Using Remote Access Histories for Thread Scheduling in Distributed Shared Memory Systems

Assaf Schuster[1] and Lea Shalev[2]

[1] Computer Science Department, Technion, Haifa, Israel 32000.
assaf@cs.technion.ac.il,
WWW home page: http://www.cs.technion.ac.il/~assaf
[2] IBM R&D Center, Center of Advanced Technologies, Haifa, Israel.
Much of this work was done while this author was still with the Department of Computer Science at the Technion.

Abstract. We developed the *remote access histories mechanism* (RAHM), a technique that can be used to dynamically, adaptively, and transparently improve the locality of memory references in distributed shared memory systems (DSMs). The RAHM helps the run-time system make correct decisions concerning the redistribution of threads to hosts – operations that typically involve high overhead.
The RAHM is independent of the memory consistency model, and comes into play only when the memory model fails to achieve locality of memory reference. The RAHM piggybacks on existing communication and adds only a negligible overhead. It is designed to work in dynamically changing systems, such as non-dedicated distributed environments.
The RAHM is fully implemented in user-space as part of the MILLIPEDE system, a Virtual Parallel Machine (VPM) designed for non-dedicated NT/PC clusters. The performance evaluation shows a potential for dramatic improvement in the efficiency of shared-memory parallel computations in loosely coupled distributed environments, due to an enhanced hit-ratio for local memory references.

1 Introduction

Improved efficiency of uniprocessor computing systems depends mainly on the principle of locality, which led to the development of memory hierarchies. Applying the same principle to parallel computing in distributed systems involves making performance-critical decisions concerning the migration of data and computation. Unfortunately, it turned out to be a lot more complicated than its uniprocessor counterpart.

Many multiprocessor DSM systems attempt to utilize the principle of locality by bringing a whole page into the local memory of a machine, whenever that page is accessed by a thread which is executed on that machine. However, this may result in the page being sent out promptly, to be used by another thread on a remote machine. Although low-level DSM hacking (such as padding pages, page-in locks, work-split on page boundaries, etc.) enables efficient manual optimization, it requires alot of effort and cannot be expected to solve the problem completely. Automatic methods can simplify this process; however, those available are static, compile-time tools, not capable of dealing with irregular problems on dynamically changing systems such as non-dedicated non-homogeneous distributed environments.

One type of problem caused by the implicit data placement in DSM systems is *false sharing*: two or more logically unrelated data objects that are used by different application components are placed on the same page. These objects may be accessed by different threads that reside in different machines, causing frequent transfer of the page between these machines. A similar situation may occur when several threads share the same data item, in which case threads that are placed at different hosts and attempt to simultaneously access the data can cause repeated transfer of the related page between the hosts, while the hosts are unable to do any useful work. Such frequent page transfer, whether caused by true data sharing or by false sharing, is called *page ping-pong*.

A popular method in DSM systems is to weaken the consistency conditions on the memory behavior. Each of the hosts contending for a memory object may get its own private copy, thus eliminating ping-pong situations. Reduced consistency, however, is not always a satisfactory solution. Sometimes weak memory cannot be applied, or the memory location must be synchronized frequently in order to keep certain variables highly consistent. In fact, frequent synchronization of the same location can cause ping-pong situations similar to those resulting from data sharing in a strong memory consistency model.

In general, weak memory models are a hassle to apply, and verifying the correctness of the resulting program is difficult at best. Some relaxed memory models attempt to assist the programmer by imposing a special programming paradigm, which, assuming the same memory behavior for all of the variables, is translated into efficient memory management. However, programmers (possibly importing already existing code) may find the imposed programming paradigm inadequate for certain tasks, and in particular too restrictive, or too permissive, for their programs or their coding style. Furthermore, because all modified variables are synchronized whether or not this is necessary, many relaxed models involve additional overhead which makes it hard to obtain speedups.

