REPORT ON A VISIT TO ICOT

by

Marc Snir
Department of Computer Science
Hebrew University of Jerusalem

1.0 INTRODUCTION

I visited ICOT for three weeks, from March 9th to March 29th. The main purpose of my visit was to collaborate with the teams working on the architecture of the parallel inference machines, the PIM-R and the PIM-D. I intended to check to what extend my work on the NYU Ultracomputer, and related work on parallel algorithms, is relevant to the work on parallel inference machines.

During my stay I had fruitful discussions with Mr. Rikio Onai, and with members of the PIM-R and PIM-D design teams: Moritoshi Aso, Kanae Masuda, Hajime Shimizu, and Noriyoshi Ito. I also had discussions with members of the design team of the PSI machine, and with other ICOT members. My stay was made pleasant by the help of Mr. Hiroyuki Kusama.

I gave at ICOT two seminars: The first one was a presentation of work done on the NYU Ultracomputer. In particular I presented those aspects that I believe to be relevant to the work on PIM, indeed to any

work on parallel computer architectures: memory interconnection network, organization of shared memory, process scheduling, parallel data structures and parallel algorithms, etc.

The NYU Ultracomputer was designed with numerical applications in mind, but it is not an overly specialized machine. As such, it can support with reasonable efficiency non numerical applications as well, and data-flow, or reduction oriented approach to logic programming. To some extend, the work on the NYU Ultracomputer complements the work done here on PIM: We have done much less work done on detailed processor architecture, and on programming languages. On the other hand, much more work has been done on basic parallel algorithms, parallel data structures, and coordination mechanisms.

My second talk was an informal discussion of issues in the design of parallel computers, based on my impressions from my visit at ICOT, and on my previous experience in this field.

I was very impressed by the advanced stage of development of the prototype PIM-R and PIM-D machines. With such powerful prototypes available it becomes possible to gather data on parallel execution of logic programming programs. On the other hand, it is important to consider these machines as being experimental tools. This implies that they should be as flexible as possible, so that different approaches can be tried, especially in the area of processor and memory management. They should be instrumented for data gathering (preferably, at the level of the microcode).

Some possible applications of these experimental tools, and some possible means for achieving this flexibility were discussed.

2.0 PARALLEL ALGORITHMS

Extensive work has been done on the detection of parallelism in sequential code. Most of this work has centered on numerical applications, and conventional languages, such as FORTRAN. The general outcome of such work is that there is limited scope for parallelism in the execution of usual, sequential code: the largest number of processors that can be efficiently used in parallel is in the range 8-16. Similar results were recently obtained for production systems (OPS5). Moreover, automatic detection of available parallelism (by a compiler) is a hard problem.

Several researchers have seen in this situation a proof that large scale parallelism is unlikely to be useful for most applications. The following arguments can be used to counter this pessimistic point of view.

1. Limited parallelism has been detected in current codes because people have considered small problems. When problem size is increased, more parallelism becomes available. A large scale parallel machine will not be used to solve problems that fit comfortably in present day machines, but to solve much larger problems. It is important, therefore, to extrapolate the behavior of existing codes when problem size is increased.

2. The use of a conventional programming language encourages a sequential style of programming, and introduces spurious dependencies into the code. By coding into a more advanced language (data flow, reduction driven, logic programming, etc.)

one increases the amount of available parallelism, and simplifies the task of detecting such parallelism.

3. Conventional algorithms may, indeed, exhibit only a low level of parallelism: After all, they are optimized for sequential execution. However, if the programmer uses algorithms that are suitable for parallel execution then a much higher degree of parallelism becomes available.

A few remarks concerning these arguments.

The use of an advanced programming language, such as logic programming languages, indeed encourages a programming style that is less sequential in nature, and facilitates the detection of parallelism. However, it is unlikely that this is sufficient to achieve efficient use of a large scale parallel machine. Parallelism must be present in the application studied; parallelism must be present in the algorithm used for its solution; finally, parallelism must be readily available in the code of this algorithm. "Conventional" Prolog programs are likely to exhibit only a low level of parallelism. This is supported by the results of simulations done at ICOT [TR-048]: As for conventional numerical code, the amount of available parallelism seems limited to 4-8.

