MILLEPEDE: Easy Parallel Programming in Available Distributed Environments *

Roy Friedman† Maxim Goldin† Ayal Itzkovitz† Assaf Schuster†

Abstract

MILLEPEDE is a project aimed at developing a distributed shared memory environment for parallel programming. A major goal of this project is to support easy-to-grasp parallel programming languages, that will also make it straightforward to parallelize existing code. Other targets are portability and availability of both the user programs (hence the shared memory support and the C-like parallel language PARC) and the system itself (which is thus implemented in user-level and using standard operating systems). MILLEPEDE supports multi-parallel-programming, and is friendly and considerate to interactive users. Locality of memory references which implies efficiency and speedups is maintained by MILLEPEDE using page and thread migration through which dynamic load-balancing and weak memory are implemented.

---

*The MILLEPEDE project is supported by Intel Academic Relations Grant, by Microsoft R&D group, and by a grant from the Israeli Ministry of Industry and Trade.
†Department of Computer Science, The Technion, Haifa, 32000, Israel.
‡Department of Computer Science, Cornell University, Ithaca, NY 14850. Much of this work was done while this author was still with the Department of Computer Science at the Technion.
1 Introduction

Moderately local distributed computing environments are commonly available nowadays to almost everybody. Classes of high performance workstations are scattered in every building of every educational institution. The computing structure of most organizations, whether companies or administrative offices, is already distributed. As an example, the main building of any of the major banks nowadays include several thousands of PCs, all interconnected by some backbone communication network. Even during working hours, most CPU cycles of these machines are wasted on waiting for interactive input, while during weekends and nights all these processors are shut-off electricity. On the other hand, most organizations have expensive, central mainframe computers that are aimed at performing tasks that require the power of more than a single workstation, e.g., sorting all ATM transactions during the previous day. Such powerful mainframes are usually loaded during non-working hours and weekends. We claim here that this redundancy can be avoided.

We suggest here parallel computing on distributed environments (ParCoDE) as the solution for many organizations that need high computing power at a relatively low cost. In fact, powerful computing environments can commonly be composed by increased utilization of the existing equipment, i.e., all the workstations in the local area network. Furthermore, we conjecture here that ParCoDE is unavoidable: once research and development in this field take the right direction, it will be a matter of just a few years until it becomes a standard, with high support from the various vendors. Supporting tools and run-time systems for ParCoDE will be as standard as the current sequential compilers. It will be a common procedure for a common programmer to write a parallel program and activate it from his local workstation. He will be guaranteed that his application will not interrupt the interactive work of other users on one hand, and on the other hand, it will utilize free CPU cycles and memory pages in any of the machines on his local network. As a result, during non-working hours the program will be using a very powerful computing environment. Clearly ParCoDE is a very cheap solution in obtaining such an environment.

The realization of the above utopia involves several changes in the current development trends of ParCoDEs. Most current systems focus on obtaining speedups when using as many processors as possible in the distributed environment. Although this is essential for recognition and publication, it usually implies the following decisions during the development.

Programming Language - Current programming languages tend to be complicated, and specially designed to be used in a specific system. Consequently, the common programmer must be explicitly trained for a specific system and programming style, which costs a lot of time and money. Moreover, the skills obtained while using one system are likely to be useless in working on other systems, making the time and effort invested in one system a waste when moving to a newer system. Also because many parallel languages do not extend the serial programming style in a natural way, the task of parallelizing existing serial code is usually quite difficult.

Operating System - Most existing ParCoDEs change the Kernel of the operating system. This incorporate new bugs into the kernel, and make the system less standard and less portable. In particular, if the operating system on which the system was developed becomes obsolete, the entire ParCoDE system becomes useless.

Message Passing vs. Shared Memory - Higher speedup is easier to obtain in a message passing environment than with shared memory, so most systems adapt the message passing paradigm. However, message passing is harder on the programmer. He is responsible for
the timing a message is sent, and for spawning threads or event handlers for incoming messages. The programmer might as well include some control messages for the implementation’s sake. Message passing is also harder to adapt when the system scales up, is impossible to efficiently program in heterogeneous systems when the relative power of the machines is not known in advance, and extra work is required when the programs are to be ported to new machines.

**Friendliness to Native Users** - Since the assumption is that clusters of workstations are mainly occupied by interactive users, the PARCoDE should not interrupt this activity. Despite its efficiency in using spare resources, it should terminate its presence on any machine in a fault-tolerant and transparent fashion when the owner of that machine claims its resources. If several applications are using PARCoDE they should all be able to maturely share the available resources. Most current systems, however, are dedicated to the parallel activity when activated, and are not sensitive to external load effects.

Thus, in order to increase the popularity of PARCoDEs, the emphasis of these systems should shift from speedup to other issues like availability, friendliness, ease-of-programming, and portability. In Millepede, we try to extend this strategy to distributed environments, where more care should be given to the high communication overhead. We thus try to maximize portability and ease-of-programming by supporting shared memory on one hand, and on the other hand, supply the programmer with special tools and techniques that enable him to help the system in overcoming the communication bottlenecks. In addition, for the sake of portability, we attempt to realize the entire system in user level including the management of the shared memory and the load balancing.

Our idea is to build a system which is general enough to support most existing parallel programming languages in an efficient way, such that the only thing that is needed in order to use a specific programming language is a cross-compiler, and usually only a preprocessor, from the source language to the Millepede interface.

In particular, we support the PARC programming language, which is a very natural extension of C to parallel programming. PARC adds few constructs to the C language, so that multiple activities can be created in run-time and execute concurrently. The use of the constructs is simple and the number of activities and their context can be determined at run-time. PARC assumes shared memory and uses the scoping rules of C. Shared variables should be declared in a higher scope level than the one in which they are used as shared. PARC also adds a barrier-like synchronization instruction and some other parallel counterparts to C constructs.

PARC is extremely useful for fast and easy parallelization of existing serial code. It was successfully implemented on the Makbilan parallel computer. We report here on the first implementation which can run PARC programs in a distributed environment.

In order to be efficient, the implementation of Millepede must deal with two fundamental issues in parallel execution: support for shared memory and dynamic load balancing. Shared memory in distributed environment would hide the network presence from the programmer, providing him with a virtual multi-processors machine, while taking care of memory consistency and data distribution among the involved workstations. Consistency can also be reduced for enhancing performance, while incorporating some changes in the programming model, sometimes forcing the programmer to include certain synchronization commands in his work.

While data is distributed among workstations, parallel activities that use the same memory location might cause massive network traffic and slow down their workstations, waiting for incoming messages. Load balancing algorithms, and more precisely, dynamic load balancing are responsible for moving a page of shared memory from a workstation to another, or moving a running activity...
from a workstation to another while keeping it running, so that the system drive to its optimal state.

One might think of two extreme scenarios where each activity runs on a different workstation. In that case, the CPU load would be very small but the network traffic would be very high and a large portion of the time, the CPUs will stall, waiting for memory to synchronize. The other scenario, is having all activities run on the same workstation. In that case there will hardly be any speedup (if at all), since only one activity can run at any given moment. Dynamic load balancing should find the balance between those two extreme scenarios, and try to bring the system to optimum during execution.

