Access Histories: How to Use the Principle of Locality in Distributed Shared Memory Systems

Assaf Schuster and Lea Shalev (Wolfovich)

Abstract

The principle of locality is the main factor contributing to the efficiency of uniprocessor computing systems. It allowed the development of memory hierarchies, where data that have higher chance of being referenced are automatically identified during running time and moved closer to the processor. Making use of this principle for parallel computing in distributed systems turns out a lot more complicated. It involves taking performance-critical decisions about the migration of data and computation.

In this work we develop a novel technique that can be used to dynamically and transparently optimize the locality of memory references in distributed shared memory systems that compute in parallel. We dub it the histories mechanism. This method helps the running-time system in taking correct decisions concerning the adaptive redistribution of memory objects and computation tasks, an operation which typically involves high overhead.

The histories mechanism is fully implemented in user-space in the MILLIPEDE system, which serves as a test-bed. The performance evaluation shows a potential for dramatic improvement in the efficiency of shared-memory parallel computations in loosely coupled distributed environments, due to enhanced hit-ratio for local memory references.
1 Introduction.

Many attempts are made to integrate the resources and services of distributed computational environments into virtual parallel machines (VPMS). While being very cheap and available to everyone, such VPMS will exhibit very high computational power and high bandwidth of I/O and communication. Parallel applications using these environments will have to be dynamically adaptive to the varying network configurations, utilizing idle resources and instantly evicting those resources claimed by their native users. In order to integrate the resources of a distributed environment some form of cooperation among its nodes (hosts, machines) is necessary [2, 3, 13, 14, 18]. Multithreaded distributed shared memory (DSM) systems seem to be most promising with respect to portability, ease of programming, and adaptive scalability.

A DSM system is an implementation of a shared memory paradigm on a physically distributed system [12, 16]. Parallel programming in this model is easy, since the DSM is a natural generalization of sequential programming. Furthermore, with a DSM it is relatively easy to parallelize sequential programs. Components of an application (i.e., sequentially executing tasks that in this work we call threads) communicate using the virtually shared memory, where local and remote data accesses are carried in a way transparent to the programmer, serviced by the underlying DSM mechanism. This makes DSM applications both easier to develop and more portable (across DSM architectures) than programs that use explicit message passing. In particular, VPMS which exhibit virtually shared memory (and may consist of the cooperation of large suits of various machines and resources), are at the same level of abstraction as that of multiprocessor machines with physically shared memory, such as symmetrically multiprocessors (SMPs). In fact, the programming paradigms are at the same level of abstraction as that of a multithreaded uniprocessor machine.

However, distributing an application – and in particular a shared-memory application – over the network has its drawbacks. Threads need to communicate and synchronize, thus imposing overhead that, due to the relatively inefficient communication, is typically very high [15]. The goal of this work is to show how this overhead can be significantly reduced, essentially by harnessing the principle of locality.

The principle of locality is one of the central issues in the design of computers. It states that when a memory location is accessed, it is most likely that it will be accessed again soon (locality in time), and that there is a high chance that some other near-by location will be accessed soon (locality in space). For this reason, the memory hierarchy was developed for uniprocessor systems, where data items (and their near-by neighborhoods) are brought “closer” to the processor whenever they are being accessed. Many multiprocessor DSM systems attempt to utilize the principle of locality by bringing into the local memory of a machine the whole page whenever it is accessed by a thread which is executed on this machine. However, in contrast to the memory hierarchy in uniprocessor systems, bringing in the page may result in sending it out promptly, as it may also be used by another thread on a remote machine.

With explicit message passing it is often possible to optimize an application “manually” in order to reduce the communication overhead. However, these program optimizations require a lot of effort. Also, they are typically intended for a particular message-passing architecture and therefore are not portable, so that the program should be rewritten for each type of environment, or even when the number of nodes changes. Automatic methods can simplify this process, however those that are available are static, compile-time tools, that are not capable of dealing with irregular problems on dynamically changing systems such as non-dedicated non-homogeneous distributed environments.

In contrast, the DSM model hides the details of the data distribution from the programmer:
Page locations are determined implicitly by the threads accessing them. This mechanism provides a convenient way to write portable applications, but also induces communication overhead that cannot be controlled by the programmer. Due to the relatively inefficient communication, this overhead may be too high for distributed execution. One type of problem caused by the implicit data placement is false sharing: two or more logically unrelated data objects that are used by different application components are placed on the same virtual page. In this case these objects may be accessed by different threads that reside in different machines, which causes a transfer of the page between these machines. The result is (a possibly rapid) communication activity between program components that are essentially unrelated. This kind of communication overhead does not exist in parallel systems that use explicit message passing.

The second kind of communication overhead may appear also in the message passing model, namely, when two or more threads share the same data item. As with false sharing, if these threads reside on different hosts, their accesses to this item are often remote and thus cause the frequent transfer of the related page(s) across the network. However, the resulting overhead in the DSM model may be a lot higher than in the message passing model because in the latter the threads are not contending for the data: their accesses to it are implicitly scheduled by the message send/receive operations. In contrast, in the DSM model two or more threads running on different hosts that attempt to simultaneously access the same data can cause repeated transfer of the related page between the hosts, while the hosts cannot even do any useful work. We refer to such frequent page transfer as page ping-pong, whether caused by true data sharing or by false sharing.

One way to significantly reduce the amount of remote data access is by redistributing the threads (and consequently and implicitly the related memory pages) according to the application’s communication pattern, i.e., according to the page access pattern. Suppose the communication pattern was known in advance for the application A, and suppose that an unlimited preprocessing time is available. The communication pattern may be represented by a weighted bipartite graph, \( G = (T, P, E) \), as follows. Let \( T \) contain a node for every thread in the application at hand, and let \( P \) contain a node per active memory page. Let \( e \in E \) connect nodes \( t \) and \( p \) and has the weight \( w_{t,p} \), if thread \( t \) access page \( p \) with (some normalized) frequency \( w_{t,p} \). Clearly, \( G \) represents the communication pattern of the application \( A \). Minimizing the communication for \( A \), involving an optimal distribution of the work and data, can thus be done by finding a multi-cut in the graph so that the sum of the weights of edges crossing all cuts is minimized. The resulting optimization problem, called multi-way cut is NP-hard, where the size of its input (the bipartite graph) is typically in the billions (number of pages and threads). Recently, some approximation schemes were found for this problem [17]. Figure 1 gives an example for a communication pattern and its three-way cut.

Unfortunately, there are several reasons for which the implementation of the above method is not feasible.

**Load Balancing.** The minimal multi-way cut does not take into account the need for some load balancing policy. Furthermore, the load balancing may depend on the dynamically changing and possibly non-homogeneous capabilities of the underlying system, as machines may become idle or loaded unexpectedly.

**Changes in Pattern.** While the computation proceeds, there are dynamic changes in its communication pattern.

**Page Replica.** A popular optimization method in DSM systems is to relax the consistency conditions on the shared memory. When this method is applied, page copies may be dynamically created and eliminated.
Figure 1: Left: The communication pattern of a system with seven threads and four pages. Right: The pattern is mapped for execution on three machines, so that threads 1 and 3 are scheduled together, 2 and 5 together, and 4, 6, and 7 together. This mapping induces a ping-pong situation involving pages 2 and 3. Although the communication relate to the sum of the weights of the edges crossing machine boundaries, it may not be as bad due to successive accesses by a certain thread to the same page that, consequently, becomes local with respect to this thread.
Partial Information. The global communication pattern may not be known in advance, i.e., it may be implied by the input. In order to circumvent this problem, one may try to model the communication pattern during running-time. In this case, however, although the remote accesses may be recorded, the related overhead for local accesses is too high, so once again we are left with incomplete information.

