Proceedings of Performance 93 , Rome, Italy, September 27 to October 1, 1993, pp. 183-204.

Performance Benefits and Limitations of Large NUMA Multiprocessors

Kenneth C. Sevcik and Songnian Zhou

Technical Report CSRI-256

October 1991
revised: November 1992

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Performance Benefits and Limitations of Large NUMA Multiprocessors *

Kenneth C. Sevcik and Songnian Zhou

Computer Systems Research Institute
University of Toronto

ABSTRACT

In scalable multiprocessor architectures, the times required for a processor to access various portions of memory are different. In this paper, we consider how this characteristic affects performance by comparing it to the ideal but unrealizable case in which the access times to all memory modules can be kept constant, even as the number of processors is increased. We examine several application kernels to investigate how well they would execute on various instances of NUMA systems with a hierarchical memory structure.

The results of our analytic model show that access locality is much more important in NUMA architectures than it is in UMA architectures. The extent of the performance penalty of non-local memory accesses depends on the variability in access times to various parts of shared memory, as well as on the amount of congestion in the interconnection network that provides access to remote memory modules. In the applications we examined, we found that it is possible to partition and locate both the data and the computation in such a way that reasonable speedups can be achieved on NUMA systems. To do this requires some additional work on the part of the programmer or perhaps the supporting software (i.e., the compiler and the run-time system). For applications that fail to achieve good speedup on a large number of processors, our modelling approach makes it possible to predict how well the application would perform on subsystems with fewer processors.

1. Introduction

Shared memory multiprocessors consisting of up to about 30 processors have been proven capable of providing very good speedups for a wide variety of applications. These multiprocessors typically have structures that lead to a uniform memory access time (UMA) for each processor to each memory module. As the number of processors in a shared memory multiprocessor is increased, memory modules are generally placed in such a way that each module is ‘‘close’’ to some processor(s) but necessarily ‘‘distant’’ from others, so that Non-Uniform Memory Access times (NUMA) are a result.

In a NUMA shared memory multiprocessor, it is more difficult than in UMA multiprocessors to obtain nearly linear speedups because execution on a single processor is able to use only ‘‘close’’ memory, for which access times are short, whereas execution on a multiprocessor generally involves each processor making accesses to a mixture of ‘‘close’’ and ‘‘distant’’ memories, which causes the average memory access time to be higher.

In this paper, we address the question of what memory access characteristics an application must possess in order to run well on a NUMA multiprocessor. Alternatively, turning the question around, given a particular application and algorithmic approach with known memory access characteristics, how many processors can be used while keeping memory access times sufficiently low so that performance is

* This research is supported by the Natural Sciences and Engineering Research Council of Canada.

acceptable? We present a high level model of a type of NUMA multiprocessor architecture with a hierarchical interconnection structure, and also a model of the memory access behavior of an application. With these two models, we study the interaction of applications and architectures, assessing the degree of memory locality needed in an application to attain acceptable parallel execution performance.

In Section 2 of the paper, we present first the architecture model then the application model. We provide an analysis of the models to determine the extent of performance degradation due to the NUMA memory structure in comparison to an ideal UMA system with uniform memory access times. Such an ideal UMA system is unrealizable in practice for large numbers of processors unless the access times to all memory modules are made quite large. Initially, we perform the analysis without specifying the particular types of interconnection components that attach the processors and memory modules. This permits us to develop equations sufficiently general to apply to a range of possible NUMA architectures. Then we specialize to a particular form of hierarchical interconnection network, and carry out a more detailed analysis that includes the contention encountered on communication paths and at memory modules. Our analytic model provides a method for estimating the performance of a specific application on a particular number of processors in a NUMA system if the basic memory access characteristics of the application are known.

In Section 3, we examine three fundamental application kernels to estimate what degree of memory access intensity and locality they exhibit. We demonstrate that modest variations to the memory access strategies employed in each algorithm can lead to substantial changes to their locality using the analysis of Section 2. We are then able to estimate the performance of each application kernel on various size subsets of processors in a NUMA system. The three applications we treat were chosen in part for their relatively intensive use of memory. It is likely that similar techniques can be applied to a wide range of applications to make them attain acceptable performance on NUMA systems.

In Section 4, we present our conclusions that the modelling approach can be useful both in studying the general interaction of applications and NUMA architectures, and in estimating the performance attainable by a specific application running on a NUMA system.

2. Models of NUMA Architectures, Applications, and Their Performance

We develop here models of a class of multiprocessor architectures and of applications that might run on them. We represent in our system model symmetric hierarchical NUMA multiprocessors where, from any particular processor's point of view, the shared memory of the system is partitioned into levels according to the access time from that processor. The system is symmetric in that the hierarchy of memory distances is the same from the point of view of each processor. Many shared memory NUMA multiprocessors can be represented by this model.

We expect that large multiprocessors (with about 30 processors or more) will be used predominantly in a multiprogrammed mode in the future (by partitioning the system's resources among concurrently executing applications). However, in this paper, we will examine only a single application at a time, estimating how it would run either on a full system or on a convenient subset of the processors of the system. Space sharing (partitioning the processors among active applications) has been demonstrated to be a good approach to processor allocation when serving multiple applications on a multiprocessor [11,12].

Our attention will be focused primarily on memory access delays due to the NUMA memory characteristics of the system being considered. As a baseline, we take a hypothetical (but unrealizable) environment where every memory access that misses cache is satisfied in local memory (i.e., a UMA structure). Every cache miss delays completion of the memory access operation. The length of the delay depends on the memory access time from the processor to the memory module being accessed. The average delay will be higher with the NUMA structure than with the baseline UMA one. (We ignore the possibility of prefetch that would overlap memory access time with useful computation, although sufficiently clever compilers are capable of invoking that strategy.)

We assume that each processor has a separate single level cache for instructions and data. Data accesses satisfied in cache take a single cycle, and accesses satisfied in shared memory require ten to one hundred cycles (or possibly more in the case of virtual shared memory supported on a distributed architecture). We will ignore instruction accesses since instruction cache hit rates are typically close to 100% for many applications, and the instruction fetch time from cache is usually overlapped through pipelining with other computational steps. Architectural differences among the types of processors used to construct multiprocessors determine the extent to which the detailed assumptions above are valid in a particular case. However, the qualitative results and conclusions we reach are not sensitive to the particular set of assumptions we make.

Our proposed modelling approach is applicable to a variety of existing NUMA multiprocessor systems, both commercial (Myrias SP-1, BBN Butterfly, Thinking Machines CM-5, and Kendall Square KSR-1) and experimental (Illinois' Cedar, Stanford's Dash, MIT's Alewife, and Toronto's Hector). There have been numerous experimental studies on such systems, but relatively few modelling studies. Markatos and LeBlanc have studied the importance of memory management policies in the BBN Butterfly [15]. Johnson investigated the role of communication locality in mesh-connected multiprocessors (such as Alewife) [14]. He obtained upper bounds on the performance in light of the communication requirements of applications. Holliday and Stumm have used detailed simulation to assess the performance of applications on large versions of the Hector structure [13]. Zhou and Brecht have studied processor allocation policies that are appropriate in NUMA systems [16].

2.1. The System Model

The general NUMA architecture that we examine is shown in Figure 1. We assume that memory modules are paired one to one with processors, but that the memory modules together in aggregate form the shared memory for all the processors of the system. At level 1 of the structure are all the processor-memory module pairs. These are grouped into sets such that memory access times to memory modules within the set are lower than those to memory modules outside the set. The clouds shown at level 2 of the Figure represent the interconnection components that join the processors within a group. An interconnection component could be a bus, a ring, or a crossbar, among other possibilities. At levels 3 and higher, there are interconnection components that link the subgroups below.

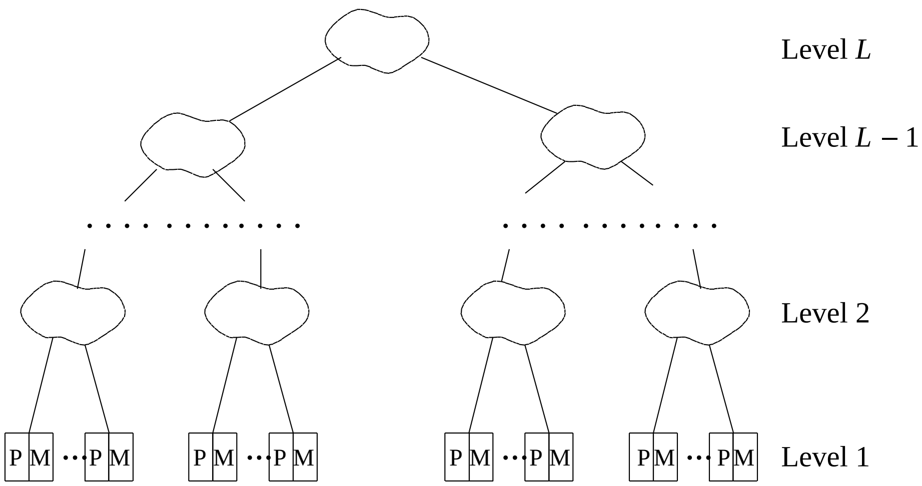

Figure 1. NUMA Architecture Model.

