STATE-OF-THE-ART PROGRAMMING ENVIRONMENTS ON PARALLEL COMPUTERS

Ian Willers
CERN
Genève, Switzerland

Summary: This paper presents some of the first thoughts of the CERN group which is helping plan the GP-MIMD project. This is an Esprit project which will develop a "General Purpose Multiple Instruction Multiple Data" architecture using up to 1000 of the next generation INMOS transputers. The four major "transputer machine" companies, Meiko, Parsys, Parsytec and Telmat, plan to adopt this as a standard architecture. I will be reporting on the software implications of such a machine in the High Energy Physics environment.

1. INTRODUCTION

Since the GP-MIMD Esprit project has not yet begun, our group has no results to report. This paper will therefore talk about the motivation for High Energy Physics to be involved with such a programme; our attempt to situate the proposed architecture within the general scheme of parallel architectures; the past influence of "Farming" techniques; how message passing was not quite so simple; the Linda paradigm and its relation to problem perception; and finally our own initial thoughts on the parallelisation of a large High Energy Physics program; the FORTRAN programmers comprehension of parallelism; and the possibility of automatic parallelisation of FORTRAN code.

2. WHY IS PARALLELISM INTERESTING?

2.1 Parallelism and Supercomputers

For some years the name Cray has been synonymous with the supercomputer. The latest model, Cray Y-MP, is a shared memory multi-processor with 8 processors which still relies on vectorization to attain its peak speed. This year there were announcements in the financial and business press of the successor to the popular iPSC/80. It has 128 processors arranged in a hypercube, is known as the iPSC/860 and is based on Intel's i860 chip. The press also noted that the Connection Machine with up to 65,535 simple processors had obtained approximately 10% of the supercomputing market. What was once a well understood segment of the computing scene now seems to lack direction.

The Cray Y-MP has a peak performance (with perfect vectorization on 8 processors) of 2.7 Gigaflops and costs between $20m and $30m. The Intel iPSC/860 has 128 i860 processors with distributed memory. Each processor has its own memory and communication is via messages passing over a communications structure known as a hypercube. The peak performance of the Intel iPSC/860 is 7.6 Gigaflops and costs about $3m! The Connection Machine has 65,535 processors obeying a single stream of instructions putting it in the SIMD category. It has a peak performance of 10 Gigaflops and a price in the $8m range.

An obvious question at this point is 'Why should I buy a Cray instead of an Intel iPSC/860 or a Connection Machine?'. One answer to this question is based on the relative difficulty of

programming each of these machines in order to obtain maximum performance. All three computers have distinctive features that make it difficult to obtain 50% or even 25% of their peak performance when running many programs. On the Cray, physicists have been unsuccessful in obtaining large gains by vectorizing large numerical codes for analyzing experimental data. However, in general, the software on the Cray is more mature, vectorization is relatively well understood, the CRAY compilers are very smart and the machine excels in problems that can be vectorized. While the iPSC/860 and the Connection Machine use less advanced software tools, the 'hard-nosed' business man believes that he can buy a lot of programming for the difference in price between them and the CRAY. The iPSC/860 has been used to successfully solve problems where the data can be distributed over many processors. And the Connection Machine has already proved itself in situations where large databases are involved. It permits its many processors to simultaneously examine database records that are stored across many disks and then accessed in a parallel fashion.

2.2 Physics Need For Computer Power

A statement from Fermilab reads 'High Energy Physicists have always wanted more computer power than they could afford to buy in the commercial market place'. CERN has predicted that 600 CERN units[1] will be needed within the next two years (see CERN Green Book, 1989). This is to be divided up as 150 units in the CERN computer center, 300 units from outside organizations and 150 units from parallel farms.

The profile of computing from experiments using the LEP accelerator at CERN is shown in the graph in figure 1 which is taken from the so-called CERN Green Book. This will take up about half of the available computing power. Also note that when the Z^0 physics has been well understood there will probably be more demanding requirements from new LEP physics. The LHC (planned Large Hadron Collider) will continue to increase demand.

1: A CERN Unit corresponds roughly to 1/8 th. of the power of a single processor in the CRAY X-MP.