In this work we focus on a different optimization domain which may significantly reduce remote data access in DSM systems. Here the improvement is obtained automatically at run-time by redistributing the threads according to the application *memory access pattern*.

In order to better understand our method, we first consider an optimization problem known as the *multi-way cut*. Suppose that the memory access pattern is known in advance for the application at hand and an unlimited processing time is available. The memory access pattern may be represented by a weighted, bipartite graph, with one side consisting of the threads, the other side consisting of the memory pages, and an arc, weighted w, connecting the ith thread with the jth page, if the ith thread accesses the jth page with (some normalized) frequency w. Minimizing the communication with an optimal distribution of the work and data can be accomplished by finding cuts in the graph that will minimize the sum of the weights of edges crossing all cuts. The resulting optimization problem (called, as we said, multi-way cut) is NP hard (see [11] for 2-approximation algorithms), where the size of its input is typically in the billions.

Although this method is deficient for several reasons, its relation to our own method is instructive. Our method succeeds in circumventing the problems given rise to by the multi-way cut, which does not, for instance, always take the need for some load balancing policy into account, nor consider the fact that load balancing may depend on the dynamically changing and possibly non-homogeneous capabilities of the underlying system. Dynamic changes in the memory access pattern are also not accounted for by this method. Furthermore, the problem is complicated by the fact that most DSM systems use page replication (in either strong or weak consistency models) to decrease DSM-related communication; after replication, what used to be an optimal cut is so no longer. Finally, the information available before run-time may not be complete; the global memory access pattern may be implied, for instance, by the input.

Attempting to avoid the partial information problem by modeling the memory access pattern during run-time raises the following problem. Although the overhead required for recording remote accesses is negligible in comparison to that of the involved communication operation, it is much too high for recording local accesses. Local accesses cannot, therefore, be recorded, and the gathered data remains incomplete.

The method we propose in this paper actually utilizes partial information about data accesses (collected at run-time) to approximate the (possibly dynamically changing) memory access pattern. The method, called RAHM (Remote Access History Mechanism), 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. RAHM collects information about the memory access pattern into a compact data structure which we call *page access history*. At any point in time, this data structure may be viewed as an approximation of the full representation of the application's memory access pattern, containing that part of the information which can be inferred by observing the application's remote memory references up to this point. Decisions about the redistribution of the computation are then made by consulting the accumulated information.

The RAHM can be applied regardless of the strength of the memory consistency model. When memory references are local due to a local copy of an item, the RAHM is inactive and incurs essentially no overhead. This is also the case when, due to relaxation of the memory consistency, there is a single remote access to the item which brings in a copy, so that all subsequent accesses will also be local.

The RAHM itself only records events that require communication and need not know anything about the memory consistency. However, its mechanism ensures that it will only "wake up" when the applied weakening of memory consistency (if applicable and exists) fails, causing remote references to a memory item that are frequent enough to require intervention. This frequency can, in fact, be viewed as the interpolation of all the important parameters: the strength of the memory model, the rate of accesses to the item, and the rate of memory synchronization operations. Thus, for example, the RAHM decision mechanism reacts identically to the case of an item accessed frequently by a thread where the memory model is weak, and to that of an infrequently accessed item in a strong memory consistency model.

The RAHM requires only limited resources and incurs, as we said, only a small overhead. It piggybacks on the network activity that is already taking place, so that the collection and storage of access information is done only when remote memory access occurs. Processing this information and using it to redistribute threads is done only 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 the rest of this section we give our assumptions about the underlying system on which RAHM is to be implemented and mention related work. Section 2 presents the basic method for choosing among the candidates for migration, when migration is required by some load balancing algorithm. Section 3 discusses ping-pong detection and treatment. Section 4 gives some performance measurements and expected improvements. We conclude the paper in Section 5.

System Requirements

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 with physically shared memory, each considered here as a single host.

We assume a page-based system: when a remote memory access occurs, the corresponding page may move or is copied 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 more general context of memory objects. Nevertheless, for the sake of uniform terminology, we refer to page-based DSM systems only.

