A Method to Evaluate the Performance of a Multiprocessor Machine based on Data Flow Principles

Ranjani Narayan
and
V. Rajaraman

Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore India.

Abstract: *In this paper we present a method to model a static data flow oriented multiprocessor system. This methodology of modelling can be used to examine the machine behaviour for executing a program according to three scheduling strategies,* viz., *static, dynamic and quasi-dynamic policies.*

The processing elements (PEs) of the machine go through different states in order to complete tasks they are alloted. Hence, the time taken by the machine to execute a program is directly dependent on the time spent by the PEs in various states during the execution of tasks. We adopt a "state diagram" approach to model the machine. This modelling scheme can be used for a class of machines, which have similar execution paradigm. By introducing "wait states" in the state diagram of a PE at appropriate places, we capture the delays that are incurred by the PE waiting on events; the events during the execution of a program being those of wait for availability of inputs, access to global memory, response from the scheduling unit and access to communication media. The communication media are modelled as queueing networks and the delay introduced by the wait state (of a PE for accessing a medium) is specified by the queueing delay of the corresponding network model. The novelty of the state diagram approach is that it facilitates faster simulated execution of programs on the machine as compared to that of conventional simulation languages. This is because, the properties of the machine are described by a set of polynomials.

1 Execution model of the machine

The multiprocessor system consists of the following components:

(i) a bank of homogeneous processing elements (PEs),

(ii) a special processing element which holds global memory — I-Structure memory[1] (known as the I-Structure processor),

(iii) three broadcast type communication media,

- (a) one for transmitting results of computation, referred to as the "result bus",
- (b) the second primarily for transmitting acknowledgements and allocating nodes to PEs, called the "acknowledgement bus",
- (c) and the third bus for I-Structure operations, known as the "I-Structure bus",

(iv) a conventional von-Neumann peripheral processor (referred to as the I/O or host processor) and

(v) a scheduler (refer to Fig. 1).

A program represented by a data flow graph is initiated for execution by allocating the nodes (of the graph) to the available PEs in the machine. This allocation of nodes to the PEs can be done by adopting one of the following three scheduling strategies.

1.1 Dynamic scheduling

In this scheme, at any instant each PE is allocated one node by the scheduler. Once a PE completes the computation of the operation specified by the node, it broadcasts the computed result on the medium. The PE on receiving the necessary acknowledgements is free to accept the next node of the graph. The process is repeated till all nodes of the graph are exhausted.

1.2 Static scheduling

In the second approach, the scheduler divides the data flow graph into as many sets of nodes as the number of PEs. Each set of nodes is allocated to a PE prior to the start of execution. A PE picks up an "enabled" node (from the set it is allocated) for execution. Once it finishes computation of the node, it broadcasts the result to the medium. The PE then picks up the next "enabled" node for execution. The PE repeats this process till it executes all the nodes allocated to it.

1.3 Quasi-dynamic scheduling

In this method the data flow graph is divided into a number of partitions by the scheduler. To begin with, each processor is allocated one partition. All nodes belonging to a partition is executed according to static scheduling policy by dividing them into a number of sets. After the completion of a partition, the next partition is allocated to the PEs. This process continues till the graph is completely executed.

2 Execution of a program

Here we consider the execution of a program according to dynamic scheduling strategy. To begin with, the host processor compiles a given source language program P to a data flow graph G_m. The compiled data flow graph is loaded into the local memory of the scheduler by the host processor.

2.1 Acyclic data flow graph with conditional constructs

We first consider the execution of an acyclic data flow graph. The scheduler allots as many nodes (of the graph) as possible to the available PEs according to data flow principles[2,3] (*i.e.* "enabled" nodes in preference to "not enabled" nodes). Those PEs which possess "enabled" nodes perform the operations specified by them and produce results. A result is broadcast by a PE on the result bus. If a result produced corresponds to an output of the graph, the scheduler picks it up, stores it in its local memory and sends acknowledgement to the PE that produced the result. If the result is not an output, then it is an input to one or more nodes of the graph. (Result that is a "write" onto the global memory is explained in a following section.) The result is picked up by