Our initial performance measurements are very encouraging, and prove that at least for a large class of programs, our goals of ease-of-use, portability, and availability can be provided while maintaining reasonable performance. Most programs achieve good speedups, while the speedups for programs that have natural parallelization is close to linear. Also, there seems to be a correlation between the problem sizes and the speedups, indicating that when the problem is large enough, the benefits of using our system outweighs the overhead imposed by it. We believe that by further optimizing the implementation, we will be able to achieve even better performance.

The rest of this paper is organized as follows: Section 2 sketches related work and similar systems. Section 3 describes the commands available with Millepede. Section 4 briefly mentions the load balancing algorithms and the structure of the distributed memory manager in our current implementation. Section 5 describes the memory management protocols. Those are also described in some more detail in Appendices A and B. Section 6 gives some benchmarks and performance results. Section 7 indicates of our current development effort towards the next generation Millepede. Finally in Section 8 we give some conclusions.

## 2 Related Work

Many systems that are intended to run parallel programs on a cluster of workstations with support for Distributed Shared Memory (DSM) have been proposed, and are currently in the process of design and development. We refer the reader to [48, 50, 46] for surveys which cover and give some classification of previous work. Here we briefly mention only certain systems which incorporate some of the goodies included in our goals. None of these systems capture all of our targets. The systems are divided into two main categories: Hardware DSMs and Software DSMs.

The Hardware DSM systems implement memory consistency algorithms in hardware, thus achieving better speedups than currently available software DSM systems. They, however, suffer from not being portable, and usually require dedicated hardware and special modifications in the operating system kernel. Several hardware DSM systems were published (APRIL [3], MemNet [26], Plus); the most known is the Stanford DASH [42, 41] which implemented directory-based algorithms for supporting release and lazy release consistencies.

Software DSM systems have been introduced since the mid eighties. Ivy [43] employed single-writer/multiple-reader mechanism together with distributed directory algorithms for efficiency. Ivy brought the idea of distributed memory that can be coherent among cluster of workstations using the invalidate-based protocol. Clouds [25] added support for object-oriented programming language, distributing objects over clusters of workstations. Clouds used invalidate-based algorithms and implemented sequential consistency. Panda [9, 10], Emerald [36] and Amber [22] distributed objects in similar way as Clouds, although the programmer could control object migration. The basic Unit of migration was an object, that contained all of its private data. Restricted use of data replications was supported. Orca [13] ran above the Amoeba operating system, supporting update-based and invalidate-based protocols. Mether [47] fought the “ping pong” effect and introduced good results.
on part of the benchmarks. *Mirage* [30] was implemented on top of SunOS. This might have given a large advantage when it was publicly released. *Munin* [19, 20] introduced a paged-based system that supports multiple consistencies in a single program. *Munin* implemented release consistency and proved it to be efficient for many kinds of programs, where there are simultaneous writers to same page in different locations (false sharing). *TreadMarks* [38] implemented improved version of the release consistency, namely lazy-release consistency that handles multiple updates in a single message. *TreadMarks* runs on a standard cluster of SGI Unix workstations, that again, like *Mirage*, gave it a large advantage. *Midway* [16, 15] implemented entry-consistency, using a massive compiler work, enclosing parallel conflicting set of memory references under same labels (like lock names). This ensured that those memory references won’t execute concurrently. *Midway* is bypassing the false-sharing problem of a paged-based system by letting the programmer decide what limitation he would give of his program’s execution. Although *Midway* got some good speedups, it is unlikely that systems which overload the programmer with such decisions will become popular. *Quarks* [21] is an ongoing project employing user-level DSM above standard operating systems (like SunOS, HP BSD and Mach). *Quarks* includes support for multiple protocols within an application on a page granularity. *Typhoon* implemented a user-level shared-memory system, paying by lots of restrictions [51]. *Adsmith* gives some support for heterogeneous computing, as it was built on top of the PVM communication layer [44], however references to shared variables are made via special function calls. The MIT *Phish* system [17] is currently operational on the CM-5 and SunOS workstations. The programming model is via *Cilk*, a language that is an extension of C, and is, however, more restrictive than PARC which is supported by *Millepede*. Finally, the Berkeley *NOW* project is a user-level implementation on a network of Unix machines [33, 8]. Programming in *NOW* is by the *Split-C* [23] programming language, while a massive compiler work is done to catch stores and indirect branches to find remote memory references. *NOW* claim that with intensive compiler optimization they can reduce the overhead on memory references to 3%-7%. In contrast, in *Millepede* we put the locality principle in the center of our concern so that no overhead is allowed with local references. Also, despite their big effort to make minimal changes in the kernel, *NOW* is still making some modifications there, which directly hurt portability.

Other systems for DSM include *Agora*, *Choices*, *Domain*, *Merial*, *Shiva*, *DiSOM* [49], and others. University of Utah is extending the *Mach* operating system (Flex-Mach project), for embedding DSM capabilities as operating system service.

In our research we attempt to implement a full user-level distributed shared memory, while integrating various reduced memory consistency strategies, and several load sharing algorithms. We try to avoid imposing tedious restrictions on the programmer. We investigate the applicability of reduced memory consistency and give some performance evaluations. Various related policies were proposed in the past, including sequential (or, strong) consistency [2], weak consistency [35, 43], release consistency [18, 20], casual consistency [4, 6], lazy release consistency [37], and entry consistency [15]. Other general applied and theoretical studies abound, see for example [52, 28, 32, 54, 31, 12, 11].

### 3 Millepede Constructs

*Millepede* supports several constructs for creating parallel activities, sharing information among them, synchronizing parallel activities, and influencing the control of parallel activities. Most of these constructs were taken from PARC and generalize similar constructs of C. However, some of the synchronization constructs that appear in other parallel programming languages but were missing from PARC, and we viewed them as necessary, were also added. We now list these constructs:
3.1 Creating Parallel activities

Millepede supports a construct \texttt{par_exec} for creating parallel activities. Its parameters are a list of functions to be executed as parallel activities (in parallel), and lists of parameters to be passed to these functions. A parallel activity \( a \) that invoked a call that was translated into \texttt{par_exec} is said to be the parent activity of the set if activities \( S \) that were initiated by \texttt{par_exec}; the activities in \( S \) are the children activities of \( a \), and are siblings of each other. In the implementation of Millepede, a parent activity is always blocked until all of its children activities have terminated.

3.2 Sharing Data Between Parallel Activities

In order to allow meaningful parallel programs, Millepede must support a paradigm for sharing and exchanging information and data between the parallel activities. The paradigm that we have chosen for Millepede is of a (distributed) shared memory. Millepede supports a construct for making a specific variable shared between a given set of parallel activities. In particular, this construct allows to add new parallel activities to the list of activities that share a given variable. This is vital for implementing parallel programming languages like ParC in which the variables are shared according to C’s scoping rules [14].

3.3 Synchronization Between Parallel Activities

Synchronization Points

Synchronization may be required in both the computational activity and the memory updates. Computational activity is controlled by the \texttt{sync} construct which blocks an activity until all of its sibling activities have either called \texttt{sync} themselves or completed execution. Synchronization of memory updates is described in Section 5.2.

