Thread Migration and its Applications in Distributed Shared Memory Systems*

Ayal Itzkovitz  Assaf Schuster  Lea Wolfovich
Computer Science Department
Technion - IIT
{ayali, assaf, wlea}@cs.technion.ac.il

Abstract

In this paper we describe the way thread migration can be carried in distributed shared memory (DSM) systems. We discuss the advantages of multi-threading in DSM systems and the importance of preempted dynamic thread migration. The proposed solution is implemented in MILLEPEDE: an environment for parallel programming over a network of (personal) computers. MILLEPEDE implements transparent thread migration mechanism: a thread in a MILLEPEDE application can be suspended almost at every point during its life-time and be resumed on another host. This mechanism can be used to better utilize system resources and improve performance by balancing the load and solving ping-pong situations of memory objects, and to provide user ownership on his workstation. We describe how some of these are implemented in the MILLEPEDE system. MILLEPEDE, including its thread migration module, is fully implemented in user-mode (currently on Windows-NT) using the standard operating system APIs.

1 Introduction

Many attempts are made to integrate the resources and services of distributed computational environments into virtual parallel machines, or: metacomputing environments. While being very cheap and available to everyone, such metacomputing environments will exhibit very high computational power, large virtually shared memory, and high bandwidth of I/O and communication. Applications using these environments will have to be dynamically adaptive to the varying network configurations, utilizing idle resources and instantly evicting those resources reclaimed by their native users.

In order to integrate the resources of a distributed environment some form of cooperation among the nodes (or, computers) is necessary [Casavant and Kuhl, 1988; Chase et al., 1989; Krueger and Livny, 1988; Kumar et al., 1987; Willebek-Le-Mair and Reeves, 1993]. Dynamic load sharing is the form of load distribution that has a potential of being efficient in a distributed system [Eager and Lazowska, 1986; Kremien, 1993]. Load-sharing algorithms attempt to assure that there are no idle hosts when there are tasks waiting for execution on other hosts. This is achieved by dynamic initial placement and by migration after startup. Multithreading further helps to achieve better load distribution between the nodes in the system by splitting the application into smaller chunks of work.

However, distributing an application over the network has its drawbacks. Components of an application need to communicate and synchronize, imposing overhead that, due to the relatively inefficient communication, is typically very high [Kumar et al., 1993]. In fact, the optimal speedup in a distributed environment is commonly obtained by using fewer processors than the total number of those that are idle and available. The exact distribution of the machines that take part in the computation must be determined dynamically, according to both the system varying capabilities and the application varying needs. The solution to all these can be found by using multithreading and thread migration: Multithreading can hide the latency by overlapping communication and computation. Thread migration can significantly reduce the amount of communication in DSM systems, by migrating threads in order to improve locality of shared data accesses.

Although most of the power of metacomputing environments will typically consist of personal machines, degrading of interactive response must be avoided. If the owner of a machine or a resource is not guaranteed to receive it at the moment he attempts to use it he will not allow “invasion” of remote execution in the future [Douglish and Ousterhout, 1991]. To this end, once again thread migration is the answer which may be used to provide user ownership in an efficient way.

Part of the machines in a metacomputing environment may be symmetric multiprocessors (SMP), which are tightly-coupled “shared-all” multiprocessor machines. In an SMP system all the components such as processors, physical memory, buses, disks and controllers are shared. A single copy of an operating system controls all components, manages the shared memory, and balances the load among the processors by dynamically reassigning processors to threads. Here, using multiple threads make it possible to utilize the processors in a transparent and efficient way.

SMP systems are becoming widely available and it is expected that this process will promote the development of parallel applications that use multithreading and shared memory.
From the application point of view, non-scalable parallel computing on SMP machines with shared memory, and scalable parallel computing on metacomputing environments with virtually shared memory, are at the same level of abstraction. Thus, given an efficient run-time support for metacomputing environments, the transition from parallel computing on SMPs to parallel computing on distributed environments is just a small, natural step.

As argued above, an efficient support for metacomputing environments must include migration of threads between machines. Unfortunately, implementing thread migration is not an easy task. In this paper we discuss the problems and complications of such implementations, with a special emphasis on the relation to a possible neighbor DSM mechanism. We describe some wrong solutions that appear in the literature and present a good solution that is implemented in the MILLIPEDE virtual parallel machine.

We then proceed to describe the way thread migration is utilized in the MILLIPEDE system. MILLIPEDE is a thread-based system for the development and execution of parallel applications in distributed environments. It presents a strong application interface, including a flexible DSM mechanism along with a dynamic thread scheduling algorithm. The thread scheduling algorithm strives to reach the optimal speedup by dynamically solving the tradeoff between minimal load and minimal communication. It also tries to minimize communication by migrating both threads and pages between machines, until maximal data locality is achieved. To this end, MILLIPEDE implements a transparent thread migration mechanism that is used by the thread scheduler.

