CONTROLLING AND PROGRAMMING THE SPHINX MULTI-SIMD PYRAMID MACHINE.

J. Méhat
Université Paris VIII, St Denis, France

A. Mérigot
Institut d'Electronique Fondamentale, Université Paris Sud, Orsay, France

ABSTRACT
This paper is concerned with the control environment of the SPHINX Pyramid Machine. First, a low overhead interlayer method, to synchronize communication of independently controlled SIMD processor meshes is presented. We show how it can be used to provide multitasking within a mesh to allow opposite data flow to cross safely. Related programming concepts for Multi-SIMD machines — control transmission — is presented.

Keywords: Pyramid Machines, Multi-SIMD, Parallel Languages, Interprocessors Synchronization.

INTRODUCTION
SPHINX is a cellular pyramidal machine primarily designed for image processing applications [4,5], currently under development as a joint effort of University Paris Sud, and ETCA Defence Research labs. It is organized as a set of stacked layers of decreasing size interconnected according to a dual network: a mesh based inter-neighbor interconnection network within a layer, and a binary tree between adjacent layers. The SPHINX processing element relies on bit serial operations and communications. It is formed of an ALU, with enhanced data transfer capabilities, a 256 bits local memory, and a set of special purpose registers. An hardware prototype pyramid 32x32 (2047 PES) is going to be available in the next future.

Two important differences between SPHINX and other pyramidal machines [1,6,7] are that each PE has 2 sons instead of 4, and that each layer receives its own instruction stream (figure 1); so SPHINX presents both an SIMD aspect, since all the PEs of a same layer share the same instruction stream and a MIMD aspect between layers. It is a Multi-SIMD machine. If we define the power of a pyramid as the number of processors at its base, SPHINX, as a binary pyramid uses 50% more PEs than a quaternary one of the same power, but presents several advantages:

— operations between layers can generally be pipelined in an efficient way, since each PE has two sons and a two input ALU;

— oct-trees as well as quad-trees can be built by grouping PEs of different layers into virtual PEs;
— thanks to the larger number of layers compared with a quaternary pyramid, we can make a more efficient use of the MIMD aspect.

Arguments against pyramid are often the following [3]: either the pyramid is SIMD, leading to a very poor layer occupation, either, for a Multi-SIMD machine, the interlayer control will slow down data transfer. We present a control scheme with a very limited overhead for SPHINX and related software aspects.

THE CONTROL STRATEGY
The instruction stream bandwidth problem draw us to spread the control over the pyramid: different layers are independently controlled. With this control scheme, communication between adjacent layers leads to a critical synchronization problem. As data are transferred between layers in a bit serial way, this type of communication implies that one must be able to realize an intercontroller synchronization within a bit serial instruction execution time (typically 100ns).
Two Levels of Synchronization

To obtain the necessary speed of synchronization, we distinguish between two levels of synchronization: a low level one insuring integrity of the interlayer transferred data with respect to the state of communication buffers according to a producer/consumer protocol, and a high level one allowing coherent manipulation of data within the whole pyramidal structure by means of information passing between adjacent layers.

The high level synchronization is necessary to take into account the constraints of process scheduling, parameter passing and interlayer message passing. To deal with the problem of high level data exchange between layers, it communicates with its two adjacent controllers. It will typically be realized with a standard microprocessor executing compiled code to run application programs generating system control and word level pyramidal instructions, translated into sequence of bit-serial instructions by a simple macro-generator and put into a FIFO for the low level synchronizer.

The low level synchronization has in charge the correct transmission between layers at the bit-serial level. It has to be done within a bit serial instruction execution time, but is very simple as one has only to consider if an instruction is executable with respect to the state of interlayer communication buffers, i.e. the buffers used as source are full at the time of the operand fetch and the ones used as destination will be free when the results will be stored.

The necessity of low level synchronization is not bound to the size of one bit of the communication buffers. If the buffer size were made greater, either through a hardware or software artifice, the problem would be actually identical: a \( n \) bit buffer permits to overcome the synchronization necessity for the \( n-1 \) first bits, but then the buffer may be full and the problem comes back.

