Performance Modelling for Parallel PDE Solvers on NUMA-Systems

Markus Nordén

August 31, 2006

Abstract

A detailed model of the memory performance of a PDE solver running on a NUMA-system is set up. Due to the complexity of modern computers, such a detailed model inevitably is very complicated. Therefore, approximations are introduced that simplify the model and allows NUMA-systems and PDE solvers to be described conveniently.

Using the simplified model, it is shown that PDE solvers using ordered local methods can be made very unsensitive to high NUMA-ratios, allowing them to scale well on virtually any NUMA-system.

PDE solvers using unordered local methods, semiglobal methods or global methods are more sensitive to high NUMA-ratios and require special techniques in order to scale well beyond a single locality group.

Nevertheless, the potential performance gain of improving the data distribution on a NUMA-system can be considerable for all kinds of PDE solvers studied.

1 Introduction

Partial differential equations (PDEs) are important components in models of systems of many different kinds. They are used to describe e.g. fluid flow, wave propagation, chemical reactions and even economical phenomena. While using PDEs is a convenient way to describe the models, it is often a challenging task to solve them, at least for realistic settings. In such cases numerical methods usually have to be used.

Solving PDEs numerically often involves large scale computations. Traditionally, parallel computers have been used for the most demanding problems, but over time they have become more widely used. In the future, most PDE solvers will probably be parallel as chip multi-processors (CMPs) [3, 22] make their way into every desktop computer. Knowledge about parallel implementation aspects will then play an increasingly important role.

A key-issue for achieving good serial as well as parallel performance of a PDE solver is to minimize the number of memory references through reuse of data in cache memories [17]. This can be achieved if the memory references of the PDE solver show good temporal and spatial locality. On NUMA-systems there is an additional form of locality, *geographical locality*, that determines how performance is affected by memory references that are not cache hits.

We have earlier performed experimental studies of performance with respect to geographical locality of multi-threaded PDE solvers on NUMA-systems [19, 23, 30, 31]. We have observed effects of the non-uniform memory system and to some extent been able to explain them. Similar studies have also been performed by other authors [7, 26–29].

Here we go one step further and try to gain a deeper understanding of the NUMA-effect by setting up a theoretical model for the memory performance. We have a very detailed knowledge of how the computer systems we use work, but the computers are so complex that it is often very difficult to predict their behaviour. It would therefore be desirable with a simplified model, specialized towards memory performance, that mimics the important features of the memory system using only a few parameters. The aim with such a model is partly to be able to explain experimental results, partly to make predictions and formulate guidelines for different situations.

For similar purposes, it is common to use simulations [12, 20] or to observe e.g. hardware counters during experiments on real hardware [4,8,9,13,35]. Such studies often focus on cache utilization, but they have been used also for studying effects of geographical locality. An advantage is that they give very detailed and accurate information for a specific experimental setup and can help improving a program.

A weakness with simulations and experiments is, however, that they usually reveal less about the influence from individual parameters on the over all performance. It is therefore desirable that they are complemented also with analytical models in order to better understand the results. This has been done for cache utilization [1, 5, 6, 15, 25] and also for communication in MPI programs [2, 11, 21, 33, 36]. In the present study we extend the use of analytical models to include effects of geographical locality for OpenMP programs on NUMA-systems.

In the present article we develop a detailed model of memory performance on NUMA-systems in Section 2. In section 3 we reduce the detailed model to a simpler model involving less detailed descriptions of computers and programs. In Section 4 we characterize four important classes of PDE solvers with respect to the simplified model and in Section 5 we study the impact on them from different hardware and software optimizations. Finally, in Section 6, we make some concluding remarks and discuss possible future extensions of our studies.

2 A theoretical memory performance model

From a memory performance point of view, the execution of a computer program consists of a sequence of memory accesses from threads running on CPUs to addresses in a virtual address space that is mapped onto physical memory. The different abstraction levels of a computer program are illustrated in Figure 1.

Each memory access has a latency, which can be reduced by the use of cache memories. Furthermore, the latency can be hidden if the data is not needed immediately and other computations can be carried out while waiting. An example where the latency often can be hidden completely is prefetching, where a memory access is issued in advance.

In order to fully capture the memory performance, a very detailed access pattern model is needed. It is necessary to include some kind of time dependency and an order of the accesses to predict cache behaviour. Furthermore, it is necessary to determine how memory accesses and computations can overlap

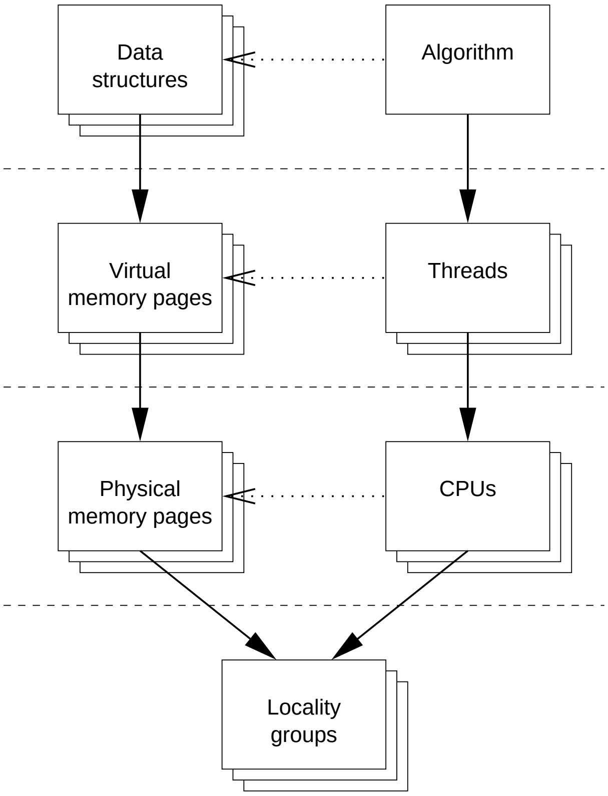


Figure 1: Different abstraction layers of a computer program. We want to solve a problem by means of an algorithm that manipulates data structures. The algorithm is implemented by means of threads and the data structures are mapped onto virtual memory addresses, which can be divided into pages. The virtual memory pages are mapped onto physical memory pages and the threads are scheduled onto CPUs, which both belong to locality groups in the computer.

each other.

