaluate them. Finally we discuss the suitability of the actor paradigm to achieve massive parallelism.

II. A machine to explore giant architecture: MEGA

One of the most attractive architectural models for AI applications are message-passing architectures: they are characterized by an asynchronous MIMD control, distributed local memories and message-passing communications.

MEGA focus on fine grained parallelism relying on thousands of processing elements interconnected in a 3D grid. VLSI technological progress makes possible the integration of the entire network node, including CPU, memory and routing hardware, on a single chip [COR87] [ATH88] [GER89]. This modular approach minimizes the number of wiring interconnections making possible the structural feasibility of machines with up to 10^6 processing elements (PE).

II.1 The processing element

The CPU executes a reduced instruction set, *MegaTalk* [CAP90], designed to allow easy compilation and compact representation of a lexical LISP dialect (as for example based on Scheme [STEE75]). A detailed description of the CPU can be found in [CAP90]. The CPU access code and data are stored in on-chip memory. This feature severely constrains the memory size (between 4K and 64K following the actual technology) but drastically speeds up the memory accesses [STA86]. Thus, all instructions, including memory accesses, are performed in one clock cycle of less than 80ns with a standard 1.5 µm CMOS technology, achieving 8 Mips [BEC90].

II.2 Network and communication

The topology is a 3D grid

The monochip also includes a hardwired router dedicated to the routing management in a three-dimensional grid. The choice of a 3D-grid is not naive: in recent works on very large parallel machines, grids have been shown to be more efficient than hypercubes for very large networks. Roughly, this is due to the fact that they take into account technologically-limited resources such as wiring density [DAL87] or available pin number [REE87].

A hardwired mailing system

With very fine-grained parallelism, the average number of instructions executed between two message-passing operations tends to be small and therefore the network delay becomes a bottleneck for the whole system. If the message delivery time is too large, the processors have to wait for messages and they will stay idle. This constraint is known as the communication-calculus equilibrium [REE87]. Therefore, when massive parallelism is concerned, software message routing as in the Cosmic Cube [SEI85] and Intel iPSC/1 [INT86] is no longer possible, because it degrades the performance strongly [GRU87]. The routing tools must be embedded in hardware such as in iPSC/2 [NUG88]. Thus the main constraint on routing strategies relies on hardwired implementation feasibility.

A new routing algorithm, the *forced routing* [GER90a], is used in MEGA. It is a tradeoff between deterministic (e.g. *greedy* routing: geometrical dimensions of the network are ordered, and routing follows this order) and randomized routing [VAL82] [LEI88]. No buffering capacity is

needed at the nodes and the algorithm is quite simple, thus well adapted to hardwired implementation [BEC90]. In non-conflicting case, messages are randomly spread on the paths of equal length insuring maximum efficiency in the use of network links. When a conflict for an output-link appears, all requesting messages are routed, possibly along directions moving them away from their destination. In this manner, when there is no contention, messages follow a shortest path and when contention increases, they are randomly spread in the network.

At programmer level, two kind of messages are offered by the hardware: *direct* and *defered* messages. A direct message embed the memory location where it must be stored. A deferred message has a fixed size and is stored at reception in a hardware managed queue. In case of overflow, the message is re-routed in the network and will come back later.

III. Actor languages

III.1 Actor concepts

Actors are a message passing based model of computation. An actor is a self-contained entity with its own processing power. The computation is performed by sending and processing messages in parallel between actors. The reception of a message triggers the execution of the current *behaviour* of the receiver. Processing a communication results possibly in:

- some simple computations (*arithmetic operations* and the *conditional* control structure);
- the sending of messages to others actors (*send* command);
- the creation of new actors (*new* command);
- the specification of the *behaviour* which will govern the response to the next message (*become* command).

An *unserialized* actor, opposed to *serialized* actor, cannot change its internal state:. the same behaviour and the same *acquaintance list* (the actors with which an actor is able to communicate by sending messages) are used to respond to any message. So message processing can be done in parallel.

The concept of actor abstracts the notion of process, function or data structure. Patterns of passing messages represent various control structures; data structures and assignment are accomplished through the mechanisms of replacement behaviour and acquaintances list [HEW76]. (We will not enter more into details, the interested reader can find references in §III.3.)

III.2 Motivations

Among the available models of parallel execution, many features make actors very attractive to be implemented on a massive MIMD machine.

Abstracting the hardware

Actor communication is very close to that of MEGA. The assumptions underlying the mailing system are the same as provided by the hardwired routing of MEGA (guarantee of delivery but arrival non-determinism). Moreover, the granularity in an actor language like *Act1* or *Act2* (message size and number, task size and number) is well fitted to the number and power of available processing elements of MEGA. Therefore, an actor language can abstract a specific MEGA machine, hiding implementation details while staying a priori near enough, to enable a pertinent driving of the hardware resources.