The low level controller will synchronize these streams, and send the instructions as soon as it is possible in terms of the producer consumer protocol. This leads to the instruction matrix presented in figure 3d. This matrix is less regular than the idealized

The histogram of a certain attribute stored in the lowest layer of the pyramid, i.e. the number of occurrences of every value of attribute, and either output it by the apex of the pyramid, or store it in the upper layer. A straightforward algorithm is the following:

\[
\text{for } i \text{ varying from 0 to the last attributes value} \\
\text{begin} \\
\quad \text{select the PEs where attribute } = \ i \\
\quad \text{count the number of selected PEs} \\
\text{end}
\]

The select operation is a point wise procedure applied to PEs in the lowest layer. The count operation uses the vertical communications to accumulate, in a logarithmic time, the number of selected PEs. The actual method is the following: at the lowest layer, PEs send the selected bit to their father. At the others layers, PEs add sequentially the bits they receive from their sons, sending the result to their father, followed by the MSB of the result stored in their carry register. The algorithm is the following, assuming the pyramid has \( \text{height} \) layers:

\[
\text{at the base} \\
\quad \text{begin} \\
\quad \quad \text{send the selected bit to the father} \\
\quad \text{end} \\
\text{at layer } l \text{ such as } \text{top} \leq l < \text{base} \\
\quad \text{begin} \\
\quad \quad \text{repeat } \text{height} - l \text{ times} \\
\quad \quad \quad \text{begin} \\
\quad \quad \quad \quad \text{send to father the sum of the bits} \\
\quad \quad \quad \quad \text{transmitted by left and right sons} \\
\quad \quad \text{end} \\
\quad \quad \text{send carry to the father} \\
\quad \text{end}
\]

Figure 3a presents the instruction matrix, i.e. the instructions executed on every layer vs. time, for one step of the histogram, computing the sum for one attribute value on a pyramid of height 5.

Thanks to the MIMD interlayer control mode, it is possible to start another step of this operation as soon as the previous step has completed in the lowest layer. This lead to the idealized version of the histogram computation presented in fig 3b. One can see that MIMD allows a large gain in computing time, as the apparent execution time will only be limited by the data output time, instead of the complete count operation time.

The high level control will execute the previously described algorithm, and send to the low level controllers the instructions streams of fig 3c. The instructions are correct in term of their relative occurrence within a layer, but the actual execution time is not fixed.

The low level controller will synchronize these streams, and send the instructions as soon as it is possible in terms of the producer consumer protocol. This leads to the instruction matrix presented in figure 3d. This matrix is less regular than the idealized
Figure 3: Pyramid occupation vs. time for different histogram operations and different synchronization mechanisms.

Figure 3a: One integration step of the plain histogram procedure.

Figure 3b: Idealized version of the plain histogram operation. The MIMD between layer control mode allows independent execution of every integration step.

Figure 3c: Instruction stream sent by the high level controller to perform plain histogram.

Figure 3d: The instructions sent to the Processing Elements after producer-consumer like synchronization performed by the low level synchronizer. This step leads to a layer occupation automatically optimized at run time.

Figure 3e: The associative histogram problem with the unidirectional synchronization scheme. The lack of data stream crossing capabilities leads to a poor pyramid occupation.

Figure 3f: The associative histogram with the bidirectional synchronization scheme. The crossing of uncorrelated upwards and downwards streams insures a better layer occupation.
version of figure 3b, but more efficient in term of pyramid occupation. More, the programmer just has to take care of the logical correctness of his program, and the synchronizer will automatically optimize its execution at run time.

On figure 3d at date labelled t, we can see a vertical data shift. Layer 3 performs an add operation and sends the result to layer 2, whilst this one performs the same operation on previously generated data and so on until the top. This means that the instruction on layer 2 can take place, not because the present state of the father's receive buffer is correct (empty), but because the instruction in the upper layer will empty the buffer, if executed.

This kind of operation where all the concerned communication buffers are full before operand fetch but freed before the result should be stored, requires non local knowledge to be performed. [2] presents a cellular automaton based mechanism able to solve this kind of problems at run time, as well as a complete description of this synchronization scheme.