MILLIPEDE is currently implemented on the Windows-NT operating system, using its support for multithreading and SMP thread scheduling. Detailed description of the MILLIPEDE system can be found in [Itzkovitz et al., 1996a].

Related Systems

We now discuss the main differences between MILLIPEDE and several other systems that support thread migration.

- UPVM is a package that supports multi-threading and transparent migration for PVM applications [Casas et al., 1994]. UPVM defines an abstraction having some of the characteristics of a thread and some of a process, called a user level process (ULP). ULPs differ from threads in that they define a private data and heap space. ULPs communicate with each other via message passing. The ULP state that is transferred when a ULP migrates includes the context, the stack, the data, and the heap.

  As in MILLIPEDE, mapping of a ULP to a set of virtual addresses is unique across all the processes of the application. The difference is that MILLIPEDE threads keep their non-local data in shared memory that need not be transferred explicitly at migration time. The memory usage of a thread triggers the migration of pages to its new location meaning that only the data that is actually used by the migrated thread is transferred on demand, thus decreasing the cost of migration in MILLIPEDE.

- Ariadne is a user-space threads system that runs on shared- and distributed-memory multiprocessors. In contrast to Ariadne, MILLIPEDE uses operating-system supported
threads (also called kernel threads in UNIX-like environments). The advantage of user-space threads is their relative portability, since they may be implemented on an operating system that does not support threads. In addition, context switching between user threads is faster than context switching between kernel threads. However, this may change in the future, since next generations of processors may support thread context switching in hardware, thus making switching of kernel threads less expensive than that of user threads.

There are two main disadvantages of user threads. First, a user thread that blocks on an internal page fault or a system call causes blocking of its process. If kernel threads are used, a thread that blocks does not prevent other threads of the same process from running. Another advantage of kernel threads is that on an SMP they are scheduled by the operating system automatically on available processors. With user threads, an application has to be modified explicitly in order to use multiple processors. In Ariadne, additional processes are created for this purpose imposing high overhead.

Similar to Millipede, thread migration in Ariadne is supported at user level in homogeneous environments. However, the mechanism of migration in Ariadne is different from Millipede. We further discuss Ariadne’s thread migration and the problems that are associated with it in Section 3.3.1.

- Amber [Chase et al., 1989] is an object-oriented DSM system that permits a single application to use a homogeneous network of computers. Each node may be a shared-memory multiprocessor. Amber supports data and thread migration; the location of objects is managed explicitly by an application. The mechanism of thread migration is essentially the same as in Millipede. The difference is that with Millipede a programmer does not have to deal with the data and threads location issues, since Millipede provides a location-independent interface and automatically improves locality of data accesses at run-time.

The rest of this paper is organized as follows. In Section 2 we discuss our motivation for using preemptive multithreaded DSM systems. Section 3 discusses some global aspects of thread migration, explains the various approaches introduced so far for its implementation, and proposes a new approach for implementing thread migration in user-space which is applicable on most existing operating systems. Section 4 gives an overview of the Millipede system and discusses its implementation of thread migration. Section 5 describes the way Millipede utilizes thread migration in order to share the load and improve the locality of memory references. Section 6 presents some measures taken with the Millipede system on a non-homogeneous environment, essentially giving examples to possible improvements of performance that are enabled by thread migration. Finally, Section 7 gives some concluding remarks.

## 2 Motivation and Discussion

In this section we discuss the advantages of the DSM model combined with multithreading. We also explain the benefits of dynamic load distribution schemes and thread migration in
multithreaded DSM systems.

**Why DSM systems**

Distributed Shared Memory (DSM) is an implementation of a shared memory paradigm on a physically distributed system [Keleher et al., 1994; Li and Hudak, 1989]. Parallel programming in this model is easy, since the DSM is a natural generalization of sequential programming. Furthermore, with a DSM it is relatively easy to parallelize sequential programs. In this model, components of an application communicate using a virtually shared memory. Local and remote data accesses are carried in a way transparent to the programmer, serviced by the underlying DSM mechanism. This makes DSM applications both easier to develop and more portable (across DSM architectures) than programs that use explicit message passing. In particular, metacomputing environments which exhibit virtually shared memory (and may consist of the cooperation of large suits of various machines and resources), are at the same level of abstraction as that of multiprocessor machines with physically shared memory. In fact, the programming paradigms are at the same level of abstraction as that of a multithreaded uniprocessor machine.

With the rapid growth of popularity and availability of SMPs, it is expected that more users will attempt to utilize the power of their machines by parallelizing them. This will lead to a growing set of available parallel applications. These applications will assume the convenient programming paradigm provided by their native multiprocessor machines; namely, multithreaded parallel computing with shared memory, that does not assume a dedicated machine. Given this expected large volume of applications, it is just a natural step to provide this interface (including in particular the DSM) also on top of physically distributed, metacomputing environments. Such metacomputing environments have the additional advantage over SMPs of being scalable to higher levels of parallelism.