Such a model will be very general, but of little practical use. On the other hand, it will be interesting from a theoretical point of view and can act as a foundation for a simplified model. In the present section we present a detailed model and in the following section we simplify it.

We notice that the total execution time, T, of a program can be split into two contributions: T_W, which is the time spent on meaningful work, and T_M, which is the time spent waiting for memory accesses to complete.

$$T = T_W + T_M$$

In this study, the focus is on geographical locality, and we will therefore focus on those memory accesses that are cache misses and how they affect T_M.

2.1 Memory access pattern

Virtual memory access pattern Let Θ be the set of threads that are running a program and let Ω be the memory footprint, i.e. the set of all addresses that are accessed by the program. Each memory access can then be identified as an edge $e \in E$ in the memory access graph $G = (V, E)$, where $V = \Theta \cup \Omega$ and $E = \{e = (\theta, \omega) : \theta \in \Theta, \omega \in \Omega\}$, and the memory access pattern $\mu \in E^N$ is a sequence of accesses.

Furthermore, let $\theta : E \to \Theta$ and $\omega : E \to \Omega$ be two functions that are used to identify the thread and the address of a memory access, i.e. $\theta(e) = \theta$ and $\omega(e) = \omega$ for $e = (\theta, \omega)$.

We also introduce a subgraph $G_\theta = (V_\theta, E_\theta)$, where $V_\theta = \{\theta\} \cup \Omega$ and $E_\theta = \{e \in E : \theta(e) = \theta\}$. Thus, G_θ will contain all memory accesses from a certain thread.

Thread scheduling and data distribution A multiprocessor consists of a set of CPUs P and a memory M. When running a parallel program on a multiprocessor, the threads have to be scheduled onto processors and the virtual memory mapped onto physical memory. We denote the thread scheduling $s : \Theta \to P$ and the memory mapping $d : \Omega \to M$.

The mapping from virtual to physical memory addresses is usually accomplished using memory pages. The virtual address space is divided into pages by a page model $\Psi : \Omega \to \tilde{\Omega}$ and each virtual page is mapped onto a physical page. A common page model is $\Psi(\omega) = \lfloor \frac{\omega}{\psi} \rfloor$ for pagesize ψ, i.e. if $\psi = 2^k$ the binary representation of the address is split into two parts, where the leftmost becomes the page number and the rightmost the offset with respect to the beginning of the page. The offset is kept unaltered during the mapping from virtual to physical addresses.

However, during the execution of the program, the thread scheduling as well as the data distribution may change. This dynamic behaviour can be resolved in the model by dividing the execution of the program into phases, where the thread sceduling and data distribution is constant during each phase.

Physical memory access pattern As mentioned above, μ is a *virtual* memory access pattern between threads and virtual memory addresses. Using the thread scheduling s and data distribution d, it is possible to translate the virtual memory access pattern to a physical memory access pattern.

Let $W = P \cup M$ be the vertices in the physical memory access graph $H = (W, \Pi)$, the edges $\Pi = \{\pi = (p, m) : p \in P, m \in M\}$ are all possible combinations of a CPU and a memory address. The vertices in V are mapped to elements in W using either s or d, depending upon which is applicable, and an edge $e \in E$ is translated mapped to an edge $\pi \in \Pi$ by the function $\pi(e) = (s(\theta(e)), d(\omega(e)))$.

Cache memories Geographical locality will, however, not be an issue for those memory accesses that hit in cache. We therefore introduce a cache model C that determines whether or not a memory access hits in cache and also distinguishes between different levels of cache memories.

In general, in order to determine whether or not a memory access hits in cache, it is necessary to know the history of memory accesses preceding it. We will however, for simplicity, use

$$C(\mu_i) = \begin{cases} k & \text{if } \mu_i \text{ hits in } k\text{-level cache} \\ 0 & \text{if } \mu_i \text{ does not hit in cache} \end{cases}$$

as a short-hand notation of the "cache status" of a memory access μ_i in μ.

2.2 Hardware topology

The topology of a multiprocessor can be described by a hierarchical set of locality groups. This has been done by Sun Microsystems in their locality group APIs [34], and this part of the model is inspired by that. It is, however, a very general model and it can be used to describe the topology of a multiprocessor from any vendor, not only from Sun. The hierarchical structure of the hardware topology for three different computer systems is illustrated in Figure 2.

Let Λ be the locality group hierarchy. Every locality group $\lambda \in \Lambda$ consists of a union of a subset of P and a subset of M, i.e. one or several CPUs and a piece of physical memory. We use the following short-hand notation to determine whether a memory access $e \in E$ occurs within a locality group or not.

$$\pi(e) \in \lambda \iff \{s(\theta(e)), d(\omega(e))\} \subset \lambda$$

Furthermore, there is an upper bound $\tau : \Lambda \to \mathbb{R}$ of the latency for a memory access between a CPU and a memory page within a locality group.