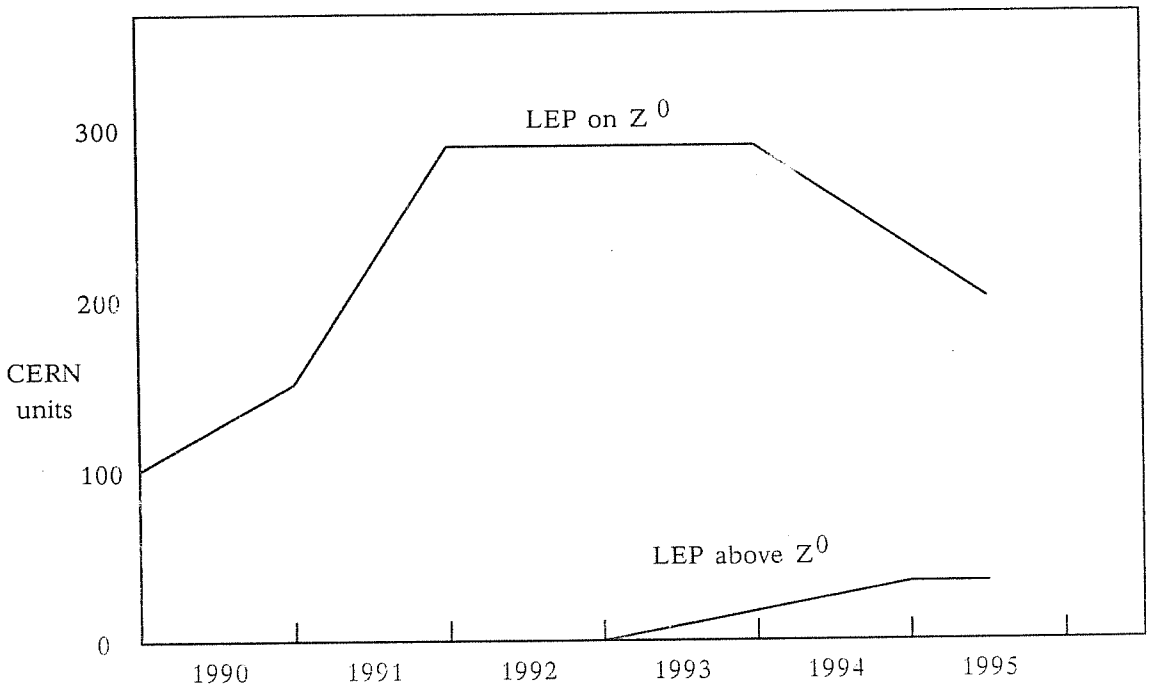

Figure 1: Total CPU needs of the LEP experments

3. CLASSIFICATION OF ARCHITECTURES

In trying to justify CERN participating in a project to produce a potentially 1000 processor distributed memory MIMD computer, we felt that we should understand the relation of such a machine to other types of parallel architectures.

3.1 Flynn's Taxonomy

Flynn's, by now, classical taxonomy gave us little insight.

SISD - Single Instruction, Single Data stream - defines the simplest computer type but rules out pipelining and superscalar machines although these look identical to the user;

MISD - Multiple Instruction, Single Data stream - defines something which, I believe, is possible but totally impractical;

SIMD - Single Instruction, Multiple Data streams - for example defines that part of the Connection Machine which is attached to a normal host but does not define the whole architecture;

MIMD - Multiple Instruction, Multiple Data streams - does not distinguish between shared memory and distributed memory.

In practical situations it is necessary to look for a more comprehensive classification.

3.2 Duncan's Taxonomy

Fortunately this was forthcoming. Ralph Duncan (See Duncan, 1990) published an article in the IEEE Computer magazine where he had tried to classify existing machines in a practical way. His

taxonomy is depicted in figure 2. As you can see it included not only the commercially interesting machines but also many machines which only appear in computer science research laboratories.

In our justification for concentrating on the distributed memory MIMD machines we felt that it was only necessary to compare with the commercially available machine types, the SIMD concept as in the Connection Machine (processor arrays) or as in ASP (associative memory), the shared memory machines MIMD and the distributed MIMD machines.