**Why dynamic load sharing**

Load distribution is necessary in a distributed system for better utilizing its computational power. Various load balancing and load sharing algorithms appear in the literature. In general, the purpose of the load balancing operation is to split the work evenly among the processors, whereas the approach of load sharing algorithms is to ensure that no processor stays idle or slightly loaded when there are heavily loaded processors in the system.

Static load distribution strategies are effective when applied to problems that can be partitioned into tasks with uniform computation and communication requirements. An additional requirement of static algorithms is that the environment is homogeneous, i.e., all machines in the system should have identical hardware parameters (such as processor speed) and similar load resulting from other activities. There exist, however, a large number of problems with non-uniform and unpredictable computation and communication requirements. Also, machines in a non-dedicated network of computers (such as a metacomputing environment) will commonly differ in their speed and load state; part of them may be even unavailable at certain times. Therefore, dynamic load distribution is essential both for efficient solving of non-uniform problems and for solving uniform problems in a non-uniform
environment. Thus, it seems that in a metacomputing environment applying either dynamic load balancing or dynamic load sharing is unavoidable.

The overhead imposed by dynamic load balancing in a large distributed system may outweigh its potential benefits for the following reasons. First, equalizing the load among all nodes in the system requires large amounts of precise, global information concerning the state of all the machines. For fairly large systems this may violate the scalability requirement. Furthermore, when the overall system load is high, load balancing strategies will cause transfer of work from highly overloaded hosts to other hosts that are overloaded as well. This may improve performance in some cases, e.g., when iterations of a loop are scheduled on a uniform system. However, with environment such as a network of workstations, this strategy will only impose additional overhead, and may even cause an unstable behavior.

In contrast to load balancing algorithms, dynamic load sharing strategies have a potential of achieving resource utilization that is almost as good at a much lower cost. Due to their relaxed requirements, load sharing algorithms may avoid the need for global information, using restricted local information only. The algorithm may do very well even if machines know the status of only part of the other machines in the system. Moreover, the information may be less precise than that needed for load balancing, and thus may be exchanged less frequently. Another advantage is that load sharing algorithms can be designed so that no overhead is imposed when all nodes in the system have enough work to do. This makes load sharing strategies potentially more efficient, especially in dynamically changing environments.

**Why multithreaded DSM systems**

Multiple threads within a process share its virtual space. Threads are the basic entity to which the operating system allocates the CPU time. On a multiprocessor system executable threads are distributed among the available processors. Therefore multithreading allows an application to take advantage of an SMP architecture by using all the processors on a node in a way that is transparent to the programmer, and is natural to a shared memory application. As long as the level of parallelism in the application exceeds that of the actual machine, it need not be changed in order to utilize multiple processors. Modern operating systems balance the load among the processors of the machine when enough threads are available, and this load distribution need not be programmed in advance. In addition to better utilization of multiprocessor machines, using multiple threads allows also better load distribution over the network, when the level of parallelism provided by the application is sufficiently high.

In an environment that does not support threads, an application should be divided into multiple processes in order to be parallelized. However, the cost of communication, synchronization, and context switching between processes, is a lot higher than that of multiple threads that share the resources in the same process. The reasons are that the threads can exchange data efficiently using the shared virtual address space, that their context is small relative to that of a process, and that their working sets may overlap, so that in many cases context switching between threads does not cause swapping, while process switch would do.

Some additional overhead may be imposed by multithreading due to the need to switch contexts. This switching commonly occurs in a remote access. The associated overhead is
thus justified, as it implies that the time one thread is awaiting for the remote access to complete (called the latency of the system) is overlapped by a computation that is carried by a different thread. In this way we avoid stalling the processor during remote accesses that may be frequent in a large metacomputing environment. When kernel threads are used this overlap of communication and computation is easy, natural and efficient, by the automatic scheduling of the operating system.

Another advantage of using multiple threads is the reduced cost of migration. Migrating threads is less expensive than migrating a process since process migration requires transferring all virtual space of a process [Zayas, 1987], while migration of threads in a DSM system requires only the transfer of memory occupied by the threads’ stacks.

Why thread migration

Dynamic initial placement of threads solves part of the problems arising from non-uniform problem or environment. However additional performance improvement can be achieved by thread migration for the following reasons:

1. Load may change quickly, causing poor utilization of processors. Therefore redistribution of the load is necessary.

2. Poor initial placement of threads may cause large communication overhead. In a DSM system this happens when threads that are executing on different hosts are using the same data. In such a case migrating these threads to one host turns the remote data accesses into local ones, thus reducing communication overhead.