We let $N_1, N_2, \cdots, N_L$ represent the numbers of processors contained in successively higher level subgroups. (By definition, $N_1 = 1$, and, by convention, $N_0 = 0$.) Thus, N_L is the total number of processor-memory pairs, and N_j/N_{j-1} is the number of subgroups within a level j group (i.e., the branching factor at level j of the hierarchy), for $j = 2, \cdots, L$. From a particular processor's point of view, there are exactly $N_j - N_{j-1}$ memory modules that can be reached by a request that must go up to a level j interconnection component of the hierarchy but not higher. We shall refer to such requests as *level j requests*. We use t_j to denote the time required for a level j request to access its target memory module. So, t_1 is the time to access the processor's local memory, and t_j is the access time to each of the $N_j - N_{j-1}$ memory modules reachable by level j requests from the processor under consideration.

To reduce the number of parameters to two for some aspects of our study, we will consider a special case in which the access times to the various levels of memory form an arithmetic progression:

$t_0 = 1$ {one cycle from cache, overlapped in pipelining}

$t_j = C + (j - 1)\, C_\Delta, \quad j=1, \cdots, L$ {cycles for level j requests}

where $C = t_1$ is the access time to the processor's local memory, and C_Δ is the change in access time between level $j-1$ and level j requests. In actual systems, the time differences are unlikely to be identical at successive levels, but the approximation is useful in order to use a single parameter to represent the "NUMAness" factor of a system.

2.2. The Application Model

As with the system, we attempt to characterize an application by a small number of parameters. These parameters indicate the frequency of memory reference instructions and the locality of the references. The parameters are:

- a the ratio of total number of instructions to memory access instructions (loads or stores);
- c the average number of cycles per instruction;
- h the data cache hit ratio, which we assume to be the same for private data and shared data;
- p the fraction of memory references to private data (in the local memory);
- f the ratio of the number of level k accesses per module that requires level k requests to the number of level $k-1$ accesses per module that requires level $k-1$ requests;
- F_k the fraction of memory accesses among those that miss the cache that require level k requests.

Note that we will use either f or $\{F_1, F_2, ..., F_L\}$ to specify the distribution of memory access requests across levels. Using f, the distribution is characterized by a single parameter, as will be done in the remainder of section 2. With locality of reference to data and a suitable memory management algorithm, we expect f to be between 0 and 1, and generally much closer to 0. A more precise characterization of the distribution can be expressed with the set of parameters, $\{F_1, F_2, ..., F_L\}$. In section 3, we will use detailed knowledge of specific applications to identify their F_k values.

While we consider the parameters above to be primarily descriptors of the application, the values do depend on system architecture characteristics as well. For example, c depends on the extent of instruction pipelining achieved, and h depends on the cache size.

2.3. Analysis of the Model

With the architecture and application parameters indicated above, it is possible to create a simple, high-level model to estimate the performance of an application on the NUMA architecture. In this section, we develop formulae for execution times and resource utilizations in the presence of memory access delays. These formulae will be extended in Section 2.4 to consider contention at the memory modules and in the interconnection network.

First, we specify the distribution of shared memory accesses (excluding cache hits) across memory modules at various distances from a particular processor. In order to focus on the overall effect of "NUMAness", we assume here that the single parameter, f, indicates the ratio of the frequency of accesses to each module requiring level k requests to the frequency of accesses to each module requiring level $k-1$ requests. If F_k is the fraction of all accesses to shared data that require level k requests, then

$$F_k = \frac{(N_k - N_{k-1})\, f^{k-1}}{\sum_{j=1}^{L} (N_j - N_{j-1})\, f^{j-1}}, \qquad k=1, \cdots, L\ ,$$

since $N_j - N_{j-1}$ is the number of memory modules requiring level j requests from the processor of interest.

With the access delay to level k defined as $t_k = C + (k-1)\, C_\Delta$, we can calculate the average memory access delay over all accesses to shared data (again excluding cache hits):

$$C_{avg} = \sum_{k=1}^{L} F_k\, t_k = \frac{\sum_{j=1}^{L} (N_j - N_{j-1})\, f^{j-1}\, t_j}{\sum_{j=1}^{L} (N_j - N_{j-1})\, f^{j-1}}\ . \qquad (1)$$

Let I be the number of instructions of the application executed on one processor. Then its execution time on the NUMA system including the time to access memory is:

$$E_{NUMA} = I \times \left\{ c + \frac{(1-h)}{a} \left[p\ C + (1-p)\, C_{avg} \right] \right\} .$$

We will compare this execution time with that which would be achieved in an unrealizable *UMA* environment with all memory accesses that are not satisfied in cache requiring C cycles:

$$E_{UMA} = I \times \left\{ c + \frac{(1-h)}{a}\ C \right\} .$$

Note that both the instruction fetch from cache and the data access time (if any) for a cache hit are included in the parameter c.

We will refer to the ratio E_{NUMA}/E_{UMA} as the *NUMA expansion ratio*, which gives an indication of the performance penalty paid because some memory accesses require more than C cycles. Based on these formulae, we can estimate the sensitivity of execution time to the various parameters in the model.

For purposes of the experiments described here, we use the parameter values listed in Table 1. We assume a four-level hierarchy with the branching factors at levels 2, 3, and 4 of the hierarchy being 6, 5, and 4, respectively. So, $N_4 = 120$, $N_3 = 30$, $N_2 = 6$, and $N_1 = 1$. The basic memory access time, C, is taken to be 13 cycles. We assume that one in every five instructions is a read or write (i.e., $a = 5$), and that the cache hit rate for data references is 50%. (The cache hit rate for instruction accesses is assumed to be 100%.) Finally, we assume that all data is shared (i.e., $p = 0$). These parameter values are intended to represent memory intensive applications.

$N_4 = 120$,	$N_3 = 30$,	$N_2 = 6$,	$N_1 = 1$	
$C = 13$,	$c = 1$,	$a = 5$,	$h = 50\%$,	$p = 0$

Table 1. Values for model parameters (ignoring contention).

Figure 2 shows how execution times depend on the values of C_Δ and f. The greater the differential between access costs of requests at successive levels (as represented by higher C_Δ values), the higher the NUMA expansion ratio. Similarly, the higher the (non-)locality of an application's memory reference patterns, as represented by larger f, the higher the NUMA expansion ratio.

2.4. Contention in Memory Access

In NUMA systems, the time required to complete a read or write includes delays incurred due to contention along the communication paths and at the memory module itself. The amount of contention depends on the specific structure of the interconnection network between the processors and the memory modules. In this section, we will postulate a particular interconnection structure and extend the basic analytic model of Section 2.3 to reflect the resulting delays due to contention. The interconnection network we consider is an L level NUMA structure such as that shown in Figure 1. The hierarchical nature of this structure makes it possible to expand the size of the system by taking several replications of a particular size system and joining them with some form of interconnection component. In this way, the number of processors used by an application can be increased substantially, while ideally the NUMA expansion ratio

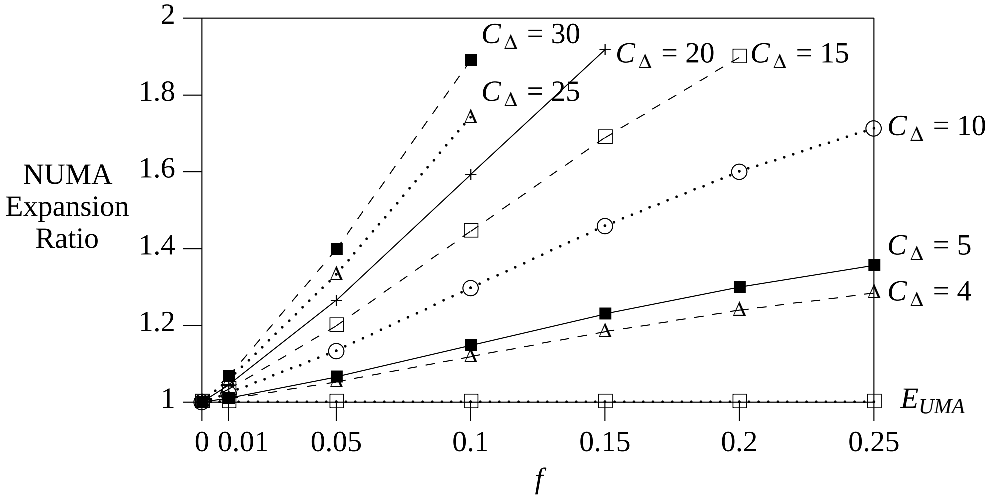

Figure 2. Performance sensitivity to C_Δ and f with no contention.
(Other parameters have the values indicated in Table 1.)

increases only a small amount.