In this work we present a method that shows how to collect and store partial information which approximate the (possibly dynamically changing) communication pattern, and how to use this information. The general technique is an adaptive distributed heuristic that strives to improve the distribution of the threads according to their “observed” memory access behavior, in order to achieve optimal locality of the DSM data accesses. It uses only partial information and limited resources, and incurs only a small overhead. The general idea is to collect information concerning the communication pattern into a compact data structure which we call page access history. Decisions about the redistribution of the computation are then taken according to the accumulated information. In order to avoid additional overhead, the method piggyback on the network activity that is taking place, as follows: Collecting and storing access information is done merely during the handling of remote memory access. Processing this information and using it to redistribute threads is done merely when sending threads is required anyway by the load-balancing algorithm, or when page ping-pong is detected.

The rest of this work is organized as follows. In Section 2 we compare our work to some previous approaches and present the system requirements. Section 3 presents the basic method, namely: how to choose among the candidates for migration, when migration is performed by some load sharing algorithm. Section 4 discusses ping-pong detection and treatment. Section 5 describes the implementation of the proposed histories mechanism as part of the Millipede system. Section 6 gives some performance measurements and improvements that can be expected when using the histories method. Section 7 lists some open issues and tunings yet to be done.

2 Related Works and The Model
Related Work
Most DSM systems do not use communication pattern information to improve locality of data references. Instead, they concentrate on smart coherency schemes using various replication methods. However, these methods can still impose high synchronization overhead, so that in some cases communicating objects should be co-located. There are few systems that relate to this issue; below we describe some of them.

Amber [3] is a multithreaded object-based system for distributed computations on networks of workstations (NOWs). As in DSM systems, components of an Amber application share single virtual address space and access remote objects transparently. The difference is that in most DSM systems data is moved (or replicated) to the location of a thread that tries to access it; in Amber, if a thread invokes an operation on a remote object, the thread is moved to the node where the object resides. All aspects of object location are left to programmer: data objects are moved only if the program moves them explicitly; an object is replicated only if it was explicitly marked immutable. Locality can also be controlled using additional primitives that make it possible to dynamically create groups of related objects that are guaranteed to be co-located. Thus, Amber’s programming model provides means to tune the performance of the application while renouncing network transparency. Therefore programming in this model is complicated. Since the application only is in charge of the load balancing and for the optimization of locality, achieving the primary
goal of Amber, namely: high performance, is even more complicated in a dynamically changing environment.

Locust [4] is an object-based DSM system for NOW environments. It gathers information about data dependencies at compile time and uses it to support producer-initiated data transfer and affinity scheduling (assigning tasks that share data to the same host). If the access pattern for an object cannot be determined at compile time, a separate owner module controls accesses to the object and no locality optimizations are applied. Thus, if even one such object exists in an application, it has the potential of imposing high communication overhead that cannot be eliminated. Therefore a large class of applications are handled inefficiently by Locust.

MCRL [8] is a multithreaded DSM system that implements a dynamic choice between data and computation migration. MCRL data objects (programmer-defined regions) are managed using fixed-home sequentially consistent protocol; when a thread calls a procedure that accesses remote data, MCRL contacts the region’s home node. The home node decides between data or computation migration and locks the region. Computation migration is chosen for all write operations and depending on the policy it can also be chosen for some read operations. If computation migration is chosen, the procedure is executed on a host that has a copy of the region; the thread does not migrate, so next time it will try to access the same data, this will again be a remote access. Since the cost of computation migration is low (because only a small part of the thread state is transferred), this policy can improve performance if the procedure accesses a single large data region. It imposes unnecessary overhead if the region is small and only one processor accesses it; if the region is large and there are several writers, this can restrict parallelism, since different threads cannot write simultaneously to different parts of the same region. The method does not take into account relationships between different regions. Consider the following situation: some thread alternately attempts to access (for writing) two different regions, but it cannot lock both of them simultaneously (because it is deadlock prone and would limit the potential degree of parallelism). Assume now that both regions are remote; then computation migration occurs each time the thread switches the region it is working on.

The Model

We assume the following type of environment and applications: The system consists of (possibly non-homogeneous, non-dedicated) machines, interconnected by some communication network. In particular, the system may include SMP multiprocessor machines, each considered here as a single host due to its physically shared memory.

We assume a page-based system, so that when a remote memory access occurs, the corresponding page may move or copy into the local memory of the issuing machine. We remark that the method presented in this work is not restricted to this framework and may be described in the general context of memory objects. Nevertheless, in order to use a uniform terminology, and in order to avoid tradeoffs related to memory overhead, we refer in the sequel to page-based DSM systems only.

Thread migration. The system is multi-threaded, and a migration module exists, capable of moving threads (that are not marked as immobile) between the machines.

Slow communication. Sending messages between machines in the network is slow. We thus assume that when communication occurs it makes sense to add some extra overhead in order to reduce future communication. In contrast, additional overhead must be avoided for memory
accesses that are local; Hence, applications which stabilize on some steady state in which locality is maximized, experience no overhead whatsoever.

**Coarse granularity.** The overhead associated with creating a thread and with initiating a remote execution is relatively high. Therefore a thread should carry a sufficient amount of computation in order to justify the cost of its creation or migration. Thus, we assume that the expected lifetime of a thread is relatively high.

**Unpredictable computation and communication requirements.** Requests for the execution of one or more threads may arrive in an arbitrary timing. No assumption is made about the memory access patterns of the threads. No a-priori knowledge is assumed about the relative amounts of communication and computation used by the applications.

### 3 Thread Scheduling

As mentioned above, a DSM system must be adaptive to the environment dynamic changes and the application varying needs. In order to fulfill this requirement one may use thread migration for sharing the load among the nodes in the system. However, improper placement of threads might degrade performance because of the high communication overhead. Therefore, the threads that are chosen to migrate when the load balancing policy is enforced must be selected with care so to minimize the resulting remote memory accesses.

In this section we present a general method that can be used in a DSM system for selecting threads that will migrate to another host so that the expected communication will be minimal. We assume that some external module decides that the migration of $n$ threads from host $H_1$ to host $H_2$ is to take place; Then, this module invokes the thread migration module. The goal of the migration is load transfer from an overloaded host to an idle or an underloaded one; however the thread selection strategy is independent of the load balancing policy that is implemented in the algorithm which initiate the migration. Thus, the load balancing algorithm is not described here; in fact, any standard load sharing algorithm can be used for this purpose.

Thus, the thread selection module is concerned only with the problem of optimizing for maximal locality. In order to carry these optimizations, it collects the information concerning remote page accesses, and in this way it “learns” the communication patterns of the threads. What make things interesting is that – for performance reasons – the information about local accesses is not recorded. Thus, the knowledge about the communication patterns is incomplete and incorrect decisions may be taken. For example, in the case that all threads which frequently access the same page are running on the same host, no information is (initially) recorded, so some of these threads might be selected as candidates for migration to another host. This will turn out to be a poor decision, since it will cause page ping-pong. However, making this seemingly-wrong decision may eventually turn out to be helpful. As a result of this decision, not only that information about the page will become available, making it possible to correct this decision, but also we will be able to keep the information that is obtained in order to improve future decisions. We proceed to describe below how all this is done.