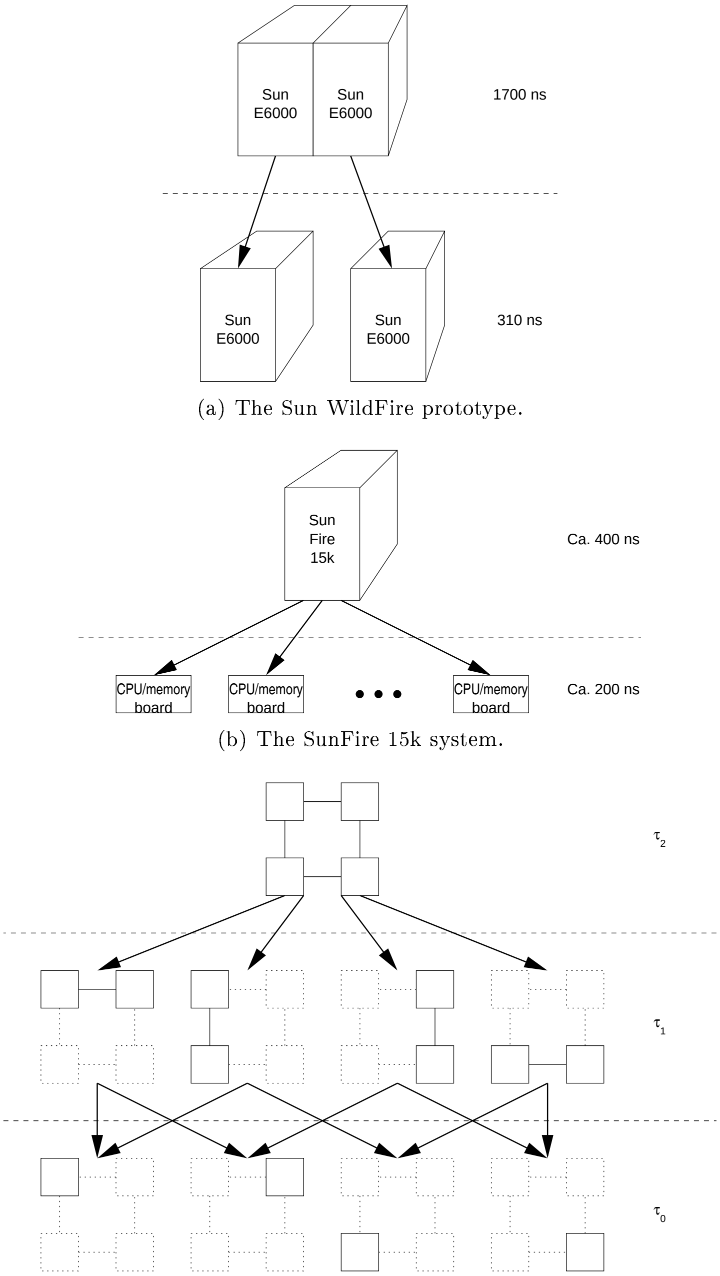
Since $\tau(\lambda)$ is in general only an upper bound of the latency, the latency can be lower within a subset of λ. This subset will then also constitute a locality group. If $\lambda_i \subset \lambda_j$, then $\tau(\lambda_i) < \tau(\lambda_j)$. A UMA system is thus a special case where the topology can be represented by one single locality group.

Since our focus in this study is on memory accesses that do not hit in cache, we will ignore the latency of memory accesses that are cache hits. For a memory access μ_i in μ, the latency will then be given by

$$l(\mu_i) = \begin{cases} 0 & \text{if } C_i(\mu) \neq 0 \\ \min_{\pi(\mu_i) \in \lambda} \tau(\lambda) & \text{if } C_i(\mu) = 0 \end{cases}$$

Furthermore, let $\tilde{\Lambda}$ be the leaves of the locality group hierarchy

$$\tilde{\Lambda} = \{\tilde{\lambda} \in \Lambda : \lambda \not\subset \tilde{\lambda} \quad \forall \lambda \in \Lambda\}$$



(a) The Sun WildFire prototype.

(b) The SunFire 15k system.

(c) A 2D hypercube, e.g. a small SGI Origin system.

Figure 2: Hierarchical locality group abstraction.

Since the locality groups in $\tilde{\Lambda}$ cannot be further subdivided, $\tau(\tilde{\lambda})$ will also be a lower bound of the latency for $\tilde{\lambda} \in \tilde{\Lambda}$.

In order to minimize the latencies of the memory accesses of a program, it is therefore desirable that as many as possible of the memory accesses occur within locality groups in $\tilde{\Lambda}$.

2.3 Performance implications

Knowing the reduced thread scheduling and data distribution above, the latency for an individual memory access can easily be determined. This latency can, however, often be overlapped, at least partly, by computations or by another simultaneous memory access from the same thread.

Let σ_i denote the time, for which the thread $\theta(\mu_i)$ can continue to execute independently after it has issued the memory access μ_i. The influence from μ_i on the total execution time of $\theta(\mu_i)$ will then be the portion of $\tau(\mu_i)$ that exceeds σ_i, or zero if it can be completely overlapped.

The memory contribution to the total execution time of a thread θ will thus be the accumulated latency, i.e. the sum of the contributions from all its memory accesses

$$t_M(\theta) = \sum_{\mu_i \in E_\theta} \max(l(\mu_i) - \sigma_i, 0) \tag{1}$$

What determines the overall performance of the program is the execution time for the slowest thread. As long as some other thread needs more time to finish its computations, the execution time of a specific thread does not influence the overall performance.

Provided that the computational work is evenly distributed between the threads, the memory influence on overall performance will be the aggregate latency, which is the maximum of the accumulated latencies of all threads.

$$T_M = \max_{\theta \in \Theta} t_M(\theta) \tag{2}$$

Furthermore, let d^* denote the optimal data distribution, i.e. the data distribution that minimizes T_M.

3 Simplification of the model

The model in the previous section is very general, but it is more detailed than desired. Therefore, we make the following simplifications in this section in order to get a simpler model that is more convenient to use.

1. We assume that no memory access is overlapped by any other instruction, i.e. that $\sigma_i = 0$.

2. We replace the detailed cache model with a measure of the cache miss ratio, which is the quotient between the number of memory accesses that are not found in cache and the total number of memory accesses. We let ρ_θ denote the cache miss ratio of a thread θ.

3. We assume that the locality group hierarchy has only two levels. This allows us to divide all memory accesses into two categories: local accesses

within a locality group on the lowest level and remote memory accesses between CPUs and memory that reside in different locality group on the lowest level. We denote the latency of local and remote memory accesses τ_L and τ_R respectively and define the NUMA-ratio ν as the quotient $\nu = \frac{\tau_R}{\tau_L}$.

4. We introduce an aggregate measure of the geographical locality, which is the quotient of the number of local memory accesses and the total number of memory accesses. We denote the geographical locality L_θ for a thread θ.

Of the simplifications above, the first is the most severe. It will affect the result significantly, in that it will no longer be a quantitative measure of execution time, but rather a qualitative measure of something "time-like". In particular for memory accesses caused by prefetch instructions, the discrepancy will be large.

The other three simplifications will only introduce small changes or no changes at all in the total result. Many NUMA-systems actually have only two levels in their locality group hierarchy in which case the third simplification does not lead to any restriction. The second and fourth simplification only mean that we replace detailed information about every single event by aggregate measures, which is completely in line with what we wish to do – compute an aggregate measure!