The amount of contention for each memory module depends on the number of cycles the memory module is busy serving a request. We denote this quantity by C_{mem}, and set its value to 10 cycles in our experiments.

In order to determine the extent of contention in the interconnection network, we need to know the effective load placed on each interconnection component when a memory request and the corresponding response pass through it. Let b_j, $j=2, \cdots, L$ represent this total service time (assuming that all interconnection components at the same level are of the same type). The value of b_j can be determined as the number of cycles spent in the interconnection component divided by the number of service units there. For example, a *bus* has a single unit that is occupied one cycle for the request and one for the response, so that $b_{bus} = 2$. For a slotted ring with k slots interconnecting k subgroups, the request and response together occupy each of the k slots for one cycle, so that $b_{ring} = k/k = 1$. Also, for a $k \times k$ crossbar, each request and each response occupy k of the k^2 switching points for one cycle, so $b_{crossbar} = 2k/k^2 = 2/k$.

Delays due to contention depend on the utilizations of the memory modules and the components along the interconnection network paths. These utilizations are the product of the service times and the throughput of memory requests at each processor in the system. This throughput is the inverse of the average time including memory access delays between successive memory accesses that miss the cache, which we denote by T. The value of T can be computed in terms of T_p, the average time to complete an access to private data, and T_s, the average time to complete an access to shared data:

$$T = \frac{a \times c}{1-h} + p\, T_p + (1-p)\, T_s \tag{2}$$

where

$$T_p = C + D_{mem}^{local}, \tag{3}$$

and

$$T_s = C_{avg} + D_{net} + F_1 D_{mem}^{local} + (1-F_1)\, D_{mem}^{remote}, \tag{4}$$

The quantities D_{mem}^{local} and D_{mem}^{remote} are the average memory contention delays incurred by local and remote memory accesses, respectively, and D_{net} is the average network contention delay incurred by (all) accesses to shared data while possibly passing through the network. Expressions for these delay terms are derived below.

Since memory accesses are generated at rate $1/T$ by each processor, and, due to symmetry, reach each memory module at the same rate, the utilization of each memory module is given by:

$$U_{mem} = \frac{C_{mem}}{T} , \tag{5}$$

where C_{mem} is the number of cycles that a memory module is busy serving a single request. To estimate the queueing time at the memory module, we observe that the expected queue length at a memory module can be approximated by $\dfrac{U_{mem} - U_{mem}^{own}}{1 - U_{mem}}$, where U_{mem}^{own} is the fraction of memory utilization generated by the processor being considered [1, see sect. 10.2]. This expression accounts for the fact that a request will never have to queue behind another request from the same processor. Since fraction $(1 - p)$ of all accesses are to shared data and fraction $(1 - F_1)$ of those are not from the same processor,

$$U_{mem} - U_{mem}^{own} \; = \; U_{mem} \, (1 - p) \, (1 - F_1)$$

Thus, for local requests, the expected delay is given by the product of the estimated queue length and the service time per request:

$$D_{mem}^{local} \; = \; C_{mem} \, \frac{U_{mem}}{1 - U_{mem}} (1 - p) \, (1 - F_1) \, . \tag{6}$$

For remote requests, we must consider requests at various levels, and take into account the amount of usage of the specific memory module by each processor. For a remote level j request,

$$U_{mem} \; - \; U_{mem}^{own} \; = \; U_{mem} \left[p + (1 - p) \left(1 \; - \; \frac{F_j}{N_j - N_{j-1}} \right) \right]$$

Averaging over various j values from 2 to L and weighting by the proportion of level j requests,

$$\begin{aligned} D_{mem}^{remote} \; &= \; \frac{C_{mem}}{(1 - F_1)} \; \sum_{j=2}^{L} F_j \left\{ \frac{U_{mem}}{1 - U_{mem}} \left[p + (1 - p) \left(1 \; - \; \frac{F_j}{N_j - N_{j-1}} \right) \right] \right\} \\ &= \; C_{mem} \, \frac{U_{mem}}{1 - U_{mem}} \left[1 - \frac{1 - p}{1 - F_1} \sum_{j=2}^{L} \left(\frac{F_j^2}{N_j - N_{j-1}} \right) \right] \end{aligned} \tag{7}$$

Similarly, we can determine the utilization of and contention delay at each interconnection component of the system. Consider first the level L (i.e., highest level) interconnection component. Requests that pass through it come from N_L processors, each generating level L requests at rate $(1-p)F_L/T$ with each request placing load b_L on the interconnection component. This leads to utilization:

$$U_L = (1-p) \, \frac{N_L \, F_L \, b_L}{T} \; .$$

For each interconnection component at level $L-1$, N_{L-1} processors each generate level $L-1$ requests at rate $(1-p) \, F_{L-1}/T$, and also level L requests at rate $(1-p) \, F_L/T$. In addition, the number of level L requests going out through a level $L - 1$ interconnection component is matched by an equal number of level L requests coming in to memory modules under it. Hence, the total utilization of the level $L - 1$ interconnection component is:

$$U_{L-1} = \frac{1-p}{T} \left[N_{L-1} \, b_{L-1} \, (F_{L-1} + 2 \, F_L) \right] \; .$$

In general, for a level j interconnection component, the total utilization is:

$$U_j = \frac{1-p}{T} \left[N_j \, b_j \, (F_j + 2 \sum_{i=j+1}^{L} F_i) \right] \; . \tag{8}$$

Once the utilizations are known, the contention delay in cycles for each passage through a level j interconnection component can be approximated as:

$$D_j = \frac{U_j}{1 - U_j} \; , \quad j = 2, \cdots, L \; . \tag{9}$$

Taking into account that level j requests and also the response to each request enter one level j interconnection component and two interconnection components at each level from 2 to $j-1$ (see Figure 1), we can express the average network contention delay for all accesses to shared data that miss the cache:

$$D_{net} = F_2\ (2D_2) + F_3\ (2D_3 + 4D_2) + \cdots + F_L\ (\ 2D_L + 4D_{L-1} + \ \cdots\ + 4D_2)$$
$$= 2 \sum_{j=2}^{L} \left(F_j + 2 \sum_{i=j+1}^{L} F_i \right) D_j\ . \qquad (10)$$

We have now established formulae for all items used in calculating T.

In the formulae above, the delays incurred due to contention for memory modules and in interconnection components are expressed in terms of the utilizations. But the utilizations depend on T, the average time between memory accesses by a processor, and T depends in turn on the delays. Contention causes T to be larger, but a larger T leads to reduced utilizations and hence a reduced amount of contention.

To estimate the actual value of T with congestion taken into account, an iterative scheme is required:

0. Calculate C_{avg} from eq. (1) in section 2.3 and initialize utilizations U_{mem} and U_j, j=2, $\cdots$,L to zero.
1. Calculate the delays D_{mem}^{local}, D_{mem}^{remote}, D_j, j=2, $\cdots$,L, and D_{net} from the utilizations using eqs. (6), (7), (9), and (10), respectively.
2. Calculate T_p, T_s, and T from the delays using eqs. (3), (4), and (2), respectively.
3. Calculate the utilizations U_{mem} and U_j, j=2, $\cdots$,L based on the new value of T using eqs. (5) and (8), respectively.
4. Repeat steps 1 through 3 until the utilizations and T stop changing.

Empirically, we have found that the iteration converges quickly as long as no resource is saturated. When the iteration has converged, the execution time can be calculated from the final values of T_p and T_s:

$$E_{NUMA} = I \times \left\{ c + \frac{(1-h)}{a} \left[p \times T_p + (1-p)\ T_s \right] \right\} . \qquad (11)$$

In a shared memory multiprocessor, there are three possible sources of increased memory access latency (and therefore longer execution time):

1) access to distant memory modules,
2) contention at the memory modules among requests from multiple processors, and
3) contention in the interconnection network.

In a NUMA architecture, all three are present. To isolate the performance impact of each of the factors, however, we define three hypothetical UMA architectures:

UMA-1:
All memory accesses incur the local cost, C, and there is no contention at memory or in the network.

UMA-2:
All memory accesses incur the local cost, but contention at the memory modules is considered based on the access locality (indicated by f) of the application.