Semaphores

Millepede supports operations on semaphores, implementing Dijkstra’s P and V operations. These constructs include: \texttt{semaphore_init}, for initializing a semaphore, \texttt{semaphore_wait}, which serves as P, and \texttt{semaphore_signal}, which serves as V. Semaphore operations are very useful for implementing mutual exclusion for accessing complex shared data structures, or to perform complex manipulations, e.g., read-modify-write, on shared variables. Typically, before accessing such a data structure, \texttt{semaphore_wait} must be called, and after finishing the manipulation of this data structure, \texttt{semaphore_signal} must be called. This ensures that only one activity can access the critical section at any given time, and that if some activity wishes to access the critical section, it will be granted permission to do so.

More complex access patterns, such as multi-reader single-writer, can be also written using semaphores.

Atomic Access to Data

The case in which data needs to be incremented atomically is very common in parallel programs. Although this can be implemented with semaphores, direct implementation of this commonly used memory access pattern is much more efficient than using semaphores. We have chosen to support the \texttt{faa} (fetch and add) construct in Millepede, which atomically increments a specified variable by some given amount and returns this variable’s old value. In later versions of the project, the \texttt{tst}
(test and set) construct will be also implemented providing the functionality of entering a critical section only if no one is there. It is similar to a non-blocking semaphore V operation.

### 3.4 Control statements

**Millepede** supports several constructs for controlling the flow of parallel programs, which are extensions of similar constructs in C, and are the same as those used in ParC. These constructs include: `pcontinue` which terminates the activity that invokes it and is therefore similar to `continue` in C, `pbreak` which terminates the activity that invokes it and all of its sibling activities, and is therefore similar to `break` in C, and `preturn` which terminates all activities within the last function that was called in the program and returns a specified value, and is therefore similar to `return` in C.

### 4 Load Sharing and Memory Management

In this section we briefly describe some of the implementation details in our system. We refer the reader to [34] for further details.

#### 4.1 Load Balancing

Our goal is to support several load balancing protocols, each with slightly different optimizations and strategies. We have initially implemented two simple protocols for load balancing: a *round-robin* scheme and a *central queue* scheme. In the round robin scheme, new threads are simply assigned to processes in a round-robin manner, regardless of their load. Naturally, this scheme is not expected to behave well. We have chosen to implement it as a basis of comparison with the other schemes we intend to develop.

The central queue scheme works as follows: A queue manager task is initiated on one of the hosts. Then, whenever new activities are created, they are first sent to the queue manager which stores these uninitialized activities in a FIFO queue. Every host has a local load manager, and whenever this load manager notes that the number of **Millepede** activities on that host is lower than some threshold, it sends a request for more activities to the queue manager. When the queue manager receives such requests, it removes the first activities from the queue and send them to the host that initiated the request. If there are no activities in the queue when a request arrives, then this request is buffered until some activity is sent to the queue. If there are several pending requests for activities when an activity is received by the queue manager, it chooses one of them according to some fair scheme.

This approach is expected to behave a lot better than the round-robin one, since it takes into account the load of the hosts. On the other hand, it is still far from optimal, since it does not take into account the load on the host which is not the result of **Millepede** activities. Moreover, this approach does not take into account the number of remote memory accesses caused by placing a thread in one machine and not in the other. We are currently developing schemes that will take these considerations into account.

#### 4.2 The Distributed Memory Manager

Our goal here was to avoid centralization of the memory management mechanism (For details about user-level implementation of the memory manager in Mach we refer the reader to [39]). To this end we have implemented a distributed memory manager as follows. The entire shared memory region
is split into equal segments according to the maximal number of hosts which are expected to take part in the program execution. Each host carry a manager which is in charge of one particular part of the total memory.

Suppose some attempt occurs to access data that is not available locally. Suppose this data item is located at an address controlled by the memory manager that is executing at a certain host. Then the access request must be sent to the host holding the data item, and the memory manager located at that host will handle the resulting activity. This design achieves the desired effect: the load resulting from external memory references is split between the processors.

Furthermore, every request for shared memory allocation is supplied from the part of the shared memory segment that is controlled by the local host where the request was issued. In this way, if several requests for memory allocation are issued concurrently, they will be serviced simultaneously, thus achieving increased degree of parallelism. This also serves to enhance parallelism and locality of reference at a later stage, when the activities which issued the requests for additional memory start accessing it. Notice that an activity naturally share the memory it is allocating with its locally spawned descendants (unless they immigrate to some remote machine), thus locality is maintained here as well.

5 Distributed Shared Memory

Since Millepede is designated to run parallel activities of the same application program on different workstations, a true shared memory cannot be used. Instead, Millepede simulates a shared memory model among the parallel activities, known as distributed shared memory. Thus, different parts of the distributed shared memory may be held by different machines with a mechanism which ensures that activities that try to access data which is located on a remote host can do so without noticing that such a remote access occurred. In particular, for reasons of efficiency, in some cases several copies of the same data object may be held by more than one machine, and it is the task of Millepede to maintain these copies coherent.

Previous research in the area of distributed shared memory, both practical [1, 32] and theoretical [12, 45], indicate that there is a tradeoff between the amount of transparency that an implementation of a distributed shared memory presents to the application, known also as the consistency model, and how efficient this implementation can be. Therefore, in Millepede we support several models of consistency in order to allow the programmer to choose between increased transparency (ease of programming) and efficiency.

Millepede is intended to support three different consistency models such that the programmer can choose between them at compile time. These models are strong consistency (also known as sequential consistency [40]), weak consistency, and release consistency.

In the strong consistency model, the fact that the memory is distributed is completely transparent to the application, i.e., it appears to the application as if it was a real shared memory.

More formally, all memory operations must appear to be executed everywhere in the same order, and operations generated by the same machine must appear to be executed in the order in which they were specified in the application program, known also as the program order [27].

The weak model, on the other hand, requires that each process will only view its own operations in program order; by this model, a process is allowed to view operations by other processes in a different order than the program order.

Release consistency is a compromise between strong consistency and weak consistency. In release consistency there are release and acquire operations. The following requirement is placed on top of the requirements of weak consistency: all operations of a certain process that appear in
program order after an acquire operation must be viewed by everyone after this acquire operation, and all operations of this processes that appear in program order before a release operation must be viewed by everyone before this release operation.

The current implementation of Millepede on Mach supports only strong consistency and weak consistency, while the implementation of release consistency is still under way. This implementation employs an invalidation based, page granularity, protocols to implement the various consistency conditions. This approach can be viewed as a generalization of virtual memory. That is, the memory is divided into logical pages, that can be mapped to physical pages in the memory of the processes and be given certain access rights. In this case, a memory access to a page that is present and does not violate the access rights is serviced entirely by the virtual memory mechanism, while any other access attempt causes a trap which is handled by a memory pager. The main difference between this approach and common virtual memory is that in our implementation there can be more than one copy of the same virtual page at the same time, and that missing copies of virtual pages may reside on a different machine, and not necessarily on the disk. The exact behavior of the pager in response to a trap depends on the particular consistency model chosen, and is described below.