The SIMD style of machine appeared to be best in regular problems with a small grain size. There is already a CERN group and a project with external partners (see Kunt and Rohrbach, 1989), called MPPC, looking at this type of computer. This left us making a comparison between shared and distributed memory machines.

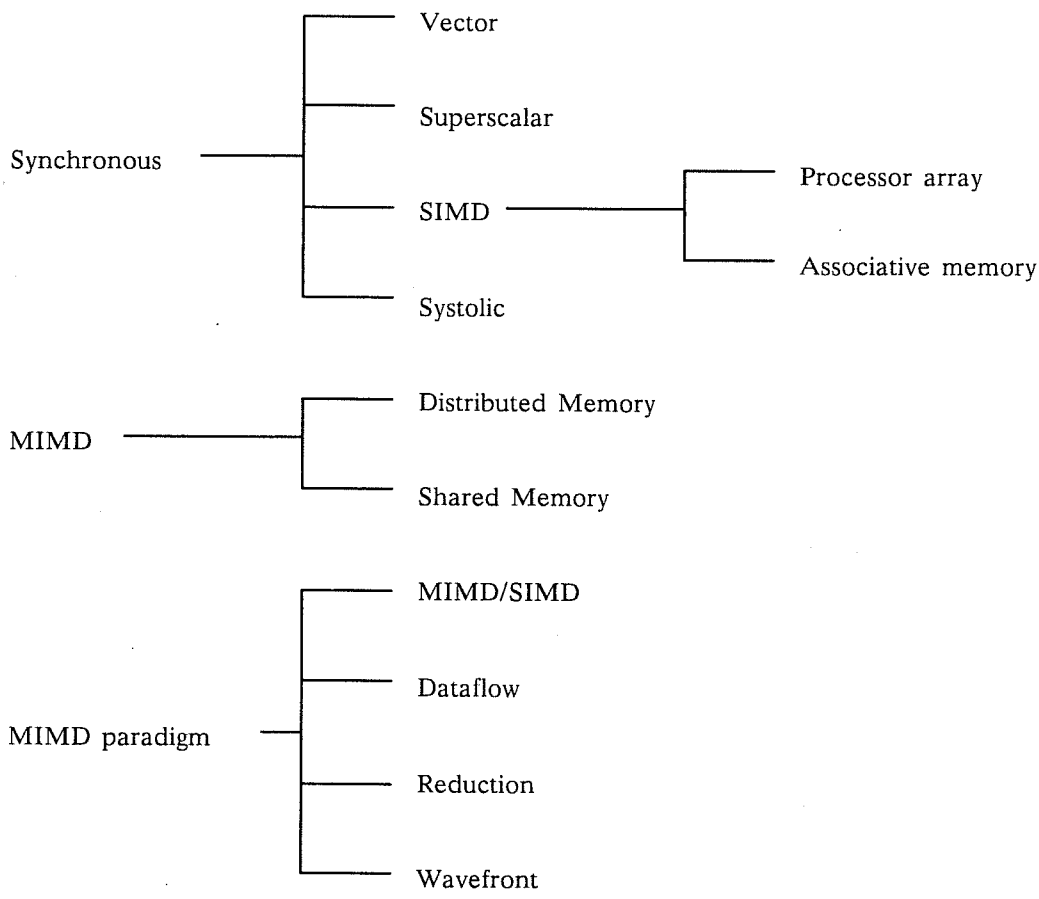

Fig. 2 Duncan's taxonomy

3.3 Shared Memory Architectures

The problem that we see with the shared memory machine is that it does not easily extend to a large number of processors. Systems work well with bus and crossbar switch technology even when cache coherency is a problem. The multistage interconnection network used in the BBN Butterfly could support many more processors and a machine with 256 nodes was constructed. However, reports on the performance of a 256 node machine indicated that the best architecture would include large amounts of cache memory where programs could run with few accesses to the main shared memory. This was therefore beginning to look like a distributed memory machine.