#### 3.1 Choosing Migrating Threads.

The thread selection module is concerned merely with the problem of optimizing for maximal locality; it receives as input both a destination for the migration and the number of threads to migrate. It starts by determining the set of threads that are allowed to migrate. A thread may fail to
fit into this category because of several reasons. It might have just started to perform some location-dependent activities, so that it cannot migrate. It might be waiting for some synchronization event, so migrating it will not contribute to sharing the load. Finally, it might have migrated recently because of a page ping-pong, so that an additional migration may cause a thread ping-pong. This issue is explained in detail in Section 4.

Once the Migration Server determines which threads can migrate, it selects such threads for which their migration is expected to minimize the amount of remote DSM references for both the source and the destination hosts. Let \( L \) denote the local host and \( R \) to the migration target host. The migration should not increase the number of remote accesses and should reduce it whenever possible. This requirement is easily met if there exist \( n \) threads that reside on \( L \), and that are using only pages that are transferred between \( L \) and \( R \), such that these pages are used by no other thread on \( L \). In this case, as a result of migrating these threads, the corresponding pages will become local on \( R \), or at least they will not be accessed from \( L \), while the other pages will not be affected by the migration.

In the general case, the selected set of threads should have the following characteristics.

Maximal frequency of remote references to pages on \( R \). This condition makes it possible to decrease the number of existing page transfers between \( L \) and \( R \).

Maximal frequency of any remote references. Since the threads are intended to migrate to a less loaded host, it makes sense to move those of them which have high communication demands. However, this condition should be given less weight than the previous one, as it does not reduce the total amount of communication.

Minimal access frequency of the threads remaining in \( L \) to the pages used by the selected threads. This condition attempts at minimizing the expected number of accesses to these pages by the remaining threads. If the threads which remain in \( L \) are not using these pages at all, the pages may become local on \( R \).

Minimal access frequency to local pages. Since local accesses are not recorded, there is no information regarding the local threads that were using the page lately. It is only possible in some cases to know that a thread was once using this page, and that the page is local. Thus, the condition is applied only when the corresponding frequency can be inferred from the history of execution on another host. The related threads, for which we know that they were once using the page, are assumed to continue using it (“inverse” principle of locality). The reasoning behind this condition is that transferring the threads that are using a page may cause the transfer of this page to \( R \); if only part of these threads are transferred, a ping-pong on this page may occur.

Evidently, these requirements may conflict, so a tradeoff should be found.

Our solution is to let the thread selection algorithm estimate for each thread the improvement that can be achieved by migrating this thread to the target host. To this end, the thread selection algorithm uses a heuristic function which takes into account the information that is available about the communication pattern of the candidate thread and the impact of migrating it on the other threads.

If there is no information concerning remote accesses (i.e., in the beginning of the execution), or if there is more than a single choice, then an additional heuristic is applied: two threads that were created successively have a higher chance of sharing data (See [6, 5]). Accordingly, successively spawned threads are preferred in such cases.
3.2 Page Information.

On each host, the local history of remote-references is collected using data provided by the DSM mechanism. As explained above, for performance reasons, only remote accesses are treated, hence no information concerning local accesses is available. For each remote access, the following are recorded: page id, peer host id, access type, and accessing thread id. This information is collected into three different types of stores, called histories, as described below.

The goal of splitting into different storage types is double-folded. As we will see, this split simplifies the retrieval of the necessary information during decision time. Importantly, it also protects the system from unstable behavior that may be caused by communication bursts: On the one hand, care should be taken to limit the amount of memory that is used to store the histories; on the other hand, enough information must be stored even in case that the memory constraints do not allow to keep the full history. In addition to the above, out-of-date information should be discarded, otherwise wrong decisions may be taken due to events that took place a long time ago and do not characterize the current behavior anymore. The details follow.

History of a page is maintained in a certain host for each page that visits it, unless the page leaves the host and does not return for “a long time”, which is a configurable parameter of time units, denoted \( T_{epoch} \). It includes the following information about a limited number of the most recent paging events (page receiving, sending or invalidating):

- Event type: an attempt to access the remote page, receipt of the remote page, or sending the page to another host.
- The remote host id.
- Event’s timestamp. The timestamp is used both to discard out-of-date information, and to determine the frequency of references to this page. It is important to note that the timestamps are strictly local: there is no global clock in the system, nor do we assume any coherency mechanism on the global ordering of the timestamps.

The length of the history is limited for each page; out-of-date information is removed from the history even when there is enough place to store it.

Access history of a thread is maintained for each thread that is currently executing locally (so when a thread leaves the host its access history goes with it). It includes the following information about a limited number of its most recent remote accesses:

- page id;
- timestamp of the access.

An access that occurred more than \( T_{epoch} \) time units ago is discarded. If the page resides locally, the thread is assumed to continue using this page, and the corresponding information is stored in the thread reduced access history. If access information is discarded because of space limitations, then its “summary” is stored in the thread reduced access history, as described below.

Reduced access history of a thread.

Additional data structure called reduced access history is required for each thread \( j \). It contains reduced or old information about remote accesses of the thread that occurred a long time ago, but are still relevant because the page became local. It also contains reduced information about accesses that occur recently, but cannot be stored in the full history because of space limitations.
The **reduced history** includes at most one entry for each page that is referenced by $j$. The entry describing page $P$ includes:

- The frequency of remote accesses of $j$ to $P$ calculated from the full access history (just before it was discarded from there).
- The timestamp of $j$'s most recent access to $P$ (if it was already deleted from the full history).

Information about pages that became local (i.e., they reside at the same host as the thread) stay in the reduced history forever, or until they stop being local - due to either page or thread migration - for $T_{epoch}$ time units.

**Migration of history.**

The history of each page is maintained locally on each host visited by the page. The access histories of a thread (full and reduced) are maintained locally on the host that is currently executing the thread. When a thread migrates to another host, its full access history is translated to the reduced format; that is, reduced information about each page in the full history is extracted and added to the reduced history. The resulting reduced access history migrates together with the thread, and is used by the target host to improve future decisions taken by its own migration module.

### 3.3 Thread selection using a heuristic function.

We describe now the general method used to select $n$ threads that will migrate from $L$ to $R$. Choosing the best set of $n$ threads seems to be infeasible because of the high computation overhead and the incomplete information. Therefore we suggest a greedy strategy: at the beginning some heuristic value is calculated for each thread, then the best thread is chosen according to this value, and the heuristic value for all other threads is revised to take into account the changes in the communication pattern that will be caused by migrating the selected thread. The process is repeated iteratively until $n$ threads are chosen.

It seems, however, that a naive “hill-climbing” strategy will not work. If $m$ threads that communicate with each other are located at the local host (where $m \leq n$), then migrating only part of them is a bad decision, while migrating all of them is a good one. Therefore, the heuristic function must take into account both the current communication pattern and the number of threads that should be chosen.

Taking into account the requirements imposed on the migrating threads and the above argument, we suggest the following generic heuristic function. The heuristic value will be the additive sum of the contributions by the individual pages, where a page contribution may be either positive or negative, depending on several factors. The contribution is increased if the page is used by threads on $R$ because the migration of this thread to $R$ will bring together the threads that are using this page, thus improving the locality of accesses to this page. For the same reason, each thread that uses this page and that was already selected for migration should increase the contribution.

In addition, the contribution should be weighted according to the frequency of accesses to this page. The reason is that the locality of accesses to a page is more important when this page is frequently accessed. This factor is more subtle than the previous one, and can be used to avoid the migration of only part of a set of communicating threads by assigning negative heuristic values, as follows: Each thread that references the page and will definitely stay at $L$ (either because it’s immobile or because there are too many threads in $L$ that use this page) should decrease the
contribution. When the additive heuristic value reaches a negative number the candidate thread will not migrate, thus keeping it at the same host with other threads with which it communicate.