CH2766 – 4/89/0000 - 0209 © 1989 IEEE

all the PEs which have nodes for which this is an input operand. The PEs which pick up the result send acknowledgements to the PE that produced the result. As the number of available PEs is limited, there could be nodes that are not alloted to any PE (*i.e.*, still in the local memory of the scheduler) for which the result is an input operand. If there is (are) such a node(s) with the scheduler, then the scheduler picks up the result and updates the appropriate node(s). Following this, the scheduler sends acknowledgement to the PE which produced the result. The PE which produced the result is relieved as soon as the required acknowledgements are received. The relieved PE is alloted the next node by the scheduler. The above operation goes on until there are no further nodes to be alloted and all the outputs of the graph are collected by the scheduler.

Consider the execution of a conditional construct. At the time of allotment of a conditional node to a PE, there could be some nodes of the two branches of the conditional construct (*i.e.*, nodes of the "then" and "else" branches) that are alloted to PEs and some nodes of the two branches not alloted to any PE (*i.e.* still in the local memory of the scheduler). At the instant when the conditional node is alloted to a PE, the scheduler stores information about the nodes of the two branches of the conditional construct (whether alloted to PEs or not). The scheduler, constantly listening to the medium (result bus), captures any result from a conditional node and checks the result. It then releases all those PEs alloted to the nodes of the "failed" branch of the conditional (if any). This is done by broadcasting "disregard" token(s) in the medium. Also nodes if any, of the failed branch, which are not alloted to any PE are deleted from the local memory of the scheduler. A merge node (of a conditional construct) is "enabled" on the availability of one of the operands.

2.2 Cyclic dataflow graph with conditional constructs

The execution of a loop is given the highest priority in this machine. A loop is initiated when at least one free PE is available in the system. A PE, alloted to a node of a loop remains alloted to that loop till the termination of the loop, in order to avoid reallotment of loop node as many times as the number of iterations. The scheduler, constantly listening to the medium captures result from a loop node when avaiable. If the result so captured indicates termination of the loop, then all the PEs alloted to the nodes of the loop are released. If the result captured does not indicate termination of the loop, then the execution proceeds as follows. If a destination node of the result produced (the destination node also belongs to this loop) is already alloted to a PE, then the result available in the medium is captured by the destination. Otherwise, if a free PE is available at this instant, the destination node is alloted to that PE and the PE is not freed to any other node outside this loop till the end of execution of this loop. If a PE is not available, then a PE already alloted to a node belonging to the same loop is freed and the destination node is alloted to this PE (in order that the execution of the loop proceeds without the destination node eternally waiting for the availability of a PE). In case a destination node is not alloted to a PE and is made to wait till a free PE is available, a deadlock could occur. A simple example of this kind of deadlock occuring in this system is when the system has only one PE and the dataflow program consists of at least one loop with more than one node in the loop.

2.3 Graphs with nested loops and conditionals

In order to support nested constructs, we have two identifiers, one for the nested loops and the other for the nested conditionals. The depth of nesting is limited only by the number of digits that can be accomodated in an identifier. The loop(s) to which a node belongs is(are) specified in the loop identifier. The nesting information grows from right to left. For instance, the loop identifier of a node belonging to loop 3 which is nested in loop 1, will contain the number 13000...0. The identifier containing the nesting information about the conditional constructs has a similar structure. To accomodate nesting of loops within conditionals and vice- versa, the two nesting identifiers are used simultaneously when the program is executed.

The outermost loop (containing the inner nested loop(s)) is the indicator for either termination of any of the nested loop(s) or for reallocating the PE. In other words, a PE which is allocated to a loop, in case of necessity, could only be reallocated to any node belonging to a loop which lies within the same outermost loop.

In the case of nested conditionals, nodes as well as any other nested conditional(s) along a branch, could be eagerly allocated (prior to the availability of the result from its corresponding decision node) to PEs. At a later instant, if a contrary decision arrives (from the decision node), then the scheduler issues "disregard" token(s) to those PE(s) holding any type of construct on the failed branch of the conditional.

A loop construct nested within the conditional is treated as any other ordinary node as regards transmission of "disregard" to a PE that holds it. But once the branch holding the nested loop succeeds, it is executed as a simple loop construct.