This suggests the following approach to parallelism:

1. Study the application field of interest (A.I, natural language processing, knowledge processing, ...). Try to define a few "kernel" problems that recur in these applications, such that

if these basic problems are solved efficiently, then the application is supported efficiently. Typical kernel applications for numerical processing are matrix computations. For non numerical processing, searching and sorting seems to have a central position. It is not clear (to me) what are the paradigmatic problems for the applications sought at ICOT.

2. Define parallel algorithms for these central problem. Use an abstract setting to define these algorithms. At that stage one can verify whether the studied problems are amenable to efficient parallel solution, and what is required (in theoretical terms) to support such solution (can we run the algorithm efficiently using a special type of communication network; can we run it using coarse grain tasks; etc.).

   Note that it is usually easy to design parallel algorithms for a fixed number of processors that are efficient on very large problems. The real issue is: How large need a problem be in order to use efficiently a given number of processors? Or, inversely, how many processors can be efficiently applied to solve in parallel a problem of given size?

3. Encode these parallel algorithms in a concrete programming language. At that stage one may verify whether the language supports efficiently parallelism: is the parallelism inherent in the theoretical algorithm still readily detectable in the coded version of it? Is such parallelism naturally expressed in the language? Do we have adequate data structures?

4. Verify whether the algorithm (e.g. its coded version) is efficiently supported by the target architecture: Are communication mechanisms adequate? Is the scheduling policy suitable?

At each stage, the analyses and simulations should not be restricted to small problem sizes. Theoretical analysis is naturally concerned with complexity as a function of problem size. The same concern should be for the analysis and simulations of code run on the target architecture. Simulations done on small programs should be used to develop an analytical model of performance, and extrapolate performance for large problems (and large machines).

This type of "top-down" approach to parallel processing is essential to the successful design of a new parallel computer architecture. Without it there can be no adequate definition of requirements, and any proposed architecture is a shot in the dark.

In addition, such study can have the following results.

1. The development of parallel algorithms to support essential computations in logic programming, and logic programming applications.

2. The definition of basic data structures, and of parallel algorithms for their manipulations, in support of such algorithms.

3. An appraisal of Prolog as a tool for programming parallel applications, and evaluation of possible extensions.

2.1 Analysis And Simulations

Decisions on the structure of PIM-D and PIM-R, and on scheduling policies, are based on simulations of actual Prolog programs. The simulated programs are typically small (due to the limitations of the existing Dec20 prolog system, and to the lack of large applications in Prolog, or concurrent Prolog).

The properties of large Prolog programs may be quite different from the properties of small ones. The (dynamic) occurrence frequency of different predicates is likely to change (less I/O ?); the (dynamic) occurrence frequency of "structured data" or "database clauses" is likely to change; programs may grow in "width" (more OR or AND branching), or in "depth".

Moreover, in order to utilize efficiently high level parallelism it is necessary to restructure algorithms. Programs restructured to run efficiently in parallel may exhibit quite different properties than conventional "sequential" programs. Note that such restructuring is not what obtains when cuts, asserts, retracts, are mechanically removed from Prolog programs. Restructuring should increase parallelism with no significant increase in the number of operations; just removing cuts from a "sequential" Prolog program usually will increase significantly the number of operations performed.

Simulations and analysis can not be performed extensively on large programs; also, it is hard to understand results obtained from simulations of composite programs. it is important to define standard parameterized benchmarks, such as exist for data processing or numerical

processing. These benchmarks should be relevant to logic programming applications. It is not clear to me what are suitable candidates. To start with I propose to consider algorithms for sorting, merging, and related data manipulations. Other candidates are algorithms for heuristic search, algorithms for alpha-beta tree search, etc. Even if these algorithms are not typical of logic programming applications, they are well understood, and will still take a significant fraction of computing time in large applications.