The contribution of a local page is calculated using exactly the same method by using the information contained in the reduced history. Of course, this information might be unavailable for some local pages, or may be available for only part of the threads that are using a certain local page. This brings us to an important question: what is the quality of decisions that can be expected from the thread selection algorithm? The next section answers to this question.

3.4 Correctness of the Migration Module Decisions.

The algorithm uses partial information because only information about remote accesses is available. Therefore if few threads are locally accessing a page $P$ it is possible that no information about it becomes available. Then nothing can prevent the thread selection module from choosing only part of these threads for migration to another host. If the threads are using the page frequently, this will be a bad decision, because it will cause repeated transfer of this page to and from the host, namely, it will cause a page ping-pong.

Clearly, incorrect decisions are inevitable. Fortunately, as a consequence of each such wrong decision, more information will become available for both hosts, because the accesses to the page will become remote and the information about them will be recorded in the histories of the corresponding threads and in the page history. Thus, at the next time one of the involved hosts will send its threads it will be able to make a better decision. The resulting global effect is some initial unstable period, during which threads migrate relatively frequently, and the system is busy in collecting information about the communication pattern. After some time, the system reaches a stable state in which the locality of page references is optimal with respect to the heuristic value calculations.

Note, however, that in the above scheme the activation of the thread selection module is done by an external module. For example, it might be invoked only when serious load imbalance is detected. This incurs a new problem, since a ping-pong situation may occur also when the load is perfectly balanced. Since ping-pong causes high overhead, it should be treated promptly when it happens. Therefore, some additional mechanism is needed to treat such situations independently. The following section describes such mechanism.

4 Ping-Pong Treatment.

As explained in Section 3.4, the optimal (or suboptimal) locality of data accesses cannot be achieved by using a “smart” scheduler, because we neither assume a-priori information about the way the threads use the DSM, nor can we obtain complete information about it during running-time. Thus, although our algorithm for the selection of migrating threads strives to improve the locality of data accesses, it may make wrong decisions. Namely, it might position threads that are using the same data on different hosts, which causes ping-pong situations. As explained in Section 3.4, this will provide the histories with more precise information regarding the communication pattern. Hence, if the same thread selection module would have been activated again, it could possibly solve the problem. The problem we face is that this module is activated by some external module (the load sharing) that is not aware of the situation. Therefore we suggest an additional mechanism, namely, the ping-pong treatment module.

The ping-pong treatment module performs the task of detecting and treating ping-ongs. It strives to transfer all threads participating in a ping-pong to the same host whenever possible (without breaking the load balance or causing instable behavior). The algorithm description follows.
When a ping-pong treatment module detects a ping-pong situation on a page \( P \), it determines (using the page history) which hosts participate in this ping-pong, i.e., which copies of the application have threads that are rapidly using the page \( P \). Then it starts a protocol that determines which host can execute all the threads which participate in the ping-pong. If such a host is found, all threads using \( P \) migrate to it. The threads' access histories are updated so that the scheduler will make better decisions in the future. If no such host is found, then no thread will migrate, and delays are used for flow control: all hosts will lock the page \( P \) for \( \Delta_P \) time units every time it is received from another host.

### 4.1 Ping-Pong Detection.

Each time an access to a remote page occurs, the ping-pong treatment module is invoked. Suppose some local thread attempts to use a remote page \( P \). The ping-pong treatment module records the information concerning this access in \( P \)'s history and in the access history of the thread, and examines the history of previous remote accesses to \( P \). If the total number of entrances and exists of \( P \) to/from the host is less than some threshold \( N_a \), then the module decides that not enough information is available to detect that a ping-pong is taking place. Otherwise the ping-pong treatment module checks whether the page is transferred to/from the host too frequently. The general criterion for detecting a ping-pong situation is described below.

Informally, a ping-pong is detected when the following two conditions are met. First, local threads attempt to use the page a short time after it leaves the host, which indicates that they are using it frequently. Second, the page leaves the host a short time after it enters, thus indicating that there are remote threads that are using this page frequently. Obviously, there are many ways to formalize these conditions. In Section 5 we describe one of them.

### 4.2 Determining Where to Migrate.

Here we address the problem of selecting the target host for transferring threads which participate in a detected ping-pong. Part of the hosts participating in the ping-pong might detect it at different times, others might not detect it at all. The objective is to choose one of the hosts and to transfer all the participating threads to that host. Unfortunately, the selection of the target host cannot be done locally at a host that detects a ping-pong, for the following reasons. First, care should be taken that all participating hosts will make the same decision. However, the hosts might have incomplete or out-of-date information about the other participants; they also do not know how many threads at the other hosts are using the page. Since this information is crucial for making a good decision, the hosts should exchange it between them. Second, more than a single page may be involved in a ping-pong. Treating them simultaneously may cause poor decisions, i.e., by selecting the same host as the target for all of these pages (thus increasing the load imbalance). Therefore, the ping-pong treatment must be handled in a case-by-case manner.

The decisions how to handle a ping-pong are thus made in a centralized way by the **Ping-Pong Server**. The Ping-Pong Server is in charge of collecting all the necessary information about a ping-pong and for resolving it. It uses global information about the load status in order to select the least loaded host as a destination for migrating the threads which take part in the ping-pong. This information may be maintained, for example, by the load-sharing module.

A copy of an application initiates a ping-pong resolution protocol when it receives a ping-pong message from the Ping-Pong Server, or when it detects a ping-pong situation that was not yet announced by the Ping-Pong Server. When the ping-pong protocol for a page \( P \) is initiated, it examines the page history of \( P \) to determine the set of threads that are using it and the corresponding
set of hosts. Note that the host might be communicating with only part of the set of hosts which participate in the ping-pong, and thus it needs to extract the full list from the page history of \( P \). Then it notifies the Ping-Pong Server the number of local threads that are using \( P \), whether they are migratable, and the list of the hosts that he knows are participating in the ping-pong (this may not be the full list, though). Once the identity of the target host for all ping-pong threads is selected by the Ping-Pong Server, it is notified to all machines hosting these threads, which in turn send them to the selected location.

The Ping-Pong Server collects information about a page ping-pong in order to make a decision concerning the ping-pong resolution. Both the global load status information and the ping-pong information that are collected may become out-of-date as a consequence of the server's decisions concerning other ping-pongs. Thus, the Ping-Pong Server does not process more than a single alarm simultaneously. Rather, if it receives a ping-pong message while processing a previous one, it ignores the new alarm, differing its handling until after completing processing of all ping-pons that were previously detected.

If the Ping-Pong Server receives a ping-pong message when it is not busy processing another one, it sends a query to each host that is mentioned in the message. The Ping-Pong Server continues collecting the information until it receives up-to-date information from all the hosts that are mentioned in any of messages that relate to this ping-pong. Once all necessary information is received the server selects the least loaded host (according to the appropriate load-sharing policy) to which all the involved threads are allowed to migrate. The server also attempts to minimize the number of threads that will migrate, which may even have higher importance than the least-loaded criterion if the final outcome is not too unbalanced. If the final number of threads that need to migrate is greater than some threshold, then the server aborts the ping-pong treatment, and no migration will be carried. In this way the server tries to keep the load balance and to avoid high migration overhead. Finally the Ping-Pong Server notifies the identity of the selected destination to all the involved hosts.

