TECHNICAL REPORT NO. 304

Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000

by

Francois Bodin, Daniel Windheiser, William Jalby,

Daya Atapattu, Mannho Lee, and Dennis Gannon

February 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

TECHNICAL REPORT NO. 304

Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000

by

Francois Bodin, Daniel Windheiser, William Jalby,
Daya Atapattu, Mannho Lee, and Dennis Gannon

February 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000

François Bodin, Daniel Windheiser, William Jalby
Indiana University, IRISA Rennes and INRIA France
Daya Atapattu, Mannho Lee, Dennis Gannon
Department of Computer Science, Indiana University

Abstract

The techniques of "load/store" memory reference modeling is based on deriving performance characteristics of the memory architecture of a computer by looking at the behavior of simple sequences of load, store and nop (null operation) instructions. The resulting data base can be used to match load/store templates against algorithm kernels to predict performance or as a source of data for testing analytical models of the architecture. In this paper we study the BBN GP1000 parallel processing system. We show how to build a subset of the load/store kernels needed to characterize the machine and illustrate the behavior of a simple model based on the data.

This material is based in part on work supported by the National Science Foundation under grant number CCR 88-03432.

1 Introduction

The goal of performance *analysis* is to understand how a computer behaves both as a complete system and as a collection of subsystems. It is through performance analysis that we can see if the implementation of an architecture fulfills the goals of its design. As a subdiscipline of computer science, performance analysis involves both the theoretical models of machine behavior as well as the experimental analysis of real hardware. On the other hand, performance *prediction* is the process of understanding how the characteristics of an algorithm relate to the constraints of an architecture. The goal is to be able to give an "a priori" estimate of how well a given computation will do on a given machine with an eye towards understanding which subsystem of the computer dominates the profile of performance for that algorithm. Unlike traditional asymptotic complexity analysis, performance prediction must be concerned with the real values of the constants in a formula for execution time in an algorithm and have analytical models of the hardware that are both accurate and easy to relate to algorithmic properties of the computation.

As both and performance prediction and performance analysis techniques mature, it should be possible to build tools that help users (and, eventually, compilers) understand why a given computations run slowly and how to redesign the algorithms to optimize performance.

Before these goals can be accomplished we need to do a lot of work to help us refine our experimental methodology and our understanding of how to relate experimental work to the hypotheses derived from our theories of machine behavior. One methodological approach to studying supercomputer behavior was invented by Harry Wijshoff, Kyle Gallivan and William Jalby at CSRD in Urbana Illinois [GGJ+89]. Called the "load/store" modeling method, it was designed to help understand the relation of the Alliant FX/8 vector instruction set to the memory hierarchy of that machine. Briefly stated, they discovered that these two subsystems can be characterized by the performance of the machine on a set of "template" sequences of vector load, store and "nop" (null operation) instructions. Furthermore, more complex computational kernels can be reduced to a sequence of such templates and performance prediction can be expressed in terms of the behavior of these basis templates.

In this paper we make a first attempt to extend this technique to a large MIMD shared memory multiprocessor that has no vector instructions. More specifically, we examine the BBN GP1000 "Butterfly" parallel processing system. Our eventual goal in this project is twofold:

1. We wish to build a set of load/store templates that can be used to automatically characterize any machine in the same architectural class (for example, the IBM RP3, the new BBN MP2000, Illinois Cedar and others).

2. We wish to find complete analytical models of the hardware which match the results of the experiments with high accuracy. These analytical models can be embedded into restructuring compilers as components of performance prediction modules that help to guide program restructuring.

These goals are far from complete. In this paper we will illustrate some of the components of the load/store templates and try to sketch a design for a more complete system. In section 2 of this paper we will describe the Butterfly architecture. In section 3 of the paper we look at the design of the initial set of templates and examine the experimental results. In section 4 we look at a simple analytical model that uses some of this data to predict performance. We conclude with some observations about the future of this project.

2 The GP1000 Architecture

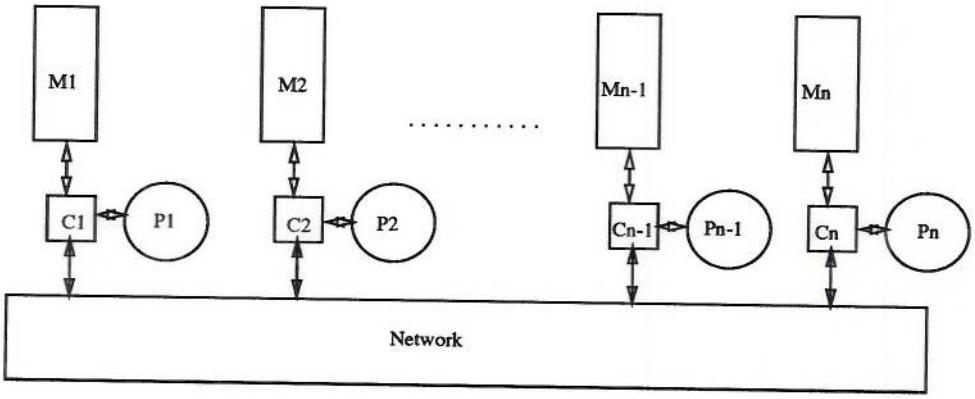

Figure 1: Butterfly Architecture

The BBN GP1000 Butterfly parallel processor [CGS+85] is a shared memory MIMD machine composed of Motorola 8 Mhz 68020 processors connected by network, as illustrated in figure 1. The network is composed by 4×4 switches (figure 2) interconnected with a shuffle type permutation. The bandwidth of each path of a switch is 32 megabit per second. Thus a 32 processors version of the switch (like the one currently available at Indiana University where

these experiments were run) has a capacity of 1024 Mbits/sec. When contention occurs in the network one of the messages proceeds to its destination and the other is retransmitted after a random period and it may try an alternate path in the network. The number of alternate paths is determined at boot time. In our case we have two alternate paths.

The processor is coupled with a 2901-based bits slice co-processor (C_i on figure 1). The co-processor mediates the interaction between processor, memory and the network. Each processor has a special link to a memory bank, provided by the co-processor. However memory management is implemented so that the memory banks constitute a single physical address space. The Uniform System software library and the tightly coupled architecture of the Butterfly compose an environment in which tasks may be distributed to processors without regard to the physical location of the data associated to the tasks. However, a single data reference is slower if the data is accessed through the network rather than directly. A special feature is provided to transfer blocks of data through the network. The GP1000 runs the Mach operating system.

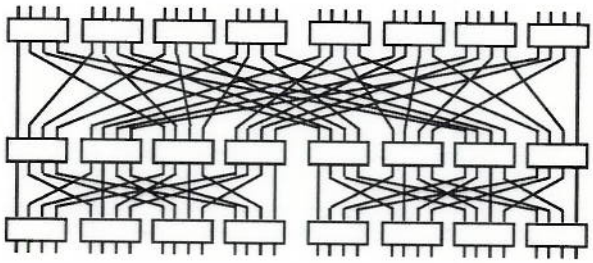

Figure 2: Butterfly network for 32 processors

Definition 2.1 *In the following we will call an* **external** *access to memory banks i a memory access done through the network (i.e. done by processor $j \neq i$), and a* **local** *access a memory access done through the special link between a processor and its memory bank.*

3 The Load/Store Kernels

Prior to this study, the previous versions of the load/store experiments were carried out on bus based multiprocessor systems. For the BBN Butterfly we needed to refocus the traditional structure of the kernels so that we could characterize a machine with a distributed but shared memory. We set out by designing several initial experiments. The goal was

to highlight three characteristics of the machine involving the performance of single word references and block transfer operations. More specifically, we wanted to study

1. the behavior of the external access: The problem here is to see the effect of the memory and network contention on the external access when a "hot spot" occurs. By hot spot, we mean a memory module that is frequently reference by a large number of processors during some phase of program execution.

2. the Interaction between local and external access: We wanted to know if external access modify the behavior of local memory access? In other words, if a processor is connected to a "hot" memory module will the external references made to that memory have a significant effect on the performance of the local requests to that memory. Conversely, we also wanted to know if external memory access are degraded by local access.

3. the behavior of the network: in the absence of memory hot spots, is there a degradation of the performance due to network conflict? For example, if one set of processors is actively using one set of memory modules, will the traffic that they create on the network have a significant impact on other, unrelated computation?