Such study will also provide realistic analytical models of Prolog programs. It is expensive to study problems concerning the efficiency of a computer architecture using simulations of actual programs. For sequential architectures, problems such as caching policies, paging policies, etc., are studied using analytical models, with parameters derived from real program simulations. Such analytical models are especially important in the first stage of development, when it is desirable to test many different options. Such analytical models can be used either for analysis, or for simulations using synthetic inputs. Example of possible applications are the study of the performance of interconnection networks; the performance of different process scheduling policies; etc.

2.1.1 An Example -

To illustrate the approach suggested, let me describe a analyses of two simple algorithms that I did with the help of mr. Shimizu. The analysis was done at the level of the theoretical algorithm, and of the Prolog code, but not at the level of a concrete computer architecture.

2.1.1.1 Mergesort -

The algorithm sorts a list of length n by splitting it into two sublists of length n/2, recursively sorting them, next merging them. If a list structure is used to store data, then list copying is a linear operation. Likewise, merging is a linear operation. The two recursive calls to sort can be executed in parallel. Assuming unbounded parallelism, the time required to perform mergesort is given by the recursion

$T(n) = O(n) + T(n/2)$

which solves to

$T(n) = O(n)$.

The number of operations executed is the same as in sequential mergesort, namely

$t(n) = O(n \log n)$.

Thus, an the average, there are $O(\log(n))$ operations available for execution at each cycle.

When p processors are used the algorithm runs in time

$Tp(n) = O((n/p)\log(n/p) + n)$.

The efficiency of this parallel algorithm (the ratio between the number of operations executed in the serial case and the number of operations executed in the parallel case is equal to

$T(n) / pTp(n) = \Omega (\log(n) / (\log(n/p) + p))$.

In order to have a constant efficiency we must have

$p = O(\log(n)\log\log(n))$

or

n = Ω((c**p)/p), for some constant c.

This algorithm, therefore, is efficient only when the number of items to be sorted is exponential in the number of processors used. This implies that mergesort is not suitable for large scale parallelism: its efficiency will decrease as the amount of parallelism increases, as the input size is not likely to increase exponentially in the number of processors.

Mr. Shimizu implemented this algorithm in Prolog, and made a dynamic analysis of its execution. The table below presents the results

n	10	20	30	40
total number of reducible subgoals	220	517	838	1181
av. number of reducible subgoals per level	2.89	3.92	4.66	4.96

The total amount of work done is proportional to the total number of reducible subgoals, and behaves like n*log(n); the average amount of parallel tasks available for concurrent execution is roughly equal to the average. number of reducible subgoals per level, and increases as log(n). Thus this concrete Prolog implementation of mergesort behaves as predicted by the theoretical analysis (there is no loss of efficiency or loss of parallelism).

Note that the amount of available parallelism would be increased (by a constant factor) if a data structure that can be copied in parallel would be used, rather than a list. Also, after the initial list has been split into more than p sublists, then the algorithm essentially sorts sequentially each of the sublists. Thus, allocation

overheads can be avoided by running a sequential sort algorithm on each sublist (equivalently, by reverting to sequential Prolog when reduction depth exceeds log p).

Clearly, the analysis and simulation can be further refined, to extract more detailed information, and reflect more accurately the machine model. Also, a comparative study of (Prolog implementation of) sort programs can be carried. We do not pursue this direction as sorting is not a main stream problem for logic programming, and the theoretical analysis of sorting algorithms is well documented in the literature.

2.1.1.2 Pattern Matching -

The simple pattern matching algorithm executes, for a text of length m, and a pattern of length p, pm-p(p-1)/2 comparisons, in the worst case. The algorithm can be run in parallel, by starting in parallel to test for a match from any point in the text. This parallel algorithm requires p comparison steps.

It is not possible to start in Prolog m tasks in constant time; if the text is represented using a conventional list representation, then the m tasks must be started sequentially. We obtain an algorithm that requires O(m) steps, assuming that $\Omega(p)$ processors are available. The algorithm can use efficiently at most O(p) processors.