A conditional nested inside a loop is treated in a slightly different fashion. An eager scheduling done on a node lying on a branch of a conditional (nested within a loop), is sent "disregard" (if the corresponding decision fails), even if a loop nesting it, is still being executed. In case the decision succeeds in the next iteration of the loop, reallocating nodes will be a necessity. But such a strategy is adopted to keep the overhead of book-keeping minimum.

2.4 Global "read"/"write" operations

Global memory is conceived as an I-Structure memory whose accesses are controlled by the I-Structure processor. This is a "write-once" memory. A global "write" request is honoured by the I-Structure processor if the request is a first time write. Further, after the "write" operation is performed, the I-Structure PE disposes any pending "read" request(s) for the location. A "write" request on a global location already written by a previous operations is flagged as an error.

A PE requiring a global data first makes a request to the I-Structure PE which checks the availability of the requested data (data will be available in a global location if it is "written" by a previous operation). If the data is available, the I-Structure processor immediately broadcasts the data (on the I-Structure bus) to the requesting PE. Otherwise the I-Structure PE puts out a "not-available" signal to the requesting PE and keeps note of the pending request. The waiting PE, on receiving the "not-available" signal, goes to the state of wait for another node. Once the I-Structure PE receives the requested data (because of a "write" operation by some other PE), it transmits the same on the I-Structure bus. The scheduler then allocates the node to another PE.

3 Modelling of the machine

Below we describe the scheme adopted for modelling the machine in terms of its basic components, *viz.* processing element, scheduler and buses. This model is used for evaluating the machine for executing programs represented by data flow graphs.

3.1 Modelling a typical processing element

Under ideal conditions (*viz.* unlimited PEs, ideal communication medium), a given program is executed by exploiting maximum parallelism resident in G_m. Every node in the graph has a PE to which it can be allocated. A PE can start execution of the node as soon as the required data are available. Thus the given graph will be executed asynchronously in minimum time. The minimum time is dependent only on the nature of G_m. In actual practice however, the number of available PEs is limited and the communication medium has a finite bandwidth. Hence under realistic conditions, the PEs spend time waiting for input, computing nodes of G_m, waiting for the availability of communication medium to send their outputs and waiting for acknowledgements. Therefore the execution time of G_m by an ideal machine gets stretched when it is executed in an actual machine.

Since the execution of a given program is carried out by a limited number of PEs in the system, the time taken to execute a program can be derived by studying the states of PEs during the execution of the program. Hence it is sufficient to compute the time spent by all the PEs in various states during the execution of a given program, to evaluate the performance of the machine. The PE state diagram will therefore include PE wait states corresponding to delay in acquiring the various buses, delay due to responses from the I-Structure PE and delay due to the scheduler etc. The various states in which a PE can exist during the execution of a node of G_m, can be identified by looking at the execution model of the machine.

3.2 Modelling the scheduler

During program execution, the scheduler is required to allocate the nodes of the graph to PEs, to keep track of the status of PEs, to transmit acknowledgements to PEs producing results if required etc. Each of these operations takes finite time, independent of the reason for which the scheduler is invoked. Thus the scheduler can be modelled as a constant delay during the execution of a program.

3.3 Modelling the buses

The buses in the system are modelled using queueing networks. The requests to a bus are queued at the input to the bus. There is no limit on the length of this queue. All the requests to the bus are honoured on a "first come first served" basis. Further there could be only one request that is serviced at any instant. Hence a bus can be considered as an infinite-buffer, single server system that uses first in first out service policy[4]. This is true of all the three buses in the system.

3.4 Average execution time of a node

The state diagram of a PE executing a node is shown in Fig. 2. There are b (b= 3) paths that a PE can possibly take to execute a node (depending on the nature of the node the PE is assigned). Let x_i represent the probability that a PE takes a particular path i. (x_i can be computed by looking at the mix of instructions, number of PEs in the system, number of nodes in graph etc.) In the state diagram, with every path there is associated a number n_i, which indicates the number of times a PE traverses path i to complete the execution of a node.