UMA-3:
All memory accesses incur cost for the most distant memory modules, $C + (L - 1)\ C_\Delta$, and the contention at the memory modules is considered, but the network contention delay is ignored.

Effectively, in UMA-1 (which we called UMA in section 2.4), we assume an infinite capacity zero delay network, which is clearly an unrealizable case. The UMA-2 case, though still unrealizable is less unrealistic than UMA-1. The UMA-3 case may be regarded as nearly realizable because an expensive interconnection network for a large number of processors and memory modules might incur substantial delays but low contention.

$N_4 = 120$, $N_3 = 30$, $N_2 = 6$, $N_1 = 1$
$C = 13$, $C_{mem} = 10$, $C_\Delta = 6$
$t_1 = 13$, $t_2 = 19$, $t_3 = 25$, $t_4 = 31$.
$b_2 = 2$ (bus), $b_3 = b_4 = 1$ (rings)
$a = 5$, $h = 50\%$, $p = 0$

Table 2. Values for model parameters (including contention).

Using the parameter values shown in Table 2, a comparison of the (realizable) NUMA case to the three (unrealizable) UMA cases is shown in Figure 3. We assume that the interconnection components are a bus at level 2 and rings at levels 3 and 4 (referred to as the *local ring* and *global ring*, respectively). All execution times are normalized to that of UMA-1 (to yield the values of the NUMA expansion ratio). The UMA-2 and UMA-3 curves increase with f because memory contention is included although network contention is not. It is interesting to observe that the NUMA cases achieve execution times close to those of UMA-2 when the f value is small, i.e., when the application exhibits good memory reference locality. If f is large and all data is shared, however, NUMA may be as bad as or even worse than UMA-3 (which has uniformly long access delays) due to contention in the interconnection network. However, unlike UMA-3, which results in poor performance for all applications, NUMA encourages application writers to enhance their applications' locality. Data reference locality is an important determinant of an application's performance in a NUMA system.

By assuming that all references are to shared data (i.e., $p = 0$), we are being pessimistic about the performance potential of NUMA systems. The performance with $p = 0.5$ (i.e., half of the data accesses are to private data) is shown in Figure 4. In this case, the performance of NUMA is close to E_{UMA-2} even for large f values.

To get a more direct feeling of the degree of locality necessary for good performance, in Figure 5, we show the fractions of the shared data access traffic to groups of memory modules at various distances. F_1 decreases as f increases; F_2 and F_3 rise then flatten out, and finally decrease for sufficiently large f; and F_4 takes off as f increases. At $f = 0.15$, F_1 is less than 40%, i.e., less than 40% of shared data references go to the local memory, yet NUMA still yields substantial improvement over UMA-3, and has a moderate NUMA expansion ratio of about 1.45, even with $p = 0$.

In Figure 6, we examine the performance sensitivity of NUMA systems to the parameters f and C_Δ, as we did in Figure 2, but this time with the effects of contention for memory and communication paths taken into account. The execution times are now normalized against the more realistic E_{UMA-2}, rather than E_{UMA-1}. Such normalization allows us to focus on the NUMA architecture, as memory access contention is independent of the NUMA structure and is reflected in UMA-2. For all but very large f and C_Δ values, the execution time expansion ratio stays below a factor of two. This result is encouraging because, from the characterization of several memory intensive applications (to be discussed in section 3), we expect that a large class of applications would perform reasonably well on large NUMA systems.

There exist interactions between utilizations of the memory units and various interconnection components, as is illustrated in Figure 7. With good locality, most data references (that miss the cache) go to local memory, leading to low average reference delay, and hence high memory utilization and low network utilization. As locality decreases, however, the utilizations of the network components increase, slowing down memory references, so, memory utilization decreases. The higher the level of a network component, the faster its utilization increases with f. For the configuration assumed in this study, the global ring appears to be the most likely system bottleneck (although burstiness of memory accesses could make other resources more critical for periods of time).

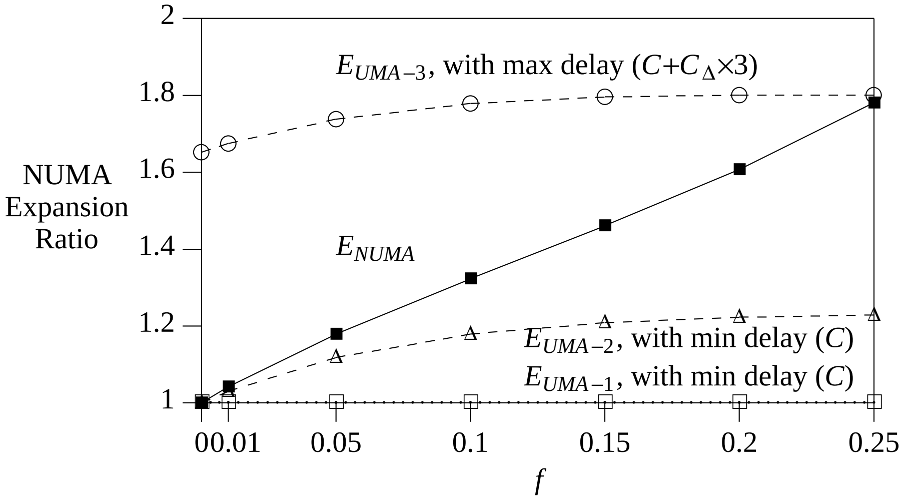

Figure 3. Comparison of NUMA to UMA with minimum and maximum delays with $p = 0$ normalized against E_{UMA-1}. (Other parameter values are given in Table 2.)

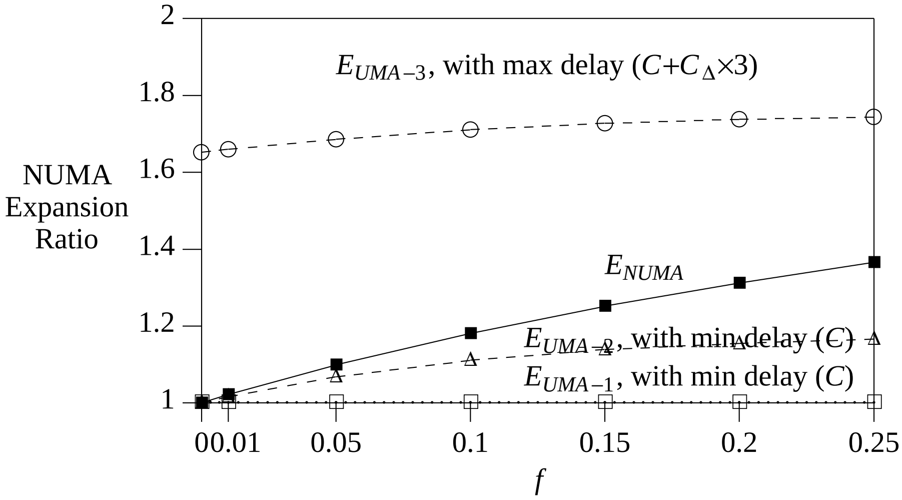

Figure 4. Comparison of NUMA to UMA with minimum and maximum delays with $p = 0.5$ normalized against E_{UMA-1}. (Other parameter values are given in Table 2.)

3. Applications Performance: Case Studies

In this section, we use the models developed in section 2 to assess the performance of three parallel applications in a representative hierarchical NUMA system, Hector. We first give a brief description of Hector with its various memory access costs. We then describe the three applications to be studied, and present their memory access characteristics, as observed with an instruction-level simulator of the Hector processor-cache subsystem. Finally, we look at the memory sharing patterns of the applications in detail, and derive performance predictions for these applications running on a Hector type of machine.

3.1. Hector

Hector is a hierarchical NUMA multiprocessor being developed at the University of Toronto [2]. The Hector system consists of four levels. At the lowest level are the processor-memory (PM) pairs built using MC88100 processor and MC88200 cache chips. Several PM pairs are connected to a parallel bus to form a Hector *station*. Several stations are then connected to a parallel slotted ring network to form a

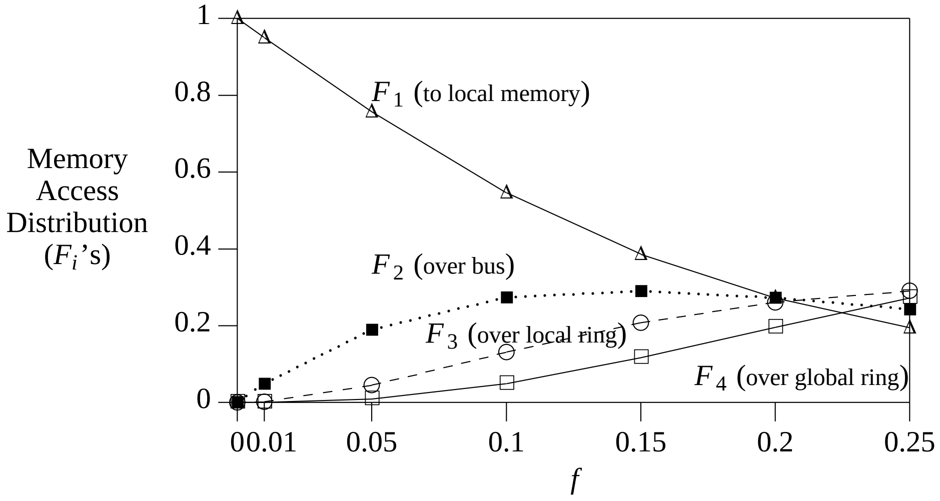

Figure 5. Memory access traffic pattern. (Other parameter values are given in Table 2.)