Thread migration. The system is multi-threaded, and a migration module exists, capable of moving threads between the machines. **Slow communication.** Sending messages between machines in the network is relatively slow; it makes sense to add some extra overhead in order to reduce future communication. Additional overhead *must be avoided*, however, for local memory accesses. Of course, applications which stabilize in 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. A thread should therefore carry a sufficient amount of computation in order to justify the cost of its creation or migration. Thus, we assume the expected lifetime of a thread to be relatively long. **Unpredictable computation and communication.** Requests for the execution of one or more threads may arrive arbitrarily. No a-priori assumption is made about the memory access patterns of the threads. No previous knowledge is assumed about the relative amounts of communication and computation used by the applications.

Related Work

Most DSM systems do not use memory access pattern information to improve locality of data references. Instead, they concentrate on smart consistency schemes using various replication methods. Few systems attempt to do beyond reducing the consistency, including *Amber* [3], *Locust* [4], *MCRL* [8], *Olden* [1], and *COOL* [2]. Most of these are not fully transparent. A comprehensive comparison is given in the full paper [12].

2 Thread Scheduling

Since it is essential that a DSM system be adaptive to dynamic changes in the environment and to varying needs of the applications, one may use thread migration for sharing the load among the nodes in the system. However, improper placement of threads results in high communication overhead. Therefore, the threads that are chosen to migrate when the load balancing policy is enforced must be selected with care, so as to minimize the resulting remote memory accesses.

In this section we present a general method that can be used by a DSM system for selecting threads that will migrate to another host, while keeping the expected communication to a minimum. We assume that some external module decides that the migration of n threads from one host to another is to take place; this module then 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 initiated the migration. Hence, the load balancing algorithm (which is described elsewhere [10]) is not described here; in fact, the RAHM can be incorporated in systems which employ any standard load sharing/balancing algorithm.

The thread selection module is thus concerned only with the problem of optimizing for maximal locality. It collects information concerning remote page accesses, and in this way "learns" the memory access patterns of the threads. However, the information about local accesses is not recorded. Thus, the knowledge about the memory access patterns is incomplete and incorrect decisions may be made. What makes things interesting is that these "poor" decisions can then be utilized for future improvements.

If, for example, all the threads which frequently access the same page occupy the same host, no information is (initially) recorded. Some of these threads may therefore be selected as candidates for migration to another host, a poor decision if the memory consistency is strong and page duplicates are not allowed at this point, or if they are allowed but require rapid memory synchronizations. At the same time, information about the page will also become available, making it possible to correct the decision and improve future decisions based on this information. We proceed in the following subsections to describe the details of this method.

2.1 Choosing Threads for Migration

The thread selection module 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 for several reasons. It may have started to perform some location-dependent activities, or be waiting for some synchronization event. Finally, if it migrated recently in an attempt to resolve a page ping-pong, additional migration may cause a *thread ping-pong* instead. This issue is explained in detail in Section 3.

Once the Migration Server determines which threads can migrate, it selects n such threads whose migration is expected to minimize the number of remote DSM references for both the source and the destination hosts. Let L denote the local host and R denote 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 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 with 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 to minimize the expected number of accesses to these pages by the remaining threads. If the threads which remain in L do not use 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 which used the page recently. Sometimes, however, it is possible to know that a thread once used 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. By the principle of locality, the threads that used the page (according to the history) are assumed to continue using it.

These requirements are clearly in conflict, necessitating a tradeoff. Our solution is to let the thread selection algorithm estimate the improvement that can be obtained by migrating each individual 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 memory access pattern of the candidate thread and the impact of migration on the other threads.

If there is no information concerning remote accesses (i.e., at 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 [5]). Accordingly, successively spawned threads are preferred in such cases.