4.3 Treating Ping-Pong When Migration is Impossible.

Transferring all the threads that are using the page \( P \) to the same place may cause unstable behavior of the system, and may thus sometimes be impossible. The simplest example for this problem is depicted in Figure 2. On each host there is a set of threads using the same data (which is not used by other sets); in addition there is a thread that uses the data of both sets. When this additional thread moves to host \( A \), it causes page ping-pons on all the pages except for these used by the threads running on \( A \). If this thread were allowed to migrate each time a ping-pong is detected, thread ping-pong would occur: the thread will be rapidly transferred from one host to another, doing almost no useful work and causing overhead that is a lot higher than that of transferring the pages. This situation clearly violates the system stability and must be prevented. In order to keep the system stable, threads are not allowed to migrate every time a ping-pong on one of their pages is detected. After one migration to resolve a ping-pong situation the migrating thread is not allowed to migrate for some predefined period of time. In addition, the total number of ping-pong migrations per thread should be limited, so that global instabilities that are not resolved by the proposed histories mechanism are avoided as well (see remark below).

Another example for unstable behavior involves two page ping-pons \( P \) and \( P' \) with some intersection of the corresponding sets of threads, such that sending both sets to the same location will seriously violate the load balance. When the ping-pong on \( P \) is resolved according to the guidelines of Section 4.2, all involved threads are sent to some selected location \( A \). This will result
Figure 2: Thread ping-pong
in a page ping-pong on \( P' \), since its corresponding set of threads are now split in at least two
locations. When this ping-pong is resolved, all involved threads are sent to some location \( B \) which
is different from \( A \). Now, again, the set of threads interested in \( P \) experience a ping-pong between
\( A \) and \( B \). Clearly, letting the server continue in resolving both ping-pongs will cause a thread ping-
pong of the whole set of threads that are interested in both \( P \) and \( P' \): they will rapidly migrate
between \( A \) and \( B \).

In general, unstable situations occur when at some point in the application a large number of
threads happen to reference the same page. One may keep these threads on several hosts, or collect
all of them to the same place, paying the additional cost of high overhead and breaking the load
balance. Either case, unstable behavior will be caused, by the detection of a ping-pong in the first
solution, or in the second option because a short time after the migration the thread scheduler will
try to re-distribute the load. Since thread migration is an expensive operation, and since significant
load imbalance should be avoided, the number of threads that are allowed to migrate when resolving
a ping-pong is limited by a configurable constant.

**Remark:** It is our belief that solving the above situations should be done via several other
mechanisms. One (optimization-time) suggestion involves sending the programmer an indication
that will enable him to relax the consistency conditions on certain variables, so that they will be
able to spawn copies. Another (running-time) method corresponds to false-sharing situations, in
which it is possible to split the falsely-shared data into two different padded pages, thus solving the
ping-pong (this might require some intermediate table for pointer handling). For space limitations
we avoid here further discussion on these directions, focusing on the suggested histories technique.

Due to all the above, treating a ping-pong may be impossible since part of the threads are
locked in their hosts because of the previous ping-pong, or since there are too many participating
threads. In these cases the threads continue running on different hosts. To decrease communication
overhead and to avoid instability (resulting from repeated transfers of the page while the threads
have no chance to do useful work), each host locks the page for some configurable time \( \Delta \) after
receiving the page. Thus, it is guaranteed that the page can be used by the local threads for at
least \( \Delta \) time units. Obviously, the optimal value of \( \Delta \) depends on the current behavior of this page;
the problem of choosing this value is the subject of further research and is not addressed here.

5 Implementation: Millipede

In order to prove that the methods presented in the previous sections can improve dramatically
the performance of a DSM system, we built MILLIPede — a work-frame for adaptive distributed
computation on network of personal workstations, \([5, 7, 10, 9]\).\(^1\) MILLIPede includes a user-level
implementation of a DSM and thread-migration mechanisms \([11]\); it provides a convenient way to
execute multithreaded DSM applications with transparent page- and job- migration. Based on the
ideas described above, MILLIPede provides a built-in optimization for data locality.

In this section we give a brief description of the MILLIPede system, and proceed to explain how
it implements the general method from Sections 3 and 4.

\(^1\)Currently MILLIPede is implemented on the Windows-NT™ operating system, see also
http://www.cs.technion.ac.il/Labs/Millipede
5.1 System Overview

Each machine in the system runs a **Millipede Daemon**: a process that is in charge of collecting and disseminating load information, of managing **Millipede** applications, and of the dynamic load sharing (Figure 3).

A **Millipede** package includes two libraries: the DSM (memory manager) and the MGS (migration server). The DSM provides the interface for allocating distributed shared memory and keeping it consistent. The MGS provides the interface for creating multiple parallel activities and for managing them; it controls their locations and performs the migration.

**Millipede** applications are written in a parallel language independent of the underlying operating system, of the number of available processors, and of the data and threads location. Currently ParC [1] (a natural parallel extension of C) is supported; porting is underway for ParFortran90, ParC++, and Java. A ParC program is precompiled; the resulting C code is compiled using a conventional C compiler and is linked with the DSM and MGS libraries. The libraries are independent of the ParC language constructs; they provide an interface that allows implementation of any similar precompiler for any other language. It is also possible to write an application in a conventional language and use the libraries directly; this (less convenient) way may be used if the application requires some exotic synchronization method that is not supported by existing precompilers.

A **Millipede** application is started by a user from the **Millipede** Daemon running on the user’s machine. The Daemon automatically distributes the application over the network. An application consists of instances (copies) of a user program running on different nodes in the system (see Figure 3). If a node is an SMP, all available processors are used in a transparent way by a single instance of an application. Instances of an application share single virtual space. They communicate in a location-independent way via the DSM mechanism and synchronize using the MGS primitives.

In the rest of the section we explain how the methods presented in the previous sections are implemented in **Millipede** and combined together with some other features (such as load sharing and eviction) to build an adaptive work-frame for distributed computing.

5.2 Millipede Daemons.

**Millipede** daemons are in charge of the dynamic load sharing. They collect and disseminate load information, identify idle workstations, and distribute the **Millipede** applications to these machines.

A **Millipede** daemon consists of the following modules.

**Idle Detector.** This module checks whether the host is idle, i.e., it is not used by its owner interactively and the load caused by the non-**Millipede** background processes is low. Each time the host becomes idle or becomes non-idle the Idle Detector notifies the Eviction Server.

**Eviction Server.** This module initiates and stops eviction of foreign applications from the host. It uses information maintained by the Application Info Manager and notifies each local instance of a foreign application when this instance should start or stop the eviction of its local jobs.

**Application Info Manager.** The Application Info Manager collects administrative information about each **Millipede** application running on the local host; This information includes the host’s unique identifier, instance identifier, main host name, system process handle, etc.

**Local Load Info Manager.** It collects load information about each application running on the host, such as the number of local jobs and the number of mobile local jobs. It also determines the global load state of the local host.
Figure 3: millipede architecture
Masters. A separate manager called Master is created for each new application that was started locally. The master is in charge of distributing its application over the network.

Master Info Manager. This module collects master information about each application that was started locally, e.g., the status of the application on each host. It also collects global load information about all hosts in the system. The information maintained by this module is used only by Masters, therefore the module is activated only if there exist applications that were started locally.

Communication. This module is used for communication with other Daemons and local Migration Servers of MILLIPEDE applications.

Eviction

The MILLIPEDE system runs on a non-dedicated network of personal computers. To make the system acceptable by the native users, user ownership must be provided in a transparent fashion. That is, a user must be guaranteed to get the full resources of the workstation when he is using it. Remote applications should not degrade interactive response, and must not slow down the execution of local processes, otherwise the users will not agree to share their resources with MILLIPEDE applications.