This lead us to believe that our decision to follow the distributed memory route was correct. However, we soon saw that programming a shared memory processor was much better understood.

3.4 Distributed Memory Architectures

There appeared to be two approaches to constructing a distributed memory machine either using general purpose processors integrated onto boards or using processors that contained the inter-processor communication. The advantage of using the Motorola 68020 or the MIPS chip was that there was a large amount of general purpose software in existence and commercial development was fast. The disadvantage was that the systems became very expensive as the number of components grew. The large number of components gave inherent unreliability to systems with a very large number of nodes.

The use of the transputer with its inter-processor links in the GP-MIMD project was therefore appropriate. The move to the next generation of transputer would give us the performance and more general switching capabilities necessary for a general purpose machine of this type. We did lose the possibility of virtual memory support. The lack of virtual memory should aid performance. In general, it can be argued, that virtual memory on such a machine is totally inappropriate. On the other hand, it does mean that a lot of real memory will be required on each node. Multiply that by the number of nodes and the memory system becomes the main contributor to the cost.

4. THE FARMING PARADIGM IN OFF-LINE HIGH ENERGY PHYSICS

The main use of parallelism in High Energy Physics is in farming. A large amount of compute intensive programs analyse physics experiments event by event. Each event is independent of other events so a simple form of farming is already effective using distributed systems. Work has taken place on how this can be extended to distributed memory machines with a large number of nodes. The farming paradigm is so successful in High Energy Physics that, many argue, there is no need to look beyond farming for any sort of parallelism since you simply cannot do better than that.

4.1 Farming in its Simplest Form

The farming paradigm has been extremely successful in applying parallelism to the analysis of physics events. The fact that every event is quite independent of any other event creates a situation which is known in computing circles as 'embarrassingly parallel'. In its simplest form a *master* program reads an event, allocates the event data to an available *slave*, collects the results from the *slave* and writes the results to tape (see figure 3). Each slave runs an identical analysis program which will handle one event after another.

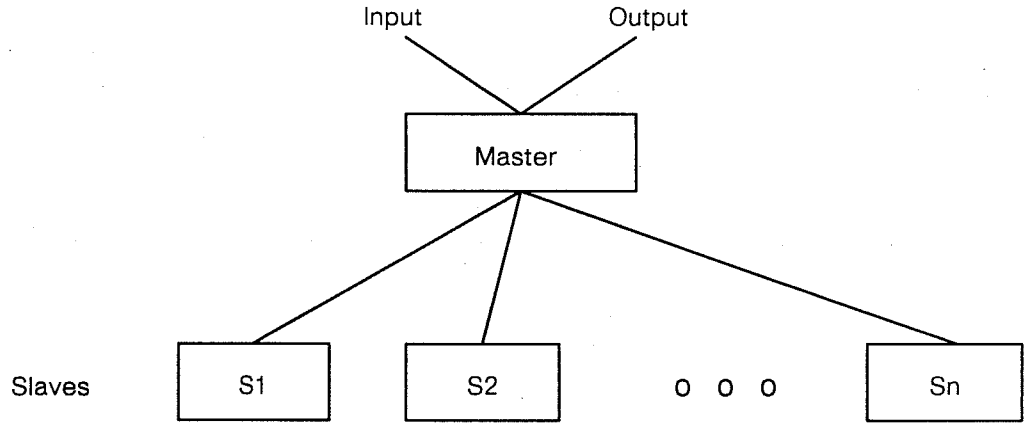

Figure 3: Farming - Master controlling slaves each receiving a single physics event.

4.2 The IBM Emulators - The Original

The first farm of processors was based on IBM instruction set emulators. The idea was pioneered at SLAC and CERN (see Kunz et al., 1984]). The original motivation did not come from ideas of parallelism but from the need to preserve software investment. At the end of the 1970's it was realized that software costs were escalating while hardware costs were diminishing rapidly. The idea of the emulator was to construct an inexpensive processor which exactly emulated a subset of some popular processor's instruction set. This subset of the instruction set was chosen so that physics code could be run. Questions of I/O were handled effectively with data being supplied by a host machine. The effectiveness of this approach is demonstrated by the large number of IBM emulator farms now in use in the physics community.