Expressive power

Actor languages provide a natural expression of the parallelism and the distribution. These expression facilities are strengthened by theoretical results showing equivalence between actors and PRAM[1] [BVN91] [EPP88] [VAL89]. It means that problems efficiently solved in the PRAM framework are also efficiently solved by an actor program. A lot of research have been done on PRAM algorithm and interesting results exist (the set of problems efficiently solved by a PRAM algorithm is not an empty one).

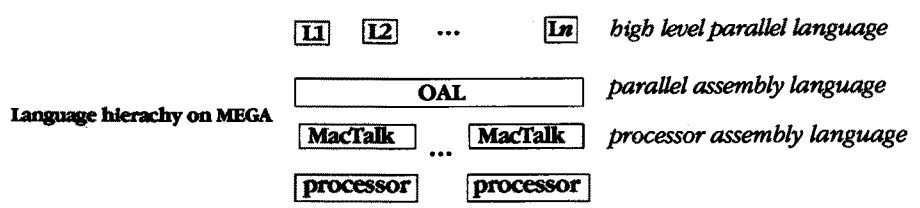
Moreover, the expressive power of actor languages make them able to **emulate execution model** of other programming paradigms. For example, *continuation* can be used to implement the function call paradigm. Function call differs from actor message sending in that sending a message does not return a value. Continuation consist in putting an additional argument in the message, the receiver of the answer that will be elaborated in response to the message. So actors used together with continuation, realize a *demand-driven evaluation* scheme [BUR81] [TRE82] (evaluation of arguments of a function call is done in parallel).

Replacement behaviour is a cumbersome but powerful construct. It allows the implementation of more sophisticated communication schemes such as those required of *eager* and *lazy* evaluation. More elaborated communication primitives (e.g. *now* or *future* in ABCL [YON86a]) can be implemented using the basic primitives "become" and "send". Explicit translation, however, is obviously to hard to be done manually: the script of an actor must be broken in several parts linked by become commands (in addition, its acquaintance list must be copied through behaviour replacement).

Actor that does not change its script, acts like an object (the "become" command is just used to change acquaintances value). The part of the script dedicated to process a certain kind of messages correspond to a method. *Delegation* can be used to achieve inheritance: when a message is outside the domain of an actor, it is dispatched to a supply actor, the delegation, that can represent the "inherited part" of the object. See also works around POOL [AME88] about the combination of structuring mechanisms of object-oriented programming with the facilities for parallelism.

Finally, actor languages are not only able to process various functional evaluation scheme but also provide an effective support for *reactive systems* [PNU86] that is, systems that does not give an output for data in input (transformational systems) but that react at their environment stimuli.

So, an actor language must be seen as a *parallel assembly language*, the target for the compilation of more sophisticated languages :



[1] More precisely, if a problem is solved by a CRCW-PRAM algorithm in time T(n) and in space S(n), then it exists an actor program solving the problem in time T(n)log(n) with high probability and in actor size complexity S(n) (cf. [BVN91]).

III.3 OAL in the actor languages family

PLASMA (Planner-like Language and System Model on Actor) is the ancestor of the family of actor languages [HEW75] and is purely sequential. Concurrence is introduced with the next generation, *PlasmaII*, and has motivated the development of the experimental language *Act1* [LIB81], a language focussed on message passing and behaviour mechanisms (the *PlasmaIII* and *Alog* [CAR84] descendents of PlasmaII are more aimed at the investigation of message filtering and logic in terms of actors). A denotational semantic of Act1 was given through the semantic of *Atolia* [CLIN81], a simplified form of Act1. Actor languages are usually seen as a kernel system to build more sophisticated languages. An example is given with *Omega* [ATT85] (a description and deduction system), and *Ether* [KOR79] (a reasoning system) built on top of Act1. They are aimed at knowledge representation and dedicated to ai-applications. Another example is *ABCL* [YON86b], an object-oriented computation model inheriting some concepts (message passing) from the actor computation model. The *Act2* [THE83] programming language blends kernel concepts from Act1, Omega and Ether to provide an extensible framework in which additional concepts (such as in the *Prelude* system) can be embedded. Act2 was effectively implemented in *Scrippter* a language running on a network of LispMachines emulating an actor-dedicated architecture: *Apiary* [HEW80]. *Act* and *Sal*, close to Act2, are used by [AGH85] to provide a transition-based semantics to actors. Sal is generally regarded as the minimal actor language and is also the basis of *OAL*.

OAL is a canonical actor language very close of *SAL* [AGH85] *TOONS* [ESP] or *SCRIPTS* [LIT90]. Some restrictions have been made in regard to SAL semantics for sake of simplicity and to respect staticity. The idea behind is to have a compiled language where most of the work (type checking, message dispatching, etc) can be done statically at compile-time.