Let T_i represent the total time taken by the PE to execute a node along a path i. T_i is the sum of all the times that a PE spends in each of the states along the path i, which consists of (a) the wait time for a node, (b) wait time for input(s) to the node (if not available), (c) compute time of the operation specified by the node, (d) wait time on the result bus for transmission of the result computed, (e) wait time for the required number of acknowledgement(s), etc. In some cases such as accesses to the global memory, execution of branch nodes etc. the PE goes through different states, as detailed in the state diagram. If t_i represents the time taken by a PE for one traversal along the path i, then

$$T_i = n_i.t_i$$

t_i is the summation of times spent by a PE in the various states along a particular path.

Hence the average time taken by a PE to execute a node along any path is

$$T_{av} = \sum_{i=1}^{b} x_i.T_i$$

3.5 Time taken for execution of the graph, G_m

A program represented by a data flow graph G_m, is executed level by level, since the nodes in a graph are assigned levels, according to the data dependencies among them. Let p be the number of PEs in the system, L the current level being executed and m_L the number of nodes in that level. At most only one level can be in the state of execution. m_L nodes in a level are executed by the p PEs in $\lceil m_L/p \rceil$ steps, each step taking T_{av} units of time. Therefore execution time of a level L is

$$e_L = \left\lceil \frac{m_L}{p} \right\rceil T_{av}$$

Let the number of levels in the graph G_m be L_m. The next level of the graph enters the state of execution only when the current level completes execution. Therefore the sum of execution times of all levels of the graph, i.e, the execution time of the graph G_m is

$$E_m = \sum_{L=1}^{L_m} e_L$$

3.6 Other Performance measures

This model can be used to measure other performance parameters also. One of these performance parameter measurement is illustrated here.

3.6.1 %-PE utilization during execution of a program : During the execution of a program, some or all the PEs are used in producing the results. If f_p denotes the total time spent by p PEs existing in the system for computing the m nodes of the graph G_m then

$$f_p = \sum_{L=1}^{L_m} \left\lceil \frac{m_L}{p} \right\rceil e_{t,m}$$

where, $e_{t,m}$ is the time of computation of a node, on an average. Then the total time spent by all the PEs in computation of m nodes is,

$$\%\text{PEutil} = \frac{f_p}{E_m} * 100$$

4 Using the model

From the expression for E_m, it is clear that the model can be used to compute the excecution time of any program conceived as a data flow graph, whose nodes are of a desired granularity. Note that representation of resident parallelism in a program is an inverse function of the granularity of the corresponding graph. Execution time of a program represented as a fine grain data flow graph has high overheads, whereas that which has coarse nodes has reduced parallelism. This model can be used to obtain the best suited representation of a given program.

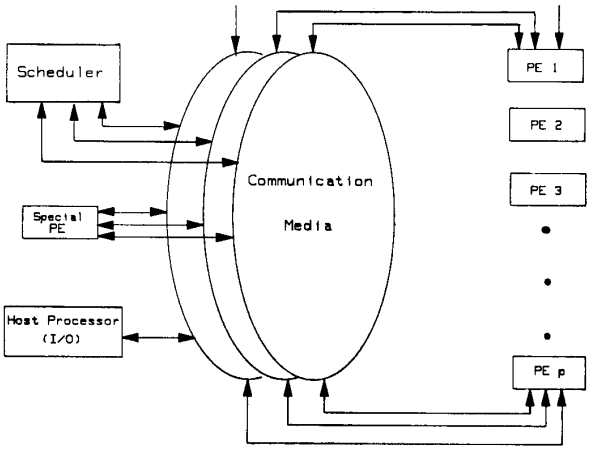

Fig. 1 Schematic diagram of the proposed machine

We use this model to study the behaviour of large programs (of the order of 10,000 fine grain nodes) and small programs (of the order of 900 fine grain nodes)[5]. In both cases we represent the programs by data flow graphs of varying grain size. We vary the number of PEs in the system between 1 and 16, the delays introduced by the scheduler between 1 and 100 and the raw speed of the communication between 1 and 10. Here we present the results.

1. The rise in execution time is only a sublinear function of decrease in speed of the communication media.
2. The machine shows poor behaviour for slow scheduler speeds.
3. Beyond 8 PEs, the machine does not show improvement in speed up that is commesurate with the number of PEs.
4. The number of nodes in the best suited graph decreases with decreasing number of PEs.