The recent arrival of fast RISC processors has provided an alternative to the IBM emulator farm, if the user gives up compatibility with the IBM 3090 mainframe. CERN did not want to manufacture processors when industry was clearly capable of producing far better results using modern chip technology. CERN approached IBM to see if a joint solution could be found. The result was the Parallel Processing Computer Server, PPCS (see IBM, CERN 1989). This machine consists of 32 IBM microprocessors that execute the IBM 370 instruction set. These processors are inter-connected using a 'crossbar' switch which gives efficient inter-processor communication. CERN has connected the machine to a standard VME bus and from there to an IBM computer. The future of the system within IBM remains unknown and its future availability as a commercial product uncertain.

4.3 Fermilab ACP

The idea of farming was further explored in Fermilab (see Nash et al. 1987) to see how this concept could be extended to a larger number of nodes. In figure 4 we can see the separation of the input tasks from the output tasks. This enables parallel input and parallel output to support a larger number of slave processors arranged using a flexible crossbar switch. The separate master program is replaced by a queue manager. When data is ready to be processed it is placed on a

queue. A slave processor removes this from the queue. When the analysis task terminates the results are placed on the output queue. The tasks that are writing the results to tape then take their data from this output queue.

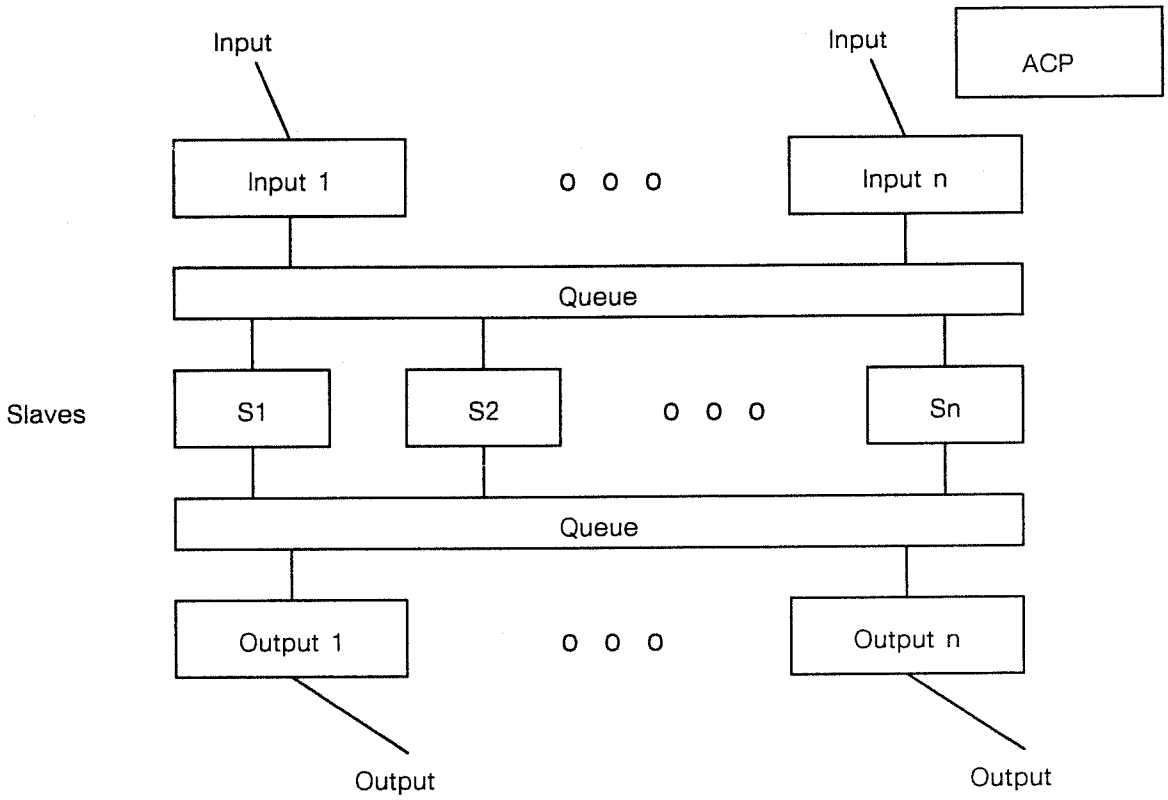

Figure 4: Farming - the general solution

The project embarked on producing two versions of the ACP machine, each of which had an architecture that mapped well onto this more general farming scheme. The first machine used Motorola 68000 processors and the second machine was planned using the MIPS R2000 RISC processors. The ACP project was successful but, in a similar way to the emulators, the group decided to abandon the building of computers in situations where this was clearly within the capabilities of industry to produce a cost effective solution.

5. MESSAGE PASSING

The transputer based software has relied on the *occam* model which is, itself, based on the theory of Communicating Sequential Processes, CSP (see Hoare 1985). This depends on the idea that processes communicate using messages. From the software point of view a closer look at message passing leads to some thoughts on how we might program the machine.

5.1 Message Passing on Distributed and Shared Memory

On a distributed memory machine there is hardware support for passing messages. On the transputer a process may initiate a send or a receive of a message. The processes block until the sending process is ready to send and the receiving process is ready to receive. In this way the message passing is used to synchronise the two processes as well as pass information.

On a shared memory machine the sending process can use the shared memory for the body of the message. The receiving process can then map the message into its own address space. On systems with copy-on-write capabilities the mapping may be done with minimum overhead. Very large messages may be passed, only a small part read by the receiving process and then the message discarded with very little overhead concerning the size of the message.

5.2 Different Addressing Modes in Message Passing