The load/store kernels are not like standard benchmarks. They are not parts of "real" computations like the Livermore loop kernels, but rather, they are synthetic loads that are designed to stress a particular aspect of the system. The advantages of this approach are multiple:

- The loops are directly generated in assembly code: this removes side effect, and the experiments are independent of the the compiler and its optimizer.

- They are simple: the interpretation is easier, only memory reference are done.

- They are systematic: most parts of the experiments are generated automatically by the load/store testing software.

Each kernel takes the form shown in the concurrent loop show below. Each process executes the body of the concurrent loop. Inside that code is synchronization to make sure that each processor start executing the body of the test at the same time. (The clock is local to a processor, but the synchronization mechanism ensures that the drift is limited to 2 cycles.) Inside the timed part of the code is a loop which contains the template.

```
Concurrent loop
 {
   Synchronization;
   t1 = get_time();
   for (i = 0; i < n ; i++)
     {
       load/store template;
     }
   t2 = get_time();
 }
```

Each load/store template is a sequence of machine instructions that carries out 32 memory references, either loads or stores. (We have not yet mixed write and reads, but this will be done in the future.) Furthermore, to make sure the timing is accurate, we have unrolled the loop to make sure that the loop overhead cost is not significant.

From now, we will only describe the load templates (the store templates are derived by replacing the load instruction by store instructions). There are only three instructions in the kernels; loads of from an address x, denoted by L_x, stores to an address x, denoted by S_x, and null operations used to cause the processor to waist a fixed amount of time and denoted by N. The kernel is just a string of these operations and we use exponentiation to denoted a repeated sequence. For example $(XY)^k$ means the string with the pattern XY repeated k times.

We used two different sets of load templates:

1. Nop
   $(L_x N^k)^{32}$ with $k = 0, 1, 2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25$ This set of templates lets us experiment with the rate at which a processor makes a request to memory. By increasing the number of nops, we decrease the memory request rate.

2. Spatial commutativity
   $(L_x^k N^k)^{\frac{32}{k}}$ with $k = 1, 2, 4, 8, 16, 32$ In this set, the number of nops is constant. However the nops and the load instructions are interleaved in different ways. These kernels are intended to highlight the effect of of the temporal distribution of the memory requests.

3.1 Memory Hot Spot Experiments

In our first experiments we used a cluster of 21 processors, one of which is the main processor running system and the main procedures (it will be called P_0). The other processors are labeled as follows.

- P_1 does only local requests (references to an array which is allocated to memory bank M_1 which is local to P_1).
- all other processors do external requests to the same memory bank M_1.

We used two sets of processing configurations:

1. P_1 is active making local requests and the number of active processors doing external requests is increased from 1 to 19.
2. P_1 is inactive and the number of active processors doing external requests is increased from 1 to 19.

The curves obtained from the experiments are plots of Mega Transactions per second on the vertical axis and active processors on the horizontal axis. A Mega Transaction is either one million data movement operations like, integer loads or floating point stores, with the exact meaning obvious from the context. The details of the results are given below.

- Figures 6 and 5 represent the total external access bandwidth, EAB in Mega Transaction per second for integer (32 bits) load and store plotted as a function of the number of active processors p by

$$EAB(p) = \sum_{i=2}^{p} MT(i,p)$$

with $MT(i,p)$ is the bandwidth obtained for processor P_i given that p processors are active. if P_i is inactive or i > p, $MT(i,p) = 0$.

- Figures 7 and 8 represent the total external access bandwidth in Mega Transaction per second for floating point (32 bits) load and store. This is the same as the above but processor P_1 is inactive in this case.
- Figures 9 and 10 represent the local access bandwidth LAB as a function of the number of active processors p for integer load and store:

$$LAB(p) = MT(1,p)$$

type	load int	store int	load float	store float
local	0.55	0.412	2.1	2.55
BBN local	0.53	0.38	?	?
external	7.125	4.375	9.088	4.736
BBN ext.	7.00	4.00	?	?

Table 1: Values obtained from the experiments for basic load/store operations in microseconds and times published in the BBN literature.

- Figures 11, 12 and 13 describe the bandwidth in the case that P_1 is doing scalar loads and stores but processors P_2 through P_{20} are doing block transfer operations.

As a normalization test we compared our measurements of the time it takes for a processor to do a local access and an external access on a system with no other processors active with numbers published by BBN. These numbers are listed in table 1. As the reader can see the agreement is fairly close.

A close look at this set of bandwidth profiles generated by these experiments reveals some interesting facts. As illustrated by figure 6 and 5 a saturation is rapidly reached for single load and store operations. The curves for different numbers of nops merge when the delay between to accepted request become greater than the time consumed by the nop operations. However there is an interesting difference between the load and the store case. The load case reaches a peak with a few processors and then drops off slightly. We observe the same phenomenon for floating point load/store.

The first idea that comes to mind to explain this difference is to argue in terms of network contention, i. e. a memory load requires 2 messages: one from the requesting processor to the PNC co-processor that controls the memory module and a second from that PNC co-processor back to the requesting processor with the value read. On the other hand, a store operation requires only one message because a reply is not needed. One might suggest that the density of network traffic would be greater for load operations than for store operations. However, if network contention occurs it must be greatly dependent on the static memory reference density which we define as follows

type	load int	store int	load float	store float
B w P_1	0.248	0.27	0.249	0.272
B	0.249	0.273	0.265	0.275

Table 2: Total bandwidth in Megatransaction per second for for external access. $B\ w\ P_1$ is for experiments with processor P_1 active and B with processor P_1 inactive

Definition 3.1 *We call a* **static memory reference density** *the ratio:*

$$\frac{number\ of\ memory\ access}{ideal\ computation\ time}$$

where ideal computation time means the best possible without conflict in the network.

If we consider that a load is two requests on the network and a store one,the computation of the density D is

$$D = \frac{req}{T + 2 * N * nop}$$

where req the number of request on the network, T the best time for an operation, and $N * nops$ is the time for the number of nops. Using this formula and the data in table 1 we get a density for a store with no nop D_s^0 of 0.028 messages per machine cycle and the density for a load with 25 nops $D_l^{25}$is 0.018 messages per machine cycle. We see that the store reaches a greater static density than load with 25 nops. If the behavior of the load operation were completely explained by the memory density we would see the same, or worse behavior for the store. This is not the case as the reader can see.

This shows that the behavior of load operation is not completely determined by the static density of request. The explanation of this difference in behavior between load and store seems to be the following. When a load request is treated by the destination co-processor does not accept other requests until it has sent the data back to the source processor. If the network is close to saturation the reply may be rejected. This can cause the other requests to try alternate paths because all the request are not accepted before the reply is sent. So the result is a saturation of the network. In other words the response time of a load is dependent on the network congestion, in contrast to the store service time which is independent of the network traffic.

The maximum bandwidth obtained in each case is summarize in table 2.

Figures 13 and 12 are results from blocks transfer experiment. Difference between load and store has not completely disappeared; the maximum bandwidth is obtained with store operations. The maximum total bandwidth obtained for load block transfer is 0.802, and for store 0.909 Megawords per second which is very near the maximum bandwidth of the network being 1 Megaword per second per processor.

Local/ External Access Interactions. Figures 9 and 10 illustrates the perturbation of local memory accesses caused by external accesses. The bandwidth of local access decreases significantly when the density of request is high (0 nops) however if only 3 nops are inserted between each load/store operations this differences becomes very small. So simple external memory accesses does not really perturbate local access, because in real programs the density obtained with 3 nops is not frequently reached. In case of floating point load/store no perturbation by external access are noticed, the density of request being smaller than for integer load/store.

In the case of integer load/store the external access are not perturbed by local access. We don't see any significant difference in the result when processor P_1 does local access or when it is inactive. However, in case of floating point load/store a difference appears in the case of load, the bandwidth reached for 5 processors is higher if the processor one is inactive as show in figure 8.

In the case of blocks transfer, an important behavior difference exists. Figure 11 illustrates the behavior of the processor P_1 when others processors do external access to the memory banks M_1. In contrast to single word access, blocks transfer decreases the bandwidth of processor one a lot. From the BBN manual we see that the memory bandwidth alway available for local access is 25% (during block transfer), however we can see that the resulting bandwidth for local access is less than 25% of the memory bandwidth. So it appears that contention occurs at the co-processor level.

3.2 Network Contention Experiments

The previous experiments focus on the behavior of one Memory module, the network and up to 20 processors. We see that there are some effects that are clearly related to network interactions, but in general these experiments characterize the bandwidth of the memory module.