2.2 Page Information.

On each host the local history of remote-references is collected using data provided by the DSM mechanism. As explained above, 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 the id of the accessing thread. This information is collected into three different types of stores, called *histories*: *page history*, *thread access history*, and *reduced history*.

The reason for splitting into different storage types is three-fold. As we will see, this split simplifies the retrieval of the necessary information during decision time. It also protects the system from unstable behavior that may be caused by communication bursts. In such cases, care must be taken to limit the amount of memory that is used to store the histories, while still keeping enough information even when memory constraints do not allow storage of the full history. In addition, out-of-date information should be discarded, lest wrong decisions be made due to past events that no longer characterize the current behavior. Finally, keeping the histories in different stores also helps dealing with page duplicates in systems employing memory models of weak consistency. The details follow.

History of a page is maintained in a certain host for each page P that visits the host, unless P leaves the host and does not return for "a very 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:

– 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; when a thread leaves the host its access history goes with it. The access history 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 (although no remote accesses can be recorded), and the corresponding information is stored in the thread's *reduced access history*, as described below. If access information is discarded because of space limitations, then its "summary" is stored in the thread's reduced access history.

Reduced access history of a thread. An additional data structure, called *reduced access history*, is maintained for each thread. It contains *old* data about remote accesses that occurred a long time ago, but are still relevant because the corresponding page became local, and *reduced-size* data about accesses that occured 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 known to be referenced by the thread. The entry describing the page includes:

- The frequency of remote accesses of the thread to the page, as calculated from the full access history (just before it was discarded), and
- the timestamp of the thread's most recent access to the page (if it has already been deleted from the full history).

Information concerning pages that became local (i.e., reside at the same host as the thread) remains in the reduced history forever, or, in the case where the pages cease to be local (due to either page or thread migration), for T_{epoch} time units.

Migration of History The history of each page is maintained locally at each host visited by the page. This promotes quick and easy page migration, as the history does not travel with 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 made by its own migration module.

RAHM **and Memory Behavior** Since the RAHM is independent of the memory model we somehow need to determine how local histories are handled when pages are duplicated, synchronized, discarded, or simply migrated. Fortunately, the division into thread and page histories makes it possible to handle everything in a natural way, and serves to clarify which module is responsible for which actions.

The page history is maintained for the local copy of the page only, and is never synchronized with other histories of copies of this page that are maintained elsewhere. If the page copy moves to another machine, or is discarded, the local history is stored for some time, waiting for another copy of the page. If such a copy fails to arrive, the history is eventually deleted.

As will be explained in Section 3, if the local copy of a page is frequently accessed by remote threads, the threads will sometimes be redistributed in order to improve locality. However, it is left for the DSM to decide whether requests to access the page or requests for its copies are sent to this host or another. The DSM also decides (according to the corresponding consistency protocol) whether to move the page to another machine, to make a copy of it, or to synchronize copies. Once again, the RAHM does not intervene in the DSM decisions regarding these events, unless it detects a ping-pong situation to be resolved. This will be explained in Section 3.

In Section 3.3 we elaborate further on the relationship between the RAHM and the memory behavior, and further explain why the page histories for copies that are maintained on different hosts are never synchronized.

2.3 Thread Selection

We now describe the general method used to select n threads that will migrate from host L to host R. Choosing the best set of n threads is infeasible because of the high computation overhead and incomplete information. Therefore, we suggest a greedy strategy: the best thread is chosen according to some heuristic value calculated at the onset of each thread; the heuristic value for all other threads is then revised to take into account the changes in the memory access 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 L (where $m \leq n$), migrating only some of them would be unwise. Therefore, the heuristic function must take into account both the memory access pattern and the number of threads that should be chosen. We suggest the following generic heuristic function, in which 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 the factors described below.

The contribution of a page to the heuristic value of a thread increases if the page is used by threads on R. This is because the migration of the additional thread to R will bring together all the threads that are using this page, thus improving the locality of accesses to it. For the same reason, any other thread which uses this page and was already selected for migration should increase the contribution. In addition, the contribution correlates with the frequency at which the currently examined thread accesses this page, because the locality of accesses is more important for frequently accessed pages.