Observation 1 *The accumulated memory latency of a thread θ is*

$$t_M(\rho_\theta, N_\theta, L_\theta) \approx \rho_\theta N_\theta (L_\theta \tau_L + (1 - L_\theta)\tau_R)$$

Rationale Since we assumed that there is no overlap, i.e. that $\sigma_i = 0$, the terms of the sum in (1) are $l(\mu_i)$. If the thread θ makes a total of N_θ memory accesses, $(1 - \rho_\theta)N_\theta$ of these will be cache hits and do not contribute to the aggregate memory latency according to the assumption in Section 2, $L_\theta \rho_\theta N_\theta$ will be local memory accesses that contribute with τ_L each and $(1 - L_\theta)\rho_\theta N_\theta$ will be remote accesses that contribute with τ_R each. The sum in (1) is thus $\rho_\theta N_\theta (L_\theta \tau_L + (1 - L_\theta)\tau_R)$. ∎

Observation 2 *Assume that all threads make approximately the same number of memory accesses and have the same cache miss ratio, i.e. $N_\theta \approx \frac{N}{|\Theta|}$ and $\rho_\theta \approx \rho$.*

The aggregate memory latency of a program is then

$$T_M(\rho, N, \theta) \approx \rho \frac{N}{|\Theta|}(L\tau_L + (1 - L)\tau_R)$$

where

$$L = \min_{\theta \in \Theta} L_\theta$$

Rationale Observation 2 follows immediately from (2) and Observation 1. ∎

We now notice that there are two sources to degraded memory performance of a program running on a NUMA-system: There will be necessary accesses to remote memory due to truly shared data and there will be unnecessary remote accesses due to missplaced memory pages that are used exclusively by a single thread.

If the data distribution is optimal, i.e. $d = d^*$, the latter form of remote memory accesses vanish and we achieve optimal locality, i.e. $L = L^*$. In this case, the aggregate memory latency cannot be improved by altering the data distribution and the cause of the performance degradation is solely the non-uniformness of the system. We quantify this by the *NUMA-factor* F_ν, which is the relative memory performance for an optimal data distribution compared to a case with no remote accesses, i.e. a UMA-system.

$$T_M(\rho, N, L^*) = F_\nu T_M(\rho, N, 1)$$

Inserting the expression from Observation 2 and solving for F_ν yields

$$F_\nu = L^* + \nu - L^*\nu$$

If the data distribution is not optimal, i.e. $L > L^*$, there will be a contribution to the aggregate memory latency also from unnecessary remote accesses. We quantify this by the *locality-factor* F_L, which is the relative memory performance of the actual data distribution compared to the optimal data distribution, i.e.

$$T_M(\rho, N, L) = F_L T_M(\rho, N, L^*)$$

or, after substitution and simplification

$$F_L = \frac{L + \nu - L\nu}{L^* + \nu - L^*\nu}$$

We also notice that

$$T_M(\rho, N, L) = F_L F_\nu T_M(\rho, N, 1)$$

In order to quantify L^*, and thereby also F_L and F_ν, we divide the memory accesses into two sets, accesses to shared pages and accesses to exclusive pages. A shared page is a memory page that is accessed by threads in different locality groups whereas an exclusive page is a memory page that may be available to all threads, but where all accesses are made by threads in a single locality group.

If there are N memory accesses in all, we denote the number of accesses to shared pages N_S and the number of accesses to exclusive pages N_E. We thus have $N = N_S + N_E$. Furthermore, let N_C be the (average) number of "consumer threads" that make accesses to a single shared page and assume that those threads make approximately the same number of accesses.

Observation 3 *The optimal locality can be approximated by*

$$L^* \approx \frac{1}{N}\left(N_E + \frac{N_S}{N_C}\right)$$

Rationale The optimal placement of an exclusive page is within the same locality group as the thread that accesses it. All accesses to that page will then be local.

A shared page can be placed in the same locality group as any of its N_C consumer threads. The probability that an access to a shared page is local will then be $\frac{1}{N_C}$.

Observation 3 follows immediately as the average locality of all individual memory accesses. ∎

4 Characterization of PDE-solvers

In order to be able to use the model in the previous section, we need to characterize the programs we are interested in. Since the focus of the present study is on PDE solvers, we have choosen four classes of numerical methods that are extensively used for solving PDEs.

Below, we assume that the numerical methods are implemented in a straightforward manner and analyze them theoretically with respect to their memory access pattern. Our purpose is to form the basis of rules of thumb for what to expect when running a certain kind of PDE solver on a certain kind of NUMA-system.

Of course, there are also other classes of methods than the four presented here. There are methods, with their roots in one of our four classes, that have been modified in a way that have changed their characteristics. We nevertheless believe that the four categories below are good representatives of typical PDE solvers.

4.1 Ordered local methods

A *local method* is a numerical method that involves only local dependencies of quantities at grid points in a close vicinity of the grid point where something is computed, i.e. the computational kernel consits of applying a sparse numerical operator. If the gridpoints are numbered such that topological locality is preserved, we call it an *ordered* local method.

Examples of ordered local methods are finite difference methods (FDM), finite volume methods (FVM) and finite element methods (FEM) on structured grids, and also FVM and FEM on unstructured grids if the vertices are numbered so as to minimize matrix bandwidth.

Proposition 1 *The optimal geographical locality of a PDE solver using an ordered local method is approximately* $L^* \approx 1$.

Proof Let M denote the total number of vertices in the computational grid. The number of memory accesses during one application of the operator will then be $N = kM$, where k is a small constant determined by e.g. the order of accuracy of the method. For a solver using a FDM on a scalar PDE, k will typically be the number of grid points in the difference stencil.

When performing the computations in parallel, the work has to be divided among the threads. This is usually done by grouping adjacent vertices of the grid into partitions and letting the threads perform all computations in one partition each.

```
do i = 1, Nrows
  y(i) = 0
  do j = row(i), row(i+1)-1
    y(i) = y(i) + val(j)*x(col(j))
  end do
end do
```

Figure 3: Pseudocode for a matrix-vector multiplication $y = Ax$, where the matrix A is stored in the CSR format using the three arrays `val` for values of nonzero elements, `col` for column indices of nonzero elements and `row` for indices of the first element of each row.