The simple Prolog code for this algorithm is given below.

```
match(T,P) :- seqmatch(T,P).
match([A|T],P) :- match(T,P).
```

```
seqmatch(T,[]).
seqmatch([A|T],[A|P]) :- seqmatch(T,P).
```

OR parallelism is used to generate the parallel match tests, that are then executed deterministically.

A simple analysis shows that there are m+3 levels (including the main one), and the number of reducible subgoals is
(m + (3-p)/2)(p+2).
This formula was verified by simulation by mr. Shimizu. Thus, the number of reducible subgoals per level, e.g. the amount of available parallelism is roughly equal to p+2, conforming to the theoretical results.

3.0 REMARKS ON PIM-R

3.1 Number Of Processes

The number of processes concurrently active during parallel execution of Prolog can be very large. Assume, for sake of definiteness, an execution tree with an OR (dynamic) branching factor of Bo, an AND (dynamic) branching factor of Ba, and depth 2d. The number of reductions is (roughly) equal to (Bo*Ba)**d. An existing process is either running, ready (scheduled, but not yet running), or waiting (suspended, and waiting for a message - from a child in our case). For each existing process one has to allocate space for the associated data structures (Task Control Block). Thus, we are interested to estimate the maximal number of processes that can exist at the same time (in either of the tree states).

Consider first a sequential execution (one processor).

1. If processes are scheduled according to a First In, First Out policy (normal queueing discipline) then the tree is traversed in breadth first order, and the number of concurrently existing processes can grow to (roughly) Bo**d. This happens when the (AND) root node is waiting, one of its children is waiting, all the children of this OR node are waiting, and so on, up to the last level.

2. If processes are scheduled according to First In, Last Out policy (stack used) then the tree is traversed in depth first order, and the number of concurrently existing processes can grow to (roughly) Bo*d. This happens at the maximum reduction depth: there is one active (OR) process, d-1 waiting (OR) processes, and Bo*d ready (OR) processes.

Consider now AND sequential, OR parallel execution, with p processors. Let

- I be the number of reductions executed per processor;
- Pr be the number of concurrently existing processes per processor;
- p be the number of processors.

We have

p*I = (Bo*Ba)**d .

If each processor schedules tasks with least reduction level first (pseudo breadth first), or each processor schedules locally tasks in FIFO order, or a global FIFO scheduler is used, then the tree is visited (roughly) in breadth first order, and the number of concurrently existing tasks can grow to be Bo**d. We obtain p*Pr = Bo**d,

so that

Pr = I**(1-a)/p**a,

where

a = log Ba / (log Ba + log Bo) .

Consider a machine consisting of p=100 processors, each executing 30 Klips. Each processor, running for 10 minutes, will execute I= 18 Mglips. Taking Ba=Bo, we have a = 1/2, and Pr ~ 400. Taking Bo = Ba**2, we have a = 1/3, and Pr ~ 15000.

If a global scheduler is used that schedules the processes in LIFO order, or in largest reduction level first then the number of concurrently existing processes is bounded by p*Bo*d, so that Pr<d*Bo, as in sequential depth-first reduction. This can still be significant for programs that are nearly deterministic. For example, with Bo=Ba=1.1, and 18,000,000 nodes, we still can have

>4000 independent tasks, in an AND sequential, OR parallel execution (so that the problem is suitable for parallel execution); the depth of the tree is more than 200.

No central scheduler is planned for PIM-R. There are two local scheduling policies that approximate depth-first

1. Pseudo depth-first: Each processor schedules tasks in its queue according to a largest reduction level first policy.

2. Pseudo LIFO: Each processor schedules tasks according to a last arrived first executed policy.

The number of concurrently existing tasks, when a local scheduling policy is used, depends also on the processor allocation policy used. It is expected that the number of concurrently existing tasks, when these two scheduling policies are used, is approximately equal to the number that would obtain in a true depth-first execution, but I do not have firm analysis.