The main advantage of the approach we have taken is that local accesses can be done completely by the VMM, and are therefore executed as fast as possible by the corresponding hardware and OS services. The drawback of this approach however, is that remote accesses always incur a trap, which is relatively costly to service. Also, due to the high granularity of this solution, the amount of false sharing is quite high, resulting in many redundant invalidations and traps.

5.1 The Strong Consistency Protocol

In the implementation of strong consistency pages can either be present or non-present, and present pages can either be marked as read only, meaning that only read accesses to these pages are allowed, or as writable, meaning that both reads and writes are allowed to these pages. In order to maintain the required level of consistency, we use the following rule: If there exist a copy of a virtual page that is marked as writable, then this is the only copy of that page. However, we allow several copies of the same virtual page if all of them are marked read-only. Whenever an attempt to access a page \( A \) which is not present on the local machine \( m_i \) is made, the pager on \( m_i \) tries to fetch a copy of \( A \) from one of the other machines that hold it. If the access is a write, then all other owners of copies of \( A \) must invalidate their copies such that \( m_i \) will hold the only copy of this page. \( A \) is then marked as writable in \( m_i \). If the access is a read and the only copy of \( A \) is marked as writable, then a message is sent to the owner of this copy to change its access writes for \( A \) to read-only, which is also the way \( A \) is marked in \( m_i \). Otherwise, the copy of \( A \) which is brought to \( m_i \) is marked as read-only.

A write access to a present but read-only page causes the pager to invalidate all other read-only copies of the same page, and then to change the access rights of the page to writable. All writes to writable copies and all reads from present pages (either read-only or writable) are handled directly by the VMM.

This scheme guarantees that every read will always return the value of the last write to the same address, which obeys the requirements of strong consistency. A more detailed description of the implementation on Mach appears in Appendix A.
5.2 The Weak Consistency Protocol

The weak consistency protocol is similar to the strong consistency protocol presented in Section 5.1, except that in this case we allow to have several read-only copies together with one writable copy of the same page. Thus, an attempt to write to a non writable page $A$ on some machine $m_i$ is serviced in the following way: If a copy of $A$ exists at some other machine $m_j$ and is marked there as writable, then this copy is fetched from $m_j$ after the access rights for $A$ are changed in $m_i$ to read-only. If there is no write only copy of $A$, but it is present as read-only in $m_i$, then it is only necessary to change the access rights for $A$ in $m_i$ to writable. Otherwise, the page is brought from some other machine that has a read-only copy of $A$ and this new copy is marked in $m_i$ as writable.

An attempt to read a non present page $A$ on a machine $m_i$ is served as follows: If there exists a writable copy of $A$ on some other machine $m_j$, then a copy of $A$ is fetched from $m_j$ and is marked as read-only in $m_i$. Otherwise, a copy of $A$ is fetched from one of the other machines which have a copy of $A$, and this new copy is marked as read-only in $m_i$.

All other cases are served by the VMM. Also, in order to guarantee that the read-only copies would be updated about the writes to the writable copy, whenever a writable copy exist, every now and then all read-only copies are invalidated. There are currently several possibilities to do this periodical invalidation. The first scheme is to invalidate the read-only copies after every $k$ updates to the writable copy, where $k$ can be chosen by the programmer. The second scheme is to invalidate the read-only pages every $t$ milliseconds, where $t$ is also chosen by the programmer. The third approach is to use special program statements to initiate such an invalidation. These statements include:

- `total_mem_sync()` - invalidates all read-only copies of all pages that have a writable copy.

- `host_mem_sync()` - invalidates all read-only copies of all pages located in the machine on which this statement is invoked and have a writable copy on some other machine.

- `page_mem_sync(addr)` - invalidate the read-only copies of the page that holds address $addr$ on the host in which this statement is invoked if there is a writable copy of this page somewhere else.

- `total_page_mem_sync(addr1, addr2)` - invalidates all read-only copies of all pages within region $addr1$ to $addr2$ for which a writable copy exists.

- `host_page_mem_sync(addr1, addr2)` - invalidates all read-only copies of all pages within region $addr1$ to $addr2$ on the machine in which this statement is invoked and for which there exists a writable copy somewhere else.

A more detailed description of the implementation on Mach appears in Appendix B.

6 Performance Measurements

Our performance measurements were taken with several benchmark programs, tested on a cluster of i486 machines, running Mach 3.0 MK78 UX39, and connected by a 10 Mbit per second Ethernet. For a given program, the speedups are measured against the serial version of the protocol that solves the same program, and compiled with gcc compiler with all of its possible optimizations (-O2 flag). Also, the results indicated in this paper were obtained with the central queue load balancing scheme, for both the strong and weak consistency protocols.
One of the main observations of our experiments is that our weak consistency protocol did not perform better, and sometimes even worse, than the strong consistency protocol. This came to us as a surprise, at first, but can be explained by the fact that most of the applications we have chosen had very little conflicting accesses to shared memory, so the benefits of weak consistency could not be noticed in them. As for programs in which the weak consistency protocol behaved worse than the strong protocol, we believe that this can be mainly attributed the following reason: The weak consistency protocol allow to execute single operations faster than the strong consistency protocol. However, in the weak consistency protocol it also takes longer for the information to be updated. Thus, although each operation may be faster, many more operations are needed in order to perform the same task, resulting in an overall higher latency. Also, it seems that working in a coarse granularity of 4 KByte pages caused a high degree of false sharing which resulted in redundant invalidations and traps, and consequently increased network traffic. We plan to overcome this problem with our expected release consistency implementation, and by employing a more fine grain update based protocols, in which updates are cheaper, and there is far less false sharing.

6.1 Matrix Multiplication

In this program we try to multiply two $N \times N$ matrices $A$ and $B$, and store the result in an $N \times N$ matrix $C$. The simplest protocol for multiplying matrices is to use the formula $c_{ij} = \sum_{k=1..N} a_{ik} \cdot b_{kj}$. This can be parallelized quite naturally in the following way: Assuming that we have $n$ machines, each machine is assigned $\frac{N}{n}$ rows in input matrix $A$ and $\frac{N}{n}$ rows in the output matrix $C$. To further optimize the performance, each activity stores the results in a segment of memory held by the machine it runs on. When all activities finish, another activity copies the various parts of the result to the real output matrix $C$. Since no two machines ever write to the same memory location, this can be done with very little communication. Thus, in this problem we expect to achieve good speedups. On the other hand, due to the structure of this problem, we do not expect the weak consistency protocol to behave any better than the strong consistency.

Figure 1 shows the running time of multiplying two $N \times N$ matrices, as a function of $N$ on 4 machines with both weak and strong consistency, and compares it to the results of the same problem sizes with the serial program. It can be seen that for small matrices (less than $100 \times 100$), the serial program performs better than the parallel version. This can be explained by the communication overhead of MILLEPEDE and the overhead associated with initiating threads and synchronizing between them. However, as soon as the matrices become larger, the serial program becomes much slower than the parallel one. In fact, as can be seen in Figure 2, for matrices of size $400 \times 400$, the speedup of our system with 4 nodes becomes close to linear.