For a network of personal computers, there is one more reason for migration being important. A user expects to receive the full resources from the machine he is using. Therefore, threads executing remotely should not degrade interactive response. To achieve this, threads should be executed remotely only on idle machines, and if a user returns before they finish, they should be stopped. Thread migration mechanism makes it possible to continue the execution of such threads on other hosts.

3 Designing Thread Migration in a DSM System

This section discusses problems that need be solved when designing a user-level thread migration in a non-distributed operating system. We make several assumptions on the underlying system. We consider operating systems that provide support to multithreading at the kernel level. Migration is only supported across machines with processors of the same architecture, running the same operating system. We assume that the migration issues are transparent to the application. In particular, migration may occur at any moment during a thread’s life-time, and not only in predefined points (where the thread is checking if it should migrate). We also assume that a conventional compiler is used, so that no extra information about threads’ state is available.
3.1 Requirements From the Operating System

The following services by the operating system are vital in order to support a user-level implementation of a combined DSM and thread migration:

- Virtual address space that is arranged identically for each instance of an application. Namely, the code and the static data reside at the same virtual addresses in each copy of a program.
- Protection of pages in virtual memory and exception handling on a protection fault.
- Interface for the creation and management of threads, including a mechanism for obtaining and updating a thread's state.
- Some mechanism for resetting the location of threads’ stacks. It should be possible to reserve a range of virtual addresses for the stack of a thread.

The reasons behind the above requirements are described below.

3.2 Restrictions on Thread Migration

Here we describe the thread state and the problems that arise when a migration of a thread occurs, i.e. when a thread is stopped on one host and is resumed on another one in the same state. We identify the restrictions on the state of the migrated thread that are necessary to make the migration possible.

Thread state consists of global data and thread-specific information: stack contents, registers values and operating system internal control information. In the DSM model global data is assumed to be allocated in shared memory, so it should not be transferred explicitly when a thread migrates (this will be done by the DSM when needed, i.e. when a migrated thread will attempt to access this data). On the other hand, the stack contents and the register values must be transferred at migration time.

The stack and the registers may contain pointers to code, global data or data in the stack. A potential problem is that these pointers may not have the same meaning on different hosts. Thus, it is necessary either to ensure that the pointers will retain their meaning, or to provide some translation mechanism. We are assuming that program code and static data are automatically placed by the operating system at the same virtual addresses in each copy of the program; DSM addresses also have the same meaning in each instance, so the only problem that should be treated is the pointers to data in the stack. This problem is discussed in detail in section 3.3.

Another important issue is the usage of the system calls. A user can not access the internal control information of the operating system, so it can not be updated or transferred when a thread migrates. Therefore, a thread that owns system resources cannot migrate. For example, a thread that entered a critical section (using the corresponding system call) and did not leave it yet owns the critical section object; its migration in this state will prevent other threads from entering the critical section. Releasing the critical section on the destination host will not make much sense, because in a non-distributed operating system
object handles are meaningful only on the host they were created on. It might be possible to redirect such calls to relevant machines, but this requires redefinition of all the system calls, and in addition increases the cost of remote execution.

Many system calls (especially those used for synchronization) cannot be used explicitly in user level in a distributed system that supports thread migration, because the location of a job may change at any time. For example, jobs cannot communicate via pipes, since they have no information about each other’s location. Even if they do have such information, the location of a job may change after a message was sent to it and before it arrives. Thus, some other mechanism of synchronization is necessary in such systems. Using DSM for this purpose might be extremely ineffective; for example implementing critical section using shared variables inevitably involves busy-waiting and in addition imposes high communication overhead associated with synchronization of these variables.

3.3 Implementing Thread Migration

We now describe and discuss several approaches to the problem of transferring the stack contents of a migrating thread.

3.3.1 A simple approach that fails

The approach described here is used, for instance, in the Ariadne system [Mascarenhas and Rego, 1996]. We will describe the method itself and the problems that it may cause, and try to explain why and when it works. The method is as follows. When a thread migrates the contents of its stack on the destination machine is copied to addresses that might differ from addresses on the source machine. Let us call stack self-references pointers that reside in the stack and reference some data that also reside in the stack. These self-references, as well as the stack pointer and the frame pointer, have to be translated when the stack is moved to different addresses. The offset that is used for this purpose consists of the difference between the stack bottom address on the origin machine and the stack bottom address on the destination machines.

The stack contains two types of self-references: saved frame pointers and addresses of stack data. The latter may reside in the stack in several ways: as parameters to functions, as values of local variables, as values of saved registers, or as intermediate values used by a compiler. The method suggested in [Mascarenhas and Rego, 1996] is to identify such references and update them (details are not provided). Saved frame pointers are easily identifiable, so they can be updated correctly. The problem is that local data addresses in the stack can not be identified in the general case. They may be everywhere in the stack; the data in the stack may be even mis-aligned (if compiler alignment must be disabled for some reason). The only way such addresses might be updated without some additional information is to prohibit the use of data types such as char that may cause misalignment in the stack, to examine the value of each aligned entry in the stack and update it if it may be a stack self-reference, and to hope that this updated value was not a non-pointer that accidentally looks like pointer to the stack data.