There are three commonly used ways of addressing in message passing:

> Direct Naming: the process A wanting to send a message to process B must name B as the recipient of the message. Process B waits for a message to be sent specifically from A.
>
> Client/Server: the process A, client, sends a message to process B. B, the server, waits for a message from anyone. B is offering a service to any process.
>
> Global Naming: the process A sends a message but does not care who receives it. Process B looks for a message and does not care who sent that message. This is the style of message passing in Linda where messages are received that match a template, or in the farming example where the master process places "work packages" in a "bag" to be picked up by any slave process.

The different styles of addressing strongly effect the programmer. In direct naming the program must be rather static in structure. When writing programs for the present transputer the programmer must "do the plumbing" first, that is he must plan the processes and the message passing before he can begin to write code. This is, of course, only good practice. However, without the client/server style of message passing it is difficult to provide system services or to write inter-communicating library packages.

In the standard client/server addressing scheme it is not obvious how to deal with different numbers of servers. In particular you may want more than one server to handle requests, or you may want a request to remain when there are no servers currently in operation. Global naming gives great flexibility but poses the problem of where to store the messages which should be globally accessible?

What we have discovered is that different styles of message passing are appropriate for different stages of a task. Certain divide and conquer algorithm will require direct naming for communicating the "work packages" but would run well with global naming when reporting results.

6. PERCEPTION OF PARALLELISM IN PROBLEMS

Up to this point it would appear that we are taking the techniques from operating systems and distributed systems, and extending them to parallel MIMD machines. Nicholas Carriero and David Gelernter (See Carriero, Gelernter 1989) have developed programming paradigms and more specifically Linda for parallel computers.

6.1 Result, Specialist and Agenda Parallelism

It has been observed that since the world around us is naturally parallel, parallel programming should naturally model that world. Unfortunately, programmers have been trained to think sequentially and the type of parallelism that is perceived in a problem will depend on how he or she thinks.

There are three types of parallelism:

> Result Parallelism: is where the programmer focuses on the final solution to his problem. The application should be planned around the data structure that is yielded as the final result. Parallelism is obtained by computing all elements of the result simultaneously;
>
> Specialist Parallelism: is where the programmer focuses on the specialist worker processes. The application should be planned around the ensemble of specialists connected into a logical network of some kind. Parallelism results from all nodes of the network being active simultaneously;
>
> Agenda Parallelism: is where the programmer focuses on the list of tasks to be performed. The application should be planned around an agenda of activities and general worker processes are set to work on each step in turn.