As expected, the weak consistency protocol does not achieve better performance than the strong consistency protocol. In fact, the weak consistency protocol is even somewhat worse than the strong consistency protocol. Recall, that for this type of problems, our implementation of weak consistency does not have any advantages over strong consistency. On the other hand, in order to guarantee a correct result, the activity that collects the final results must perform explicit memory synchronization, which even adds to the execution time of the entire program.

6.2 Shortest Distance in Graphs

This test program uses Dijkstra’s algorithm to find the shortest path between two vertices in any graph in the case of a finite digraph, whose edges are assigned non-negative lengths [29]. For any vertex $s$ of the input graph $G$, Dijkstra’s algorithm finds the length of the shortest directed path from $s$ to every other node, with computational complexity of $O(|V|^2)$, where $V$ is the set of nodes
Figure 1: Execution time of matrix multiplication with a 4-node system configuration. The graph shows execution time as a function of the input matrix size.

in $G$. The input is a matrix whose $(i, j)$-th element is the length of the directed edge from vertex $i$ to vertex $j$ (if defined, and zero otherwise).

In order to find the shortest directed path between every pair of nodes in $G$, we apply this algorithm to each of the nodes in $G$, or a total of $V$ times. The complexity of this algorithm becomes $O([V]^3)$, and the result is stored in a $[V] \times [V]$ matrix, where the $(i, j)$-th entry stores the shortest distance from vertex $i$ to vertex $j$.

This is trivially parallelized as follows. Each Millepede activity $a_i$ is assigned a node $v_i$ and performs the Dijkstra’s algorithm with $v_i$ playing the role of the source $s$. The only interaction among the participating activities is in sharing the input and output matrices.

Figure 3 presents the runtime of the program as a function of the number of vertices in the graph, while Figure 4 shows the speedups achieved. As in the case of matrices multiplication, we have no benefit of weak memory management usage.

6.3 Linear Equation Solver

In the following test program we have implemented a parallel version of Jacobi’s algorithm for solving a set of linear equations [24]. The goal is to solve a set of linear equations of the form $Ax = b$, where $A$ is an $n \times n$ matrix of real numbers, $b$ is a vector of $n$ real numbers, and $x$ is a vector of size $n$ of the unknowns, which are to be determined by the program. In this algorithm, new values of the elements of $x$ are computed iteratively until the solution converges (i.e., the largest absolute value of the difference

$$
\sum_{j=1}^{n} a_{i,j} x_j - b_i , \quad i = 1, 2, \ldots, n
$$

becomes less than some acceptable predefined value).
We use the notation $x^t_i$ to represent the value of the $i$th component of $x$ computed on the $t$th iteration. Thus, given $x^t_i$, $x^{t+1}_i$ is computed as follows:

$$x^{t+1}_i = \frac{b_i - \sum_{k=1}^{t-1} a_{i,k} x^t_k - \sum_{k=t+1}^n a_{i,k} x^t_k}{a_{i,i}}, \quad i = 1, 2, \ldots, n.$$ 

where the initial approximation is usually assumed to be $x^0_i = 0$ ([24]).

This important problem serves as a test case for many DSM systems ([5, 6, 43]). In Jacobi’s method, one does not make use of the partial results until the iteration is completed, which creates a potential for good speedups of parallel executions.

A concurrent execution of Jacobi’s method is illustrated in Figure 5. The vector $x$ is placed in shared memory so that every host can read and update its values. One of the threads starts the calculation and distributes the work among other processors. Every host receives a local copy of $x$, $A$ and $b$ (all of them reside in the shared memory and are initialized by the root thread). Since the calculation of the $t$-th approximation of $x_i$ depends only on the values of $A$, $b$ and the $(t-1)$-th approximation of $x$, every node can calculate new values for some members of vector $x$ without a need to communicate with other processors. After calculating the new value for $x_i$, each of the threads notifies the root thread that is has finished. The root thread then checks the precision of the result, and if necessary, restart this process all over again, until the required precision is obtained.

The bottleneck of this implementation is in verifying the precision of the result. To check how close we are to the solution every time new values of $x$ are calculated, the main thread has to obtain the new values of all elements of $x$, perform multiplication and create new threads to continue the program execution. This may take a long time, especially since thread creation and termination
Figure 3: Evaluation of Dijkstra’s algorithm with a 2-node system (the left graph) and a 3-node one (the right graph). Here size is the number of vertices in the graph.

Figure 4: Speedups obtained in Dijkstra’s algorithm evaluation. The left and the right graphs present results of the work on nodes 2 and 3 respectively.
Figure 5: Parallel execution of original Jacobi’s algorithm.
incurs a substantial overhead. Moreover, for a system with $n$ linear equations, each host should hold a matrix of size $n^2$. However, the kernel of Mach has a very limited space for storing memory pages that were obtained from the memory manager. When this space is full, the kernel returns a page to the memory manager for every new page it is asked to store. This phenomena is known as threshing. As a result, a great deal of time is needed in order to read all values from an $n \times n$ matrix of real numbers.

To avoid this annoying overhead, we slightly modified the parallel execution of the algorithm, as shown in Figure 6. Now, every host executes a fixed number of internal iterations without precision control between every two consecutive precision checks of the root thread.

Figure 6: Parallel execution of the modified algorithm.

Note that in order to achieve good performance, we still need to find the optimal number of iterations to be performed between any two consecutive precision checks. Moreover, it is unclear how this modification would influence the convergence of the original algorithm. To determine the optimal number of iterations between every precision check, we have conducted several tests, whose results are summarized in Table 1 and Figure 7. In these tests we have solved a system of 500-equations using a 3-node system. We call the iterations between two consecutive precision checks internal iterations and the precision checks external iterations.

Note that despite the fact that the number of internal iterations is the same, the number of external iterations required for convergence while using the weak consistency protocol is far larger.
Table 1: Dependence of external iterations on the number of internal ones in the program execution under the modified Jacobi’s algorithm. The linear system of 500 equations is solved with a 3–node system. The serial solving of the same system of equations converges in 30 iterations.

<table>
<thead>
<tr>
<th>Number of internal iterations</th>
<th>Number of external iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>strong</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 7: Solving of 500–equations linear system with 3–node system by the modified Jacobi’s algorithm. The obtained speedups depend on the number $K$ of the internal iterations.
than the number of external iterations needed while using the strong consistency protocol. As described before, the reason for this is that the values of the various elements of $x$ are updated less frequently in the weak consistency protocol than with the strong consistency, which results in more iterations.

Figure 8 presents the achieved execution time and speedups as a function of the number of equations under examination. The number of internal iterations in the experiments were set to 2, as Figure 7 shows that this number is the best for the program running under the weak memory management, although it is not the optimal one for the strong policy.

6.4 The Traveling Salesperson

The following test program solves the Traveling Salesman Problem (TSP): given a set of cities and prices of flights between every two cities, it is required to find a route that visits each city exactly once and has a minimal cost. More precisely, this program takes as input a directed weighted graph consisting of $N$ nodes, numbered $0, 1, \ldots, N-1$, in which all edges have a positive weight, and produces as output a minimal weight Hamiltonian path in the graph.