Since we are dealing with local methods, the computations by a thread will depend on values at the grid points inside its own partition and grid points in adjacent partitions that are located close to the border. Accesses to data at grid points in the interior of a partition will thus be to exclusive pages. Accesses to data at grid points in the vicinity of the border to another partition will be either to shared pages or exclusive pages depending on whether the partitions belongs to threads in the same or different locality groups.

This leads to $N_S = k'\varepsilon$, where k' is another small constant closely related to k and ε is the edge cut of the partitioning of the grid.

Now, let δ be the number of spatial dimensions and let M be as defined above. The edge-cut between the parts of the grid that CPUs in different locality groups are responsible for will then be $\varepsilon = \mathcal{O}\left(|\tilde{\Lambda}| M^{\frac{\delta-1}{\delta}}\right) \ll M$, where $|\tilde{\Lambda}|$ is the number of locality groups.

Since $\varepsilon \ll M$, we have $N_S \ll N$ and thus $N_E \approx N$. Therefore, $L^* \approx 1$. ∎

4.2 Unordered local methods

An unordered local method is a local method, as described in the previous section, where the numbering of the gridpoints do not preserve topological locality.

Examples of unordered local methods are FVM and FEM on unstructured grids with a random numbering of the vertices. The computational kernel of such PDE solvers typically consists of a matrix-vector product

$$y = Ax$$

where the matrix A is sparse and stored in a compact format, such as the compressed sparse row (CSR) format [14].

Besides CSR, there are other storage formats for sparse matrices. If implemented carefully, their characteristics with respect to memory access pattern should not deviate too much from CSR. In the present study we assume that the CSR format is used.

The algorithm for computing the matrix-vector product is presented in pseudocode in Figure 3. When parallelizing the computations, the iterations of the outer loop is usually divided among the threads so that each thread is responsible for a contiguous set of rows of A.

Proposition 2 *The optimal geographical locality of a PDE solver using an un-*

ordered local method is approximately

$$L^* \approx \frac{2 + \frac{b}{|\tilde{\Lambda}|}}{2 + b}$$

where b is the number of integers or floating point numbers a cache line can contain and $|\tilde{\Lambda}|$ is the number of locality groups.

Proof When computing the matrix vector product, the elements of the result y will be computed and stored one by one. This means that accesses to y and the row pointers of A will be comparatively rare and that we can expect a reasonably good cache utilization. We will therefore neglect those accesses.

The other accesses to A and x will, however, be important. The accesses of A will be very structured since each thread typically is responsible for a contiguous set of rows of A. The data of these rows are stored contiguously in memory and accesses thus involve mainly exclusive pages. The accesses to x, on the other hand, will be very unstructured due to the unordered numbering of gridpoints. All threads will access all of x and the memory pages used to store it are thus shared among all threads.

Let M denote the total number of vertices in the computational grid. The number of non-zero elements of A will be kM, where k is a small constant determined by the numerical method and the average number of neighbours of a grid point. For each element in A, there will be two memory accesses; the column index and the value, i.e. in all $2kM$ accesses. These accesses will exhibit good spatial locality, since they are stored contiguously, and many of them will be cache-hits. Let b be the size of a cache line as defined above, we then have $N_E \approx \frac{2kM}{b}$, where a common size of a cache line is $b = 4$.

The accesses to the elements of x will be unordered and we cannot expect many of them to be cache hits. We thus have $N_S \approx kM$. Furthermore, all threads will access all of x and we have $N_C = |\tilde{\Lambda}|$. Proposition 2 then follows from Observation 3. ∎

4.3 Semiglobal methods

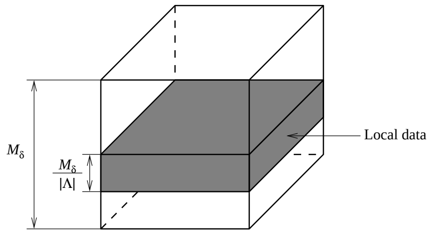
A *semiglobal method* is a method where the computation of a quantity at a certain grid point involves dependencies of quantities at all other grid points, but where the global dependency is reduced by e.g. dimensional splitting.

Examples of semiglobal methods are pseudospectral methods.

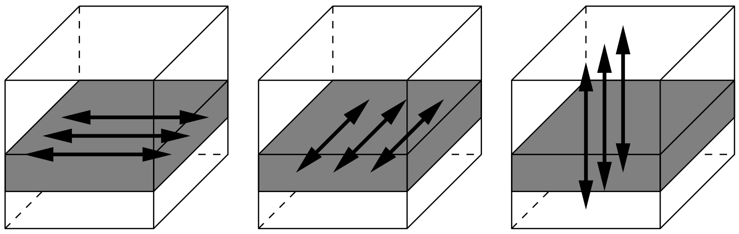
Proposition 3 *The optimal geographical locality of a PDE solver using a semiglobal method in δ dimensions is approximately*

$$L^* \approx \frac{\delta - 1}{\delta} + \frac{1}{\delta|\tilde{\Lambda}|}$$

Proof The computations of a semiglobal method can be divided into δ subphases, one for each dimension. This is illustrated in Figure 4. The computations in all dimensions except one will be local within a partition and will thus have an optimal locality $L_1^* = \ldots = L_{\delta-1}^* = 1$. In the remaining dimension, all threads will access all data and $L_\delta^* = \frac{1}{|\tilde{\Lambda}|}$, as shown in Proposition 4. The



(a) Data distribution



(b) First dimension (c) Second dimension (d) Last dimension

Figure 4: Illustration of a semiglobal method. Local data and computations of threads in one locality group are shown. The computations in all dimensions except one will involve only local data, whereas the computations in the last dimension will involve data from all locality groups.

total optimal locality will be the average of the locality for each dimension and Proposition 3 follows. ∎

Alternating-directions implicit methods [32] may seem to be semiglobal methods, but they differ in that they in addition to the solution also have a coefficient matrix, for which it should be possible to achieve good geographical locality. NUMA-effects should thus be less pronounced than for truly semiglobal methods.

4.4 Global methods

A *global method* is a numerical method where all threads use all data, e.g. for solving n-body problems. Clearly, such methods exhibit poor geographical locality. We can now quantify this more precisely.