These types of parallelism are not restricted to programming. A factory built house is an example of result parallelism. The walls, roof assembly, staircases etc. are all built in parallel in the factory and assembled on site. Houses are often built using the specialists for each task. The plumber and the electrician can both work at the same time. Barn raising is an example of agenda parallelism where the community turns its attention to one a list of tasks in turn.

6.2 Live Data Structures, Message Passing and Distributed Data Structures

These three types of parallelism map nicely onto three methods of programming. Result parallelism corresponds to using live data structures, Specialist parallelism to message passing systems and agenda parallelism to distributed data structures.

In result parallelism the data structure is constructed and a process embedded in each element. The task to be performed by each process is to produce the required value to which it is bound within the data structure. The live data structure determines the program structure, every concurrent process is locked inside a data object and the job of the process is to produce the value for that data element.

In specialist parallelism the process structure determines the program structure. A collection of concurrent processes communicate by messages and every data object is locked inside a process. This corresponds to the *occam* or CSP model.

In agenda parallelism groups of identical workers receive tasks according to an agenda of activities. When the worker has terminated his task he indicates his result and requests more work. This corresponds to the farming model.

6.3 Linda

Nicholas Carriero and David Gelernter created the Linda programming system. Linda has two great advantages – it is computer language independent and can be described on one slide. The operations manipulate tuples of information which are placed in and retrieved from tuple-space. This corresponds to the global addressing for messages that we described earlier.

Linda has the following four basic operations:

> out (t) – causes tuple 't' to be inserted into tuple space;

ins (s) – causes some tuple 't' that matches the template 's' to be withdrawn from tuple space. If no tuple 't' exists when in (s) executes, the process suspends until one is, then proceeds as before;

rd (s) – is the same as in (s) except that the matching tuple remains in tuple space;

eval (t) – is the same as out (t) except that the tuple 't' is evaluated after it enters tuple space.

The use of "eval" enables the programmer to create live data structures. For example, an array of factorials could be generated by the code:

```
for ( i=1; i<10; i++ ) {
    eval ( "factorial", i, factorial ( i ) )
}
```

The "eval" routine returns immediately before the factorial has been evaluated. Any attempt to read the value of a factorial e.g. in ("factorial", 6, ? value), will cause the calling routine to block until the factorial of 6 has been evaluated.

7. OUR THOUGHTS ON PARALLELISM

The preceding part of this paper has concentrated on those aspects of parallelism that we have learned from the experience of others. This has proved to us that parallelism is not easy but the rewards can be high. Now I would like to describe some work done by two members of our group: Adrian King's plans to parallelise a large FORTRAN program and Andre Schneider's work on FORTRAN paradigms and the automatic parallelisation of FORTRAN code.

7.1 Parallelisation of a Real Application

Adrian King will attempt to parallelise the GEANT detector simulation package, developed at CERN and used by the High Energy Physics, HEP, community (see King, 1990). It is a large computationally intensive FORTRAN program. The plan is to port the program to the GP-MIMD machine and, at the same time, to re-structure the program so that it is easier to port to other parallel systems. The code consists of over 150,000 lines of FORTRAN and it is estimated that over 50% of all HEP cpu time will be spent running this package during the next 10 years. In order to port the code a large part of the CERN library will have to be converted to run on the next generation transputer. In particular the ZEBRA data management package will have to be re-written in order to spread the data over many processors.

The analysis of a single event takes up to one minute on a CRAY X-MP. The next generation accelerator, known as the Large Hadron Collider, LHC, will complicate this task by a factor estimated to be about 400. This requires an increase in computer performance that is only possible with a 1000 node GP-MIMD type of machine. The pure farming approach to this problem would exclude any type of interactive work since six hours would be required before the first results could be analysed.

We estimate that a man-year is required to parallelise this program. Much of the work is of a tedious nature and our aim now is to see how much of this work can be automated. However, it

does appear that knowledge of the semantics will be required before a complete job can be accomplished. The algorithm will be of the divide and conquer type. The program follows tracks of particles, when a particle has an interaction which results in further tracks those tracks may be processed separately. Another interesting element to this problem is that the amount of computing time required to analyse a track is roughly proportional to the energy of the track, so we have the possibility of crude load balancing. Finally, there is the collection of the results which must be collected into one place and written to some large data store.