The notable exception, with respect to the SAL semantic, is that the argument of a become command can only be a behaviour name (a new-expression in SAL terminology): in SAL, an actor can "become" another actor, meaning that all mail of the first is forwarded to the second. This can be done explicitly by hand in OAL but not implicitly. Implementation of such a "become" implies dynamic type checking.

We have to note the absence of *unserialized actors*: indeed, qualifying as unserialized an actor, as no influence on its implementation (message received by an unserialized actor are implicitly serialized for their processing, cf. §IV). A feature we have added in OAL, is actor *suicide*, for explicit release of resources owned by an actor. A more detailed description of OAL with its semantics can be found in [FOW90].

IV. Implementation issues

We describe here a possible implementation of OAL on the MEGA machine. Although OAL is not actually running on a MEGA machine, OAL programs can be processed and evaluated on a dedicated simulator. Several simulators of MEGA exist serving different purposes: *VLSI* simulation, evaluation of *MegaTalk* (the RISC instruction set), evaluation of the *forced routing* strategy, evaluation of the OAL implementation (see further).

IV.1 The actor representation

Each instance of an actor is represented by a data structure residing on a unique processor. This data structure contains the acquaintances of the actor, a link to the message input queue and to the current script. Because OAL "become" is static, we are able to determine at compile time

the set of behaviours reachable from a given one and so we can compute the size needed to store the largest acquaintances list. So, acquaintances are accessed just as structure fields.

The current script consist of pieces of code glued together with a switch statement which makes the dispatch of the messages. Each case of the switch process a message type.

Each actor is referenced through an *auid* (actor unique identifier). The auid of an actor is composed at creation time and is the juxtaposition of a local number (an index in the actor table of the processor) and the processor address (3 bytes corresponding to its x,y,z coordinates). So the size of an auid is 5 bytes and more precisely for boundary alignment reasons, 3 words. It was considered to compact the processor address to gain 3 bits and to restrict the actors number in a processor to 512. With these assumptions, the size of a auid is lowered to 2 words. But the gain in memory resources is lost by the coding/decoding operations necessary to compact/ uncompact the processor address when sending a message.

IV.2 The management of the CPU

The processing of a message by an actor creates a task. A task is managed through a *workspace*. The workspace is a structure 16 bytes long that contains the PSW (program status word of the task), the entry in the actor table referring to the actor responsible for the task, a reference to the acquaintance structure and a reference to the message. 8 additional bytes are used for temporary results computed during the processing of a message.

Tasks are scheduled following a simple *reactive* strategy as in the original operating system of the Cosmic Cube [SEI88]. Messages are processed in their order of arrivals. Because actor scripts are ensured to end in a bounded time, no preemptive mechanism is needed. The scheduler is part of a common *toolbox* resident on each processor (the toolbox summarizes the common routines needed to perform an OAL program). Actors resident on the processor are accessed by the toolbox through the actor table. Previous evaluation [GER90c] has showed that the size of the code of the scheduler (workspace management and scheduling) is about 0.5K.

IV.3 The mailing system

The purpose of the mailing system is to implement OAL messages on top of MEGA messages. The problem is not to split too long messages (cf. §IV.5) but to ensure the unlimited buffering assumption of the language.

The "send" statement in an OAL script submit the message to the mail service of a processor. Actions performed by the mail service take place between task scheduling. Each OAL message results in 3 MEGA messages managed by the toolbox.

When the toolbox has to send an OAL message, it first requests of the (toolbox of the) receiver the right to send its message. The response is delayed until the receiver has room enough to store the incoming message. The acknowledgement contains the remote memory address used to receive the OAL message and cause the final sending. Three messages have been exchanged: one "deferred" and two "direct" (see §II.2).

It appears more and more that the constraints ensured by the actor mailing system are very weak, making cumbersome even the coding of a trivial algorithm. Languages like ABCL or CANTOR [ATH87] assume a stronger constraint on message arrival: message order is preserved

for messages sent from one actor to another one. This can be implemented adding a stamp field to each message. A table is linked with each actor, keeping track of number of messages sent to a given receiver. This count is used to give a value to the stamp field of the request message. The receiver delays its acknowledgement until it receives the previous communications.

IV.4 Load-balancing strategy

Load-balancing consist in spreading the work among the processors with the goal of maximizing task processing throughput (the overall throughput is considered as more important that the response time of an individual actor) [AMS84] [REE87]. As processors have very small memory resources, load-balancing is also crucial for MEGA not only to speed-up the program execution, but *to make possible* the processing of memory consuming application.

To reach these goals, we have to adjust two opposite effects: task processing throughput is increased and processor memory usage is reduced if the tasks are distributed among the available processors. But distributing the tasks also increases buffering and communication cost, slowing down task processing and using memory. So, a good load balancing must preserve *locality*.