MB	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}	P_{17}	P_{18}	P_{19}	P_{20}
Config 1	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2
Active	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Config 2	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2
Active	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Config 3	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2
Active	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Config 4	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2
Active	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Config 20	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2	20	19	1	2
Active	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 3: Configuration for network testing:
The first line (config) of each raw gives the memory bank number accessed by the processor , and the second line active processors (noted 1 if active 0 if not).

To get a better idea about how the network relates to the behavior of the machine we must design load/store templates that use more than one memory module. The first test of this type we have done are listed below.

1. Each processor does an external request to memory such that only one processor accesses one memory bank. We compare the total bandwidth obtained by increasing the number of processor. The total bandwidth must be a linear function of the number of processor because there is no sharing of data paths. Degeneration may result if either the permutation describing the connection between processors and memories cannot be routed by the network without conflict or if alternate paths are used when a request is rejected.

2. The second experiment is the same as the first but, in this case we increase the contention. Only 4 memory banks are used for 20 processors. The configuration used are illustrated by table 3.

The curves obtained from these experiments are respectively:

- Figures 14 and 15 which shows the total external access bandwidth, for integer load/store for experiment 1.

- Figures 16 and 17 which shows the same as the previous but for the second experiment (table 3).

In the first experiment one processor accesses only one external memory bank and one memory bank serves only one processor. The behavior of the network is linear only if the density of request is small (25 nops between stores). Note that the difference between the behavior of load and store still appears.

Figures 16 and 17 is the same experiment but with more contention (A memory bank serves 5 processors and the total request rates should be just enough to saturate it.) In this case we see an important degradation of the network and a collapse after 18 processors in the case of load. Furthermore, even at it best we see the total bandwidth of the 4 memory banks is far below that of 4 times the bandwidth of one memory bank. Consequently the network is now limiting behavior. For blocks transfer this phenomenon still exist but is less critical, the networks behavior is closer to a linear behavior for the first experiment and collapse for a greater number of processors for the second experiment.

This experiment seems to shows a "tree saturation effect" similar to that described in [PN85] for multi-stage blocking network. That is the saturation is propagated from the last switch connected to the memory back to the processors, inducing perturbation for other accesses to the memory. We believe there other models which account for this behavior, but we are not decided as to which is best.

There are a great deal more experiments that we have yet to complete before we have a complete set of templates that describe this type of shared, distributed memory computer. In the last section of this paper we will return to the topic of deciding which experiments are still to be run and we will try to show how the methodology will extend to other machines.

4 Predicting Performance on the GP1000

Once we have an established experimental methodology we can test models that attempt to describe machine behavior. Performance prediction, however, demands that we are able to model both algorithms as well as machines. To illustrate how one might approach the problem of predicting performance on the BBN GP1000 from the data we already have, we shall look at applying a simple model developed by K. Hwang and F. Briggs [HB84] to analyze interleaved memory systems and show how it can be adapted to the GP1000. Then

we apply that model to a simple algorithm, matrix vector multiply, to show that the model can be used to make performance predictions.

To analyze the behavior of the GP1000 memory system, we will view the execution of a parallel program on this a machines as follows. Label each memory module with an index in the range 1 to P. Assume a program has been partitioned to run in parallel on such a system where the data structures in the program have been distributed over the memory modules in some static manner determined by the compiler. During execution of the program each processor will access local memory and make remote references to the global memory modules. For now, assume each processors execution can be viewed as an exponentially distributed random process which generates strings of addresses to the external memory modules. We can view the k^{th} memory module as a server with a request queue of length L and a constant service rate of M_s words per cycle. Let $p_{i,k}$ be the probability that processor i makes its next external memory reference to module k. We shall assume that the program has been partitioned uniformly over the processors so that $p_{i,k} = p_{j,k}$ for all i and j so that the first subscript can be suppressed. Define R to be the average fraction of the time that a processor is NOT waiting for a memory reference to be completed from external memory. $1 - R$ is then the fraction of the time a processor is idle waiting for an external memory reference to complete. Also let Z_k be the fraction of time spent by each processor waiting for a request to memory module k. We would like to determine R but this depends on the program being executed. In particular, R is a function of the static density of external memory references in the code and the distribution p_k of those reference over the memory modules. If we consider the activity of one processor it can be characterized to be repeating the following pattern of activity. It computes for a number of cycles, then it makes an external memory request. On most systems it takes time to prepare the external request and ship it over the network to the memory module (figure 4). When the memory reference request arrives at the memory module the request is queued and eventually served by the memory module. The request is then returned over the network and decoded by the processor which returns to the computing state. Let *work_cycles* be the total time spent computing and let *request_cycles* be the total time preparing external references and the time the reference messages are traversing the network. Now let *static_cycles* be the total

$$static\_cycles = work\_cycles + request\_cycles.$$

Define

$$M_r = \frac{n}{static\_cycles}$$

which is the fraction of time a code spend doing the overhead of external memory references not counting the time spent waiting in queues and having external memory references serviced by the remote memory module. If we neglect the interaction between processors we have the following theorem due to Hwang and Briggs.

Theorem 4.1 *[HB84]. Let $\rho_k = Pp_k\frac{M_r}{M_s}$ for $k = 1...P$. The average time each processors spends doing computation, initiating, and decoding message R is related to the time spent waiting, Z_k, by the following equations.*

$$(1 - Z_k)^P + \rho_k R = 1$$

$$R + \sum_{k=1}^{P} Z_k = 1.$$

Proof The second equation is simply the statement that the total fraction of the time a processor spends busy and waiting on all memory references is exactly 1. The first equation is much more subtle. Following Hwang and Briggs, let

$$i_{k,j} = \begin{cases} 1 & \text{if processor j is not waiting for memory k} \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, let C be the average compute-memory request cycle time for the system, as show on figure 3, and let X_k be the probability that module k is busy. We have

$$X_k = 1 - E(i_{k,1}i_{k,2}...i_{k,P})$$

where $E(v)$ is the expected value of the random variable v. X_kM_s is the rate of completed requests to module k which, when the system is in equilibrium, is the same as rate of submitted requests to module k. On the other hand, $\frac{1}{C}$ is the rate of submitted requests and $\frac{Pp_k}{C}$ is the average rate of submitted requests to module k. Consequently we have the equality,

$$X_kM_s = \frac{Pp_k}{C}$$

For a given processor, each machine cycle is either a "work" cycle where the processor is accessing local data or computing, or it is a "request" cycle where it is waiting for an external

memory reference. By looking at the static code we can compute M_r which is the ratio of the number of requests to static cycles. Consequently,

$$\frac{1}{M_r} = \frac{static\_cycles}{number\_of\_requests}$$

and

$$\frac{1}{C} = \frac{number\_of\_requests}{total\_cycles}$$

so

$$R = \frac{static\_cycles}{total\_cycles} = \frac{1}{M_r C}$$

or

$$\frac{1}{C} = M_r R$$

Combining these equations gives

$$X_k = \frac{P p_k}{M_s C} = \rho_k R$$

and

$$E(i_{k,1} i_{k,2} ... i_{k,P}) + \rho_k R = 1.$$

But, because $i_{k,j}$ is binary, we have by symmetry

$$E(i_{k,j}) = 1 - Z_k$$

for all j. We now make a critical approximation by assuming that all the processors have noncorrelated activity and we get

$$E(i_{k,1} i_{k,2} ... i_{k,P}) = E(i_{k,1}) E(i_{k,2}) ... E(i_{k,P}) = (1 - Z_k)^P$$

and the result follows.

■

This theorem allows us to predict performance behavior of a given computation on a given machine if we know M_s the remote memory service time in terms of machine cycles, the number of processors P, the static external reference rate M_r and the the distribution of external references p_k.

4.1 Computing M_s and M_r

Definition 4.2 *Let $t_{i,j}$ be the time take by the processor i to execute r_i external memory request when $j-1$ other processors are referencing the same module. The absolute time Δ for an external request (including the time to send the and decode the network messages, but not the time used by the destination memory to service the request) is equal to*

$$\Delta = \frac{t_{i,1}}{r_{i,1}} - \frac{1}{M_s}$$

The memory service rate $\frac{1}{M_s}$ is

$$\frac{1}{M_s} \leq min_{j=1}^{P} \left(\frac{max_{i=1}^{j} t_{i,j}}{\sum_{i=1}^{n} r_i} \right)$$

A minimum value of $\frac{1}{M_s}$ is given by a configuration with two processors, one accessing a local bank and the other accessing the same bank in external. We can then to compute the number of cycles stolen to the processor accessing in local. If we do this computation we found a value minimum for the integer load $\frac{1}{M_s^l}$ with:

$$\frac{1}{M_s^l} \geq 1.523 \ 10^{-6}$$

or

$$M_s^l \leq 0.6565$$

megatransactions per second. This value is a minimum because load or store operations takes more time to complete than the memory request being served. Consequently, the processor that steels the cycles is able to use the free time of the memory.

