MVA'90 IAPR Workshop on Machine Vision Applications Nov. 28-30,1990, Tokyo

SimPar - a Programmer Oriented Software Environment for the Pyramidal Vision Machine SPHINX

Edwige E. Pissaloux, Samir Bouaziz, Alain Mérigot, Francis Devos

Institut d'Electronique Fondamentale, CNRS LA 022
Université Paris XI
91405 Orsay Cedex, France

1. Abstract.

In this paper we present the minimal software environment -SimPar- for Multi-SIMD massively parallel processing system (SPHINX machine). It describes the parallel process management at software levels, and introduces the programming style for parallel software design, named co-programming. The presented results can be easily applied to design of any parallel system with Multi-SIMD or MIMD control.

2. Introduction.

The advent of parallel MIMD and Multi-SIMD systems has created the demand for software techniques to help utilize of this new processing power.

Much effort has been devoted to the development of parallel languages for as well MIMD and SIMD machines and there exists a wide variety of compilers ([1-3]). However, programming Multi-SIMD machines is still a problem as one has to program independent control processors to perform a fine grain coherent data manipulation. This leads to problems in language design, basic software development and algorithmic studies. In all these cases it is required to have a robust simulation and development programming environment.

This paper presents the software simulation environment SimPar developed for the pyramidal vision computer SPHINX. The Multi-SIMD control strategy of the machine influences the SimPar architecture, so we will start with its brief presentation. We will continue by giving an overview of the SimPar and will finish by showing an example implementation of a basic image processing operation.

3. The Multi-SIMD control strategy of the SPHINX machine.

The SPHINX, (SPHINX for Système Pyramidal Hiérarchisé pour le traitement d'Images Numériques), is a massively parallel pyramidal computer dedicated to image processing, currently being developed by Paris Sud University, in Orsay and ETCA, Defence Research Laboratory in Arcueil (France).

The SPHINX machine is organized as a set of stacked layers of PEs (Processing Elements) of decreasing size interconnected by means of a dual topology : a mesh-based four-neighbour interconnection network within a layer and a pure binary tree between layers.

Figure 1 shows the organization of the SPHINX machine with its distributed Multi-SIMD (MIMD) control strategy, implemented by means of a linear network of controllers, one per layer.

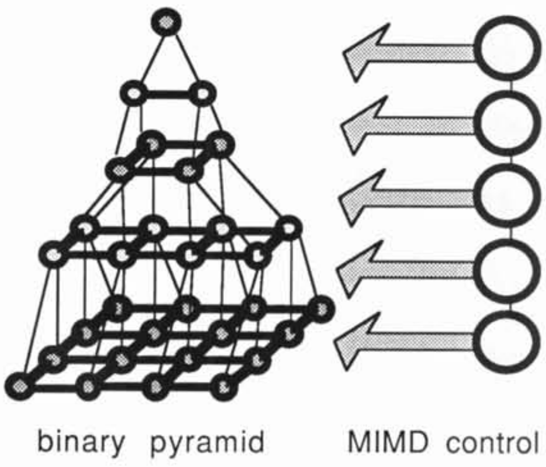

Figure1 : Architecture of the SPHINX

To fulfill temporal control constraints, whilst still having enough programming flexibility, layer controllers are subdivided in two parts :
• a high level controller (HLC), which provides an interface between the host computer and the pyramid; it runs user programs execution which generates low level instruction and data to its SIMD layer;
• a low level synchronizer (LLC), which supplies instructions received from the HLC the pyramid, and controls temporal occurrence of instruction execution with respect to an object based communication strategy between layers.

To guarantee high temporal performances of the pyramid, an asynchronous communication model has been implemented. It uses a dual-port memory attached to each sun-father junction of the binary tree, where interlayer communication channels are implemented. Thus the asynchronous communication scheme can be implemented and it will largely improve execution efficiency as well as programming flexibility of complex image analysis algorithms.

At present, this control scheme is under hardware implementation; the SimPar environment uses its functionnal principles.

4. The SimPar environment.

The SimPar is a software environment developed in C on a Unix machine to simulate the SPHINX computer. It uses a fast swap mechanism between pseudo processes (later just called "processes") simulated within an UNIX process. Its main purpose is to provide :
• a low level interface for firmware development when debugging SPHINX microcode (scheduler for low level synchronizer);
• a support for algorithms prototyping and for basics of the SPHINX operating system development (scheduler for process management).