The crossing of data movements.

The previous model works very well when all the data movement are of the same direction, but it presents some weakness in other cases. Let us consider the following associative histogram problem: each PE at the base of the pyramid contains an attribute value, and we want to associate it, in the PE memory, with the number of occurrences of its value in the base. The vertical connections of the PEs are used both to compute the number of occurrences in the top of the pyramid in a time logarithmic to the number of PEs of the base, and to project the so computed sum from the top PE to the PEs of the base. The computing necessary for each value involves two distincts data movements: an ascending one to compute the sum and a descending one to project it.

There is two way to realize this operation on a pyramid. The first one avoids data crossing by means of the following algorithm:

```
for i varying from the first to the last value
begin
count the number of PEs where attribute = i
send the result down
at the base begin
where attribute = i begin
associate the number with the attribute
end
end
```

As it is shown in figure 3e, this leads to a very inefficient use of the pyramid, most of the layers being idle whilst the data is sent downwards.

Another method consists to allow the crossing of the upwards and downwards data stream. Indeed, these streams are logically uncorrelated, and one can perform the nth integration, while the n-1th result is sent down. One possible result is shown in figure 3f.

The drawback of this solution, when using the previous synchronization method, is the possible turning up of deadlocks when two data transfers are coexisting. The figure 4 shows the top layers and their communication buffers computing the first bits of the sum in the associative histogram problem. When the first bit of a sum is computed at the top of the pyramid, the 2d bit can't be computed at the top before this 1st bit has been consumed by the level 1. At the same time, the instruction to consume this bit can't be executed by level 1 before the one which compute the 3rd bit has been executed. The two layers are in deadlock.

```
Figure 4: Deadlock emergence in data reverberation.
```

Of course, one could imagine to synchronize statically the layers at compile time, and to schedule the instructions in a way avoiding deadlocks. This solution leads to several drawbacks. First, it implies that the state of the layers in term of occupation must be well known, and accordingly, all the previously scheduled routines must have completed, disabling any use of pipeline. Second, this leads to an unstructured instruction stream, that imposes a very large data transfer rate, that causing bandwidth problems. Third, this static synchronization is not always possible if we want to be able to use data driven algorithms. For instance, if the global operation termination is controlled by a run time criterion as a convergence test on a layer, it is clearly impossible to perform any kind of static scheduling. We now present a synchronization method able to solve this kind of situation.

Two FIFO for the synchronizer

A proper automatic deadlock-free run time synchronization of data crossing requires that the upwards and downwards instruction streams are generated by two independent (pseudo) parallel processes. The synchronization mechanism should be able to choose the best candidate according to the

```
communication buffers state. The algorithm should be like the following:

```plaintext
in parallel begin
    for i varying from 0 to n begin
        count the number of selected PEs
        where attribute == n
    end
    for i varying from 0 to n begin
        at all levels except base begin
            send downwards the result of the count operation
        end
    at base begin
        where attribute = i begin
            receive the value emitted by the upper level
        end
    end
end
```

To perform this, we propose to have two FIFOs in which the synchronizer fetches instructions. One contains the instructions taking part to the ascending data movement, and the other the instructions for the descending one (Figure 5). The synchronizer selects one of the two FIFOs. As long as the execution of the instruction on the top of the FIFO is compatible with the state of the interlayer communication buffers, it is sent to the PEs of the layer. When the instruction must be delayed and the other FIFO is not empty, the synchronizer switches the context in the PEs of the layer and uses the instructions of the other FIFO.

The use of two FIFOs allows a deadlock free coexistence of the two processes in the same layer, provided they only carry data either upwards or downwards: let's imagine that in the layer $i$ we have a process $P$ which is blocked in a deadlock. We will suppose that $P_i$ is an ascending process but the demonstration is easily extended to the case where $P_i$ is a descending one. $P_i$ may be blocked either because a process $P_{i-1}$ in the layer $i-1$ doesn't produces the data $P_i$ needs, or because a process $P_{i+1}$ doesn't consumes the data $P_i$ produces. As $P_{i-1}$ and $P_{i+1}$ are ascending processes, they produce their instructions in the FIFO of the ascending processes. As they are the only ascending processes in their layers, their instructions are on the top of their FIFO, and they must be blocked because of the ascending processes of layer $i-2$ or $i+2$ to which the same reasoning may be applied. There can't be a deadlock as long as there is no interaction between ascending and descending processes.

### Vertical communication as resources

Some processes don't fit in the ascending/descending scheme we just described. For example, the process at the top of the pyramid in the associative histogram program, both consume data from the layer 1 like an ascending process, and produce data for the layer 1 like a descending one in the same instruction. We label such a process as bidirectional, and activate it according to the following protocol: in each layer, each direction of communication is considered as a resource; an ascending process needs the ascending resource, instantiated by the communication buffers from the lower layer and to the upper layer, a descending process needs the descending resource and a bidirectional process needs both resources. A process is run in a layer only if the resources it needs are available. That way in a layer we can have either at most one ascending and one descending process, or a bidirectional one.

Bidirectional processes are necessary to have crossing data streams performing useful work, as they permit exchange of data between them, but their existence allows occurrences of deadlocks, as is usual when two processes are communicating together via blocking input and output primitives. Proper programming allows the avoidance of deadlocks, which are easy to detect as they are always caused by bidirectional processes.

### THE EXPRESSION OF COOPERATION

Independently of the control strategy, one has to provide an effective way to specify the cooperation of different layers, which directs as well how high level synchronization is realized, as how the programmer specify process creation. The control model described above relies on anonymous ports, as any other communication scheme would have implied to carry identifying information about communicating processes with any bit of data. Despite this constraint, one has to let the communication expression specify as
explicitly as possible the identity of the processes involved.

To express the cooperation of the layers, we propose to use the same scheme that is used for data movement: pyramid algorithms are conceived in term of data movements between layers, and then a task is defined which migrates, accompanying the moving data. We choose not to execute that migration via process controllers, because process migration implies that the process environment migrates with the process, and also because data exchange between PES from different layers involves two processes: one sending the data and a second one receiving them.

The task migration is accomplished on a local to a layer base, in a way we call control transmission, after data transmission. The creation of the task involves three distinct steps: at first, a process is created in the layer initiating the data movement; that process creates in the neighbor level another process and then sends data to the newly created process.

For example, a step of the sum for the histogram will be programmed with:

```
process sum(n) begin
    if current layer = base then begin
        create process sum(1) in the upper level
        send the selected bit to the upper level
    end else if current layer = top then begin
        sum the n bits received from lower level
    else begin
        create process sum(n+1) in the upper level
        send to the upper level the n+1 bit sum of the n bit received from lower level
    end end
at base create process sum(0)
```

The associative histogram operation can be programmed in the way. One only needs to activate at the upper layer a descending process to carry data downwards.

Control transmission presents several advantages: involving only transmission between adjacent layers, it can be realized by a point to point communication between adjacent controllers.

The environment of each process is clearly stated in the program, as each process communicates either with the process which created it, or with a process it has created, or both. Accordingly, provided the algorithm is correct in terms of data transfer i.e. the same amount of data is produced and consumed by communicating processes, the consumer will always be the right one, even though transfers are made through anonymous ports.

Deadlocks coming from process creation obey the same mechanisms as data communication. One bidirectional processes can, directly or indirectly create deadlocks by process creation requests.

The use of layers being controlled by process execution, it can be dynamically adapted to the state of data in the PEs, allowing a better exploitation of the occupation of the layers.

CONCLUSION

We have presented a synchronization method to solve the problem of the Multi-SIMD control of the SPHINX Pyramid Computer. Based on a recently designed chip, we are building a 32x32 machine, with a controller relying on these principles. We have as well, designed a high level language—pyr-e—based on C for which a compiler is on the way.

AKNOWLEDGEMENTS

This work was supported by ETCADRET. Its results from many helpful discussions with Patrick Greussay, Philippe Clermont, Francis Devos and Ni Yang.

REFERENCES