The next factor, which is more subtle, 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 is immobile or because there are too many threads in L that use this page) should decrease the contribution. When the additive heuristic value becomes negative, the candidate thread will not migrate, thus remaining at the same host with the other threads in the set.

The contribution of a local page is calculated using exactly the same method, according to the information contained in the reduced history. Of course, this information may not be available for some local pages, or may be available for only some of the threads that are using a certain local page. This raises a question concerning the quality of decisions that can be expected from the thread selection algorithm. We discuss this question below.

2.4 Correctness of the Migration Module Decisions

As already mentioned, incorrect decisions concerning migration are inevitable. However, as a consequence of each such wrong decision, more information will become available, because the accesses which have become remote will then be recorded in the histories. Thus, the next time one of the involved hosts transfers its threads, it will do so with improved decision making capability.

The global effect is an initial unstable period when the threads migrate relatively frequently and the system is busy collecting remote access data. Ideally, after some time, the system knows enough about the memory access pattern, so that a series of correct decisions bring it to a stable state in which the locality of page references is optimized with respect to the heuristic value calculations.

Note, however, that in the above scheme the thread selection module is activated externally. For example, it might be invoked only when serious load imbalance is detected. This introduces a new problem: ping-pong situations which are caused by incorrect decisions may appear even when the load is perfectly balanced. Since ping-pong causes high overhead, an additional mechanism for prompt treatment of it is required.

3 Ping-Pong Treatment

As explained in Section 2.4, the optimal (or suboptimal) locality of data accesses cannot be achieved merely 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 run-time. Thus, our algorithm for the selection of migrating threads is bound to make mistakes, which reveal themselves in ping-pong situations. We thus suggest an additional mechanism, the *ping-pong treatment* mechanism, whose role is to *detect* and *correct* wrong decisions made by the thread scheduler.

When a ping-pong situation is detected, the ping-pong treatment module uses the corresponding page history (for the copy which resides in the host sending the alarm) to determine which hosts take part in this ping-pong, i.e., which copies of the application have threads that are rapidly accessing (the same copy of) the page. Then it initiates a protocol that determines which host can accommodate all the threads involved in the ping-pong. If such a host is found, all the threads involved migrate to it. The threads access histories are modified in order to avoid repeating those bad decisions which resulted in the ping-pong. If no such host is found, then other mechanisms are applied in order to reduce overhead and stabilize the system (see full version [12]).

3.1 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 exits of P to/from the host exceeds a certain threshold, then the module checks for a ping-pong situation according to the following general criteria.

Informally, a ping-pong is detected when the following two conditions are met. First, local threads attempt to access the page a short time after it leaves the host, indicating frequent access. Second, the page leaves the host a short time after it arrived, indicating frequent access by remote threads. Obviously, there are many ways to formalize these conditions. In the full paper we describe one of them and its implementation [12].

3.2 Determining Where to Migrate

The objective is to select a target host to which all the threads participating in a detected ping-pong can be transferred. Unfortunately, the selection of the target host cannot be made locally at a host that detects a ping-pong, because the participating hosts have different (local) information and may thus make different decisions regarding the same ping-pong. Furthermore, a host might have incomplete or out-of-date information about the other participants; it also does not know how many threads at the other hosts are using the page. Finally, more than a single page may be involved in a ping-pong, and selecting the same host as the target for all them would adversely affect the load balance. Ping-pongs are thus handled in a centralized manner by the *Ping-Pong Server*, which is in charge of collecting all the necessary information about each ping-pong, resolving them one at a time.