4.1. The SimPar for low level simulation.

The low level of simulation is particularly useful for machine microcode and machine behaviour testing at every clock period. It relies on the following assumptions :

• Although the different layers are independently controlled, the instruction execution is synchronous within the complete pyramid.
• Execution of serial code on the high level controller takes a null time. As generated low-level instructions are buffered in a FIFO, this hypothesis, although optimistic, is correct in many cases.

These assumptions allow a simple management of execution time in the following way. At the lowest level, SPHINX is programmed by means of MAP, a programming language that allows to use as well pyramid microinstructions and C control structures. The translating program can generate actual code for the final machine or a simulation program by means of a generation of a sequence of basic RTL-like data movements for every pyramid macroinstructions. These basic data movements are used either in a sequential simulation of the pyramid behaviour or in a parallel simulation on the Connection Machine [5] using the PARIS library. In order to properly simulate the temporal occurrences of instructions these simulating calls are intermixed with synchronizing procedural calls that take place whenever a clock cycle has elapsed, in order to maintain a proper relative time within the different layers and when interlayer synchronization takes place for data exchange.

Synchronizing sequences will use SimPar facilities to maintain a coherent state of the different layers of the pyramid. The following scheduling algorithm guarantees that every instruction generated by the sequential simulator will correspond to instruction generated by the real parallel machine, and maintain a proper time on every layers :

```
while the layer instruction queue is not empty
    fetch the pending instruction
    while this instruction is not executed
        if the low level synchronizer can accept it
            if an interlayer synchronization is required
                update the layer time as the maximum
                        value on both layers
            send the instruction to the synchronizer and
                update the layer time
            mark the instruction as executed
        swap to process on next layer
```

Each instruction sent is executed by the process simulating the scheduler. It is possible to trace instruction execution once finished or at every clock period (thus enabling very fine execution control). Debugging and profiling tools to monitor instruction queues, watchdogs, execution statistics, etc... can be easily inserted in this methodology.

4.2. User Interface of SimPar.

SimPar manages several processes within a single Unix process. This management is realized by means of a set of functions and macros written in C and assembly language.

The process scheduler is an essential part of process management in the SimPar. It provides several functions allowing the implementation of the SimPar's functions and user's algorithms.

SimPar's process management is performed using the following functions :

• void	SimParCallProcess()
• int	SimParGetProcessId(),
	SimParSwap(),
• Process	*SimParGiveNextProcess()

The user's processes are chained in the FIFO queue. The scheduler manages this queue in a fashion determined by user's program. Altough there exists a default scheduler toilored to our simulation needs, it is still possible for every user to write his own scheduler.

Current functions of the scheduler are :
• it adds a new process when user calls SimParCreateProcess() function;
• it suppresses a process when user's C function (a SimPar process) reaches SimParReturnIs() function;
• it swaps to another process whenever it encounters a SimParSwap() call.

For low level simulation purposes this last function is generated by the low level code generators for every simulated clock period (or instruction execution according to the granularity of the simulation) to allows a concurrent trace of the different controling processes. However this functionnality implies a very low level programming language and is dedicated to basic library development, not to algorithmic studies.

To allow fast algorithm prototyping, user level function have been introduced. They realized a simple data exchange between adjacent layers, then swap to the destination layer. These are SendUp(), SendDown(), GetUp() or GetDown() function. They realize an object based asynchronous communication. Altough less efficient than optimized assembly level coded routines, code written using these functions could also run unchanged on the real hardware. An example program using these functions is presented in the following section.

5. Design of the parallel software in the SimPar environment.

In the SimPar environment, a program is defined as a set of, communicating or not, functions implementing the same algorithm.

This definition serves to implement user's functions by means of enhanced co-routine principle ([7]), in a new style named co-programming ([8]).

In the SimPar environment, the user's functions are written in C and use the SimPar process management routines.