Consider the following example. Let us suppose that the nodes in the system use perfectly synchronized clocks; an application orders events of some type using timestamps. A thread
performs an operation \textit{get\_time} that returns the number of milliseconds that passed from some predefined moment. The thread stores the obtained value in a local variable \( t \) that resides in its stack. At this point the thread is preempted, and later it migrates to another host. If the value of \( t \) is in the range of stack addresses of the thread, it will be updated as if it were a stack reference. If now the thread will store the variable \( t \) as a timestamp of an event, the event ordering may become incorrect.

Another problem with this approach is that general purpose registers may also contain pointers to stack data. It is claimed in [Mascarenhas and Rego, 1996] that this occurs only when compiler optimizations are used, but this claim is clearly incorrect. Thus, values of registers must be updated too, causing the same problem as that of identifying references to stack data.

We believe that translating the state correctly in the general case when this method is used is impossible without compiler support. A natural question to ask is how this method works in systems that use it. The answer is that the probability of correct operation is high, given that only aligned data is used, that migration is initiated by the migrating thread itself (thus eliminating the problem of temporary addresses in registers), and that there is a little amount of non-pointer values on the stack. However, these limitations do not guarantee correctness of the state translation in the general case.

### 3.3.2 A popular approach

Since translating the pointers is impossible without extensive compiler support, it is necessary to ensure that the pointers will retain their meaning after migration. To achieve this, the segment of virtual memory occupied by the stack on one host is reserved for it on all other hosts, so that the stack contents can be copied to the same addresses when a thread migrates.

The popular method (incorporated, e.g., in [Chase et al., 1989; Casas et al., 1994; Dubrovski, 1996]) for reserving memory for stacks is as follows. A region of virtual memory starting at a predefined location is reserved for the threads' stacks on every host; each thread is assigned a unique identification number that is used to find the thread's slot in the stack region. A newly created thread is forced to use the proper slot as its stack. Moreover, this slot can be allocated from the DSM, so the stack need not to be explicitly transferred; it will be transparently brought by the DSM mechanism when needed. This method is very easy to implement when user-level threads are used, so that a programmer has control over the locations of thread stacks. With kernel-level threads, the situation is more difficult since in this case the stacks are usually allocated by the operating system. This problem may be solved in the following way. The register context of a newly created thread is changed so that the thread will use the proper slot instead of the original stack; this is performed before the thread starts executing and before any values are written into the stack.

This method can be used in many systems; however, it has a serious disadvantage. Namely, it is based on the assumption that the operating system behavior does not depend on initial location of thread stacks. This is not the case for some existing systems; for example, the Windows-NT operating system checks validity of a threads' stack pointer in certain cases, and if it decides that the stack pointer is illegal it just terminates the program. It definitely will decide so if a thread will use a stack at location other than the original
one (registered by the operating system). Moreover, even if the assumption above is true in some operating system, it may be violated in its future versions. Thus, this approach lacks portability.

### 3.3.3 Our approach

We solve the problem described above by using stacks allocated by the operating system while ensuring that these stacks will occupy the same addresses on all hosts.

A user application defines blocks of code that can be executed in parallel. These blocks are called jobs. The jobs are executed by separate threads. Instead of creating a thread each time a new job is spawned in a user program, a predefined number of threads called workers are used to receive jobs and execute them. The workers are created on each host at initialization time and run until the application completes. Since the virtual space of all copies is initially arranged identically and all instances perform their initialization in the same way, the copies of the same worker running on different hosts get their stacks at the same addresses. In this way the addresses are reserved for the stacks. A job that was already started by worker $i$ can be executed on any copy of this worker, i.e., on worker $i$ at any other host. To make sure that migration is always possible, at most one copy of each worker is executing a job at any given time. All idle workers are suspended.

As with the previous approach, the number of threads that can be created simultaneously on all nodes in the system is limited since the threads share a single address space among all hosts. If this limit is too low, a single application would not benefit from a massively parallel architecture. However, this problem may be eliminated when 64-bit architectures will be used, so that the limit on the number of threads will be high enough.

### 4 Architecture of the MILLIPEDE System

In this section we give a brief overview of the MILLIPEDE DSM system and its relation to the MILLIPEDE thread migration mechanism. MILLIPEDE is a user-level implementation of a multi-threaded DSM system with transparent page- and job- migration. The current implementation of MILLIPEDE is on Windows-NT operating system, and employs our proposed design for thread migration.

#### 4.1 Assumptions

MILLIPEDE was designed for the following type of environment and applications:

- **Homogeneous environment.** A network consisting of machines with processors of the same architecture is assumed, so that the representation of the program’s state is the same on all machines. Processors may differ in their speed. The network may include SMP machines.