When the MILLIPEDE Daemon detects that a user starts working on the workstation or that the load caused by non-MILLIPEDE processes is too high, it initiates an eviction of the foreign applications. When evicted, an application attempts to transfer all its jobs to the master host, which is assumed the home machine of the user who initiated the application. For part of the jobs it may be impossible because they perform some location-dependent activities. Therefore, they can still remain at the host until they complete these activities; Once completed, they notify the MGS, which migrate them to the master host.

5.3 Parallelism vs. Communication in Millipede

There are several aspects of the parallelism vs. communication tradeoff in the MILLIPEDE system. One of these stems from MILLIPEDE's support for running multiple applications simultaneously. The general objective is to compromise between maximum parallelism and minimum communication. Thus, since different applications are not communicating, they should run on different hosts whenever possible. On the other hand, the communication between the components of the application should be minimized without causing load imbalance.

The control over the parallelism and the communication is done cooperatively by the Daemons and the Migration Servers in the following way. The Daemons distribute the applications over the network and determine the initial number of jobs for each application on each host. They strive to find an optimal assignment of hosts to applications, namely, to achieve sufficient load sharing using a minimal number of application instances.

The mission of the Migration Servers is to optimize the communication within the application with respect to the decisions of the Daemons. That is, the Migration Servers try to minimize the amount of communication caused by the DSM mechanism without breaking the load balance achieved by the Daemons. The way the Migration Servers do these optimizations is based on our method as described in the previous sections; the details specific to the current implementation are described below.
5.4 The Thread Migration Module

The Migration Server in each instance of an application maintains the jobs that are using this instance, collaborates with the local Daemon and the other MGSSs on scheduling the jobs, and collects necessary information from the DSM mechanism in order to improve locality of data accesses. The MGS module contains also functions that provide a mechanism for communication between mobile jobs.

5.4.1 Relation between the DSM and the MGS libraries

Thread migration mechanism in MILLIPEDE is based on the assumption that all non-local data that is used by a thread resides in the DSM. Using the DSM is location-independent. Therefore, when a thread migrates, only its stack and context should be transferred.

The MGS collects information that is provided by the DSM in order to assist decisions regarding migration. Basically, the DSM mechanism informs the Migration Server about remote page accesses. This information is used by the MGS to determine if threads should be redistributed in order to decrease communication. In some cases the MGS also affects the decisions taken by the DSM, by advising it to lock a page on a certain host for a short while. This helps stabilizing the system when remote data accesses are causing high communication overhead, but thread migration is not possible.

In addition, the MGS uses the DSM mechanism to store part of the necessary information. For example, the MGS uses the shared memory to keep track of the location of the running threads. This significantly simplifies the task of keeping this information consistent, as threads may migrate several times.

5.4.2 Thread Migration Policy

Thread migration in MILLIPEDE is transparent to the application. A thread may be suspended at almost any moment and resumed on another host. Thread migration occurs in the following cases:

- **An overloaded node sends work to an underloaded one to decrease load imbalance.** Migration of this type is initiated by the MILLIPEDE Daemons; the MGS selects the threads that will be transferred using a heuristic function that estimates the expected communication improvement. This function was designed using the general method presented in Section 3.

- **Threads that cause high communication overhead are brought together.** This is the implementation of the ping-pong treatment method presented in Section 4.

- **Remote threads are evicted by the machine when the native user starts using it.** This provides user ownership in a non-dedicated distributed environment of personal workstations.

It might seem that there’s a fourth type: when migration is used to utilize resources that become available on hosts that become idle. However, when a host becomes free this is sensed by the local Daemon that is present at this site, which initiates a request for work. If, as a result, threads migrate to this host, then this is a load-sharing operation, obviously of the first migration type.
5.4.3 The Heuristic Function for the Selection of Migrating Threads.

We now describe how the page history is used for the selection of \( n \) threads that will migrate from the local host \( L \) to the host \( R \). The process presented here is an example of the general method presented in Section 3.3.

We use the following notations.

\( S_P \): The set of local threads using the page \( P \). A thread is assumed to use \( P \) if \( P \) appears either in its full or in its reduced access history.

\( c_P \): A “weighted count” of the local threads in \( S_P \). Each mobile thread that uses \( P \) is counted once in \( c_P \); each immobile thread (i.e., a thread that currently cannot migrate) is counted twice. The effect is to simulate a situation with a larger number of threads in \( S_P \), which reflect the fact that not all threads using this page can migrate, by making migration less attractive in the heuristic value calculation.

\( S_j \): The set of pages used by \( j \) (as can be inferred from history).

\( T_j^R \): The average time between remote accesses of thread \( j \) to pages on host \( R \) in the last \( T_{epoch} \) time units.

\( T_P \): The average time between remote accesses of the local threads to page \( P \) during the last \( T_{epoch} \) time units.

After removing out-of-date information from the page histories, the migrating threads are chosen using the following method. First, for each local thread \( j \) the value of the heuristic function \( h(j) \) is calculated using

\[
h(j) = h_1(j) + a \cdot h_2(j),
\]

where \( a \) is a configurable parameter that is used for tuning the weights of the contributions \( h_1(j) \) and \( h_2(j) \). \( h_1(j) \) and \( h_2(j) \) are calculated using

\[
h_1(j) = \frac{T_{epoch}}{T_j^R},
\]

\[
h_2(j) = \sum_{P \in S_j} w_p^j \cdot \frac{T_{epoch}}{T_P} ; \quad w_p^j = b + n - c_p .
\]

Intuitively, \( h_1 \) gives greater weight to threads that accessed the pages on \( R \) more frequently. It becomes zero for threads that did not access pages on \( R \) for more than \( T_{epoch} \) time units.

\( h_2(j) \) estimates the decrease in page transfers to and from \( L \) resulting from migrating \( j \) out of \( L \). \( h_2 \) consists of the individual contributions of all pages that are used by \( j \). The contribution may be either positive or negative, depending on the number of local threads that are using the corresponding page. It becomes negative when the number of local threads that will not migrate and that are using the page exceeds some configurable threshold \( b \) (set to 1 in the current implementation). The contribution of a page that does not have information in the full history is calculated using exactly the same method by using the information contained in the reduced history.

The contribution is also weighted according to the frequency of remote accesses to the page, so that the contribution is more significant if the page is accessed more frequently.

After calculating the heuristic values, the migrating threads are chosen according to the heuristic values, as follows. While not enough threads are selected the following procedure is performed:
1. The thread \( j \) having maximal value \( h(j) \) is chosen, and it is determined whether \( j \) is migratable.

2. For each page \( P \) that is used by \( j \) the value of \( c_P \) is updated as follows.
   If \( j \) can migrate then \( c_P = c_P - 2 \) in order to increase \( h_2 \) for the threads that use this page.
   
   If \( j \) cannot migrate then \( c_P = c_P + 1 \) in order to decrease \( h_2 \) for these threads, thus reflecting on the fact that not all threads using this page can migrate.

3. For each thread \( j' \) that was not selected yet, its heuristic value is revised, taking into account the forecast concerning paging information after \( j \) will migrate. This involves re-calculating \( h_2(j') \) using the new values of \( c_P \).

### 5.4.4 Ping-Pong Treatment

**Millipede**'s MGS includes a ping-pong treatment module that was described in Section 4. **Millipede**'s Ping-Pong Server resides in the main instance of the application. All instances include a Ping-Pong Detection Module and communicate with the Ping-Pong Server as was explained above. In order to avoid breaking the load balance, the Ping-Pong Server uses load state information that is maintained by the Master Info Manager of the local **Millipede** Daemon. The MGS of each host that is involved in a ping-pong situation reports to the local Daemon thread transfers that are initiated by the Ping-Pong Server.