The way our solution works is as follows: The program creates $N-1$ parallel activities numbered $1, 2, \ldots, N-1$ respectively. Activity $i$ checks the routes starting with the edge $0 \rightarrow i$. It creates recursively routes starting with this edge by adding nodes that have not yet been visited by the (partial) route. Each time a node is added the weight of the newly formed route $R$ is compared to the minimum of weights of all routes already checked (by all activities). If the weight of $R$ is equal or larger than the minimum, the activity aborts. Otherwise, this activity continues by adding another edge. The execution finishes when all activities either abort or find routes that include all edges.
Table 2: Runtime of the TSP program with 14-node graph as input. Different system configurations and memory consistency are used.

<table>
<thead>
<tr>
<th>System configuration</th>
<th>runtime, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>strong</td>
</tr>
<tr>
<td>fast</td>
<td>300</td>
</tr>
<tr>
<td>slow</td>
<td>380</td>
</tr>
<tr>
<td>fast, fast</td>
<td>210</td>
</tr>
<tr>
<td>fast, slow</td>
<td>210</td>
</tr>
<tr>
<td>slow, fast</td>
<td>240</td>
</tr>
<tr>
<td>fast, fast, slow</td>
<td>180</td>
</tr>
<tr>
<td>slow, fast, fast</td>
<td>155</td>
</tr>
</tbody>
</table>

Table 3: Results of the TSP program execution under strong memory management. Several different inputs (14-node graph) are used.

<table>
<thead>
<tr>
<th>Number of hosts</th>
<th>runtime, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_1</td>
<td>input_2</td>
</tr>
<tr>
<td>1</td>
<td>300</td>
</tr>
<tr>
<td>2</td>
<td>210</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
</tr>
</tbody>
</table>

The global minimum is kept in a shared variable which is updated whenever an activity finds a route that weighs less than the previous minimum. Note that such an update must be protected with semaphores, since it is possible that more than one activity will find a smaller route than the previous minimum roughly at the same time. Also, when using weak consistency, all other copies of the variable which holds the minimum are invalidated each time it is updated by someone, to make sure that all activities have the most recent value of the minimum. Finally, to improve performance, the weight of the minimal route calculated by the local activity is kept in local memory.

The results of our measurements are presented in Tables 2, 3, 4, 5. At the time that these measurements were taken, we only had three operational machines in our system, and one of them was slower by about 10% than the others. It turned out that the order in which these machines appeared in the configuration greatly influenced the performance. The reason for this is that the first processor in the configuration is the one that starts executing activities. Whenever new activities are created, they are assigned to other machines in the order in which they appear in the configuration. Hence, the run time of the program depends on this ordering when the machines are not the same.

Table 2 present results of running this program with the same input but with different configurations. In this table, slow means the slowest machine and fast means one of the faster machines. In Tables 3, 4 and 5, the results for one node and two nodes were obtained using fast machines while the result for three nodes were obtained using the configuration {fast, fast, slow}. 

18
<table>
<thead>
<tr>
<th>Number of hosts</th>
<th>runtime, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>input_1</td>
</tr>
<tr>
<td>1</td>
<td>1250</td>
</tr>
<tr>
<td>2</td>
<td>700</td>
</tr>
<tr>
<td>3</td>
<td>570</td>
</tr>
</tbody>
</table>

Table 4: Results of the TSP program execution under *strong* memory management. Several different inputs (15-node graph) are used.

<table>
<thead>
<tr>
<th>Number of hosts</th>
<th>runtime, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>input_1</td>
</tr>
<tr>
<td>1</td>
<td>4300</td>
</tr>
<tr>
<td>2</td>
<td>2270</td>
</tr>
<tr>
<td>3</td>
<td>2290</td>
</tr>
</tbody>
</table>

Table 5: Results of the TSP program execution under *strong* memory management. Several different inputs (16-node graph) are used.

7 Work in Progress and Future Work

We now describe the work that is being carried, along with our future plans on the way to the next generation *Millepede*.

Operating System, Portability, Heterogeneity

First, we decided to focus our development on a user-level parallel run-time system. Second, we have decided that the next generation *Millepede* will be based on Microsoft’s Windows-NT. Among other advantages, these decisions are meant to achieve maximum portability for commonly available environments. Trying to implement a user-level system which will port to even part of the zoo of Unix domain is impossible. Also, extending our current Mach-based system will necessarily have to deal with the bugs, the inefficiency, and the restricted popularity of Mach (as we experienced them in the current *Millepede*).

A user-level implementation will enable everybody to conveniently install and use our system. Most problems with doing it under NT are found in handling the page tables, the page-access capabilities for the various shared memory consistency protocols, the memory organization with different protocols for different pages, and the thread migration.

Unfortunately, the decision to migrate pages and threads makes hardware parameters important (e.g., page size, endian, processor state). Although this seems to further restrict the heterogeneous support of *Millepede*, once the system is operational, little work will be required in order to adapt it to other processors. In addition, we also plan to get some support on the way to heterogeneity, fault-tolerance and security by using connecting software layers such as Horus [53] and Transis [7].
Multi-Programming

We intend to support multi-programming parallel computing on empty environments. However, once an interactive user claims the machine’s resources, the Millepede activities will be evicted in a background mode. A computation is initiated by a dedicated daemon which will start (on local demand and when the load permits it) a distributed parallel computation. Once the load allows a machine to take part in a computation, the distributed load-sharing algorithm will provide it with work to do. Note that at any given point a machine may be running several parallel applications which may have been initiated in several different places. The load-sharing algorithm will have to make sure that they do not block necessary resources from each other.

Efficiency, Reduced Memory Consistency

Although the ease-of-programming and the portability that come with shared memory seem to trade with efficiency and speedup, most of the problems that require parallelization experience good performance on our system even with naive programming. The reason is that the locality principle which is explicitly taken care of by the programmer in message passing systems is taken care of automatically by the underlying shared memory mechanism that we support. Sometimes, however, there is a need to “help” the shared memory mechanism in using locality. We thus decided that the programmer will explicitly differentiate local variables from shared ones. This seems easy on the programmer on the one hand, and on the other hand increases reference locality as local variables are mapped to local stacks. Furthermore, stronger support for various memory consistencies is provided in the next generation Millepede. The specific consistency class is a parameter in the shared variable definition, hence we can dynamically revise our set of available protocols. For example, this design enables the programmer to define two shared variables \(X\) and \(Y\) that will both be handled using release consistency, however using different protocols: say, \(X\) will use an update-based protocol whereas \(Y\) will use an invalidate-based protocol.

The strong support for various consistency types provides the programmer with tools to enhance the efficiency of his program. The common procedure for the development and testing of a program is to design it using the simulator, then run it in the distributed system with strong-consistency shared variables only. Finally, if desired, the programmer may carry optimizations which gradually weaken the consistency requirements of key shared variables, thus enhancing locality of reference.

Due to our positive experience with this policy on Millepede, local memory access is handled by the VMM only, and thus cannot be recorded. This presents some difficulty for the optimization of the page/thread migration policy of the load sharing algorithm.

Other Goodies

Finally, we mention in passing that other extensions are now under development, which include extending PARC to a parallel object-oriented language called PARC++, develop load sharing algorithms that attempt to improve locality of memory reference, and include lower granularity update-based protocols.