The following example shows an implementation of the histogram algorithm on the SimPar. This algorithm is a straighforward implementation deriving directly of the histogram definiton. It loops over every different possible values of the gray level. For every value, it selects the pixels with this value, then calculates the number of selected pixels in the pyramid base. This summing operation is executed in $\log_2(N)$ steps, and the calculated result is sent back to its controller by the top process of the pyramid of *h* layers. Any intermediate layer gets the partial result calculted by the underlying layer, adds received values and sends the obtained result to its father.

```
main()

{
        int i, histogram[MAXVALUE]

        PyramidAdr Image,    Buffer;
        SimParInit();

        for(j=0; j<MAXVALUE;j++)
        {
                SimParCreateProcess(process_base, *p,
                        DimStack, h,j);
                for(i=1; i < h; i++)
                        SimParCreateProcess(process_inter,
                                        DimStack, i);
                histogram[j]=SimParCreateProcess
                                (process_top, DimStack, 0);
        }
        SimParScheduler();
        SimParClose();
}


process_base(Image, val)

{
         PyramidAdr Select ;
         SpxEqualConst(Image,val,Select);
         SendUp(Buffer, Select);
         SimParReturnIs();
}


process_top()

{
        PyramidAdr Sum , LSon, RSon, Carry;
        GetDown(Buffer, LSon, RSon);
        Add_with_Carry(Sum, LSon, RSon);
        SimParReturnIs(GetFromPyramid(Sum));
}


process_inter()

{
        PyramidAdr Sum , LSon, RSon, Carry;
        GetDown(Buffer, LSon, RSon);
        Add_with_Carry(Sum, LSon, RSon);
        SendUp(Buffer,Sum);
        SendUp(Buffer,Carry);
        SimParReturnIs();
}
```

Different scheduling techniques could have been used by the user to manage the successives values. For instance, instead of creating a process for every value to compute, a unique process could have been initiated on every layer that would have processed all the successives values. SimPar is used to experiment different programming methods for MultiSIMD processing.

6. Advantages of the SimPar environment.

The following caracteristics of the SimPar environment allow easy testing of machine microcode, easy writing and debugging of high level software :

• the SimPar environment allows parallel program writing in the co-programming style;
• the SimPar and its running processes are viewed as one process by Unix; consequently the number of Unix processes does not grow, the use of the Unix process manager is eliminated, and the programmer can finely control all his processes;
• Inter process communication is easily realized by means of global variables instead of complex Interprocess Communication Mechanisms ;
• the programmer works in a software environment defined by himself: he may write his own communication or synchronization mechanism ;
• fine synchronization is easily "observable" by the programmer;
• it allows debugging of parallel programs by means of standard Unix debuggers like dbx;
• it allows the entrapment of the execution of instruction on every machine cycle;
• the processes can be executed on a simulator as well as on the real machine without any major modification;

Because the SimPar facilitates parallel software design we say that SimPar is a programmer oriented environment.

7. Conclusion.

This paper presented the software environment SimPar developed for the pyramidal Multi-SIMD vision machine SPHINX. This environment includes both : a machine simulator and basic software integrating several operating system functions.

At an algorithmic level the SimPar allows the design of parallel programs to be a set of concurrent processes for which the SimPar implements user's supplied synchronization mechanism.

The SimPar approach can be used to design the software of any level for any (massively) parallel computer with Multi-SIMD and MIMD control strategy.

8. References.

[1] Sharp, J.A., An introduction to distributed and parallel processing, Blacwell Scientific Publication, 1987
[2] Evans, D.J. (Ed), Parallel Processing Systems, Cambridge University Press, 1982
[3] Perrott, R.H., Parallel programming, Addison-Wesley,
[4] Méhat, J., Mérigot A., Proc. on the 2nd Symp. on the Frontiers of Massively Parallel Computation, pp.423-428 (Fairfax, Virginia,1988)
[5] Rougerie, E., Technical Report, Univ. Paris XI, 1990
[6] Ni,Y. , Mérigot A., Devos F., in V. Cantoni (ed.), Progress in Image Analysis and Processing (World Scientific,1990),pp.759-766
[7] Knuth, D.E., The art of computer programming (Addison-Wesley, 1976)
[8] Pissaloux, E.E., Bouaziz, S., Mérigot, A., Devos, F., The Euromicro Journal, vol. 30, Numbers 1-5, August 1990, pp. 569-576
[9] Tanimoto, S.L., Ligocki, T.J., Ling R., in L. Uhr, (ed.) Parallel Computer Vision (Academic Press,1987), pp.43-83
[10] Schaefer, D.H., Ho P., Boyd J., Vallejos C., in L. Uhr, (ed.) Parallel Computer Vision (Academic Press,1987), pp. 15-42
[11] Cantoni V., Levialdi S., in L. Uhr, (ed.) Parallel Computer Vision (Academic Press,1987), pp.3-13