Using the experiments described in the previous sections we can deduce that the approximate values of M_s for the load and store of a floating point value are

and for integer load and store we have

$$M_s^l \geq 0.2325 \quad M_s^s \geq 0.2619$$

The value of Δ is

$$\Delta^l = 2.75 \ 10^{-6} \ sec$$

and

$$\Delta^s = 0.52 \ 10^{-6} \ sec$$

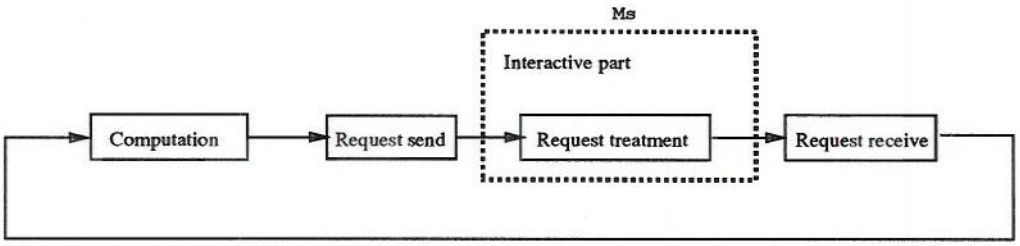

Figure 3: Computation cycle considered in the analytical model

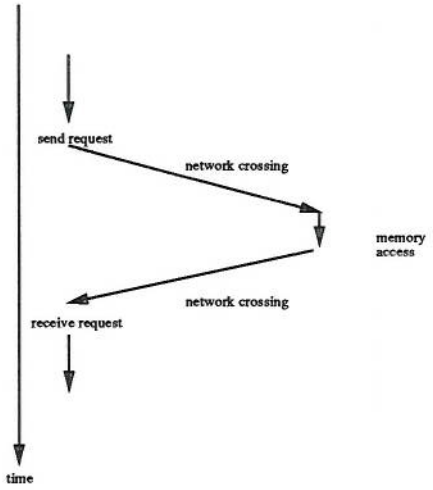

Figure 4: Memory reference decomposition

4.2 A Simple Example

Given these estimates of M_s we can now propose an experiment to derive the performance of a simple example. Consider the problem of a Matrix times vector product. Let A be an n by n array distributed such that column i is stored on memory module $imod(P)$. Consider the simple matrix vector multiplication where x and y are vectors of length n.

```
coloop(i = [1:n]){
   int j;
   for(j = 0; j < n; j++)
        y[j] = y[j] + a[i][j]*x[j];
}
```

(The coloop statement is a parallel loop in the dialect of parallel C used at Indiana University). There are a total of n^2 references to elements of A and $\frac{n^2}{P}$ come from each

processor. Elements of the variable x are referenced n^2 times and elements of y are referenced $2n^2$ times. In order to apply the model we need to compute the rate of reference to each memory module p_i, $i = 1, P$. Let us consider four cases

1. Both the variables y and x are in the memory module of processor P_1 and the array a has been distributed uniformly over the remaining modules. In this case, the variable y, which accounts for one half of all the references and x which accounts for one quarter of the rest is located in the same place. Consequently we have $p_1 = 0.75 + \frac{1}{4P}$ and $P_i = \frac{1}{4P}$ for $i = 2, P$.

2. The variable y is in memory P_1 and x is in memory P_2 and a is uniformly distributed. In this case $p_1 = 0.5 + \frac{1}{4P}$, $p_2 = 0.25 + \frac{1}{4P}$ and $p_i = \frac{1}{4P}$ for $i = 3, P$.

3. The variable y is local to each processor and x is in memory module P_1 and a is distributed. In this case $p_1 = 0.5 + \frac{1}{2P}$ and $P_i = \frac{1}{2P}$ for $i = 2, P$.

4. Both x and y are held as local copies in each processor and a is distributed. In this case $p_i = \frac{1}{P}$ for $i = 1, P$.

We seek the value of R and the speed-up $S_P = PR$ in each case. All that is left to do is compute M_r for each case and solve the equations in described in the theorem. To accomplish this we have cheated slightly. The computation of M_r requires that we know the static memory reference rate which means we need to know the total number of external reference (which is obviously $4n^2$) and the execution time based on external references but not counting the service time. To do this we ran the program on one processor with all local references. Call this time T_0. To correct for the fact that the array reference were supposed to come from external memory we have computed M_r as

$$M_r = \frac{4n^2}{T_0 + 4n^2(\Delta - 0.5)}.$$

The term Δ adds the cost of sending each array reference through the network and subtracting 2.0 deducts the cost of the local memory reference in T_0. This expression is valid in the case that all references to the array are non-local which is valid for cases 1 and 2 above. In case 3 the correct formula is

$$M_r = \frac{2n^2}{T_0 + 2n^2(\Delta - 0.5)}$$

type	P=1	P=2	P=4	P=8	P=16
x and y in P_1	0.60	1.09	1.72	1.89	1.93
actual speedup	0.75	1.02	1.21	1.45	1.70
y local	0.8	1.39	2.66	4.41	4.81
actual speedup	0.62	1.39	2.38	3.99	3.94
x in P_1 and y in P_2	-	1.13	2.38	2.80	2.87
actual speedup	0.488	0.9438	1.738	2.272	2.37
x and y local	0.82	1.64	3.27	6.53	13.05
actual speedup	0.84	1.68	3.297	6.37	12.07

Table 4: Predicted and actual speed-up for the Matrix times vector example

and in case 4 we have

$$M_r = \frac{n^2}{T_0 + n^2(\Delta - 0.5)}.$$

Using these values in the equations in the theorem we obtain the speed-up predictions given in table 4.

As the reader can see, each of these predictions results in an estimate that is high for large numbers of processors (and off by as much as 24% in the worse cases). Furthermore the failure is the greatest in the cases where the hot spots dominate performance. A reasonable explanation of this behavior would be based on the observation that the model does not take into account the true behavior of the network when the density of traffic is high or if there is a hot spot. In particular, the assumption that the memory element queues the reference requests does not accurately reflect the fact that in the real hardware the references are dropped and have to be reissued. Furthermore we have not taken care to integrate the effect that the external references have on reducing the bandwidth available to the local processor.

5 Conclusion

In this paper we have described a set of experiments to systematically measure the behavior of a shared memory multiprocessor based on a distributed architecture. However to get a complete loads/stores kernel for shared memory multiprocessor some extension to the kernel

have to be done:

- Hot spot with mixed single word and block loads/stores.
- Experiment with the side effects of a hot spot on other part of the network.
- Experiment with the symmetry of the network (measure the effect of the permutation on performances).
- Measure the effect of the network routing strategy.

A first approximation analytical model for performances prediction has also been presented. This model lacks of accuracy, not considering the network side effect (i.e. the network crossing is assumed to be constant). An analytical model suitable for performance prediction must be able to incorporate both network effect and memory contention in a coupled way.

Future works include the development of the previous comments, and also the experiments of other machine such as the new BBN machine. This first study has proven that the loads/stores model is a suitable tool for the low level analysis of multiprocessor architectures. The results of this analysis allows a better understanding of the machine behavior, but also constitutes a guide for the development of new analytical model closer to the real machine by highlighting important bottleneck of the machine. This paper illustrates this fact, by showing exactly where the model fail, and also its validity domain.

References

[CGS+85] W. Crowther, J. Goodhue, E. Starr, R. Thoma, W. Milliken, and T. Blackadar. Performance measurement on a 128-node butterfly parallel processor. *International Conference on Parallel Processing*, pages 531–540, 1985.

[GGJ+89] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff. Behavioral characterization of multiprocessor memory systems: A case study. *ACM Sigmetrics Performance Evaluation Review*, 17:79–88, May 1989.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* Mc Graw-Hill, 1984.

[PN85] G. P. Pfister and V. A. Norton. Host spot contention and combining in multistage interconnection networks. *International Conference on Parallel Processing*, pages 790–797, 1985.