Motivated by the discussion in Section 4.3, **Millipede** locks a DSM page in the host for \( \Delta \) time units after the page is received from another host. The current value of \( \Delta \) is a predefined parameter. However, it is obvious that the optimal value is equal for different applications, for different pages, or even for the same page at different times. Determining the value of \( \Delta \) dynamically for each page (the subject of further research by the **Millipede** group) might help to achieve additional performance improvement.

The ping-pong detection condition for a page \( P \) is described below. This is just one possible way to determine formally whether the communication overhead associated with a certain page becomes "too high". We use the following notations (where "average" is taken on the \( N_t \) last transfers of \( P \) to/from the host):

\[
\begin{align*}
T_{\text{unused}} & \text{ - the average time between sending } P \text{ to another host and an attempt to access it (before it is brought back).} \\
T_{\text{waiting}} & \text{ - the average time from an attempt to access } P \text{ (when it is owned by another host) to receiving } P \text{ back.} \\
T_{\text{useful}} & \text{ - the average time from receiving } P \text{ to sending it to another host.}
\end{align*}
\]

We make the following observations concerning these values.

1. The times depend on machines' hardware parameters such as CPU speed.
2. \( T_{\text{useful}} > \Delta \) even if a request for the page was received immediately after the page was brought to the host.
3. \( T_{\text{waiting}} \) grows when the network load grows (meaning that decreasing communication overhead is critical).
4. When more than two hosts participate in a ping-pong, and the page rapidly moves between two of them, the requests for the page that are initiated by a third host might ping-pong between them, “chasing” the page, and thus causing even higher communication overhead. Furthermore, this request can be delayed at each visited host for $\Delta$ time units or more, in which case $T_{\text{waiting}}$ grows accordingly.

Therefore, taking all these factors into account, a ping-pong is detected if $T_{\text{unused}} < T_{\text{waiting}}$ and $T_{\text{useful}} < T_{\text{waiting}}$. More precisely, the Migration Server decides that ping-pong takes place when

$$\frac{T_{\text{unused}} + T_{\text{useful}}}{T_{\text{waiting}}} < r_0$$

(4)

where $r_0$ is some configurable threshold. We remark that although we do not know of a precise way of doing it, taking the fraction is aimed at eliminating the dependency of $T_{\text{useful}}, T_{\text{unused}}$ and $T_{\text{waiting}}$ on the configurable parameter $\Delta$.

6 Performance Evaluation

As described in the previous section, the millipede system includes a user-space implementation of the histories mechanism. In this section we present the results of our experiments with the millipede system, focusing on the effects of the histories mechanism on the locality of memory references.

In general, testing the millipede system for just one parameter at a time is very hard, sometimes impossible. The reason for that is that millipede is designed to be very flexible, promptly responding to environmental changes. Thus, when one parameter reaches (is fixed to) some value with which it handles the situation inefficiently, other parameters would adapt their behavior accordingly. During this section we give several examples for this phenomena. In Section 7 we elaborate on some on-going research that is aimed at finding solutions to some of the resulting restrictions.

The Traveling Sales-Person Problem

The Traveling Salesman Problem (TSP) is an example of an NP-hard optimization problem in graph theory. Given a connected graph with weighted edges, the shortest Hamiltonian path should be found, i.e., a path traveling through all the nodes of the graph, so that the sum of the edge weights along the path is minimal. We give here a brief description of the parallel algorithm used to find an exact solution for this problem. Basically, the solution to the TSP problem is to scan a search tree having a node for each partial path of any Hamiltonian path which starts in a given node of the input graph, see Figure 4. More precisely, for a node representing the path $i_0 \rightarrow i_1 \rightarrow \cdots \rightarrow i_k$, its children are the nodes representing all paths of the form $i_0 \rightarrow i_1 \rightarrow \cdots \rightarrow i_k \rightarrow s$, where $s$ is different than $i_1, \ldots, i_k$. Thus, each leaf of the tree represents a Hamiltonian path in the input graph, and the objective is to find a leaf that represents the shortest path.

In the parallel algorithm work is divided among threads; each of these threads receives a set of initial paths (tree nodes), where its mission is to search for the minimal path in sub-trees of these paths. Each thread performs a DFS-style search in each of its subtrees; the search is exhaustive in the worst case. In order to optimize the search, all threads use a shared variable to store the weight of the shortest path; a thread cuts off the search in a certain subtree if the weight of the partial
Each thread uses a certain amount of dynamically allocated memory. This memory should be allocated in the DSM to make thread migration possible. Depending on the DSM allocations size, some of the threads might get their memory on the same pages, which is a typical example of false sharing. We compared three different variants. In the first variation, denoted NO-FS, false sharing is avoided by allocating more memory than necessary (the allocations are padded to precisely fit into a page). In the other two variants, each page is used by \( k \) threads. One of them, called FS, uses no optimizations, whereas the other one, called OPTIMIZED-FS, uses optimizations for data locality by enabling the histories mechanism.

**Applying locality optimizations**

As one can see in Figure 5, treating false sharing using our method is about as effective as avoiding false sharing by enlarging the DSM allocations; both improve the performance substantially compared to the case in which false sharing is not treated (in fact, almost to the point of optimal speedup). As expected, the relative improvement resulting from the use of the history mechanism becomes higher when the number of hosts increases, because there are more false sharing situations to resolve.

One may ask: Why use the optimization when it is so simple to avoid false sharing by padding? There are two answers. First, in many cases it is impossible to increase the size of all allocations up to the size of a virtual page. The reason is that a program can use a lot of small memory chunks; allocating each chunk on a separate page may not be feasible: it degrades performance by enlarging the working set of the process, potentially causing swapping to the disk and even running out of disk space. In addition part of these pages might be replicated on other hosts, so that more memory and disk space is consumed on all hosts that run this application. Therefore optimizing such applications for locality (when no padding of the allocations is done) is critical. The second
reason is that exactly the same problem, namely that of higher communication overhead, exists when some threads are sharing and accessing the same variable, i.e. there is *true data sharing*. In this case bringing these threads to the same host is the only way to avoid high communication overhead.
Figure 5. The effect of the locality optimizations using the histories mechanism for the TSP. In the NO-FS case false sharing is avoided by aligning all allocations to page size. In the other two cases each page is used by 2 threads: in FS no optimizations are used, and in OPTIMIZED-FS the history mechanism is enabled.

Table 6. This table contains statistics regarding applying locality optimization to the TSP application with false sharing for different $k$ (number of jobs contending for a page). Applying locality optimizations decreases dramatically the number of DSM messages (page lookup and transfer). The added overhead imposed by ping-pong treatment mechanism and increased number of thread migrations is negligible.
Table 6 summarizes the results of running the TSP algorithm on six machines with different numbers of threads contending for the same page.

EXPLAIN BETTER ABOUT $k$!!!

The table shows a dramatic reduction in the network traffic when the history mechanism is activated: the number of DSM-related messages, which reflects on the miss-ratio (ratio of remote to total number of accesses), drops by a factor of 30 to 40! Note that the number of extra messages that are added by the history mechanism itself is negligible compared to the improvement in the number of DSM-related messages.

MAKE THE ESTIMATION ABOVE PRECISE!!!
COMPLETE THE TABLE!!

Locking DSM pages - for how long?

Here we study the stabilizing impact of locking a DSM page in a host for $\Delta$ time units each time it is received by the host. The value of $\Delta$ is currently determined in a static fashion, depending merely on the type of the page: The pages allocated and used by the application are locked for $\Delta_{\text{default}}$ time units, while the pages used by the MGS library for maintenance purposes are locked for $\Delta_{\text{MGS}}$ time units. Here, the goal of our experiment is to demonstrate the importance of tuning the value chosen for $\Delta$. The experiment also exemplifies that this choice should depend on the behavior of specific pages, rather than being static.