We can reach two conclusions:

1. Depth-First scheduling (usually) consumes much less space than breadth-first scheduling. This is probably even more marked when only one solution is needed. Also, as programs are written with a depth-first search mechanism in mind, depth-first search will be more efficient. Simulations done for the PIE machine leads to the same conclusion.

2. The number of concurrent processes generated is very dependent on the type of scheduling used, and on the dynamic properties of the program executed. No scheduling policy can keep it small for any type of problem.

The analysis presented is very rough, as it assumes that all executed branches have the same depth. A more accurate analysis, taking into account actual dynamic properties of large Prolog programs, is needed.

Also, one should refine this analysis to estimate the number of processes in each state.

3.2 Process Control

The previous section indicates the need for controlling the number of processes generated during parallel execution of Prolog. Since the number of concurrent processes is heavily dependent on dynamic properties of the Prolog program it does not seem feasible to have this control at compile time; the number of generated processes must be controlled at run time.

One approach to this problem is that of controlling process creation by using a suitable scheduling policy. We have already pointed out that First In, Last Out will be adequate for programs with large nondeterminism. This, however, might not be enough for nearly deterministic programs. In any case, the scheduling policy does not influence the total number of processes generated (in a pure Prolog program). Process generation has a significant overhead, as it requires resource (processor and memory) allocation. Thus, it is desirable to reduce the total number of processes created.

This result can be achieved by "chunking": Each clause is, potentially, an independent process. However, the system may "chunk" several such independent processes together. Such chunk of processes will be executed serially on one processor. Since they run on the same processor they may share structures.

Chunking may be done at compile time, based on the static properties of the program. However, since the static properties of a program may be quite different from its dynamic properties, this is better done at run time.

Follows several possible chunking policies:

1. Chunk together AND related tasks; this is the default mode for Parallel Prolog on PIM.

2. Chunk together OR related tasks; this is the default mode for Concurrent Prolog on PIM.

3. Chunk together a node and all its descendants. For example, execute program sequentially when reduction level exceeds a fixed threshold.

We envisage a system where chunking is based both on static analysis of the program (e.g. the amount of sharing between AND related tasks, the amount of OR branching, etc.), on dynamic properties of the currently executed task (e.g. depth, number of unbound variables, etc.), and system-wide parameters (number of processes). The process scheduler has to decide on execution mode whenever a new goal is generated, and whenever a literal is selected. Currently the decision is fixed. We advocate an architecture where this decision is taken at run time, based on static information available at compile time, and on local and global dynamic information available at run time.

Complex analysis can be done at compile time, to decide on optimal chunking. This analysis will be used to generate the code that decide

at run time whether to generate independent tasks, or chunk. This code must be efficient, so that it will use only limited information on run time environment: mainly reduction level, and possibly restricted information on present goals and bindings. In addition global information, such as number of existing processes may be used.

I assume that decision on chunking is reversible: processes that have been chunked together can be latter separated. If the chunking policy is efficient, then the need to reverse a chunking decision will arise seldom, and one may be willing to accept a significant overhead whenever chunked processes are split.

It is often claimed that the granularity of Prolog programs is too small for efficient execution on a large scale parallel machine. One possible solution is provided by a programming language that provides for large grain modules. The other alternative we suggest is akin to program restructuring, when done statically; the possibility of late decision gives even more flexibility.

There have been a few suggestions on the literature of possible algorithms for chunking, and no analysis of their performance on large programs. I view this issue as an important research topic.

3.2.1 Implementing Chunking -

Chunking can be implemented in several ways

1. Several processes are chunked together by running them on the same processor. This saves communication over the network, but do not save the overhead of copying structures, and creating

new task control blocks.

2. Several processes are chunked together by using structure sharing, rather then copying. The processes are executed sequentially one one processor, as in usual sequential Prolog implementations. Such method saves the overhead of copying. However, it is now difficult to reverse the chunking decision: it is difficult to restore the environment of a process that is not currently running.