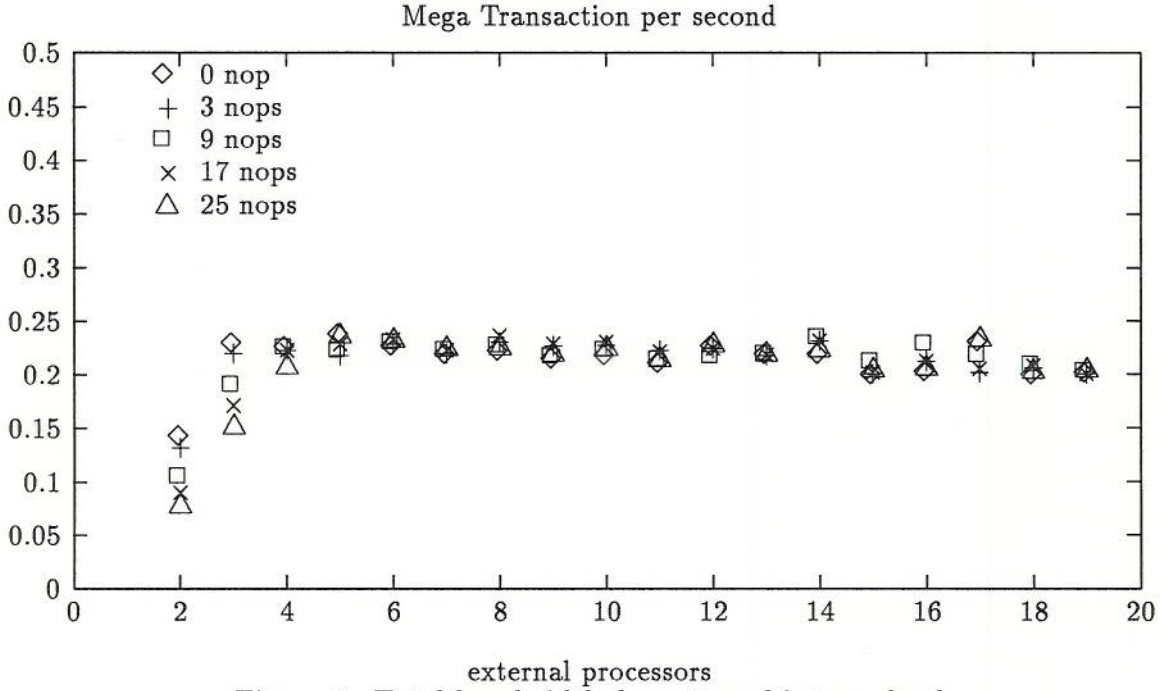

Figure 5: Total bandwidth for external integer load

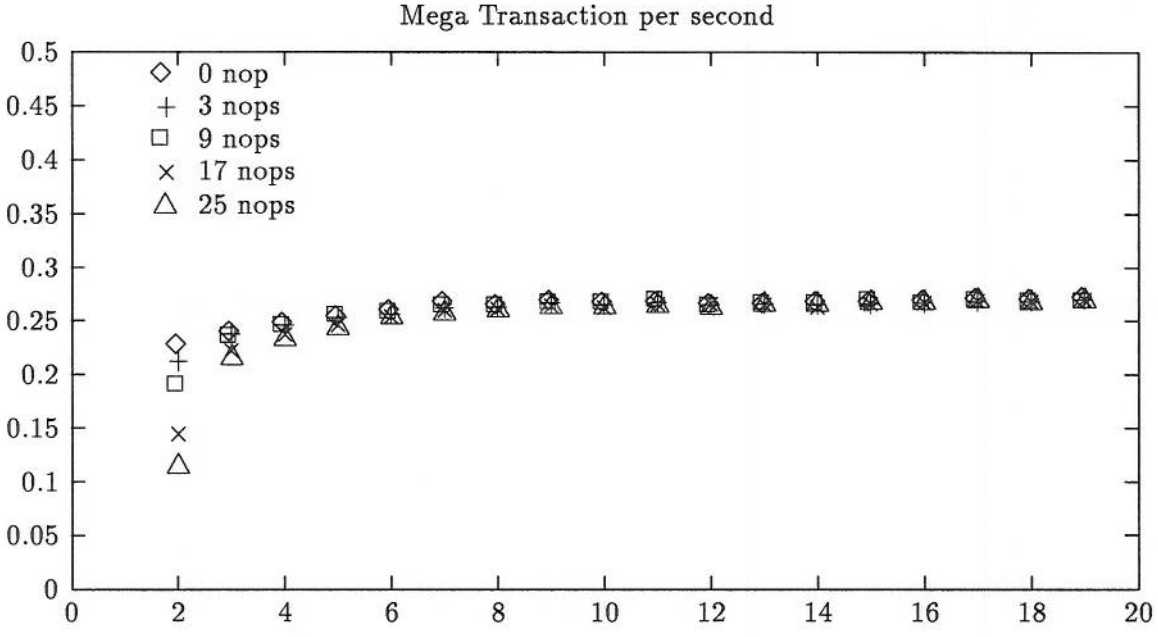

Figure 6: Total bandwidth for external integer store

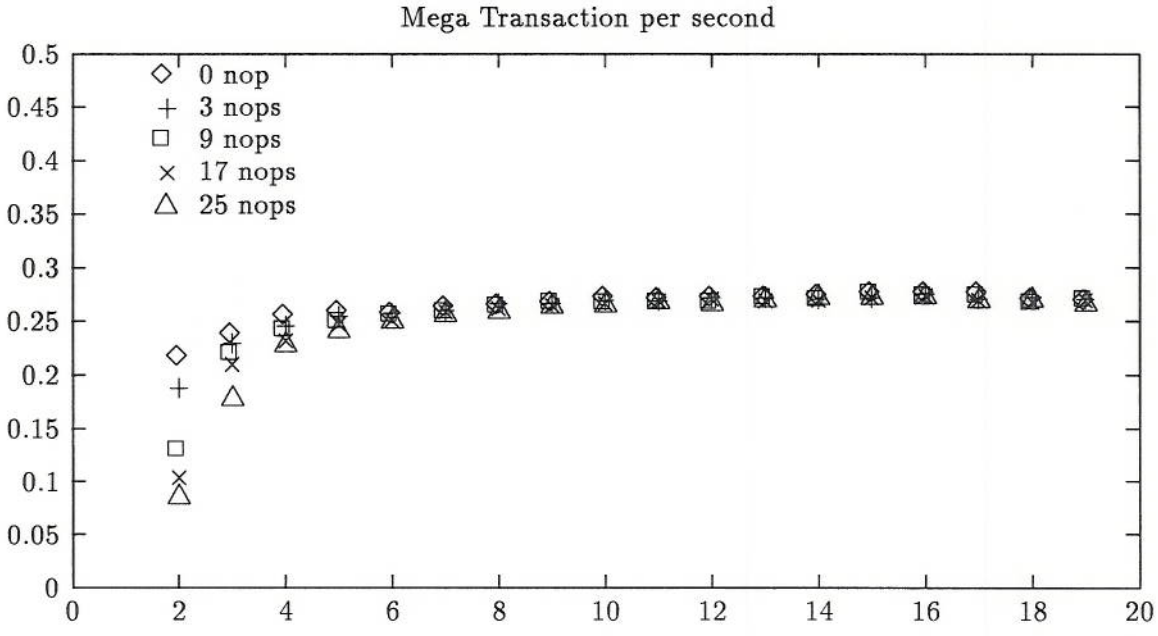

Figure 7: Total bandwidth for external float store

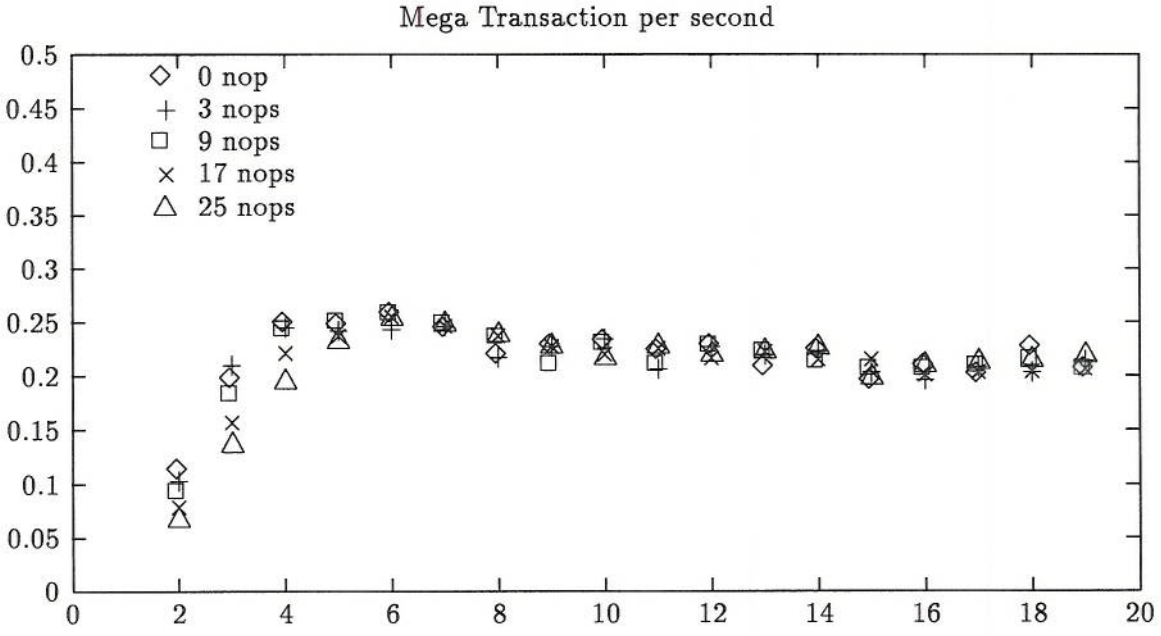