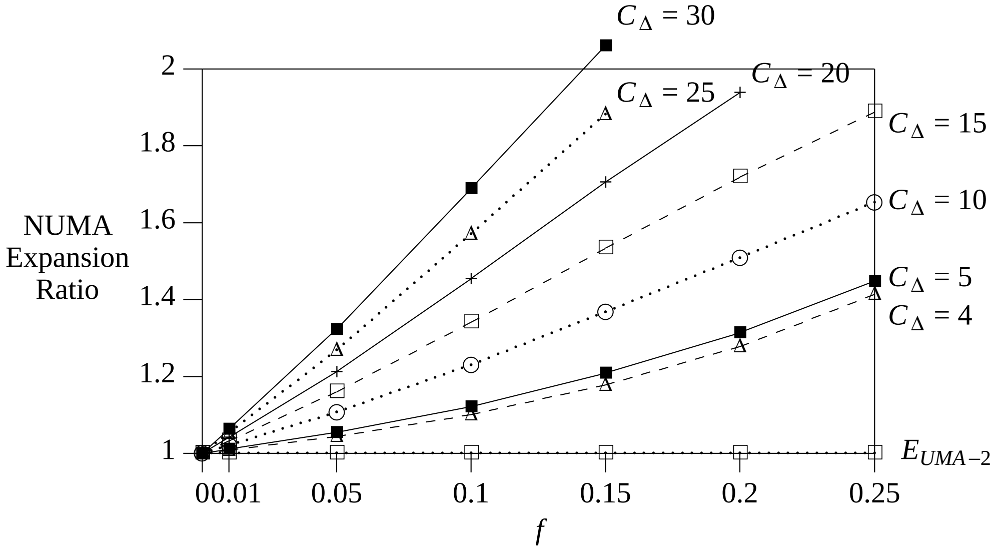

Figure 6. Performance sensitivity to C_Δ and f normalized against E_{UMA-2}.
(Other parameter values are given in Table 2.)

local ring. Finally, several local rings are connected to a similar parallel ring to form the *global ring*. The maximum configuration contains 256 processors formed with 8 PMs per station, 8 stations per local ring, and 4 local rings on the global ring. The read access latencies from a processor to memory units at various distances assuming no contention at the memory modules or in the interconnection network are shown in Table 3 [3]. If the number of components connected to local or global rings change, then memory access delays change, since each ring segment adds one cycle to the round-trip delay. As may be observed in Table 3, the access latencies in Hector form an approximate arithmetic progression, and the ''NUMAness'' of Hector is relatively mild compared to other hierarchical NUMA systems, such as DASH [4] and Butterfly [5]. In this section, we assume a configuration of six processors per station, five stations per local ring, and four local rings per global ring, for a total of one hundred twenty processors. This is the same structure as was assumed in section 2.

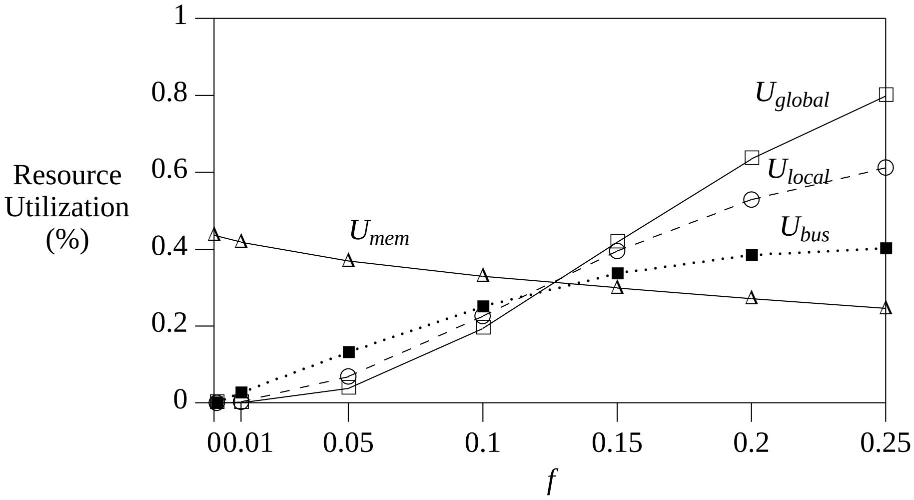

Figure 7. Utilizations of memory, bus, local ring, and global ring with different locality values. (Other parameter values are given in Table 2.)

Memory location	**Hector Read latency**	**Values used in the model**
local memory (t_1)	11	13
memory over station bus (t_2)	15	19
memory over local ring (t_3)	20	25
memory over global ring (t_4)	28	31
memory service time (C_{mem})	8	10

Table 3. Hector memory access latencies and the corresponding model parameter values (in cycles, assuming no contention).

3.2. Applications and Their Characterization via Simulation

To study applications performance in hierarchical NUMA systems, we consider the parallelizable cores of three common parallel applications. They are dense matrix multiplication (MM), successive over-relaxation on a two-dimensional grid (SOR) [6], and LU decomposition. We choose these three as representatives of a class of algorithms that are

(1) easy to understand,

(2) commonly used in scientific computing, and

(3) relatively memory intensive (in that they involve large amounts of data, and there is a high proportion of memory accesses).

Also, the nature of these algorithms is to sweep across large matrices of data, with little locality of access in data references. Darema-Rogers, Phister, and So studied the memory access patterns of several applications and observed that, for scientific applications, most of the data is shared, but the frequency of references per item is low [7].

The characteristics of these applications are discussed below:

MM: The shared data structures are two matrices, *A* and *B*, to be multiplied to produce a third matrix, *C*. For parallel computation, the *C* matrix is divided into *P* strips of rows, which are assigned to *P* threads running on *P* processors.

SOR: The Successive Over-Relaxation method for solving partial differential equations is an iterative method used to compute each element of a large 2-D matrix as a function (the average) of its four neighbors. The grid is divided into *P* strips of columns, and one strip is assigned to each processor.

LU: LU decomposition is the first (and most compute intensive) step in solving a system of linear equations using a non-iterative method. The columns of the coefficient matrix are assigned to the *P* processors in a round-robin fashion. The *P* processors take turns computing pivot columns, then the rest of the columns are updated by the *P* processors in parallel.

We used a simulator of the Hector processor and cache to extract the values of some of the parameters used in the application model developed in Section 2.2. Instead of simulating the execution of the entire applications, we chose to simulate only the application cores of the algorithms, which account for almost all of the required computation. The application cores were written in C, and were compiled with the GNU C compiler for MC88000 and with the optimization option turned on. The statistics on data cache hit ratio and frequencies of memory references are shown in Table 4*. In general, the memory reference characteristics of the applications stabilize beyond a certain problem size when the data sets no longer fit in the cache.

Application	Cache hit ratio	Problem size	No. of procs	Distn. of instructions between memory accesses								
				mean	1	2	3	4	6	9	11	18
MM	49.9%	512x512	64	5.00	49.9				0.1	49.8	0.1	
SOR	64.5%	512x512	64	5.84	16.7	33.3	16.7			16.7		16.7
LU	59.5%	512x512	64	2.73	0.1	66.6		33.2			0.1	

Table 4. Some characteristics of the three sample applications.

3.2.1. Cache behavior

For MM on large matrices running in a system with modest cache (e.g., each MC88200 chip contains 16 KB cache), the cache hits mainly result from the repeated use of a row in *A* in calculating a row in *C*. Accesses to *B* are sequential by column and result in few cache hits, since, by the time a certain *B* element is reaccessed, it has already been replaced in the cache. In Hector, the cache is also helpful in loading the entire cache line of 16 bytes upon a cache miss, which is somewhat more efficient than loading the 4-byte words in the line individually. To avoid assuming optimizations in implementation so that our results are more general, we do not consider whole cache line loading (hence the near 50% hit ratio for MM).