3. Several processes are chunked together by partially sharing their respective control blocks (and running them on one processor). There are many possible compromises: For example, one can generate the same structures that are generated for a new process with copying policy, but refrain from copying that part of the PTB that has not changed.
Specifically, we assume that a new header and a new variable area is created; we also assume that the structure area is not required to be consecutive: the address of each structure is absolute. Note that this is anyhow the case for structures stored in the SMM; if the SMM becomes writable that will be the case for most (all?) structures.

We call such policy <u>lazy copying</u>: A new PTB is created, but copying is deferred until need arise, e.g. until the task is dispatched to another processor. It is quite easy to generate a new separated PTB from the information available in the "skeleton PTB" described above.

3.2.2 Pipelining -

One possible benefit of flexible chunking is the ability to run AND related clauses in a pipelined fashion (the forwarding method). Consider a clause of the form

p :- q(x,y) , r(y) , s(z)

Presently, a process is created with goal q,r,s; for each value returned by this process, a separate process is created with goal r,s; for each value returned by this last process, a process with goal s is created. if the different processes with goal r,s that are generated as children of the first goal are chunked together, then it is possible to keep only one copy of their shared information, using a "skeleton PTB". This effectively means that the goals qrs, rs, and s are processed in a pipelined manner.

3.2.3 Load Balancing -

Note that there is a tradeoff concerning the number of processes created: The higher is the number of processes created, the higher is the overhead of process creation and resource allocation. On the other hand, the higher is the number of processes created, the easier it is to obtain a good load distribution in the system.

Analytical researches [Kruskal,Weiss] have shown that for most distributions it is preferable to have a small number of processes: the average waste of time occurring when processors idle, waiting for last processor to terminate is small compared with the overhead of splitting processes into subprocesses, and allocating these. This suggests to revert to sequential backtracking once the number of active nodes in the

computation tree is order of the number of processors. It is clearly desirable to study this issue by analysis and simulation.

3.3 Structure Memory Module

The Structure memory Module (SMM) supports three operations on lists: Read CAR (returns a pointer or an atom), read CDR (returns a pointer or an atom), read list (returns an entire list). Writes (CONS) are not supported. When an entire list is returned from memory the SMM is busy for many memory cycles. This is undesirable: it means that the busy time of the SMM has a large variance and, therefore, the average queueing delay is large. Is it possible to design an SMM such that accesses to large structures will be spread over several modules, without increasing too much the communication overhead?

The simple solution is to interleave memory space across many memory modules. Then the access of a list will require many successive memory access operations from the processor, generating a significant communication overhead. I propose an alternative tentative solution. It has not been evaluated carefully, so that I am not sure it represents an improvement.

We assume that each node in a structure has one pointer pointing to it; if there is more than one pointer pointing to a node, then this node might be replicated. The SMM's are connected in a circular manner: module i communicates with modules i+1 and i-1 (mod n). In addition, the SMM's are connected through a multistage interconnection network to the PE's. If a node is stored in module i then its CAR is stored in module i-1, and its CDR in module i+1. When a list access is performed,

module i receives a message containing the address of the root of the list. If one of the fields of the node is NIL, or atom, a value is returned to the PE; if one of the fields is a pointer, a message with the pointer value and the PE number is sent to the adjacent SMM; the process is repeated there recursively. Indirect memory access can be done in the same fashion: a message is sent to the memory module containing the pointer. This SMM sends a message to its neighbor with the pointer value and PE number. The SMM that receive the message answers the request.

If one wishes to support CON's (write operations) as well then one either have to duplicate structures whenever the two CONsed lists are not in correct SMM's. Alternatively, one can have global communication between SMM's (a pointer does not necessarily point to a location in an adjacent SMM). However, if it is the case that few new structures are created, so that most pointers still point to a location in an adjacent SMM, than one can use a global communication network between the SMM's that is optimized for communication between neighbors. An example of such network is a ring shifter. The same SMM interconnection network will also be used for garbage collection.