Figure 8: Total bandwidth for external float load

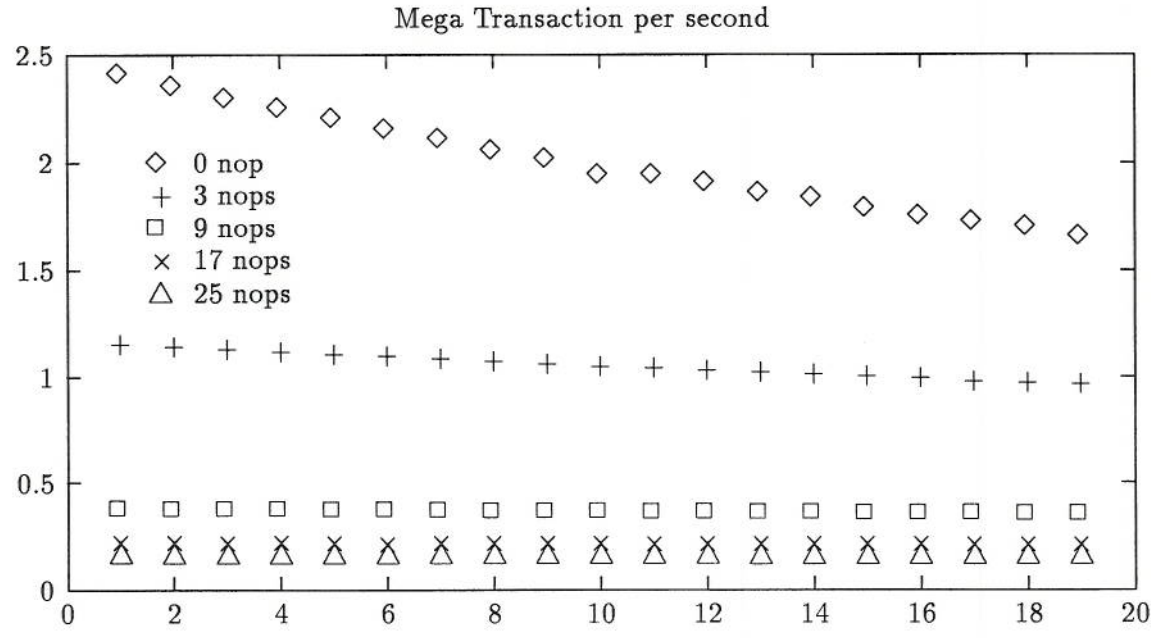

Figure 9: Bandwidth for local integer store

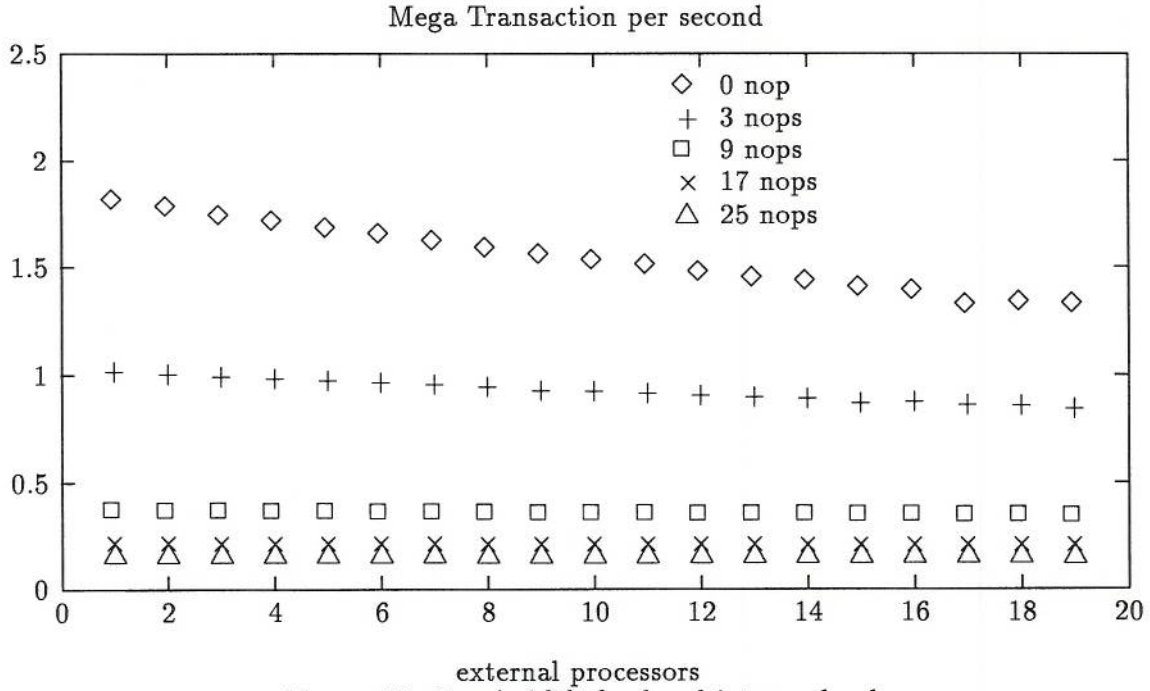

Figure 10: Bandwidth for local integer load

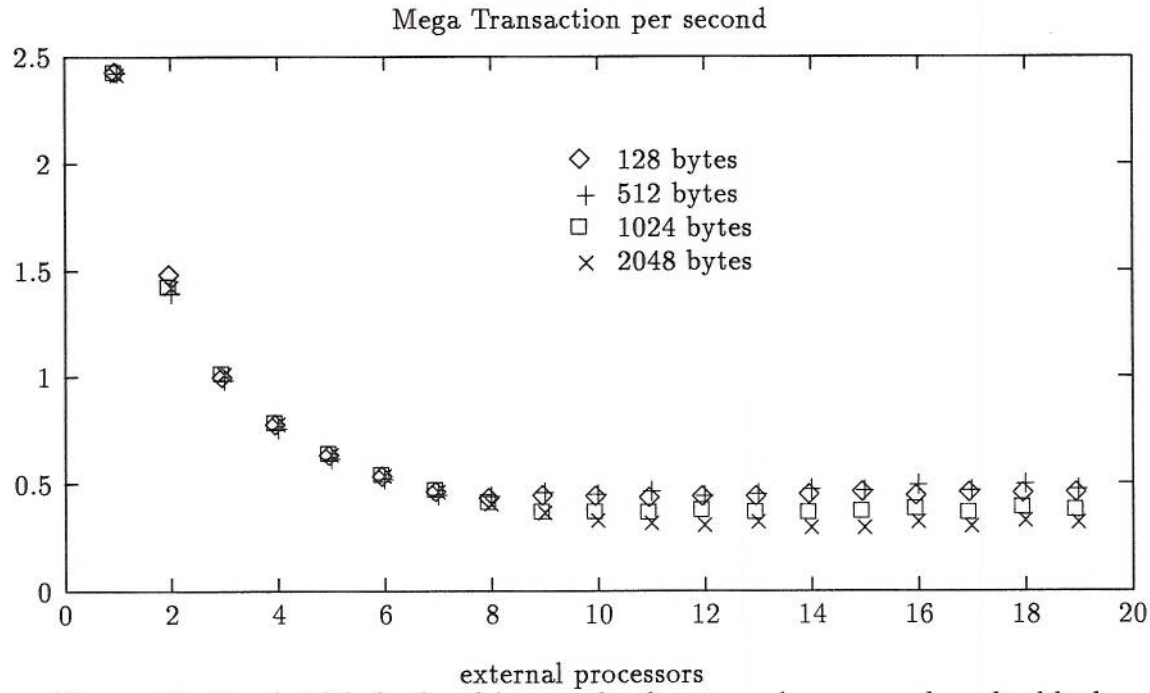

Figure 11: Bandwidth for local integer load, external accesses done by block