The cache performance for SOR and LU is somewhat better, because some of the matrix elements are reused in close succession. For instance, in SOR, updating a matrix element involves accessing its four neighbors, some of which may have just been accessed while updating some preceding elements. For LU, the pivot row is used in updating all the other rows for which a processor is responsible.

3.2.2. Memory access frequency

Given the data-parallel nature of the algorithms, it is not surprising to observe that the number of instructions between successive memory references is low. For LU, it is below three (i.e., on average, more than one in every three instructions is a load or store). Such applications are definitely strongly affected by cache and memory system performance. For RISC processors such as MC88100, most instructions execute in a single machine cycle, and extensive pipelining may yield an effective execution time that is close to one cycle for most of the instructions, depending on the instruction sequence. Therefore, the average number of machine cycles between memory references may be not much higher than the mean number of instructions shown in Table 4.

---

* Our simulator ignores cache conflicts. This is reasonable if there is hardware cache consistency mechanism, and the amount of read-write sharing of cache lines is low (which is the case with MM, SOR, and LU).

3.2.3. Memory access locality

Our simulator does not consider any particular memory hierarchy and memory management policies. However, an analysis to be performed in Section 3.3 can help us in identifying the approximate F_i values for these three applications. For the A and C matrices in MM, the strips can all be localized because each of them is accessed by one processor. The between-neighbor sharing at the edges of the strips is minimal. The B matrix is read-only, and can be distributed and replicated (using methods such as those to be described in Section 3.3). Similarly, most of the data elements in the matrix in SOR are accessed by a single processor, with between-neighbor sharing only at the edges of the strips. For LU, the rows are statically assigned to the processors. Sharing occurs with the pivot rows. Since they are read by every processor after they are computed, they can be replicated in the local memory of each processor. We therefore expect the f value to be low for all three applications, particularly if replication is done where appropriate.

For applications less dependent on large data sets, we expect that both of their cache hit rates and average numbers of cycles between memory references to be higher than those for MM, SOR, and LU. Hence, if these three applications can be made to run effectively on a NUMA system, then we expect that a broad class of other applications can be made to do so, as well.

3.2.4. Parameterization of the NUMA Model

The parameter values used in section 2 were motivated by the characteristics of Hector and the simulation analysis of the three chosen applications, although the model itself is more widely applicable. A four level hierarchy of 6 processors per station, 5 busses per local ring, and 4 local ring on the global ring (system) reflects a typical Hector configuration. The memory access costs, C, C_{Δ}, and C_{mem} closely match those in Hector (see Table 3). The cache hit ratio for data accesses, h, is close to those of MM, SOR, and LU.

3.3. Performance of the Three Sample Applications

3.3.1. Matrix Multiplication

Consider the calculation of a product matrix, C, of dimension $n \times m$ from matrices A $(n \times z)$ and B $(z \times m)$. The approximate number of arithmetic operations called for by the conventional algorithm is about $2nmz$ (half multiplications and half additions). The numbers of memory access operations are at least $(n + m)z$ reads plus nm writes.

Here we will assume that the calculation is divided among P processors by having each processor do the calculations necessary to form n/P rows of C. (We assume that n is enough larger than P that we may ignore the fact that the quotient is not necessarily an integer.) If processor i calculates rows $\frac{(i-1)n}{P} + 1$ to $\frac{in}{P}$ of C, then it must read the corresponding rows of A and all of B (a column at a time). We will assume that matrices A and C are placed in memory so that rows $\frac{(i-1)n}{P} + 1$ to $\frac{in}{P}$ of each are located in processor i's local memory. We assume that B is evenly distributed (either by rows or by columns) among the local memories of the processors.

As a consequence of this placement of the matrices in memory, the nmz/P reads of elements of A and the nm/P writes of elements of C are all to local memory, but at least some of the reads of B must be to non-local memory. For convenience of analysis, we regard data in A and C, as well as local copies of B created by some of the techniques to be discussed below, as private, and ignore the sharing of pages of data between neighboring processors. Notice that this classification of private and shared data is different from the one we used in Section 3.2; however, the difference does not affect our performance results. Below we examine three ways of arranging the accesses to B. The three differ in the extent to which the number of reads to ‘‘distant’’ memory modules in the NUMA structure is minimized.

(1) Oblivious

Each read of an element of B is performed from the location where it resides. So, fraction $\frac{N_i - N_{i-1}}{P}$ of the $\frac{nmz}{P}$ memory accesses to obtain elements of B correspond to level i requests for

$1 \le i \le L$. For the value of parameter p (fraction of accesses to private data), in order to simplify our analysis, we regard each processor's sections of the A and C matrices to be private even though they are logically shared, since processors access non-overlapping portions (ignoring false sharing at boundaries) of those matrices. We note that each processor makes mnz/P reads to A (private), mnz/P reads to B (shared), and mn/P writes to C (private). Consequently,

$$p = \frac{z+1}{2z+1} .$$

(2) Local Copy of Columns of B

Each processor uses each element of B once for each row of C that it calculates. If each column of B is read once from its initial location and copied into local memory, then the number of remote accesses is cut by a factor of n/P. This leads to $mz\left(\frac{P-1}{P}\right)$ remote reads and the same number of local writes to copy the elements of B to local memory, so that all $\frac{nmz}{P}$ reads of B for the calculations will be to local memory. Of the $mz\left(\frac{P-1}{P}\right)$ reads of elements of B, fraction $\frac{N_i - N_{i-1}}{P-1}$ require level i requests, for $1 \le i \le L$.

Counting the numbers of memory accesses to the matrices made by a processor, we note that there are $mz\frac{P-1}{P}$ reads to B (shared), an equal number of writes to create a local copy of B (private), $\frac{nmz}{P}$ reads to A and the local copy of B (private), and, finally, $\frac{mn}{P}$ writes to C (private). Consequently,

$$p = \frac{2\,zn + z\,(P-1) + n}{2\,zn + 2\,z\,(P-1) + n} .$$

(3) Logarithmic Distribution of B

In a hierarchically structured system, such as the one we are considering here, the most efficient means of providing every processor with a local copy of a matrix B is based on a pattern of communication that emulates ''logarithmic broadcast'' used in distributed memory systems. Consider how a single element of B might reach the local memory of each processor in a four level hierarchy. Let (a, b, c) designate the processor located at branch a at the top of the hierarchy, branch b at the next level, and branch c at the lowest level. If the element is initially on processor (x, y, z), then it can be read directly by processors (a, y, z), for $a = 1, \cdots, N_4/N_3$, $a \ne x$ at cost t_4 in order to place a copy on one processor in each major subgroup. Next, for $a = 1, \cdots, N_4/N_3$, processors (a, b, z), for $b = 1, \cdots, N_3/N_2$, $b \ne y$ can read the element from (a, y, z) at cost t_3, to place a copy on each sub-sub-group of processors. Finally, in a four level hierarchy, for $a = 1, \cdots, N_4/N_3$, and $b = 1, \cdots, N_3/N_2$, processors (a, b, c), for $c = 1 \cdots N_2$, $c \ne z$ can read the element from (a, b, z) at cost t_2.

In summary, the complete distribution of the one element involves $\frac{P}{N_i} - \frac{P}{N_{i-1}}$ reads with level i requests, $2 \le i \le L$, for a total of $P-1$ remote reads and an equal number of writes to local memory at cost t_1. This determines the distribution of all mz remote reads of B done by each processor. Note that doing logarithmic distribution does not change the fraction of local reads, it just changes the distribution of levels of the read requests from strongly biased toward level L to strongly biased to level 2 (see Table 6). Hence, the p value for the Logarithmic method is the same as that for the Column method.

From the distributions determined above, it is possible to specify the parameters needed to estimate performance. Table 5 shows parameter values for each of the three variations of the matrix multiply algorithm. At the top, the value of p is shown, which is the fraction of memory accesses to private data. The bottom part of the table shows the distribution of the remaining $(1-p)$ fraction of accesses across the levels of memory. The numbers shown are for the case of four levels of memory. For a system with L levels of memory, the number of processors is $P = N_L$, and the values of $F_1, F_2, \cdots, F_L$ determine the distribution.