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

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

## 4.2 System Overview

Each machine in the system runs a **Millipede** Daemon: a process that is in charge of collecting and disseminating of load information, of managing **Millipede** applications, and of the dynamic load sharing (Figure 1).

**Millipede** package includes two libraries: the DSM and the MGS. The DSM library provides interface for allocating distributed shared memory and keeping it consistent; it supports various memory consistency protocols (see [Itzkovitz and Schuster, ]). The MGS (Migration Server) library provides interface for creating multiple parallel activities and for managing them; it controls their locations and performs the migration.

**Millipede** applications are written in a parallel language independent of the underlying operating system, of the number of available processors, and of the data and threads location. Currently ParC [Ben-Asher et al., 1996] (a natural parallel extension of C) is supported (porting is underway for ParFortran90, ParC++, and Java). A ParC program is precompiled; the resulting C code is compiled using a conventional C compiler and is linked with the DSM and MGS libraries. The libraries are independent of the ParC language constructs; they provide an interface that allows implementation of any similar precompiler for any other language. It is also possible to write an application in a conventional language and use the libraries directly; this (less convenient) way may be used if the application requires some exotic synchronization method that is not supported by existing precompilers. The interface is further explained in [Itzkovitz et al., 1996b].

A **Millipede** application consists of instances (copies) of a user program running on different nodes in the system (see Figure 2). If a node is an SMP, all available processors are used in a transparent way by a single instance of an application. Instances of an application share single virtual space. They communicate in a location-independent way via the DSM mechanism and synchronize using the MGS primitives.

An instance of an application consists of the following parts (Figure 2):

- A pool of workers: system threads that receive user jobs and execute them.
- Memory manager: threads needed to keep the DSM consistent.
- Migration Server (MGS): threads that take part of the decisions whether to migrate, and handle the migration of jobs to and from the host.
Figure 1: MIIPIEDE structure

Figure 2: Instance of a MIIPIEDE application
4.3 Relation Between the DSM and the MGS Libraries

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

The MGS collects information provided by the DSM to make decisions on migration, and in some cases also affects decisions of the DSM mechanism as described below. The DSM mechanism passes to the Migration Server information about remote page accesses. The MGS uses this information to determine if threads should be redistributed to decrease communication. In some cases the MGS may affect behavior of the DSM by advising it to lock a page on the local host for a short time. In this way it is possible to stabilize the system when remote data accesses are causing high communication overhead, but thread migration is not possible.

The MGS also uses the DSM mechanism to store part of the necessary information. The MGS of each instance should keep track of the location of each running thread. Since a thread may migrate several times, keeping this information consistent on each host may be expensive. We solve this problem simply by using the DSM to store the threads’ locations.

4.4 Thread Migration in MILLIPEDE

4.4.1 Migration policy

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

- An overloaded node sends work to an underloaded one to decrease load imbalance.
- Threads that are causing high communication overhead are brought together.
- Remote threads are evicted by the machine when a native user starts working on it.

MILLIPEDE uses history of remote page accesses for making decisions on migration, where the objective is to minimize the amount of communication. The MGS “learns” about the communication pattern of the threads by recording remote page accesses. What make things interesting is that – for performance reasons – the information about local accesses is not recorded. Thus, the knowledge about the communication pattern is incomplete and incorrect decisions may be taken. For example, in the case that all threads which frequently access the same page are running on the same host, the MGS is not (initially) informed about it, so it may choose one of these threads for migration to another host. This will be a poor decision, since it will cause repeated transferring of this page between the hosts (a page ping-pong). However, information about this page will become available, making it possible to correct this decision and to keep the obtained information in order to improve future decisions. The detailed description of the algorithms that are used to decide that migration will take place and to select threads that will migrate can be found in [Schuster and Shalev (Wolfovich), 1997].
4.4.2 Migration implementation

The thread migration is implemented in user-level in Windows-NT, using standard Win32 API. The same implementation may be used in the Windows-95 environment as well. As we explained in Section 3.3.3, a pool of workers is used, where workers are threads that receive user jobs and execute them (Figure 2). The workers are created in each instance of an application when the MGS library performs its initialization; they run until the application completes. The copies of the same worker running on different hosts get their stacks at the same addresses (Figure 3); therefore a job that was already started by worker i can be executed on any copy of this worker, i.e. on worker i on any other host. To make sure that migration is always possible, at most one copy of each worker is executing a job at any given time. All idle workers are suspended.

The problem of using system calls is solved by providing a location-independent interface and by migrating only the jobs that do not own operating system resources and are executing user-level code, so that their state can be simulated on another host. The details follow.