8 Conclusions

Our conclusions which were derived from our experience with the Mach-based Millepede system are now in the process of being implemented in the new NT-based system. Basically, our approach of supporting user-level portability and availability rather than efficiency proved successful. Most
programs would achieve high speedup due to the locality of memory access that is supported by the page migration mechanism. Yet, for certain problems such as our linear equations solver (see Sec. 6), it seems that a stronger support for reduced consistency is desirable. In particular, it seems that release consistency is important for problems where writing may share pages. Sometimes a closer examination of programs would reveal that the programmer used the weak protocol to simulate release consistency (by artificially padding the data to fit into pages). As described above, the next generation will heavily support this direction.

The load-sharing algorithms that are currently under development are expected to solve most of the remaining false-sharing and ping-pong problems by unifying the page and thread migration policies. These are expected to further improve performance, although special care should be given that they do not thrash, and that their extra overhead will not degrade system performance.

Our experience with the distributed memory manager exceeded our expectations. As a result of the distributed design the system accelerated on many of the programs that we ran. Indeed, one of our general conclusions is that improving locality of memory access is generally more important than improving the balance of the CPU load. Certainly this is the result of two main features of our current implementation: that local memory access does not suffer extra overhead, and that the communication media is very slow.

**Acknowledgments:** The ParC2Millepede preprocessor was first implemented by Gady Haber. An improved version was coded by Nedim Fresko and Ophir Holder. Current version was rewritten from scratch by Mor Nagari and Gabriel Reish.
References


A Details of Strong

Recall that the strong consistency protocol that we use allow to have either a single writable copy or multiple readable copies of any single page at the same time. In the rest of this section we describe in details how this policy is enforced.

A.1 First Access to a Page

The first time that any client tries to access, either for reading or writing, a page which does not exist, the kernel sends a `data_request(..)` call to the MM. Upon receiving the `data_request(..)` call, the MM creates the requested page and supplies it as readable and writable. (See illustration in Figure 9). When the kernel receives the page, it sends a `supply_completed(..)` message back to the MM in order to confirm that the page has arrived.

In order to avoid any inconsistencies in case of concurrent requests for the same page from two different clients, we must protect changes to the page with locks. This is imposed by having the MM locking the page’s mutex before the page is sent to the kernel, and unlocking the page only after receiving the acknowledgment from the kernel. Also, any attempt to get a reference for the page must first obtain a lock on the page’s mutex, which guarantees the desired non interference policy.

Figure 9: The first `data_request(..)` call.

A.2 Trying to Read an Initialized Page

If a client wants to read from a page which is not cached on the local host, an appropriate `data_request(..)` message is sent to the MM. Depending on the access rights of the existing copies, there are two possibilities to handle this request.

If there is at least one `read_only` copy of this page in the system, then the MM simply requests from one of these hosts to send its copy of the page to the MM, so the MM can relay this copy as `read_only` to the initiator of the action (Figure 10). Otherwise, there is a single `writable` copy of the page located somewhere (Figure 11). In this case, the kernel of the current owner of the page changes its access rights to `read_only`, and sends a copy of this page to the MM, which then passes the copy to the requesting host as `read_only`.

26
Figure 10: The `data_request(..)` call under the strong management. A client task attempts to `read` the data from the page which has multiple `read only` copies.
Figure 11: The \texttt{data.request}(...) call under the \textit{strong} management. A client task attempts to \textit{read} the data from the page which is stored as \textit{writable} on another node.
Figure 12: The \texttt{data.request(\ldots)} call under the strong management. A client task attempts to \texttt{write} while the page has multiple \texttt{read\_only} copies on other nodes.
A.3 Trying to Write to an Initialized Page

When a client wants to write to an initialized page which is not cached on the local host, an appropriate `_data_request(..)` message is sent to the MM. Again, depending on the access rights of the existing copies, there are two possibilities to handle this request. If there are some `read-only` copies of the page (and, therefore, there is no `writable` copy), then the MM sends a `_lock_request(..)` message to one of the hosts that holds a `read-only` copy of this page (Figure 12). This call conveys commands to return the page and flush it from the host’s physical memory. Upon receiving the page, the MM invalidates all other `read-only` copies (if there was more than one such copy), and provides the returned page as `writable` to the host that requested this page.

If there is a `writable` copy of this page (Figure 13), then this copy must be invalidated, and the host that requested the page should be supplied with the most recent version of this page, marked as `writable`. This is again achieved by sending the owner of the page a `_lock_request(..)`. Note that in this case there are no other copies of the page, so there is no need to invalidate anything else.

![Diagram](https://via.placeholder.com/150)

Figure 13: The `_data_request(..)` call under the strong management. A client task attempts to `write` while the `writable` page is stored on another node.

Also, if a host that would like to write to a page holds a `read-only` copy of this page, then it can change the access writes of this page to `writable` after receiving the appropriate authorization from the MM. But there is no need for that host to first flush this page and send it to the MM.
as would have happened otherwise, since this host already holds an updated copy of this page and will continue to hold it after the end of this interaction with the MM (Figure 14).

Unfortunately, the implementation as described above could create a deadlock in the following scenario: Consider the situation in which a client on host \( j \) tries to read or write to a page which has only read-only copies such that one of them is located at host \( i \) (Figure 15). Assume also that immediately after the MM has locked the mutex of the page, and before the \_lock_request\()\ call that the MM issued due to the request of host \( j \) is delivered to the Kernel of host \( j \), a client of host \( i \) tries to write to this page. Since the copy of the page at host \( i \) is still marked read-only, the kernel of host \( i \) sends a \_data_unlock\) message to its MM, and blocks until it gets a \_data_return\) message from the MM. In order to serve the \_data_unlock\) message, the MM must first lock the page’s mutex by calling \_mutex_lock\). However, the mutex is still locked by the MM that started serving the request of host \( j \). On the other hand, the latter MM will only release this mutex after it receives a \_data_return\) message from the kernel of host \( i \). However, the kernel of host \( i \) can only send this message after receiving the appropriate \_lock_request\) call. But the kernel of host \( i \) cannot receive this message, since it is blocked waiting to hear from its MM, which is blocked on the mutex.

We have solved this problem by creating a new thread for each \_data_unlock\) request. This way, if this thread is blocked on a page mutex, there is still a thread to handle incoming requests. The only other issue which has to be taken into account is that is such a thread blocks on a mutex, there is no way to guarantee that it will be the first one to be awakened when the mutex variable is unlocked. Thus, by the time a thread that was created in order to serve a \_data_unlock\) request wakes up, it is possible that due to other requests, the copy of the page is already writable. Therefore, when this thread wakes up, it must check if the page is not already writable, and if it is, then this thread can terminate immediately.
Figure 15: The possible _data_unlock(..) deadlock under the _strong_ management.
B Details of Weak

In this protocol, for every page that has been accessed at least once, there exists one writable copy, and possibly several read-only copies. We now describe in details how this is achieved.

B.1 First Access to a Page

At the beginning of the execution, there are no memory pages in the system. Pages are created by the memory manager as necessary (i.e., when threads attempt to access data items within pages not currently present). The kernel – MM interaction for the first access to a page is the same as shown in Figure 9.