Because OAL tasks are very small (few lines of code), the cost of migrating a task during its execution, is too expensive. Creating a task on another processor from where the concerned actor resides, implies at least to copy actor acquaintances and so are too much expensive too. The only solution is to keep the task on the same processor as the concerned actor. Thus, load-balancing is achieved through the distribution of actors.

Our current hypothesis is that an actor cannot migrate during its life from a processor to another for the following reasons. As a matter of fact, actor migration must be transparent to the programmer: actors are light weight dynamic entities and the user must be unaware of their physical positions. The decision to move an actor is thus done by the system. It is not possible to tell each actor concerned by the move because these actors are not explicitly known by the system (this is the inverse of the relationship maintained through the acquaintance lists). So when the system has to move an actor, it must leave a *forwarder* that ensures the mail delivery. And the benefit of moving the actor (reducing the task load of one processor) is lost by the increase of the network load and will not reduce memory usage.

After this consideration, it must be clear that, within our framework, only few solutions exist to control the load-balancing. The only parameter upon which we can act is the physical processor allocated to an actor at creation time (statement "new"). *Static load-balancing* is possible in OAL with the optional keyword "at" which explicitly gives the processor where to create the new actor. *Dynamic balancing* is done in absence of the previous keyword. It consist of the creation of the actor on a processor selected following some criteria (cf. §IV.6).

The creation of an actor on an another processor uses a communication protocol between toolboxes. For sake of simplicity, the simulator assumes that actor scripts are available on each processor. Protocol for code migration have not been considered.

IV.5 The management of the memory

Data structure and space limits

The expected average number of arguments in a message is 2.5 (cf. fig. 2) and we must add the auid of the receiver. That leads to a typical message not exceeding 20 bytes length: thus there is no problem to code an actor message in a MEGA message of maximum size of 64. Pathological messages resulting in overflow of this hard limit, are compile time errors.

The acquaintances list of an actor have an expected average length of 16 bytes (see the following table: the size of the OAL programs corpus is not significant enough so we consider the indication extrapolated from a large sized object-oriented application). The data structure representing an actor must be increased by the pointer to the current behaviour script. The average length of an actor also does not exceed 20 bytes. This implies a maximum of 200 actors resident on a small MEGA processor with 4Kb local memory and 3000 actors for a 62Kb local memory. Obviously, this estimation must be reduced taking into account the space occupied by the code, the space used for the input queues and the space used for the common services. With the hypothesis of 1000 actors per processor, a small cube of edge 10 may support one million actors and a full cube of edge 100 may support 10^9 actors.

	OAL examples library	object-oriented application (in C++)
acquaintances or slots in byte	**5.2** 223/127	**15.6** 97+126/33+33
arguments number	**2.4**	**2.4** 1051args/435methods

Static analysis of typical code providing evaluation of expected actor size and arguments message number. The ratio (223/127) for the OAL library refer to 223 acquaintance for 127 new statements. In the "object" column, 97+126 refer to 97 references to other objects and 126 slots of basic types. 33+33 stand for 33 root classes and 33 derivations. For transposition in the actor scheme, a reference is counted for 6 bytes and a reference is added for each derivation class (delegation). A basic type is coded in 2 bytes.

The Garbage Collector problem

Actor programs create a lot of short-lived actors (cf. to simulation results) and a *garbage collector* is needed to minimize the amount of memory used. The "suicide" OAL feature permits explicit deletion of resources used by an actor but is dangerous from a software engineering point of view: if a message arrives after the death of an actor, unpredictable results occur (dead actors are similar to pending reference in C). So, "suicide" can be only used to remove actors statically known as useless (for example, the programmer knows that some kind of actors have only two messages to process as arising in a dichotomic search). Therefore, an implicit system is necessary for garbage collecting distributed memory.

Here are the properties we expect from a good GC algorithm [COU89] [LIB83]: locality and minimization of communications, minimization of synchronisation needed, easy processing of the message on the way, earlier deletion of inacessible resources, independence from physical parameters (network size and topology) and cheap processing.

Mark-scan parallel garbage collection algorithm [HUD82] [HUG85] has several drawbacks. The first is certainly that they do not respond the important property of maximizing available memory: actors may be deleted long after they become inaccessible. In addition, synchronization constraints are needed between processors.

Reference counting methods [BEC86] [BEV87] [WAT87] answer to the previous objection and also solve the problem of "flying" references (actor referenced in a message on the way). But they have other shortcommings: actors part of a circular structure, are not deleted at all. Moreover, because actors are light entities, the cost of an additional counter is high.