Jobs are not allowed to use system calls explicitly, unless they notify the MGS. Suppose a job wants to display some data on a graphic window. Then it cannot migrate from the moment it starts to create the window and until it finishes closing it. The MGS should be informed about it; otherwise it may choose this job as a candidate for migration. Therefore, before it performs location-dependent activities, a job must notify the Migration Server. The MGS library provides functions to avoid/enable migration. These functions may be used at language-implementation level to prevent migration when executing location-sensitive code. Note that a typical computation-intensive application (that is the most natural candidate for porting to millipede) will rarely need to use these functions explicitly.

As was shown in Section 3.2, jobs cannot synchronize using the operating system interface. Therefore the MGS provides a general mechanism for inter-mobile-job communication, or MJEC (for millipede Job Event Control), which is described in detail in [Itzkovitz et al., 1996b]. MJEC solves the problem of obtaining job locations by storing them in a shared array (that resides in the DSM). MJEC can be used to easily implement all the known synchronization protocols (semaphores, barriers, condition-variables, monitors, etc.) in a location-independent way. Together with some basic interface functions that are used for creating and managing jobs, the interface supplied by MJEC is very flexible and powerful. It is designed to support the convenient implementation of various parallel languages, where the implementation is independent of the operating system and of location issues (see [Itzkovitz et al., 1996b]).

The global design of millipede makes it possible for the MGS to transfer a job in an extremely simple way. The Migration Server of the sender instance suspends a job and if migration is enabled for this job, sends its worker id, context and contents of its stack to the Migration Server of the receiver instance; otherwise it resumes the job locally. The design ensures that the proper worker on the receiver instance is idle, and its stack resides in the same addresses as on the sender instance. Thus, the Migration Server of the receiver instance simply copies the stack of the job and its context to the proper worker and resumes it.
Figure 3: An example of the virtual memory of a MILLIPEDE application running on 3 hosts. Job 1 is executed on instance 0; Job 2 is executed on instance 2; worker N is free.
4.5 MILLIPEDE Daemons.

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

A MILLIPEDE daemon consists of the following modules.

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

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

**Application Info Manager.** The Application Info Manager collects administrative information about each MILLIPEDE application running on the host, such as its unique identifier, instance identifier, master host id, system process handle, and so on.

**Local Load Info Manager.** It collects load information about each application running on the host, such as number of local jobs and number of mobile local jobs. It also determines the global load state of the local host.

**Masters.** A separate manager called Master is created for each new application that was started locally. The master is in charge of distributing its application over the network.

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

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

4.6 Parallelism vs. Communication in MILLIPEDE

There are several aspects of the parallelism-communication tradeoff in the MILLIPEDE system. MILLIPEDE supports running multiple applications simultaneously. Our objective is to compromise between maximum parallelism and minimum communication. Since different applications are not communicating, they should run on different hosts whenever possible. On the other hand, the communication between the components of the application should be minimized without causing load imbalance.

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

The mission of the Migration Servers is to optimize the communication within the application with respect to the decisions of the Daemons. That is, the Migration Servers try
to minimize the amount of communication caused by the DSM mechanism without breaking the load balance achieved by the Daemons.

The algorithms used by the Migration Servers in order to minimize the communication are described in [Schuster and Shalev (Wolfovich), 1997]. In the following section we describe in detail the algorithms used by the Daemons to distribute an application over the network.

5 Distributing an application

The daemons are in charge of determining the set of hosts for executing an application. The objective is to execute different applications on different hosts whenever possible, since different applications do not communicate, while the components of the same applications do. Therefore, if an underloaded host already runs a certain application, then the algorithm tries to send it additional jobs of the same application. Only if this is not possible or the host is not executing any application, a new application instance may be started on this host.

Each application is executed on a subset of the available hosts. Initially only the main copy of the application is created; if, as a result, the local host becomes overloaded, and the overall load of the system is sufficiently low, additional copies of the application are eventually created on underloaded hosts. The reverse process is initiated when the load decreases so that there exist more than a single underloaded host running the application. In this case two such hosts are chosen; the application copy of one of them is forced to migrate the jobs to the other copy and is disabled in order to make it possible for another application to use this machine.

5.1 Information Policy.

In order to achieve speedups, only idle or slightly loaded hosts should be used for remote execution. Therefore, certain amount of global information about the hosts' load state is needed to decrease the number of incorrect decisions. However, maintaining exact information about all hosts in the system is extremely expensive. Therefore each host reports to peers only significant changes in its load state. In addition, since the decisions on distributing an application are made by its master, only master hosts need the global load information. Thus, each host that becomes a master or stops being a master reports this change to all the hosts in the system; each host maintains a list of all master hosts and reports to them about all significant changes in its load state. Since coarse-grain applications are assumed, the state of the hosts is expected to change seldom, so that the overhead associated with this policy is relatively low.