WHICH FIGURE???

Ping-pong detection parameters

Here we show the impact of tuning the sensitivity of the ping-pong detection, i.e., the minimal number of recent remote accesses that should be recorded for a page before checking for the ping-pong condition on this page. Figure 7 shows the results of decreasing the sensitivity on OPTIMIZED-FS. In OPTIMIZED-FS most remote accesses are ping-pong situations, hence reducing the sensitivity monotonically increases the execution time.

In order to test a somewhat more involved algorithm, we use a variant of the TSP application, called MODIFIED-TSP, as follows. In MODIFIED-TSP, the global variable holding the weight of the optimal path is no longer updated by each thread every time it finds a better path. Rather, the best value found by each individual thread is kept by it in a shared variable. An additional thread wakes up every once in some predefined time interval, reads the shared variables of all the working threads and update the global variable.

In the original TSP application, the global minimum weight variable was causing ping-pong, but this ping-pong was not treated because too many jobs were involved in it. In the modified application, each private minimum variable is accessed only by its owner and by the common updating thread, therefore if ping-pong will be detected on this variable, it will be treated. This is obviously needless, so if ping-pong detection is too sensitive, unnecessary thread migrations take place. On the other hand, if true ping-pons are not detected, significant page migration overhead is imposed. The results appear in Figure 8. The graph is not as “smooth” as desired, because more parameters tend to affect it, and in particular the measure for a host being over- or underloaded. Nevertheless Figure 8 does give an idea for the importance of tuning the ping-pong detection sensitivity.
Figure 7.

The ping-pong detection parameter at axis X is a minimal number of recent remote accesses that should be recorded for a page before checking for ping-pong condition on this page; thus the sensitivity is maximal for minimal number of the parameter. Best results are achieved at maximal sensitivity, since all pages are accessed frequently.

Figure 8.

Since part of the pages are accessed frequently and part - only occasionally, maximal sensitivity causes unnecessary ping-pong treatment and significantly increases execution time. On the other hand, if the sensitivity is too low, real ping-pongs are missed, so their high communication overhead is not eliminated.
7 Conclusions and Open issues

The implementation of the principle of locality in uniprocessor systems is very simple and efficient. Its implementation in multiprocessor distributed systems, though not less profitable, turns out to be a lot more complicated. Yet, we have seen in this work an adaptive, transparent mechanism that can be used for this purpose, and for which the benefits are dramatic with respect to the efficiency of parallel applications. We believe that this direction hides many more ideas yet to be discovered, and that it will eventually make parallel computing on common and available clusters of workstations both efficient and simple.

Some of the issues that need further investigation were cited throughout the work. The most interesting of these is the interaction between the history mechanism and the load balancing algorithm.

7.1 Interaction with the Load-Balancing Algorithm

As mentioned above, applying the history technique does not depend on the specific load-balancing algorithm. In contrast, as we now show, the effectiveness of the method can be further enhanced by enabling tighter cooperation with the load-balancing algorithm.

Consider for example the following load-balancing algorithm that is implemented in MILLIPEDE, see Section 5. The basic idea is that hosts are classified to overloaded, underloaded, and normally loaded, so that the underloaded hosts ask for work from the overloaded ones. The way a host is determined underloaded or overloaded may employ various methods for determining the corresponding thresholds.

Suppose for example that a host is determined underloaded when it is running less than two runnable threads, and is determined overloaded when it is running more than four runnable threads. With these thresholds an underloaded host asks an overloaded one for at most three threads. If the number of threads in a certain false-sharing is, say, five, then an underloaded host asks for three threads, thus it gets three out of five threads that are all rapidly communicating. This obviously causes a ping-pong: the ping-pong is detected after a while and is resolved in moving all the involved threads to the same place. However, since now the target host is overloaded, then due to some load-balancing operations, part of these threads might move again to other hosts; thus causing ping-pong once again.

This procedure will repeat itself for as long that the false sharing exists, unless we forbid sending only part of a set of threads that are known to be communicating with each other even if the machine hosting them is overloaded, which will keep the load unbalanced. This clearly incurs a locality vs. load-balancing tradeoff, involving fine tuning and collaboration of the heuristic parameters and the load-balancing algorithm, in order to reach optimal decisions.

One suggestion that is implemented in the MILLIPEDE system uses migration count as a stabilizing tool: each ping-pong handling (in fact, each migration) increases a migration count of the involved threads, so after few ping-pong detections the threads are not allowed to migrate anymore. This is helpful for example when threads take part in several ping-pongs simultaneously. Unfortunately, applying this mechanism in the situation described above will prevent the good results that are expected from the history mechanism, since when migration is disabled, the threads do not necessarily run on the same host, so the ping-pong will remain unresolved. Clearly, the history mechanism could benefit from some interaction with the load-balancing algorithm which could, for example, adapt the threshold values.

Another scenario is that the heuristic function refuses to separate communicating threads which reside at the same host. Thus, after some initial rearrangement no migration will take place
because the underloaded hosts ask for chunks of work which involve number of threads which is just too small. In this case extreme load imbalance may occur: some hosts finish their part and stay completely idle, while others still have a lot of work that could otherwise be split. For instance, the system may reach a state in which 20 threads execute on one of the machines. The threads are divided into two groups of 10 threads each, where all the threads in each of the groups are communicating intensively with each other, but no communication exists between the groups. Clearly, one of these groups should be sent to one of the underloaded hosts, which may be prevented by the fact that 10 threads is greater than the overloaded threshold parameter of the load-balancing algorithm. Once again, this requires some smart mechanism which will adaptively identify and resolve this problem.

7.2 Other Open Problems

We give here several additional parameters and issues that call for further research (or even simple tuning). This is definitely only a partial list.

- Dynamically determining the optimal number of hosts that should be employed in the execution of a certain application according to various parameters, e.g., the total number of threads and the communication overhead.

- Testing paging heuristic parameters. Clearly, all parameters that are used to give weights for the heuristic value calculations need fine tuning. Furthermore, the heuristic function itself has some obvious attractive alternatives. All these may need a lot of experimental study.

- Dynamic calculation of $\Delta$, the time a page is locked on a host upon arrival. $\Delta$ might depend on the number of threads that are using a page and on the behavior of these threads. However, this will change the observed behavior of pages and will make it more difficult to analyze (as the decisions are taken locally, while the behavior depends on unknown decisions of remote hosts).

- When should data be removed from the reduced access history? If a page is not transferred any more, it stays in the reduced histories of all the threads that were using it, and that continue running on a host that has a copy of the page. There are two possible reasons for this situation. First, the page may stop being in use. In this case its effect on the heuristic value is harmful. Second, the page is still in use, but all the threads that are using it run locally. In this case, it is essential to keep this information. Obviously, some mechanism is necessary to find out which situation is taking place.

- Global data – pages used by all threads should not be taken into account; only $\Delta$s should be used as a stabilizing factor. In the current implementation this is solved by ignoring pages that are used by “too many” threads. As described in Section 7.1 this approach restricts the effectiveness of the whole method.

Acknowledgments

The authors are grateful to Ayal Itskovitch for his help in implementing the history mechanism in the MGS module of the Millipede system. Ayal implemented and designed the Millipede system, including its DSM support for the MGS and various other helpful features. The authors would like to thank also Shiri Manor and Yariv Tal who helped in carrying the performance measurements.
References