Proposition 4 *The optimal geographical locality of a PDE solver using a global method is approximately*

$$L^* \approx \frac{1}{|\tilde{\Lambda}|}$$

Proof Since all threads access all data, we have $N_C = |\tilde{\lambda}|$, $N_S = N$ and $N_E = 0$. Proposition 4 then follows from Observation 3. ∎

There are also other reasons that make global methods inefficient and they are often replaced by e.g. multipole methods [16].

The optimal geographical locality for the four categories of PDE solvers is summarized in Table 1.

Table 1: Optimal geographical locality for the different categories of numerical methods

Method category	Optimal locality
Ordered local	$L^* \approx 1$
Unordered local	$L^* \approx \frac{2+\frac{b}{\lvert\tilde{\Lambda}\rvert}}{2+b}$
Semiglobal	$L^* \approx \frac{\delta-1}{\delta} + \frac{1}{\delta\lvert\tilde{\Lambda}\rvert}$
Global	$L^* \approx \frac{1}{\lvert\tilde{\Lambda}\rvert}$

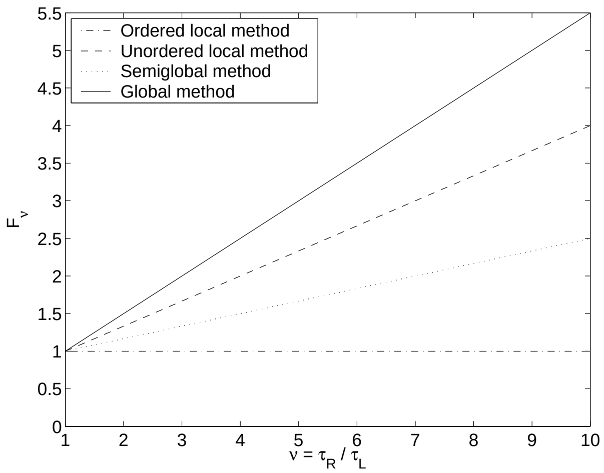


Figure 5: Impact of NUMA-ratio ν on the NUMA-factor $F\nu$ according to the model for the four categories of PDE solvers under the assumptions $|\tilde{\Lambda}| = 2$, $b = 4$ and $\delta = 3$.

5 Impact of the results

In the previous sections we derived an approximate expression for the aggregate memory latency of a program, T_M, as well as a NUMA-factor F_ν and a locality-factor F_L. We also noticed that there are two necessary conditions that need to be fulfilled in order to get good locality: the memory access pattern should not be "speckled" ($L^* \approx 1$) and the operating system should exploit this property ($L \approx L^*$).

There are many actors working on the minimization of the aggregate memory latency. Computer architects try to minimize memory latency and NUMA-ratio, compiler writers and operating system developers try to adapt the data distribution to the memory access pattern of programs and application programmers try to optimize the memory access patterns.

Furthermore, the users of the programs want to use as many CPUs as possible in their computers and to get good scalability.

5.1 NUMA-ratio

Figure 5 shows how the NUMA-factor, F_ν, depends on the NUMA-ratio, ν, for the different classes of PDE solver in the previous section. The computer system is assumed to consist of two localitygroups, $|\tilde{\Lambda}| = 2$, and have a cache line size

$b = 4$. We also assume that all computations concern settings with three spatial dimensions.

The size of a cache line is assumed to be $b = 4$, which is the case for many commercial systems.

We see that different programs show different sensitivity to high NUMA-ratios. For a PDE solver that uses a local numerical method on a structured grid, we will have $L^* \approx 1$. This means that the NUMA-factor F_ν is close to one and that the NUMA-ratio ν will have little influence on the aggregate memory latency if the data distribution is adapted to the memory access pattern. This has been shown experimentally in [19, 29, 30].

PDE solvers that use local methods on unstructured grids can show the same behaviour, if the vertices of the grid are ordered to reduce the bandwidth of the adjacency matrix. This has also been studied experimentally [24]. Such a PDE solver is categorized as an ordered local method.

For a PDE solver that uses an unstructured grid with unordered vertices, on the other hand, the aggregate memory latency will show a much stronger dependency on the NUMA-ratio. This is due to the low optimal locality L^*, or "speckled" access pattern, that makes it virtually impossible to adapt the data distribution to the access pattern. This was observed for the FVM solver in [19].

PDE solvers using semiglobal methods, such as pseudospectral methods, show similar behaviour but are somewhat less sensitive to high NUMA-ratios since only the computations in one dimension will involve remote accesses. The optimal locality of a semiglobal method can be improved further if special techniques are used. In the PS solver in [19], for example, the solution matrix is transposed repeatedly, which makes it possible to adapt the data distribution better to the memory access pattern.

Global methods, finally, has the lowest optimal locality of the four categories of PDE solvers and show consequently the highest sensitivity to high NUMA-ratios.

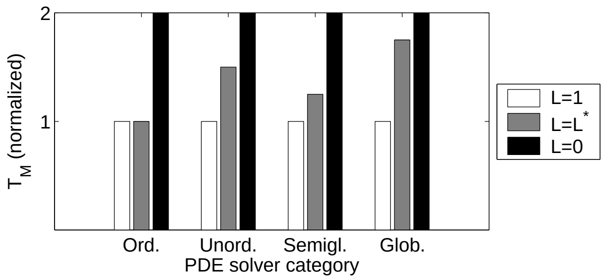
5.2 Data distribution

In Figure 5 we assumed that the data distribution was optimal, i.e. $L = L^*$, and focused on the NUMA-ratio ν. Here we turn our attention to the impact from data distribution instead.

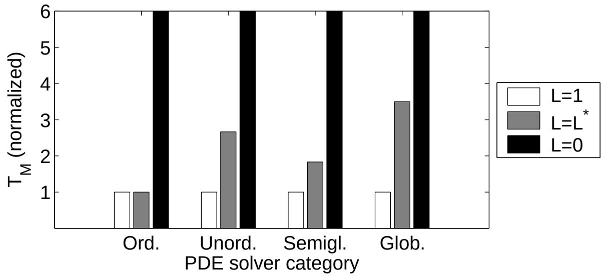
Figure 6 shows normalized aggregate memory latencies for the different kinds of PDE solvers. The parameters in the subfigures are choosen to mimic a Sun Fire 15k system [10] and the Sun WildFire prototype [18] respectively.