3.4 Exception Handling

It is harder to debug parallel programs than sequential programs. It is, therefore, essential to have hardware support for good exception handling mechanisms. Such mechanisms will allow to localize exception handling to any desired level. They will also support efficiently tracing and other debugging mechanisms (the design of efficient

debugging software for parallel systems is an entire issue by itself).

Exception handling mechanisms on a sequential machine are global: they have access to the entire job environment, and can use it to base their decisions; this environment does not change while the error is processed.

It is possible to build a "global" exception handling mechanism for a parallel machine as well: the global environment is implicitly available in the tree of processes; it can, in principle, be accessed by any process; it is in particular feasible for a process to access the environment of its ancestors.

Note, however that such access requires interprocessor communication. Moreover, the information in the TCB of a process may change while an error is processed at one of its descendants. Thus, global exception handling seems hard to support.

A local exception handling mechanism supports abnormal termination at each process; the programmer may specify error handlers for each type of error at each process; such error handler may in particular terminate the process abnormally, and return information on the error to the parent (the default being abnormal termination, with an exception flag returned). Such exception handling mechanism has to be provided by the machine language (KL1), and be supported by the firmware.

A process terminates after all its children have terminated. If an error occurred in some child, then the parent process can be terminated (abnormally) even though some children are still active. A similar situation occurs in Concurrent Prolog, if OR related tasks are executed

in parallel: if one child committed, and executed its body, then the parent process can be terminated, even though other children are still evaluating their guards.

In the currently proposed implementation a process that terminates while its children are still active is blocked, and its control block is not deleted; the children pursue their activity, and terminate normally; the parent process is deleted only when all its children have terminated.

This can cause inefficient use of processing resources. Moreover, this does not provide any means to terminate a process that deadlocks, or exceeds its allocated resources. Mechanisms that support process killing have to be provided.

If "infanticide" is rare, then child killing can be handled by software, by running "process garbage collection" code at prescribed intervals: Processes that have a dead parent are marked as dead, then deleted. Such code can probably be run without halting the machine by using concurrent garbage collection algorithms; more research is needed on this issue.

Dead process collection is facilitated if each process has pointers to all its children. In such case a parent can kill its children when it terminates abnormally (or when one of its children is committed, in Concurrent Prolog).

Such reverse links will also be useful for debugging.

4.0 KL1

In discussions with the PIM design group I found agreement that Prolog, and GHC, while being elegant languages with a clean semantics, lack many facilities that are needed to develop efficient parallel code, and to design an operating system for a parallel machine. Several of the desirable features for the machine language of PIM are

1. More efficient communication mechanisms. Presently communication is uniquely done through shared logical variables; this is not sufficient and inefficient for system programming.

2. Powerful modularization features. We already discussed the performance advantages of coarser grain tasks. A language that provides good modularization facilities encourage the programmer to design its code using modules with little interaction; such modules are natural allocation units. Also, good modularization is important for correct development of a large system.

3. Powerful data structures. Pure Prolog has only lists as data structures. List access is inherently serial; such data structure does not provide good support to parallel algorithms. The addition of vectors, sets, functions (tables), etc., would facilitate development of efficient parallel codes.

4. Facilities for real time programming (e.g. synchronized communication).

5. Facilities for exception handling. These have been already discussed.

6. Facilities for explicit resource control. For critical code, the programmer will want the ability to control explicitly the allocation of resources, e.g. the mapping of processes to processors.

Some of these different features are found in different Prolog versions: Parlog has facilities for modularization; Dr. Shapiro proposed a notation for the mapping of processes to processors; set operations occur in different versions of Prolog. The integration of all these features in one language with clean semantics is no mean issue. Dr. Reeves suggested in our discussion, that rather than attempt to incorporate all these feature in one language, one should use a two level system: at one level processes written in pure Prolog; at the next level communication mechanisms between such processes, using CSP-like semantics. Such approach, which is similar to the approach used for ESP, merits serious consideration.

The design of the machine language is the most important aspect of the design of a machine architecture. It has to be done with much awareness to the applications it will be used for, and to the technological constraints of hardware.