A large amount of research are currently done in distributed garbage collection area. At this point, the garbage collector of OAL is an open question. Future work would be done to evaluate more precisely existing algorithms.

IV.6 Simulation

The simulator

The MEGA simulator used in this study consist of a set of *C++* classes [STR87] modelling a MEGA machine: `processor`, `memory`, `message`, and `task`. Messages are similar to events in a discrete event simulation. An additional kind of object was defined, the basic `actor`, able to transform a message in a task.

The compilation of an OAL program results in a set of C++ classes inheriting from the classes `actor` and `message`. A new class inheriting of `actor` is defined for each ***head behaviour*** in an OAL program (a head behaviour corresponds to a behaviour appearing in a "new" statement). A derivation of `message` is produced for each kind of message recognized by a behaviour.

The C++ compilation of the translation of an OAL program is linked with the simulator itself to produce the final programme. Size, topology of the network and dynamic load-balancing strategy, are run-time parameters. The results of an execution are the messages received by the ***external*** actor `Output` and some statistics summarizing memory occupation, processor activity, network load, etc.

The adopted simulation scheme has several advantages. The compiled approach enables simulation of bigger size than possible through the direct interpretation of an OAL program (especially w.r.t. the number of processors). For example, one of our test program generates a binary tree of 32000 leafs and sorts it in parallel. This program mimics a realistic application, requiring more than 9.10^6 message exchanges. The simulation of this program on a 512-processors sized MEGA is possible and take less than 8 hours on a Sparc.

Moreover, the compiler allows a strict "type checking" (no run-time error of kind "unexpected message type") and furnishes interesting characteristics (actors size, average argument number, etc). At last but not least, the translation time is not annoying for the user (700 OAL lines/s) and the generated C++ is in a human readable form.

The simulations

The goal of the simulation was to quantify the execution of actors programs in term of resource consomption in order to compare with respect to efficiency the actor execution model against other paradigms.

Two main dynamic load-balancing strategies are under investigation. The first strategy consists in minimizing the number of actor residing on a processor (L1); the second one minimizes the number of current tasks on a processor (L2). These numbers are immediatly available on a processor. The following table shows the results for two applications (a ***beta-reduction*** [HIL86] of

4096 elements and a parallel bitonic sort [GIB 88] of 512 elements) and for various network size (this last parameters can correspond to a variable processor load). Each entry of the table gives the results for the *L1* and the L2 strategies. The L2 strategy is not always the best with respect to the throughput (the criterium "max processed task and max waiting task per processor") but minimizes maximum memory occupation.

L1 L2	1	512	1000	4096	8000	**network size**
Parallel summation of 4096 elements	*90K / 90K* 90K / 90K	*1 / 80* 6 / 40	*1 / 80* 3 / 40	*1 / 80* 0 / 40	*0 / 80* 0 / 40	**memory occupation** average / max
	0 / 0 0 / 0	*0 / 1* 0 / 1	*0 / 1* 0 / 1	*0 / 1* 0 / 1	*0 / 1* 0 / 1	**message path lenght** average / max
	16K / 4K 16K / 4K	*6K / 4* 1386 / 2	*6K / 4* 1386 / 2	*6K / 4* 1386 / 2	*6K / 4* 1386 / 2	**"througput"** max processed tasks / max waiting tasks
Parallel bitonic sort of 512 elements and array management	*21K / 21K* 21K / 21K	*49 / 12K* 138 / 11K	*25 / 12K* 79 / 4K	*6 / 12K* 25 / 12K	*3 / 12K* 13 / 12K	**memory occupation** average / max
	0 / 0 0 / 0	*0 / 10* 1 / 20	*0 / 10* 1 / 22	*0 / 10* 1 / 28	*0 / 10* 1 / 28	**message path lenght** average / max
	21K / 769 21K / 769	*12K / 512* 11K / 492	*12K / 512* 189 / 4K	*12K / 512* 513 / 12K	*12K / 512* 513 / 12K	**"througput"** max processed tasks / max waiting tasks

The most important remark about the previous results is that increasing the network size has very little effect to *maximum caracteristics* (memory occupation, waiting tasks...). That is, the load-balancing strategy is unable to spread the computation load homogeneously over the network. The reasons of this bottleneck will be discussed in the next section.