The white bars corresponds to the case that all memory accesses are local, i.e. $L = 1$ as would be the case on a UMA-system. All results are normalized with respect to this. The grey bars represent an optimal data distribution, i.e. $L = L^*$, and the black bars show a worst-case scenario where all data is located within a single locality group, i.e. $L = 0$, e.g. after a serial initialization.

In almost all cases, in particular for ordered local methods on systems with high NUMA-ratio, the potential performance gain of finding an optimal data distribution is considerable. Only when using a global method on a NUMA-system with low NUMA-ratio and a large number of locality groups can the effort of improving geographical locality be questioned.

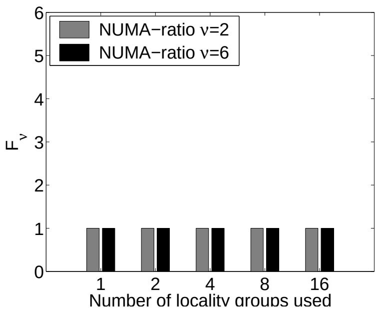


(a) NUMA-ratio $\nu = 2$ and four locality groups.

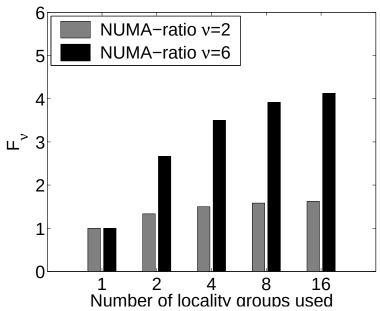


(b) NUMA-ratio $\nu = 6$ and two locality groups.

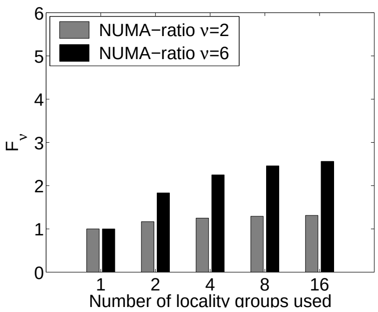
Figure 6: Normalized aggregate memory latency for a naive (serial initialization) data distribution ($L = 0$) and an optimal data distribution ($L = L^*$) compared to the UMA-case ($L = 1$) for the four categories of PDE solvers. The parameters in (a) are choosen to mimic a Sun Fire 15k system and in (b) to mimic the Sun WildFire prototype.



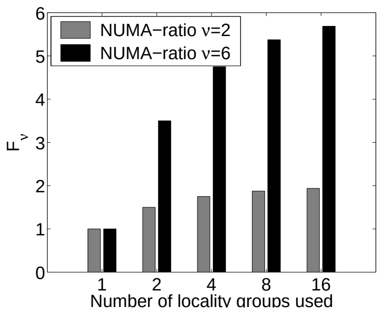
(a) Ordered local methods



(b) Unordered local methods



(c) Semiglobal methods



(d) Global methods

Figure 7: Scalability, i.e. how F_ν depends on the number of locality groups that are used. The most dramatic performance degradation occurs when going from a single locality group, i.e. a "UMA subsystem", to two locality groups. Ordered local methods are "immune" to the NUMA-effect whereas the other methods show more pronounced effects on the system with the higher NUMA-ratio.

5.3 Scalability

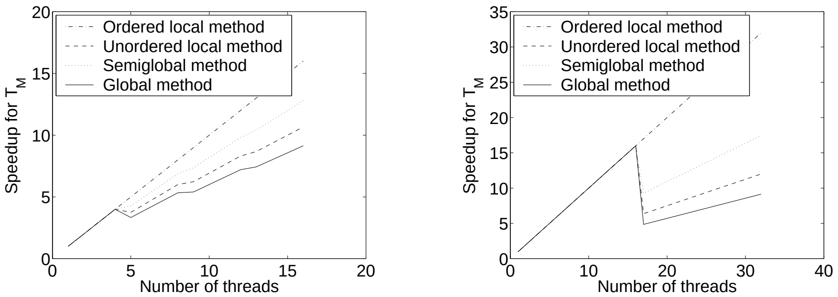
Finally, we turn our attention to scalability of the memory performance. The reason for building NUMA-systems at all is that it is difficult to build scalable UMA systems. It is therefore reasonable to believe that it will also be difficult to build large NUMA-systems with many locality groups without increasing the NUMA-ratio or introducing more levels in the hierarchical hardware topology.

Here, we look on scalability from a user perspective when running a PDE solver on a specific NUMA-system. What happens when we use more CPUs and more locality groups?

Figure 7 shows how the NUMA-factors, F_ν, depends on the number of locality groups that are used, i.e. how the overhead caused by remote accesses for an optimal data distribution increases when the number of threads, and hence CPUs and locality groups, is increased.

We have already noticed that ordered local methods are immune to high NUMA-ratios if given an optimal data distribution. Consequently, the NUMA-factor is not affected when increasing the number of locality groups.

For the other methods, we see that the most dramatical increase in NUMA-factor occur when we go from using a single locality group to using two locality groups, i.e. when we leave what is in practice a UMA system. When increasing



(a) NUMA-ratio $\nu = 2$ and four CPUs per locality group.

(b) NUMA-ratio $\nu = 6$ and 16 CPUs per locality group.

Figure 8: Speedup of T_M, i.e. T_M when using a single thread divided by T_M when using several threads, for the four PDE solver categories. The parameters in (a) are choosen to mimic a Sun Fire 15k system and in (b) to mimic the Sun WildFire prototype.

the number of locality groups further the increase in NUMA-factor is more modest.

We can thus expect that the most dramatical change in speedup occurs when the number of CPUs is increased beyond what can be confined within a single locality group. This is also shown in Figure 8, where the speedup of the aggregate memory latency predicted by our model is shown for the four different kinds of PDE solvers.

The results in Figure 8 may seem dramatic, but this only concerns the theoretical memory contribution to over all performance. When studying a real PDE solver on a real computer, there will also be a contribution from arithmetic work that scales perfectly unless there are load balancing problems. Furthermore, some of the memory latency will be hidden by overlapping computations and concurrent memory accesses. The NUMA-impact on the over all speedup will then be less dramatic.