5 Conclusion

In this paper, we presented a method to model a data flow oriented multiprocessor system that incorporates dynamic scheduling policy. The model is based on a processor state diagram. By associating wait times for each of the state (during the execution of a program) delays incurred by the processor waiting on events are captured. This scheme of modelling can be extended for studying the performance of the machine using static and quasi-dynamic scheduling strategies as well[5].

References

[1] Arvind, R. S. Nikhil and K. Pingali, "I-structures: Data Structures for Parallel Computing", *Proc. workshop on Graph Reduction*, Los Alamos NM, Sep.-Oct. 1986.

[2] J.B. Dennis, "Data Flow Supercomputers", *Computer*, Nov. 1980, pp. 48-56.

[3] Arvind, V. Kathail and K. Pingali, "A Dataflow Architecture with Tagged Tokens", *Technical Memo 174*, Lab for Computer Science, MIT, Sept. 1980.

[4] L. Klienrock, "Queueing Systems (vol. II) Computer Applications" John Wiley and Sons, New York, 1976.

[5] Ranjani Narayan, "Performance Analysis of a Multiprocessor Machine based on Data Flow Principles", Ph.D. Thesis, Dept. of Computer Science and Automation, Indian Institute of Science, India, 1989.

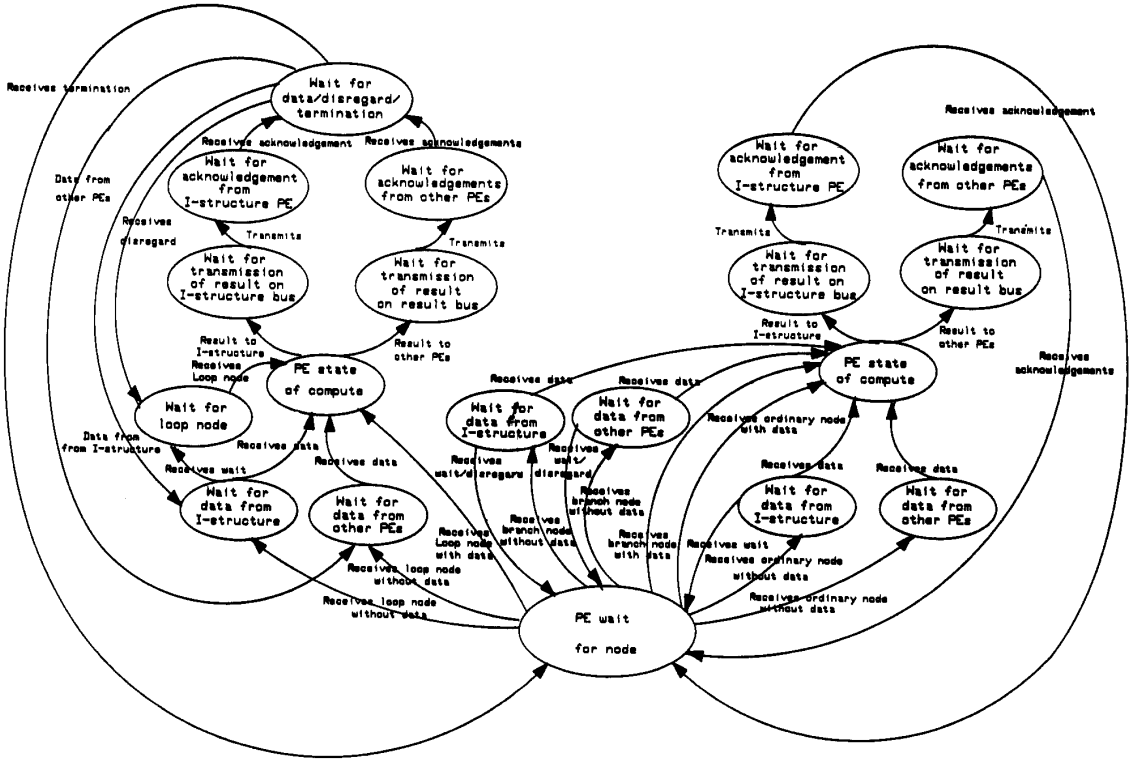

Fig. 2 State diagram of a PE during execution of a node.