The simulation also help us to verify an important hypothesis about load-balancing. The previous simulation assumes that the load-balancing algorithm possesses complete knowledge of the global network state. In practice only limited information is available. However, it appears that state knowledge of nodes within a small distance suffices to approximate the result provided by the global knowledge hypothesis (the *event horizon* effect) [REE87] [LIN85]. The following table compares the two load-balancing strategies under the two hypothesis in case of extreme contention (small network, big job). The conclusion is that local hypothesis favours average characteristics a little bit but leads to more heterogeneous processing. This was predictable because if the event horizon is too large, the load charge reduction is offset by the increased cost of communication.

global local	L1	L2	
// bitonic sort of 32000 elements on 512 processors	*4 / 100* 3 / 786	*4 / 156* 3 / 710	**/K memory occupation** average / max
	10 / 21 1 / 16	*5 / 21* 1 / 21	**message path lenght** average / max
	5000 / 4 868 / 32	*59 / 6* 113 / 29	**/K max processed tasks/ max waiting tasks**

The previous programs create respectively 8K, 60K and 620K actors (i.e. more than the available processors). This stresses the idea of having an efficient garbage collector. Using the "suicide" feature in OAL, reduces the number of living actors to 1 at the end of the beta-reduction and to 3000 for the second application . But this technique is not always possible.

V. Conclusion

Actor implementation on MEGA

The simulation we are currently working with shows the feasibility of implementing actors on MEGA. The severe constraints on memory space can be managed through a relevant load-balancing strategy that minimizes processor memory occupation and load charge.

However, the test programs we have developed in OAL are space and time consuming in regards to other parallel language implementations (refer to [GER90c]), and this, even in the same functional programming style (there is for example no need for garbage-collecting processes in a demand-driven execution scheme). Consequently, the use of actors to achieve massive parallelism is questionable.

Actors and (massive) parallelism

To analyze the suitability or unsuitability of actors to massive computing, we first make a distinction between three kinds of parallelism usable to speed-up the execution of a program [SAN90]. *Data-parallelism* is the ability to process homogeneous set of data in an atomic operation. It is the parallelism exploited for example in *Lisp [HIL85] or ParalationLisp. *Control-parallelism* is the parallel processing of multiple threads of control. This parallelism is present in Occam, //Pascal, etc. The *flow-parallelism* is the one used through the multiple stages of a systolic architecture using a pipe-line effect.

The actor mechanisms are primitive enough to handle the various sources of parallelism in an application. Actors are able to express and to use the three sources of speed-up: one actor per "data element" correspond to the data-parallel paradigm, one actor per process correspond to the parallel control scheme while acquaintances list can be used to represent the fan-out of a pipe-line stage. However, this expressive power sacrifices the language efficiency on realistic computers. Indeed, opposite characteristics have to be joined to implement the actor paradigm. For example, data-parallel operations need dynamicity and accommodate well to synchronicity while control schemes are static and fitted to asynchronous execution model. Restricting also the actor programming to a given style, cannot gain against the more constrained assumptions made in one of the more specialized scheme. For example, the cost of doing an operation for each of n data-elements (*alpha-notation* [HIL86]) is the cost of the scalar operation in a data-parallel language based on a SIMD model. To achieve the same result within the actor language, n messages must be exchanged.

In addition, resource management cannot be done in a fine manner: the usual resource management unit is the task, as being the resource consumer. But a task does not correspond to an actor, it correspond to the handling of a message by an actor. Actor management is a poor substitute to task management (some actors are the receiver of thousand messages while other are very ephemeral entities, waiting for just one message before dying). Also an actor cannot be viewed as cluster of tasks (no communication locality in this cluster, no control over CPU/network, memory/network tradeoff, ...).

In conclusion, the simulation we are currently working with, rules out the idea to use an actor language as a fine-grain parallel assembly language, a harness for more elaborated system. Although actors express very well distributed application and concurrency, they are not able to manage resource consumption in a fine way.

Acknowledgement

The MEGA project is developed within "Computer Architecture and VLSI Design" Research Group. The authors do thank the other members of this group: Dr D Etiemble and Dr J-P Sansonnet for their outstanding contribution to the project, F. Capello who worked on the architecture of the CPU and J-L Bechenec for VLSI development and many helpful discussions. We also thank the referees for their comments and helpful corrections. This work is currently supported by the french national research program on New Computer Architectures (PRC-ANM) and by DRET under grant #89342320047050. The stay of Mr. Fowler was possible thanks to the Erasmus EEC programs.

Bibliography

[AGH85] G. Agha, "*Actors: a model for concurrent computation in distributed systems*", AI tech. rep. 844, MIT, 1985.

[AME88] P. America, "*POOL-T: A Parallel Object-Oriented Language*", in Object-Oriented Concurrent Programming, eds. A. Yonezawa, M. Tokoro, MIT Press 1988.

[AMS87] J. Amsterdam, "*Load Balancing Strategies for the Apiary*", dissertation for the degree of Bachelor, Hardvard College, May 1984.

[ATH87] W.C. Athas, "*Fine Grain Concurrent Computations*", Tech. Rep 5242, Dep. of Computer Science, California Institue of Technology, May 1987.