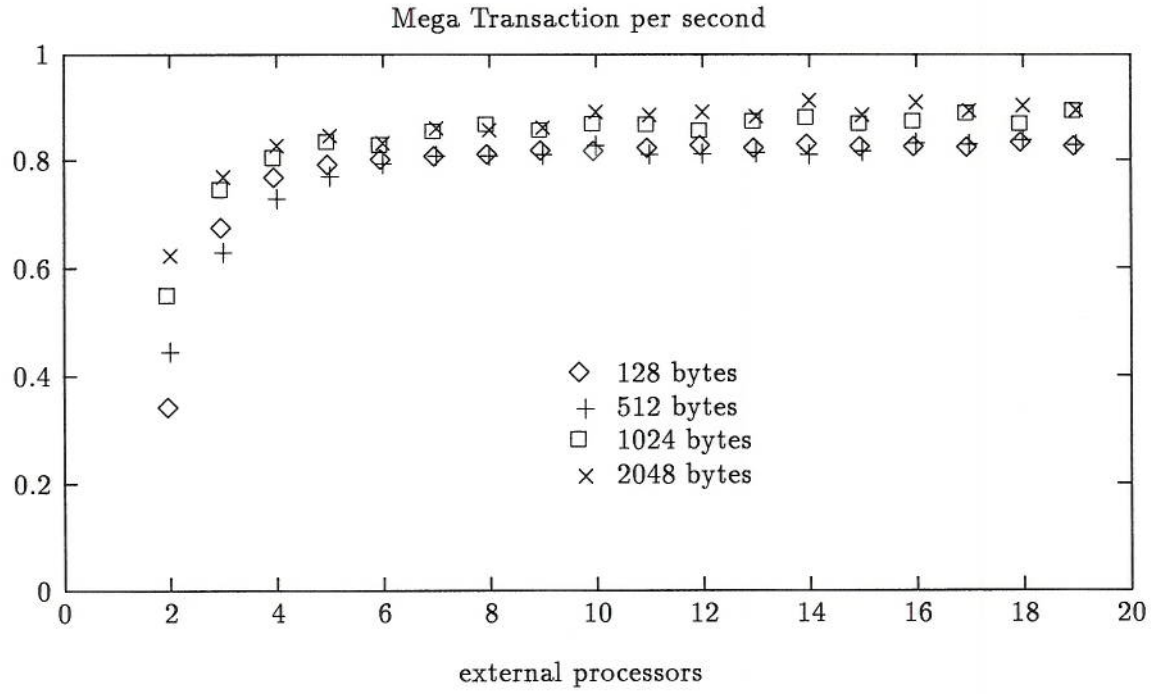

Figure 12: Bandwidth for integer store access by block

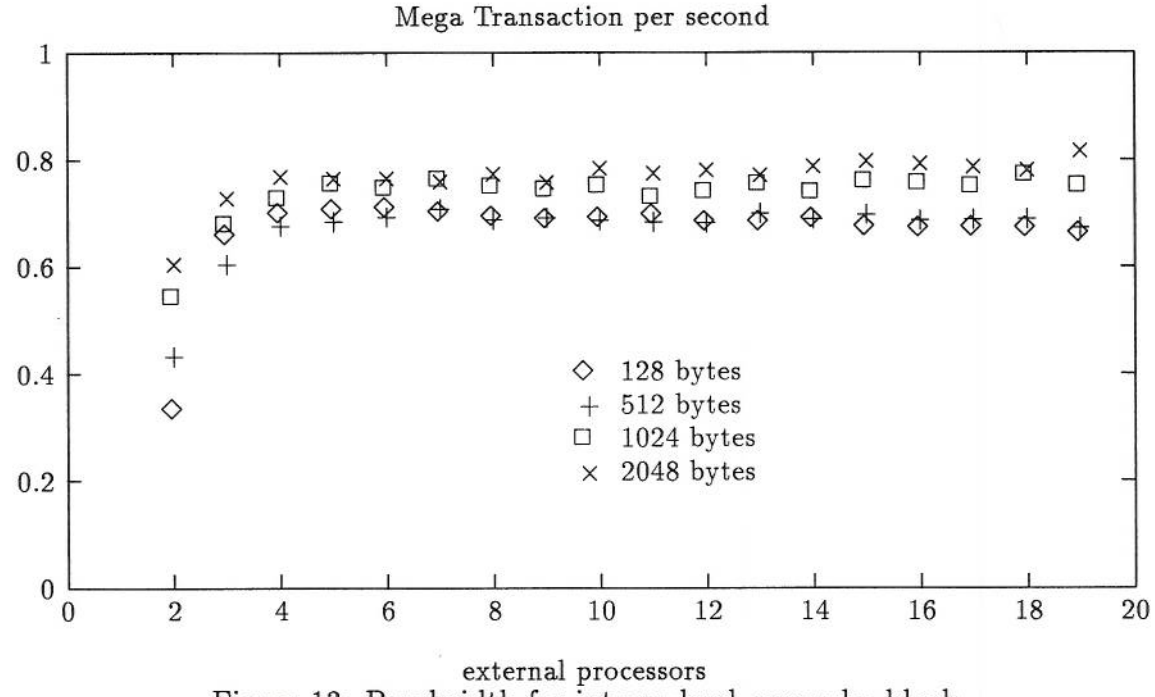

Figure 13: Bandwidth for integer load access by block

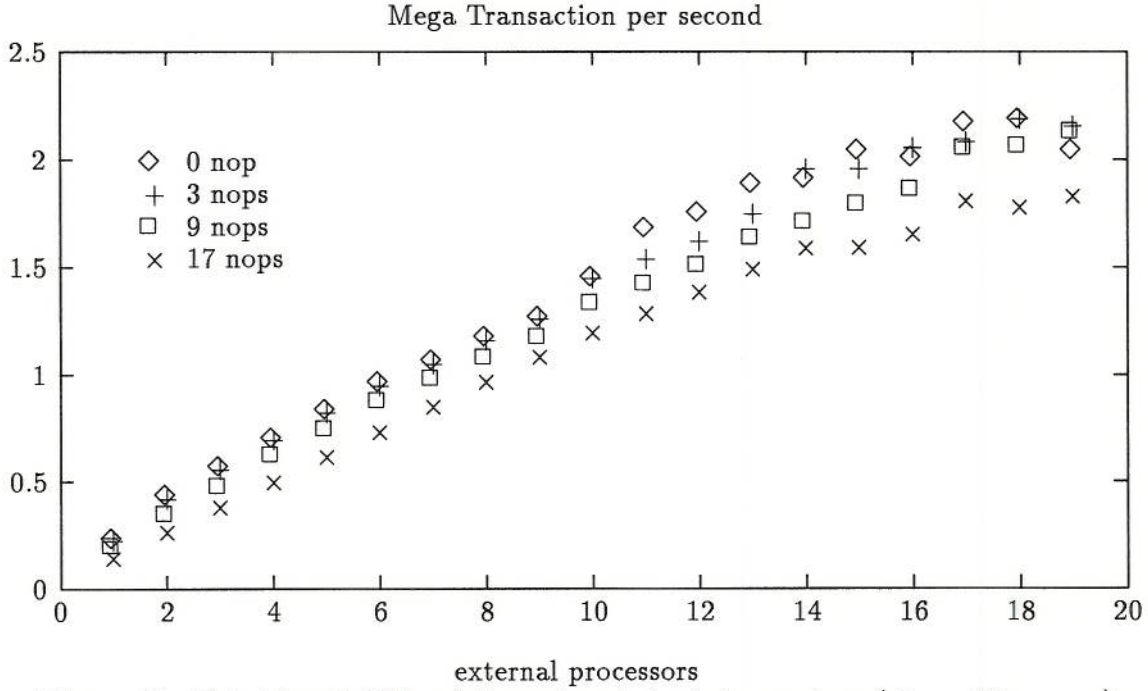

Figure 14: Total bandwidth of the network for integer store ($P_i \rightarrow M_{i+1 mod 20}$)