B.2 Trying to Read an Initialized Page

Figure 16 illustrates the protocol for handling requests to read a page which is not cached on the local host. In this case there is one writable copy of the page. Denote the host that holds the client which attempts to read as host_\(j\), and the host which holds the writable copy as host_\(i\). As always, a \_data_request(\_) is sent to the MM. The MM needs to obtain a copy from host_\(i\) and send it to host_\(j\) as read-only. This is done by sending a \_lock_request(\_) message to host_\(i\). However, in this case, host_\(j\) is requested to keep its cached version of the page as writable, and just send a copy of it. Note that even if there are other read-only copies, the MM will always seek the writable copy, since it is guaranteed to be the most updated copy regarding recent writes to the page.

As in other cases, after sending the \_lock_request(\_), the MM blocks on a condition_wait() call, until the desired copy is received from the kernel of host_\(i\). Also, during the entire interaction, the mutex variable of this page must be locked, to prohibit inconsistencies in the case of several concurrent accesses.

B.3 Trying to Writing to an Initialized Page

A write access to a non-cached page is processed similarly to a read access. The only difference is that in this case the access rights on host_\(i\) are changed to read-only while host_\(j\) receives the page as writable (see also Figure 17).

On the other hand, if the page is already marked as read-only on the local host, then the kernel of this host must send a \_data_unlock(\_) message to the MM. (See illustration in Figure 18.) The MM then sends \_lock_request(\_) message back to the kernel, in which the kernel is requested to flush the page, but not to send a copy of it to the MM. After the kernel flushes this copy, it is faced with the problem of trying to write to a page which is not cached on the local host. For this, it can simply use the protocol which is illustrated in Figure 17, and was described already.

B.4 Updating the read-only pages

An apparent advantage of our weak memory manager is that it allows one thread to write to a page while the page is being read form by other threads. (There can be several readers, but at most one writer for each page). However, due to this feature, reads are not guaranteed to return the most up to date value. In particular, it is possible that a certain host will hold the same read-only copy of a page, and will never be updated with the changes to the data in this page. Hence, care should be taken to occasionally update all read-only pages.
Figure 16: The `data_request(..)` call under the weak management. A client task attempts to read data from the page.
Figure 17: The `data_request(..)` call under the weak management. A client task attempts to write on the page.

Figure 18: The `data_unlock(..)` call under the weak management.
One way of propagating such updates, it to send an update message after each write to all read-only copies of this page. However, since our implementation does not catch local accesses, e.g., writes to writable copies, we cannot use this solution.

Alternatively, it is possible to have periodic, or user initiated, invalidation of the read-only copies. After a page is invalidated, the following attempt to read form this page will result in a page fault, which can be handled by the MM according to the protocol described in Section B.2, which gets the most updated copy of the page to the local host.

We consider two possibilities to determine how often to invalidate read-only pages:

### B.4.1 Timed Invalidations

This method is implemented inside the memory manager module. A thread of the memory manager is awakened at predefined intervals in order to invalidate all read-only copies in the system that belong to this manager (Figure 19).

![Figure 19: Update of read-only copies at predefined intervals.](image)

This solution poses a challenge to find the best frequency in which invalidations should take place. If the invalidations are not frequent enough, then it will take longer for updated data to propagate, which will slow down the computation. On the other hand, high frequency of invalidation can over load the system, and effectively slow down the execution. Also, parallel activities do not know when the next invalidation will take place. Hence, in order to avoid using stale data, or even worse, uninitialized data, parallel activities may need to use spin locks all the time, which
complicates the task of programming, and in practice can even cause unnecessary delays in the execution.

### B.4.2 Application Based Invalidations

We provide several primitives for the applications so whenever the application needs updated data, it can initiate invalidations of *read-only* pages. The idea is that whenever a user needs to read the newest value of any data item, he/she can invoke an appropriate memory synchronization function, which will guarantee that the *updated* data will be received.

Memory synchronization is performed in the following way. It is the responsibility of the user to supply the exact description of data items (i.e., their addresses) to be updated with their latest value. Both the kernel and the memory manager are involved in the message passing dialog at this point. During a *memory_sync* function call, a *synchronous* message is sent from the MACH kernel to the memory manager (which may be running on any host).

![Diagram](image)

**Figure 20**: Updating *read-only* copies: memory synchronization scheme

The arrows on Figure 20 are as follows:

1. A thread calls the “*memory_sync*” function, implemented at the *Millepede* layer.
2. MACH remote procedure call (RPC) during execution of the “*memory_sync*” function.
3. A message is sent from the kernel to the memory manager as a result of this RPC. Since this message is synchronous, the “*memory_sync*” function execution is suspended until an answer from the manager arrives (arrow “6”).
4. The memory manager sends a request to the kernel to invalidate *read-only* pages.
5. The kernel confirms fulfillment of the invalidation request.
6. The memory manager acknowledges fulfillment of the “*memory_sync*” request.
7. The kernel returns the control to the “memory.sync” function.

8. Control returns to the thread.

As a result, the thread can access any item on the invalidated pages. A page fault will occur and a new copy of the page will be fetched to him. Notice, that every time the weak memory manager processes a page fault, it takes a new copy of the page from the owner of the writable copy (since the memory manager does not save pages by itself). Remember, that as we do not allow multiple writable copies of the same page, this copy always contains the newest values of all data.

Several messages in the above dialog are optional. For instance, if a thread asks to update pages which are not stored on the host at all, or which are present as writable, then Messages 4 and 5 will be unnecessary. In any case, Messages 3 and 6 must be sent, since only the memory manager knows the exact distribution of all pages in the system.

As we show further, any thread could update not only pages on its own host, but also pages located on other nodes in the system. From a ParC user program point of view, this method is preferable when any thread may possibly update some data and would like to inform all the others about it. In this case, Message 4 will be sent to all the hosts that possess read-only copies of that data, and the memory manager will wait for confirmation from all of them (Message 5).

Several memory.sync functions are currently implemented. Recall that activating these functions is only meaningful under the weak memory manager. However, since a ParC user can experiment with the system and his ParC program using various types of memory management and load balancing protocols, we decided to allow the usage of memory.sync functions under other memory managers as well. Therefore, if a user chooses a DSM algorithm other than the weak forgets to remove the memory synchronization functions from the program code, the program will compile and run without error messages or warnings. Instead, only Messages 1 and 8 (Figure 20) will be sent in this case. Millipede can automatically recognize the type of memory manager used during the memory.sync function call, and can therefore immediately return control to the thread without sending any further message to the memory manager.

We have implemented the following functions:

**total_mem_sync()** A very heavy operation which leads to invalidation of all read-only pages in the system that belong to all memory managers on all nodes; no parameters are required.

**host_mem_sync()** Causes to invalidation of all read-only pages that belong to all memory managers and are located on the host on which this function is executed (and only those pages); no parameters are required.

**page_mem_sync(adr)** Causes to invalidation of a read-only page that contains the address (adr) on the host where this function is executed, and only this page.

**total_page_mem_sync(adr1, adr2)** Causes to invalidation of read-only pages within the region from *adr1* to *adr2* on every host in the system.

**host_page_mem_sync(adr1, adr2)** Causes to invalidation of read-only pages within the region from *adr1* to *adr2* on the host where this function is executed (and only those pages).