6 Conclusions

When running a parallel PDE solver on a NUMA-system, performance is degraded by remote memory accesses. In this paper we have set up a detailed model of the memory performance of a PDE solver running on a NUMA-system. Due to the complexity of modern computers, such a detailed model inevitably is mainly of theoretical interest. Therefore we also developed a simplified model, which is an approximation of the detailed model that mimics the important features of the memory system using only a few parameters. This allows us to describe different kinds of NUMA-systems and PDE solvers conveniently and we can see what influence different parameters have.

The central parameters in the model are the NUMA-ratio, ν, the locality, L, and the optimal locality, L^*. Computer architects try to minimize the NUMA-ratio when building new systems, application developers try to write programs with as high optimal locality as possible for the choosen method and memory management systems try to find a data distribution that has a locality that is as close to the optimal locality as possible.

The parameters can be used in our model to compute two measures; the NUMA-factor, F_ν, and the locality factor F_L. Together, they describe the performance degradation when using a NUMA-system compared to a UMA-system. More precisely, the former measure describes the inevitable influence from the NUMA-system itself while the latter describes the additional overhead caused by a suboptimal data distribution.

Using the simplified model, we showed that it is possible to make a data distribution yielding almost ideal geographical locality for PDE solvers using ordered local methods. This makes them very unsensitive to high NUMA-ratios and allows them to scale well on virtually any NUMA-system.

For PDE solvers using unordered local methods, semiglobal methods or global methods, the optimal data distribution gives more moderate geographical locality. This make them more sensitive to high NUMA-ratios and therefore they show limited scalability beyond a single locality group.

In spite of this, our results show that for realistic NUMA-ratios, the potential performance gain of replacing a naive data distribution (i.e. the first touch principle for a serial initialization phase) by an optimal data distribution can be considerable.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[2] K. Al-Tawil. Performance modeling and evaluation of MPI. *Journal of Parallel and Distributed Computing*, 61:202–223, 2001.

[3] L.A. Barroso et al. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, 2000.

[4] E. Berg and E. Hagersten. SIP: Performance tuning through source code interdependence. In *Proceedings of Euro-Par*, 2002.

[5] E. Berg and E. Hagersten. StatCache: A probabiloistic approach to efficient and accurate data locality analysis. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2002.

[6] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. Technical Report 2005-028, Uppsala University, 2005.

[7] J. Bircsak et al. Extending openmp for NUMA machines. In *Proceedings of Supercomputing*, 2000.

[8] H. Brunst, W.E. Nagel, and H.C. Hoppe. Group-based performance analysis for multithreaded smp cluster applications. In *Proceedings of Euro-Par*, 2001.

[9] B.R. Buck and J.K. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In *Proceedings of Supercomputing*, 2000.

[10] A. Charlesworth. The Sun Fireplane system interconnect. In *Proceedings of Supercomputing*, 2001.

[11] D. Culler et al. LogP: Towards a realistic model of parallel computation. In *Proceedings of Principles and Practice of Parallel Programming*, 1993.

[12] L. DeRose et al. SIGMA: A simulator infrastructure to guide memory analysis. In *Proceedings of Supercomputing*, 2002.

[13] L. DeRose and F. Wolf. CATCH – a call-graph based automatic tool for capture of hardware performance metrics for MPI and OpenMP applications. In *Proceedings of Euro-Par*, 2002.

[14] A. George and J.W. Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981.

[15] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory bahavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[16] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73:325–348, 1987.

[17] W.D. Gropp et al. High-performance parallel implicit CFD. *Parallel Computing*, 27:337–362, 2001.

[18] E. Hagersten and M. Koster. WildFire: A scalable path for SMPs. In *Proceedings of International Symposium on High-Performance Architecture*, 1999.

[19] S. Holmgren et al. Performance of PDE solvers on a self-optimizing NUMA architecture. *Parallel Algorithms and Applications*, 17(4):285–299, 2002.

[20] H. Johansson, D. Wallin, and S. Holmgren. Analyzing advanced PDE solvers through simulation. In *Proceedings of Applied Parallel Computing*, 2004.

[21] D.J. Kerbyson et al. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing*, 2001.

[22] P. Kongetira et al. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[23] H. Löf, M. Nordén, and S. Holmgren. Improving geographical locality of data for shared memory implementations of PDE solvers. In *Proceedings of International Conference on Computational Science*, 2004.

[24] H. Löf and J. Rantakokko. Algorithmic optimizations of a conjugate gradient solver on shared memory systems. *To appear in International Journal of Parallel, Emergent and Distributed Systems*, 2006.

[25] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of Measurement and Modeling of Computer Systems*, 2004.

[26] D.S. Nikolopoulos et al. Is data distribution necessary in OpenMP? In *Proceedings of Supercomputing*, 2000.

[27] D.S. Nikolopoulos et al. A transparent runtime data distribution engine for OpenMP. *Scientific Programming*, 8(8), 2000.

[28] D.S. Nikolopoulos, C.D. Polychronopoulos, and E. Ayguade. Scaling irregular parallel codes with minimal programming effort. In *Proceedings of Supercomputing*, 2001.

[29] L. Noordergraaf and R. van der Pas. Performance experiences on sun's wildfire prototype. In *Proceedings of Supercomputing*, 1999.

[30] M. Nordén, S. Holmgren, and M. Thuné. OpenMP versus MPI for PDE solvers based on sparse numerical operators. *Future Generation Computer Systems*, 22:194–203, 2006.

[31] M. Nordén, H. Löf, and S. Holmgren. Geographic locality and dynamic data migration for openmp implementations of adaptive pde solvers. In *Proceedings of International Workshop on OpenMP*, 2006.

[32] D.W. Peaceman and H.H. Rachford, Jr. The numerical solution of parabolic and elliptic differential equations. *J. Soc. Indust. Appl. Math.*, 3(1):28–41, 1955.

[33] G. Rodriguez, R.M. Badia, and J. Labarta. Generation of simple analytical models for message passing applications. In *Proceedings of Euro-Par*, 2004.

[34] Sun Microsystems. *Programming Interfaces Guide*, 2003.

[35] H.L. Truong et al. On using SCALEA for performance analysis of distributed and parallel programs. In *Proceedings of Supercomputing*, 2001.

[36] M.S. Wu et al. Performance modeling and tuning strategies of mixed mode collective communications. In *Proceedings of Supercomputing*, 2005.