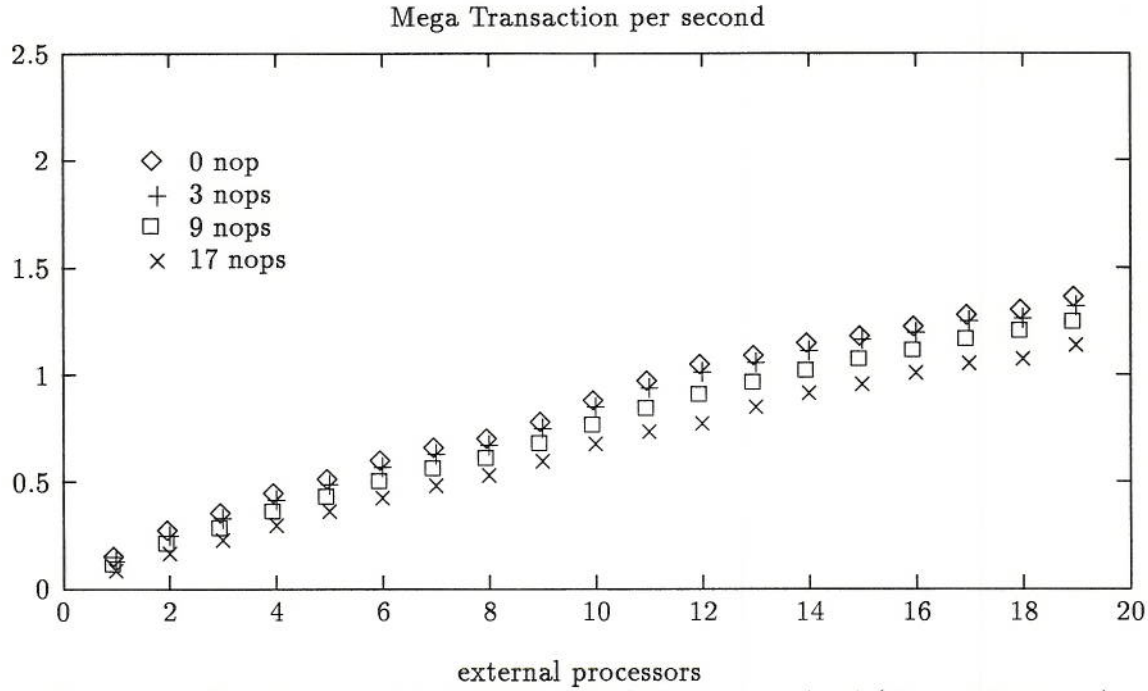

Figure 15: Total bandwidth of the network for integer load ($P_i \rightarrow M_{i+1 mod 20}$)

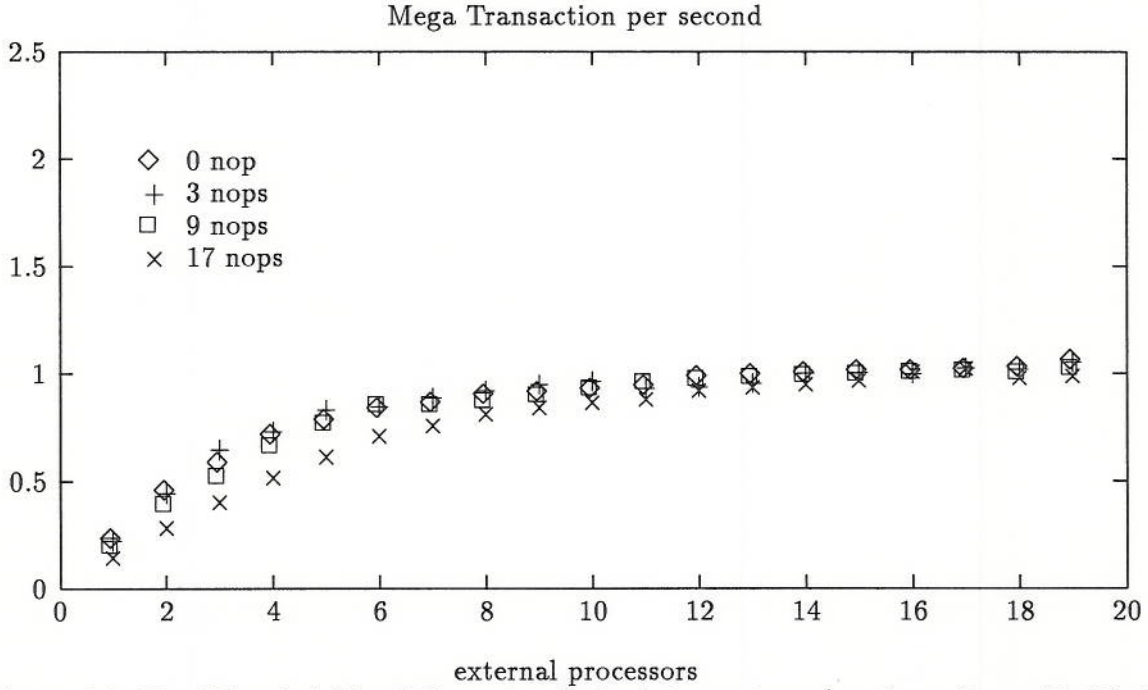

Figure 16: Total bandwidth of the network for integer store (configuration of table 2)

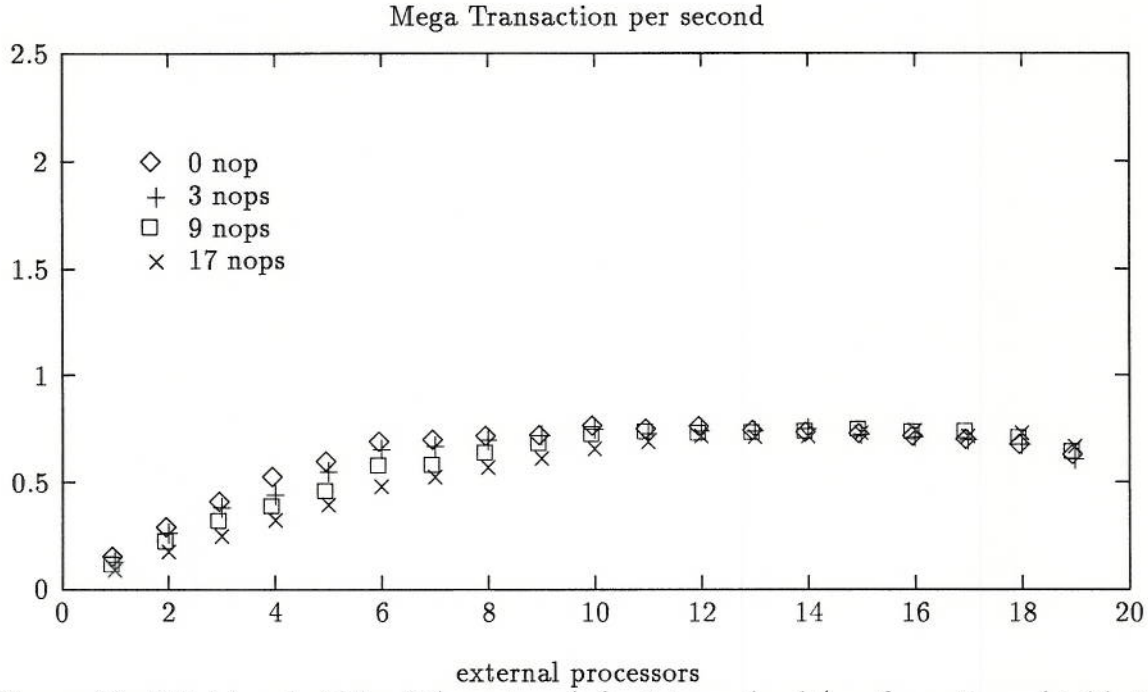

Figure 17: Total bandwidth of the network for integer load (configuration of table 2)