Each host contains a ping-pong treatment module called the *Ping-Pong Client*, which initiates a ping-pong resolution protocol when it detects a ping-pong situation, or when it receives a ping-pong message from the Ping-Pong

Server. When the ping-pong protocol for a page (copy) P is initiated, the Ping-Pong Client 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 may be communicating with only some of the other hosts involved in the ping-pong, and thus the list of participants that is locally available is not necessarily complete. The Ping-Pong Client then sends the Ping-Pong Server a message containing the number of local threads that are using P, a flag indicating whether or not they are migratable, and a list of the hosts known to be participating in the ping-pong.

When the Ping-Pong Server gets the first message concerning a detected ping-pong situation, it sends a query to each host that is mentioned in the message. The Ping-Pong Server continues collecting information until up-to-date information from all the involved hosts has been received. If the number of threads involved is below a certain threshold, then the server selects the target host whose load is low enough to accommodate all of the threads involved. Otherwise, the server may decide to abort the ping-pong treatment in an attempt to keep the load balanced and avoid high migration overhead.

3.3 Relation to the Memory Consistency Revisited

As explained in Section 2.2, the decisions as to which copy of which page is accessed, modified, copied, discarded, etc., are left to the DSM, which implements the desired memory consistency. Since two different histories for two local copies of the same page are never synchronized, these decisions also determine which page copy's history is chosen for modification. This approach, in which the histories are not synchronized, was originally planned to simplify information retrieval, to avoid communication overhead, and to avoid clock synchronization and event ordering issues. As it turned out, this approach is what also enabled the system to adapt to different memory consistency protocols.

Consider, for instance, the case of four hosts with two copies of a single page. One copy is accessed by all the threads on hosts A and B, resulting in a ping-pong between them, and the second copy is accessed by all the threads on hosts C and D, also resulting in a ping-pong. This situation can be a result of several memory configurations and behaviors. It may occur when there is a true sharing of the threads on A and B, a true sharing of those on C and D, and both data items are falsely shared on the same page.

The aforementioned situation could occur if the required memory behavior guarantees *Coherence* (accesses to each variable are globally seen in the same order), but there is no strong connection between the operations on different variables. Suppose, more specifically, that the threads at A and B frequently access the same variable a, and the threads at C and D frequently access another variable b, and a and b share the same page. The DSM decides whether to send requests for the page from A to B and vice versa, or to send them between C and D only; i.e., the underlying shared memory implementation is "smart enough" to propagate modifications only to the places where they are required.

In order to resolve the ping-pong between A and B, only the page history on these hosts should be examined, and likewise for the ping-pong between C and D. Since no communication exists between the copies, the Ping-Pong Server does not try to synchronize all the local histories of the page, but only those of the hosts which take part in the ping-pong.

The memory behavior in the above example was designed to support efficient execution of at least two independent sets of threads which access the page simultaneously at remote locations. However, if *both* page histories were to be consulted, or synchronized at times, the Ping-Pong Server would then try to transfer all the threads from all four hosts to the same location, perhaps exceeding this host's capacity and interfering with the load balancer.

The current approach allows the RAHM to coexist efficiently with different memory behaviors. Of course, if in the above example a machine were hosting threads which access both variables a and b, then the joint page history would mention both of them, and the method, flexible as it is, could not adapt. Further work is required in this direction.

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. We refer the reader to the full version of the paper [12] where we describe mechanisms that can be applied in order to keep the system stable.

4 Implementation and Performance Evaluation

In order to prove that the methods presented in the previous sections can dramatically improve the performance of a DSM system, we used MILLIPEDE – a work-frame for adaptive distributed computation on networks of personal workstations [7, 6, 9].[1] MILLIPEDE includes a user-level implementation of a DSM and thread-migration mechanisms [10]; it provides a convenient way to execute multi-threaded DSM applications with transparent page- and thread- migration. Based on the ideas described above, MILLIPEDE provides a built-in optimization for data locality.

For lack of space we omit in this extended abstract the details of how MILLIPEDE implements the general method from Sections 2 and 3, they can be found in the full paper [12]. We proceed to present preliminary results of our experiments with the MILLIPEDE system, focusing on the effects of the RAHM on the locality of memory references. A full account of our measurements can be found in the full version [12].