7.2 The FORTRAN Programmer's Paradigm

We would like to present the FORTRAN programmer with a paradigm which appears natural to him (see Schneider, 1990). We can see that an approach such as simply applying the *occam* model as an extension to the FORTRAN language will not succeed with our CERN FORTRAN programmers. This is independent of whether the extension is in the form of subroutines, thus preserving the language, or governed by pre-processors that require new keywords or stangely formatted comments.

The present proposal that is being presented to our programmers is that the extension is based on the distribution of data and remote procedure calls. Parts of programs are to be partitioned into modules that contain data and code that logically belong together. Data may be read by any other processor as long as it is not locked, but remote data may only be changed by calling a procedure within the module that contains the data. While we are not convinced that this is "the solution", it does seem to be closer to the way the FORTRAN programmer thinks about parallelisation.

7.3 Automatic Parallelisation

Finally, we are looking at the prospects of automatic parallelisation (see King, Schneider, 1990a). Automatic program parallelisation for shared memory machines has concentrated on splitting or expanding "DO loops" so that they run on many processors. The results of running this type of paralleliser on CERN code has not been very encouraging. On top of this we are convinced that the strategies developed for shared memory machines will not apply to distributed memory machines since the resulting grain size of the code is too small. Small grain size implies a big communications overhead. The new strategy that we are proposing to follow is based on a data oriented partitioning scheme. The partitioning is directed by information about data lifetime, data interdependence and data duplication.

The implementation will be in terms of a preprocessor which will generate an appropriate dialect of parallel FORTRAN as supported by the target processor. In the case of the GP-MIMD machine this will undoubtedly be a FORTRAN that has been extended using the *occam* model. We will be satisfied if this approach has any success on distributed memory machines. However, there is a trend to look at the idea of distributed memory machines having some form of shared memory. In this case compilers will not need to assign data to processors but the data will migrate to processors where it is needed. It is clear that a good first guess as to where the data should lie will produce more efficient results. This work could lead to an algorithm for placement of data in such a future machine.

8. CONCLUSION

There is a large amount of work to be done in understanding and developing the topic of general parallelisation. The good vectorising compilers that are now in general use were first developed in

the 1960's but only now perform well and in a user friendly manner. We are climbing onto the shoulders of the previous workers in this field before launching into our own developments. This paper has mainly described the work of others as we see it affecting our plans for writing and supporting software applications on a GP-MIMD machine with up to 1000 nodes.

I would like to thank those members of the CERN GP-MIMD project, especially Adrian King and Andre Schneider but also Bob Dobinson and Paul Burkimsher, who have contributed to the ideas in this paper.

REFERENCES

Carriero, N. and D. Gelernter, 1989, How to Write Parallel Programs: A Guide to the Perplexed, ACM Computing Surveys;

CERN, 1989, Computing at CERN in the 1990s, CERN DD;

Duncan, R., 1990, A Survey of Parallel Computer Architectures, IEEE Computer;

Hoare, C.A.R., 1985, Communicating Sequential Processes, Prentice Hall;

IBM CERN Joint Project, 1989, PPCS, CERN;

King, A., 1990, Parallelising GEANT, CERN;

King, A. and A. Schneider, 1990a, Automatic Parallelization of FORTRAN Code for a Distributed Memory MIMD System, CERN;

Kunt, M. and F. Rohrbach, 1989, The Massively Parallel Processing Collaboration: The MPPC Project, EPFL Lausanne;

Kunz, P.F. and M. Gravina, G. Oxoby, P. Rankin, Q. Trang, P.M. Ferran, A. Fucci, R. Hinton, D. Jacobs, B. Martin, H. Masuch, K.M. Storr, 1984, The 3081/E Processor, CERN DD report;

Nash, T. et al., 1987, The ACP Multiprocessor System at Fermilab, Computer Physics Communications;

Schneider, A., 1990, Distributed FORTRAN: A First Definition, CERN;