[ATH88] W.C. Athas, C.L. Seitz, "Multicomputers : Message-Passing Concurrent Computers", IEEE Computer, vol. 21, n° 8, August 1988, pp 9-24

[ATT85] G. Attardi "*Building Expert Systems with Omega*", DELPHI, tech. rep. ESP/85/2, 1985.

[BEC86] M.J. Beckerle, K. Ekanadham, "*Distributed Garbage Collection with no Global Synchronisation*", IBM research report RC 11667 (#52377) january 1986.

[BEC89] J-L. Béchennec, "*MegaPack : a 3D Packaging for Massively Parallel Computers*", LRI-Archi TR 89-07-1989

[BEC90] J-L. Béchennec, C. Chanussot, V. Neri and D. Etiemble, "*VLSI Design of a 3-D Highly parallel message passing architecture*", International Workshop on VLSI design for Artificial Intelligence and Neural Networks, Septembre 90

[BEV87] D. I. Bevan, "*Distributed Garbage Collection Algorithm using Reference Counts*", ACM trans. on prog. lang. and syst., vol 2, n°3, july 87.

[BUR81] W.F. Burton, M.R. Sleep, "*Executing functional programms on a virtual tree of processors*", Proc. ACM Conference on Functionnal programming langages and computer Architecture - 1981 pp 187-194

[BVN91] F. Baude, G. Vidal-Naquet, "*Actors as a parallel programming model*", to appear in STACS91.

[CAR84] F. Carré, "*Alog: acteurs et programation en logique*" (Alog: actors and logic programming) Thèse de docteur ingénieur, juin 1984 (in french).

[CAP90] F. Cappello, J-L Bechennec, D. Etiemble "*A RISC Central Processing Unit for a Massively Parallel Architecture*", EUROMICRO 90, Amsterdam, August 90

[CAP91] F. Cappello, C. Germain, J-P. Sansonnet, "*Design of a reduced instruction set for massively parallel functional programming*", LRI-Archi TR 90-07, also submitted to publication.

[CLIN81] W.D. Clinger, "*Foundation of Actor Semantics*", PhD thesis, MIT May 1987 (ai-tr-633).

[COR87] R. Cornu-Emieux, G. Mazaré, P. Objois, "*A VLSI asynchronous cellular array to accelerate logical simulations*", proc. of the 30th. Midwest Internationnal Symposium on Circuit and Systems, 1987.

[COU89] A; Couvert, A. Maddi, R. Pédrono "*Object Sharing in Distributed Systems - Principles of garbage collection*", IRISA, INRIA report 963, January 1989 (in french)

[DAL87] W.J. Dally, "*Wire-Efficient VLSI Multiprocessor Communication Networks*", 1987 Stanford Conference on Advanced Research in VLSI, 1987, pp 391-415

[DAL88] W.J. Dally, "*Fine-Grain Message-Passing Concurrent Computers*", proc. of the Third Conference on Hypercubes Concurrent Computers and Applications, vol. 1, Pasadena, January 19-20, 1988

[EPP88] D. Eppstein, Z. Galil, "*Parallel algorithmic Technic for Combinatorial Computation*" Ann. rev. Compt. Sci., 3:233-283, 1988

[ESP89] Esprit Project P440, "*Final Report*", December 1989.

[FOW90] J. Fowler, "*Studies of algorithms adapted to a network of dynamic processes*", University of Edinburgh, M.Sc Report September 1990.

[GER89] C. Germain, J-L. Béchennec, D. Etiemble, J-P. Sansonnet, "*A New Communication Design for Massively Parallel Message-Passing Architectures*", IFIP Working Conf. on Decentralized Systems 1989, North-Holland ed.

[GER90a] C. Germain, J-L Béchennec, D. Etiemble, J-P. Sansonnet, "*An Interconnection Network and a Routing Scheme for a Massively Parallel Message-Passing Multicomputer*", Third Symp. on Frontiers 90 conference on Massively Parallel Computation, October 8-10 College Park, MD

[GER90b] C. Germain, J-L. Giavitto, "*A Comparaison of Two Routing strategy for Massively Parallel Computers*", 5th International Symposium on Computer and Information Science, Capadoccia, Nov. 90.

[GER90c] C. Germain, J-L. Giavitto, J-P. Sansonnet, "*Implementation d'un paradigme de programmation fonctionelle sur une machine massivement parallele*" (implementation of a paradigmatic functionnal programming style on a massively parallel computer), LRI-Archi TR 90-07, also submitted to publication (in french).