In general, testing the MILLIPEDE system for just one parameter at a time is very hard, sometimes impossible. MILLIPEDE is designed to be very flexible, promptly responding to environmental changes. Thus, when one parameter reaches (or is fixed to) some value that causes the system to handle the situation

[1] Currently MILLIPEDE is implemented using the Windows-NT operating system; see also `www.cs.technion.ac.il/Labs/Millipede`

inefficiently, other parameters adapt their behavior accordingly. In the full paper we give several examples of this phenomenon.

The Traveling Salesman Problem (TSP) is an example of an NP-hard optimization problem in graph theory. The algorithm we use is not the most efficient one can find for this problem. It was selected for its suitability as a simple "platform" for testing the way RAHM handles various memory access patterns.

The basic 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. In the parallel algorithm work is divided among threads, so that the subtree of each node $0 \rightarrow i$ is searched by $n-2$ threads.

We compared three different variants of the program. In the first variation, denoted *NO-FS*, false sharing is avoided by allocating more memory than necessary, i.e., the allocations are padded to fit precisely into a page. In the two additional variants, the $n-2$ threads that search the paths starting with $0 \rightarrow i$ store their private data on the same page. The variant called *FS* uses no optimizations, whereas the one called *OPTIMIZED-FS*, uses optimizations for data locality by enabling the RAHM.

As one can see in Figure 1, 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 RAHM becomes higher when the number of hosts increases, because there are more false sharing situations to resolve.

One may ask: why use optimization when it is so simple to avoid false sharing by padding? There are two answers: first, the RAHM also solves the communication caused by true data sharing (which cannot be avoided by padding); and second, in many cases it is impossible to increase the size of all allocations to the size of a virtual page simply because this will overflow the working set.

Table 4 summarizes the results of running the TSP algorithm on six machines with a varying number k of threads sharing the *best-result-until-now* variable (thus contending for the same page). Since the load-balancing mechanism does not know which threads share the same variable, it sends them elsewhere in a non-deterministic fashion (which depends on which under-loaded machine is faster in asking for work). The result is a random placement of the threads in the hosts, which may represent arbitrary initial conditions for many applications with different memory access patterns.

The table shows a dramatic reduction in the network traffic when the RAHM is activated: the number of DSM-related messages, which reflects on the DSM *miss-ratio* (ratio of remote to total number of accesses), drops by a factor of 30 to 40! If the TSP problem involved more cities, this factor would grow exponentially. Note that the number of extra messages that are added by the RAHM itself is negligible compared to the improvement in the number of DSM-related messages.

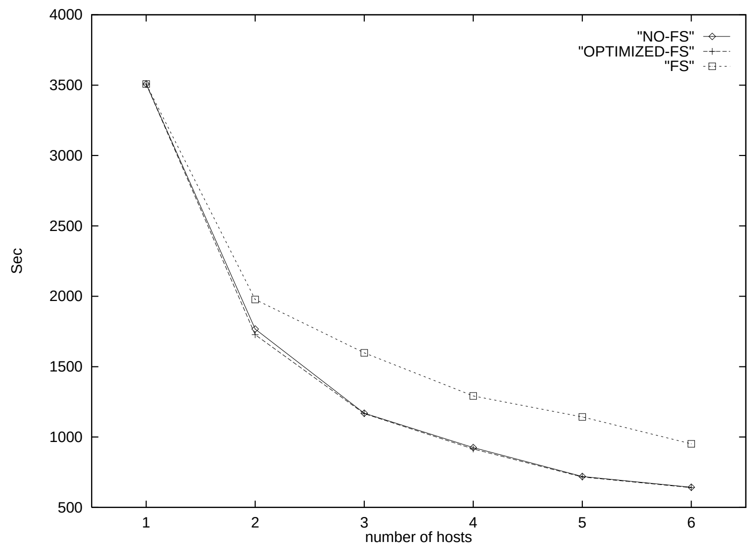

Fig. 1. The effect of the locality optimizations using the RAHM for the TSP. In the NO-FS case false sharing is avoided by aligning all allocations to page size. In the other two cases each page is used by two threads: in FS no optimizations are used, and in OPTIMIZED-FS the history mechanism is enabled.