	Oblivious	Column	Logarithmic
p	$\frac{z+1}{2z+1}$	$\frac{2zn + z(P-1) + n}{2zn + 2z(P-1) + n}$	$\frac{2zn + z(P-1) + n}{2zn + 2z(P-1) + n}$
$1-p$	$\frac{z}{2z+1}$	$\frac{z(P-1)}{2zn + 2z(P-1) + n}$	$\frac{z(P-1)}{2zn + 2z(P-1) + n}$
F_1	$\frac{1}{P}$	0	0
F_2	$\frac{N_2 - 1}{P}$	$\frac{N_2 - 1}{P - 1}$	$\frac{P - P/N_2}{P - 1}$
F_3	$\frac{N_3 - N_2}{P}$	$\frac{N_3 - N_2}{P - 1}$	$\frac{P/N_2 - P/N_3}{P - 1}$
F_4	$\frac{N_4 - N_3}{P}$	$\frac{N_4 - N_3}{P - 1}$	$\frac{P/N_3 - p/N_4}{P - 1}$

Table 5. Parameter values for the three variations of matrix multiply.

	120 processor system (global ring)			30 processor system (local ring)			6 processor system (station)		
	Oblivious	Column	Log	Oblivious	Column	Log	Oblivious	Column	Log
p	.5004	.9256	.9256	.5004	.9770	.9770	.5004	.9959	.9959
F_1	.0083	.0000	.0000	.0333	.0000	.0000	.1667	.0000	.0000
F_2	.0417	.0420	.8403	.1667	.1724	.8276	.8333	1.0	1.0
F_3	.2000	.2017	.1345	.8000	.8276	.1724	-	-	-
F_4	.7500	.7563	.0252	-	-	-	-	-	-

Table 6. Sample parameter values for the three variations of matrix multiply.

As numerical examples, we show the p and F_i values of the three distribution methods for systems of various sizes in Table 6. The locality improvement of the Logarithmic method over the others may be clearly observed. Consequently, the performance of the Logarithmic method is the best, as shown in Figure 8, in which the process execution time expansion ratios for the Column and Logarithmic methods for matrix multiply are normalized over UMA-1 and compared to UMA-2 and UMA-3, over a range of C_Δ values on 120 processors. The Logarithmic method yields execution times close to those of the (unrealizable) UMA-2, while those of the Column method are higher. The Oblivious method generates enormous traffic at the highest levels of the network; its execution times are even worse than those of UMA-3 (which has longer memory access times, but no network contention), and are beyond the range of Figure 8. Figure 9 shows that on smaller systems (such as those with 6 and 30 processors), the performance differentials between the Column and Logarithmic methods become small, while the Oblivious method remains definitely inferior.

3.3.2. Successive Over-Relaxation (SOR)

Consider applying the SOR algorithm on an n by m rectangular grid. At each iteration, each element is updated as a function of the values of its four neighbors. Thus, in each iteration, there are five reads (the element and its four neighbors) and one write. This means that there are $6mn$ memory access operations per iteration.

We assign a block of m/P columns of the grid to each processor. (We call this algorithm, which was used in the experiments described in section 3.2, 1-D Grid.) To minimize the distance of remote accesses, blocks of adjacent columns are allocated to processors on the same station wherever possible,

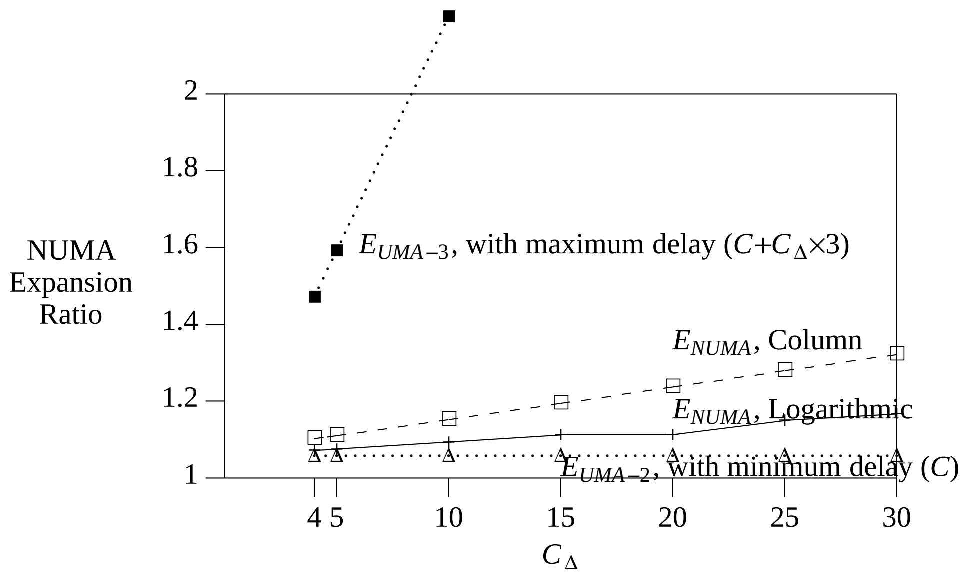

Figure 8. Performance of MM (600 × 600) on a 120-processor system normalized against E_{UMA-1}.
(All parameters from Table 2 except for C_Δ and F_i's.)

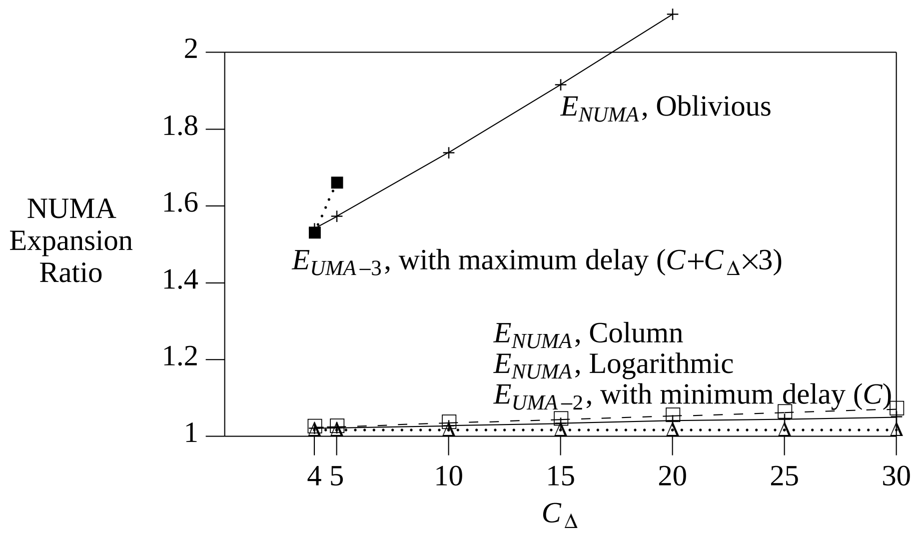

Figure 9. Performance of MM (600x600) on a 30-processor system normalized against E_{UMA-1}.
(All parameters from Table 2 except for C_Δ and F_i's.)

and similarly, the columns allocated to processors in the same local ring form a contiguous portion of the grid. Then we can observe that for every element in the grid (ignoring inconsequential boundary conditions), five of the six memory operations are certain to be local. Thus, we will take p to be 5/6. The sixth memory access will be local for all elements not in the first or last column assigned to a processor, but will be remote for the others. Since $2P$ of the m columns are either the first or the last assigned to a processor, the fraction of elements for which the sixth access is remote is $2P/m$, and the fraction that is local is $(m-2P)/m$. Finally, the remote accesses have different distances. There are N_4/N_3 of the P borders where the communicating processors are on different local rings. Similarly, there are N_4/N_2 borders where the communicating processors are on different stations, N_4/N_3 of which were accounted for above. Because of this, fraction $1/N_3$ of the $2P/m$ remote accesses are level 4 requests. Fraction $1/N_2 - 1/N_3$ are level 3 requests, and the remainder, fraction $1/N_1 - 1/N_2$, are level 2 requests.

The reference pattern of the 1-D Grid algorithm is summarized in Table 7, together with numerical values for the 120-processor system. From Figure 10, we can see that the algorithm's performance, as a function of C_Δ, is very good relative to the various UMA cases.

	1-D Grid	120-proc system
p	5/6	0.8333
$1-p$	1/6	0.1667
F_1	$\frac{m-2P}{m}$	0.6
F_2	$(\frac{1}{N_1} - \frac{1}{N_2})\ \frac{2P}{m}$	0.3333
F_3	$(\frac{1}{N_2} - \frac{1}{N_3})\ \frac{2P}{m}$	0.0533
F_4	$(\frac{1}{N_3})\ \frac{2P}{m}$	0.0133

Table 7. Parameter values for the 1-D Grid algorithm for SOR.