[GIB 88] A. Gibbons, W. Rytter, "*Efficient parallel algorithms*" Cambridge University Press - 1988, (chap. 5)

[GRU87] D.C.Grunwald, D.A. Reed, "*Benchmarking Hypercubes Hardware and Software*", Hypercube Multiprocessors 87, 1987, pp 169-177

[HEW75] C. Hewitt, B. Smith, "*A PLASMA Primer*", rough draft, 13:17 1975, MIT-AIL

[HEW76] Hewitt C., "*Viewing Control Structure as Patterns of Passing Messages*", MIT Artificial Intelligence Memo 410, December 1976

[HEW80] Hewitt C., "*Apiary multiprocessor architecture knowledge system*", prooc. of the joint SRC/Univ. of Newcastle upon Tyne Workshop on VLSI, Machine Architecture and Very High Level LAnguages, October 1980.

[HIL85] W.D. Hillis, "*The Connection Machine*", The MIT Press, 1985

[HIL86] W.D. Hillis, G.L. Steele, "*Data Parallel Algorithms*", CACM vol.29 n°12, December 1986.

[HUD82] P. Hudak, R.M. Keller, "*Garbage collection and task deletion in distributed applicative processing systems*", proc. ACM conference on Lisp and Functionnal Programming, 1982 pp 168-178.

[HUG85] J. Hughes, "*A Distributed Garbage Collection Algorithm*" proc. ACM conference on Functional Programming Languages and Computer Architecture, Nancy 1985, LNCS 201.

[INT86] INTEL Scientific Computers,"Intel iPSC System Overview", Order n° 310610-001, 1986

[KOR79] W. Kornfeld, "*Using Parallel Processing for Problem Solving*", AI Memo 561, MIT, december 1979.

[LEI88] F.T. Leighton, B. Maggs, S. Rao, "*Universal Packet Routing Algorithms*",29 st IEEE Symp. on Foudations of Computer Science,1988, pp 256-269

[LIB81] H. Lieberman, "*A preview of Act-1*", AI Memo 625, MIT AI Laboratory, 1981.

[LIB83] H. Lieberman, C. Hewitt, "*A real-time garbage collector based on the lifetimes of objects*", CACM vol. 26 n°6, pp 419-428 June 1983.

[LIN85] F.C.H. Lin, "*Load Balancing and Fault Tolerance in Applicative Systems*", Ph.D. Dissertation, Dep. of Computer Science, Univ. of Utah, 1985.

[LIT90] L. Litzler, M. Tréhel, "*The kernel of an actor language for a multi-transputer system*", ISMM Lugano, june 1990.

[NUG88] S.F. Nugent, "*The iPSC/2 Direct-Connect Communications Technology*", 3° Conf. on Hypercube Concurrent Computers and Applications, 1988

[PNU86] A. Pnueli, "*Application of temporal logic to the specification and verification of reactive systems : a survey of current trends*", LNCS, 1986.

[REE87] D.A. Reed, R.M. Fujimoto, "*Multicomputer Networks - Message-Based Parallel Processing*", The MIT Press, 1987

[SAN90] J.-P. Sansonnet, "*Concepts d'Architectures Avancées*", Tome 1, cours de DEA de l'Université d'Orsay, LRI 1990 (in french).

[SEI85] C.L. Seitz, "*The Cosmic Cube*", Com. ACM, vol. 28, n° 1, Jan. 1985, pp 22

[STA86] W. Stallings Ed., "*Reduced Instruction Set Computers Tutorial*", IEEE Computer Society Press - 1986

[STE75] G.L. Steele Jr., G. Sussman, "*Scheme : An interpreter for the extended lambda calculus*", MIT AI Lab memo 349 - 1975

[STR87] B. Stroustrup, "*The C++ Programming Language*" Addison-Weslay, 1987.

[THE83] D. G. Theriault, "*Issues in the design and implementation of Act2*", tech. rep. ai-tr èé!, MIT, June 1983.

[TRE82] P. Treleaven, D.R. Brownbridge, R.P. Hopkins, "*Data driven and Demand Driven Architectures*", ACM Computing survey Vol 14 n° 1 - 1982

[VAL82] L.G. Valiant, "*A scheme for fast parallel communication*", SIAM Jour. on Computing, vol. 11, n° 2, Mai 1982, pp 350-361

[VAL89] L.G. Valiant "*Bulk-synchronous parallel computers*" Prooc. of the A.I. and Message Passing Architecture Conference, p 15-22, London, 1989. J. Wiley.

[WAT87] P. Watson, I. Watson, "*An efficient garbage collection scheme for parallel computer*", proc. of PARLE II, LNCS 259.

[YON86a] A. Yonezawa, E. Shibayama, H. Matsuda, T. Takada, Y. Honda "*Modelling and Programming in an Object Oriented Concurent Language ABCL/1*", Research report C-75, Dept. of Information Science, Tokyo Institute of Technology, Nov. 86.

[YON86b] A. Yonezawa, H. Matsuda, E. Shibayama "*An Approach to Object Oriented Concurrent Programming: a language ABCL*", Proc. of the third Workshop on Object-Oriented Languages, Paris 1986.