5 Conclusions and Open issues

The RAHM – a dynamically adaptive, transparent mechanism that can be used to improve decisions concerning thread scheduling in a DSM system was presented. It was shown to have the potential to significantly improve the locality of memory references in such systems. Although the RAHM is described in terms of *pages* and *threads*, the same ideas can be applied in the case of threads which use *Remote Message Invocation* (RMI) in order to access remote *objects*. This will give us a mechanism that will optimize the migration decisions in a distributed environment supporting object oriented programming and concurrent execution.

There are many issues in the paper that clearly require further work. These include the interaction of the RAHM and the load balancing algorithm, the automatic tuning of parameters such as Δ, the ping-pong detection sensitivity, and additional stabilizing mechanisms.

References

1. M.C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proc. of the 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, July 1995.

k	Optimization enabled?	Number of DSM-related messages	Number of ping-pong treatment messages	Number of thread migrations	Execution time
2	Yes	5100	290	68	645
2	No	176120	0	23	1020
3	Yes	4080	279	87	620
3	No	160460	0	32	1514
4	Yes	5060	343	99	690
4	No	155540	0	44	1515
5	Yes	6160	443	139	700
5	No	162505	0	55	1442

Table 1. This table contains statistics regarding applying the RAHM to the TSP application with false sharing for different k (number of threads contending for a page). Random placement of the threads in the hosts represents arbitrary initial conditions for many applications with different memory access patterns. In all cases tested, applying locality optimizations dramatically decreases the number of DSM messages (page lookup and transfer). The table shows that the added overhead imposed by the ping-pong treatment mechanism and the increased number of thread migrations is negligible.

2. R. Chandra, A. Gupta, and J.L. Hennessy. Data Locality and Load Balancing in COOL. In *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 249–259, May 1993.
3. J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP)*, pages 147–158, December 1989.
4. T. Chiueh and M. Verma. A compiler-directed distributed shared memory system. In *Proc. of the Int'l Conf. on Supercomputing*, July 1995.
5. A. Dubrovski, R. Friedman, and A. Schuster. Load Balancing in Distributed Shared Memory Systems. *International Journal of Applied Software Technology*, 3, March 1998.
6. R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. Millipede: Easy Parallel Programming in Available Distributed Environments. *Software: Practice & Experience*, 27(8):929–965, August 1997. Preliminary version appeared in Proc. Euro–Par, Lyon, August 1996, pp. 84–87.
7. T. Hershman. Millipede Virtual Parallel Machine. *BYTE*, pages 3–4, May 1998.
8. W.C. Hsieh, M.F. Kaashoek, and W.E. Weihl. Dynamic Computation Migration in DSM systems. In *Proc. of Supercomputing*, November 1996.
9. A. Itzkovitz, A. Schuster, and L. Shalev. Millipede: Supporting Multiple Programming Paradigms on Top of a Single Virtual Parallel Machine. In *Proc. HIPS Workshop*, Geneve, April 1997.
10. A. Itzkovitz, A. Schuster, and L. Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *To appear in the Journal of Systems and Software*, 1998. (Also: Technion TR LPCR-#9603).
11. J. Naor and L. Zosin. A 2-Approximation Algorithm for the Directed Multiway Cut Problem. Manuscript (naor@cs.technion.ac.il).
12. A. Schuster and L. Shalev. Using Remote Access Histories for Thread Scheduling in Distributed Shared Memory Systems. In *12th Intl. Symp. on Distributed Computing*, Andros, September 1998. A Technion/LPCR TR-#9701, January 1997.