5.0 FUTURE COLLABORATION

I see the possibility for extensive collaboration with ICOT on the issues mentioned in this report. Many members of the Computer Science

Department at the Hebrew University, including myself, are actively engaged in research on parallel and distributed processing. This includes work on the design and analysis of parallel and distributed algorithms, work on distributed operating system, work on the semantics of distributed processing, work on distributed databases, work on coordination protocols, work on the design and evaluation of parallel computer architectures, etc.

I see the possibility for collaboration in the following areas.

1. Design and analysis of parallel algorithms. This includes in particular

    1. coordination algorithms;

    2. parallel data structures;

    3. search algorithms

    4. graph manipulation algorithm (the latest will be required in knowledge processing systems).

2. Design and analysis of parallel computer architectures. In particular

    1. design and analysis of interconnection networks; and

    2. design and analysis of memory structures;

3. Design and analysis of scheduling policies for parallel computers.

Such collaborative research would be facilitated by improved communications. In particular, it would be very helpful if ICOT gained access to CSNET, or other similar network. My department has been connected to CSNET for a year, and this connection has greatly increased the amount of collaboration with colleagues from the US: It enables us to pursue joint work initiated during personal visits, by the exchange of written comments, and versions of papers in preparation.

Curriculum Vitae

Marc Snir

Born in 1948 in Paris, France. In Israel since 1960.
Israeli Citizen.
Married and father to two children.
Military service 1967-1969.

Homeaddress:
Ramot 619, Jerusalem, Israel
(02-861190)

Businessaddress:
Institute of Mathematics and Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel. (02-585439)

Degrees:
B.Sc., Hebrew University of Jerusalem, 1972.
Ph.D., Hebrew University of Jerusalem, 1979.

Positions:
Hebrew University of Jerusalem, Teaching Assistant 1974-1977, Instructor 1977-1979.
University of Edinburgh, Dept. of Computer Science, Research Fellow 1979-1980.
New York University, Dept. of Computer Science, Ass. Professor 1980-1983.
Hebrew University of Jerusalem, Senior Lecturer 1982-.

Holding the John Cahan chair of Computer Science at the Hebrew University.

Grants:

1981 - NSF grant for research on Communication Complexity of Parallel Algorithms ($30000).

1983 - Elta Electronic Industries grant for research on Parallel Computer Architectures for Signal Processing ($2800).

1984 - National Council for Research and Development grant for research on Top Down Design of Large Hardware Systems ($40000).

1984 - Elta Electronic Industries grant for the development of a Syntax Oriented Microcode Editor ($28000).

Chairman of the Research Center on Parallel Processing and Special Computers. The center is funded by Elta to an annual amount of $65000.

Research interests:

Advanced computer architectures
Design and analysis of parallel algorithms
Software for logic design
Computational complexity

Research activities:

While at Edinburgh I contributed to ongoing research on Very Large Scale Integrated (VLSI) circuits. At NYU I was a senior member of the team that designed the NYU Ultracomputer, a parallel processor prototype with a novel architecture. I continue to contribute to the research in the Utlracomputer project during my visits in the US.

At the Hebrew University I continue my research on theoretical and practical aspects of parallel processing. I have established a research laboratory on digital hardware design. This laboratory supports research on software for digital hardware design, and research on parallel computer architectures. A parallel computer prototype is currently being designed. Fruitful collaboration has been established with several Isreali electronic manufacturers. One of them (Elta) funds a center for research on parallel processing in our department.

I have introduced new courses in the area of parallel processing, parallel computer architectures, and computer design, and have been active in many areas in the administration of the Computer Science Department.

Three PhD students and four MSc students are involved in different aspects of these research activities.

Invited lectures and sejours:

IBM Yorktown Heights (1977, 1982, 1983, 1984)
Paris VI (1980)
Univ. of Toronto (1982)
Harvard Univ. (1982)
IBM San Jose (1983)
Univ. of Berkeley (1983)
Univ. of Illinois (1983, 1984)
Carnegie-Mellon Univ. (1984)
Columbia Univ. (1984)