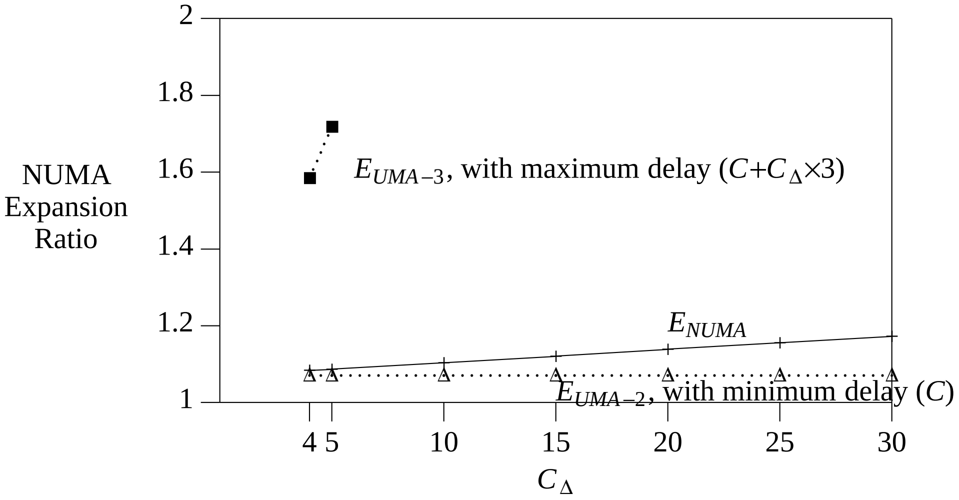

Figure 10. Performance of SOR (600×600) on a 120-processor system normalized against E_{UMA-1}. (Other parameter values are given in Table 2.)

3.3.3. LU Decomposition

We consider here a standard form of LU decomposition. Columns are distributed cyclically among all the processors in order to maintain as much load balance as possible throughout the execution. If we label the elimination steps from n to 2, then at step j, elements in one row are used to modify the $j-1$ rows below. In a multiprocessor context, it may pay to copy all the elements of the pivot column to each processor once they are calculated, so that they need not be reread remotely for each use. We consider cases both with and without the copying of the pivot column.

(1) Without Copying the Pivot Column

At step j, the $j-1$ remaining elements in the pivot column must be calculated (requiring two local reads and one local write), then $j(j-1)$ matrix elements must be updated (requiring three reads, one of which is remote to a pivot column element, and one local write). Summing these quantities for j from 2 to n, we find that the pivot column element reads account for fraction $\frac{2(n^2-1)}{5n^2+9n-17}$ of all the memory accesses, and the remainder are guaranteed to be local. Hence, we take p to be $\frac{3n^2+9n-15}{5n^2+9n-17}$. Because the remote accesses are evenly distributed across all P processors, the distribution parameters across the levels are exactly the same as in the Oblivious case for matrix multiply.

(2) With Copying the Pivot Column

In this approach, at stage j, the $j-1$ pivot elements are calculated as before, but then each processor copies the pivot elements to its local memory (requiring one remote read and one local write per element). Then the $j(j-1)$ matrix elements are updated with all four required memory accesses being local. Again, summing these quantities over j from 2 to n, we find that the fraction of remote reads is reduced to $\frac{3(n-1)}{8n^2+15n-23}$, with the remainder being local. The remote memory accesses are still distributed across all memory modules (approximately) evenly, so the fractions of remote traffic at the various levels is the same as in the No-Copy case.

	No-Copy	Copy
p	$\frac{3n^2+9n-15}{5n^2+9n-17}$	$\frac{8n^2+12n-20}{8n^2+15n-23}$
$1-p$	$\frac{2n^2-2}{5n^2+9n-17}$	$\frac{3n-3}{8n^2+15n-23}$
F_1	$\frac{1}{P}$	$\frac{1}{P}$
F_2	$\frac{N_2-1}{P}$	$\frac{N_2-1}{P}$
F_3	$\frac{N_3-N_2}{P}$	$\frac{N_3-N_2}{P}$
F_4	$\frac{N_4-N_3}{P}$	$\frac{N_4-N_3}{P}$

Table 8. Parameter values for the two variations of LU decomposition.

The results of the above analysis are summarized in Table 8. Calculations for the default 120-processor system shows that the No-Copy algorithm produces unacceptable performance, whereas the Copy algorithm yields performance close to that of UMA-2. The reason for this can be seen in Table 8. When n is at least 20 or so, the No-Copy approach has about 40% of its memory accesses requiring remote requests, the majority of which are at level L, where Copy has a negligible fraction of its memory accesses being remote. In smaller systems with 6 or 30 processors, the performance difference between the two approaches still exists, but is less dramatic. Our results are consistent with measurements performed by Medved on the Hector prototype [8].

4. Conclusions

In this paper, we have used an analytic model, supported by some experimentation, to assess how well scientific applications with large data structures will execute on NUMA architectures. We found that important application parameters include (1) the frequency of memory access instructions, (2) the cache hit rate for data references, (3) the fraction of memory requests to private data, and (4) the degree of locality in accesses to shared data. Important parameters for the architecture include (1) the access time to local memory, (2) the pattern with which memory access cost increases in the NUMA hierarchy, and (3) the structure of the interconnection network that supports the access to various memory modules from a given processor.

Relative to the ideal, but unrealizable, UMA architecture in which all memory accesses cost the same as the local accesses and all contention in the interconnection network is ignored, we found that the NUMA system performance is surprisingly competitive. For a range of application characteristics, the execution times on the NUMA system appear to be only modestly higher than on the ideal but unrealizable UMA architecture.

We examined three specific applications in detail. We found that straight-forward versions of the applications, which ignore memory structure, often would not execute acceptably on a large NUMA system. However, we found that, by changing the way the applications execute to avoid some of the remote memory accesses, the performance of all the three applications could be improved substantially. Using Hector as an example, we observe that large NUMA systems with mild ‘‘NUMAness’’ can be built. Only minor changes were require to make the three sample applications perform well on a NUMA system, despite the frequent data references and the poor data reference locality they exhibited. From this, we conclude that NUMA architectures show promise for effectively supporting large parallel applications.

Our model can be used to investigate other applications with respect to how well they would execute on architectures of various structures. It is only necessary to estimate the ten or so descriptive parameters for each such application.

5. References

[1] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik, *Quantitative System Performance* (Section 10.2), Prentice Hall, 1984.

[2] Z. G. Vranesic, M. Stumm, D. Lewis, and R. White, Hector: A Hierarchically Structured Shared-Memory Multiprocessor. *IEEE Computer 24*, 1 (Jan. 1991).

[3] M. Stumm, Measurements of Latencies and Resource Usages in the Hector Multiprocessor System, CSRI Tech. Note, University of Toronto (August 1991).

[4] A. Gupta, J. Hennesey, K. Gharachorloo, T. Mowry, and W. Weber, Comparative Evaluation of Latency Reducing Tolerating Techniques, *Proc. 18th Annual International Symposium on Computer Architecture* (May 1991), 254-263.

[5] Butterfly Parallel Processor Overview. Tech. Report No. 6148, BBN Laboratories, Cambridge, Mass. (1988).

[6] William H. Press et al., *Numerical Recipes: the Art of Scientific Computing*, Cambridge Univ. Press (1989).

[7] J. Medved, Comparison of Message Passing and Shared Memory Parallel Programming, CSC2227S Course Project Report, U. of Toronto (May 1991).

[8] F. Darema-Rogers, G. Phister, and K. So, Memory Access Patterns of Parallel Scientific Programs. *Proc. 1987 ACM Sigmetrics Conf.* (May, 1987), 46-57.

[9] L. Brochard, et al., Designing Algorithms on Hierarchical Memory Multiprocessors. *Proc. Supercomputing 1990* (1990), 414-427.

[10] E. Zimran, M. Rao and Z. Segall, Performance Efficient Mapping of Applications to Parallel and Distributed Architectures. *Proc. International Conference on Parallel Processing*, Vol.II (1990), II-147-154.

[11] A. Gupta, A. Tucker, and S. Urushibara, The Impact of Operating Systems Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications, *Proc. 1991 ACM Sigmetrics Conf.* (May 1991), 120-132.

[12] C. McCann, R. Vaswani, and J. Zahorjan, A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors, to appear in *ACM TOCS* (1993).

[13] M. Holliday and M. Stumm, Performance Evaluation of Hierarchical Ring-Based Shared Memory Multiprocessors, Technical Report CS-1992-18, Duke University (1992).

[14] K. L. Johnson, The Impact of Communication Locality on Large-Scale Multiprocessor Performance, *Proc. 18th Annual International Symposium on Computer Architecture* (May 1991), 254-263.

[15] E. P. Markatos and T. J. LeBlanc, Memory-Conscious Scheduling in Shared-Memory Multiprocessors, Computer Science Dept., University of Rochester (1991).

[16] S. Zhou and T. Brecht, Processor Pool Based Scheduling for Large-Scale NUMA Multiprocessors, *Proc. 1991 ACM Sigmetrics Conf.* (May 1991), 133-142.