5.1.1 Load indicator

Load state of a host is determined by two factors. The first is the CPU utilization by non-millipede processes and the presence of interactive work on the machine. In order to provide the user-ownership feature, millipede avoids using a host for remote execution if its native load (caused by non-millipede applications) is high or if the host is used for
interactive work. The second factor is the load caused by \texttt{MILLIPEDE} applications. Since we assume CPU-intensive applications, the load state of a host is determined by the total number of \texttt{MILLIPEDE} jobs running on the host. Other possible sources of information are:

- The number of jobs waiting for synchronization (e.g., when using ParC statements such as \texttt{sync}).
- The CPU time consumed by the jobs and by paging or migration of threads.
- Memory utilization.
- Network utilization.

Using these factors to determine the load would possibly provide better estimation. However, it would impose higher overhead, so that it would not necessarily improve the speedups. Since we concentrate on other issues in this research, the problem of load indexes is still open in \texttt{MILLIPEDE}.

\subsection*{5.1.2 Host states}

The state of a host is determined according to the two load factors described above. If the background load is too high or the machine’s owner is working interactively on it, the machine will be come \textit{evicting}. In this case, regardless of the second state component, namely, the load of the host, the machine is not used for remote execution (this may change in future implementations). However, if a \texttt{MILLIPEDE} application was started locally, it may still run on both the remote and the local machines; E.g., if the local machine is overloaded (i.e., there are too many local \texttt{MILLIPEDE} jobs), the masters of the local applications will try to migrate part of these jobs to other machines.

The host load state is determined according to the load that is caused by \texttt{MILLIPEDE} applications. Two thresholds – low and high – are used to evaluate the host load state. We call a host \textit{underloaded} if its load is below the low threshold, and \textit{overloaded} if the load is above the high threshold, and \textit{normal} otherwise. The thresholds are constant for each host and depend only on its hardware parameters, such as number of processors and their speed.

Since the algorithm strives to execute different applications on different hosts, the master hosts should also have some additional information. They should be able to determine which applications are running on which host, and whether they are enabled or disabled (as explained below). Thus, the state of a host is characterized also by the number and the type of applications that are using it.

\subsection*{5.1.3 Information dissemination.}

A new host receives the list of all the master hosts from a host that is chosen dynamically when the system comes up. It then sends the first state message to all the master hosts. Additional state updates are sent each time the host state (as defined above) changes. The state update message contains load of the host and the list of applications using it. For each such application the message contains its type (enabled or disabled) and the corresponding number of jobs. Note that a change in the number of jobs is not reported immediately.
Rather, it is reported with the regular state updates to all masters; in addition, when a particular application crosses a load threshold, an update is sent to the master host of this application.

The number and the type of applications running on a host are not expected to change frequently; only these changes and significant load changes are reported to the master hosts. In addition we assumed that the expected job lifetime is high and the number of workstations is not too large. Therefore the overhead imposed by the policy described above is relatively low. This overhead might be further reduced by using a filtering algorithm that prevents a daemon from sending unnecessary load updates when the host load state oscillates near one of the threshold values. This optimization is not yet implemented in millipede.

5.2 Master Protocol

We now describe the scheduling algorithm that the masters use for distributing their applications. This algorithm decides which, and how many, threads will be assigned to each host. The master common data structures (on a host) are updated each time the daemon receives a state update message. Then each master running on this host makes migration decisions regarding its application (if there are underloaded hosts in the system).

Let us describe the response of a master of application \(a\) which resides at host \(H_0\) to a state update message from a host \(H_1\). We denote the number of jobs belonging to the application \(a\) that run on the host \(H_j\) by \(n_j\). We denote the low and the high load thresholds of the host \(H_j\) by \(l_j\) and \(h_j\), respectively.

5.2.1 Treating an overloaded host

When the master receives a message from an overloaded host \(H_1\), it checks if that host is running the application \(a\). If not, it takes no further action. Otherwise the master tries to initiate a transfer of all or part of \(a\)'s jobs out of \(H_1\), depending on the number of these jobs (denoted \(n_1\)).

The master makes the decision in the following way. It determines the number of jobs to transfer (denoted \(n\)), which depends on \(n_1\) and possibly on other load parameters. If \(n_1 < l_1\), the master tries to evict \(a\) from \(H_1\), i.e., it attempts to transfer all jobs of \(a\) from \(H_1\). If the transfer succeeds, the master disables \(a\)'s application copy in \(H_1\). In this case \(n = n_1\). Otherwise it tries to transfer excess jobs from \(H_1\). In this case the exact number of jobs to be sent depends on both \(n_2\) and on load thresholds of the target host \(H_2\) (that is chosen as described below): \(n = \min\{n_1/2, \frac{l_2-h_2}{h_2}\}\).

The master then looks for an underloaded host that can receive the jobs. It may create an additional copy of \(a\) on some underloaded host if this is necessary. However, its objective is to avoid creating redundant instances and to avoid executing several different applications on the same host. The master therefore looks for a preferred underloaded host using the following precedence order.

1. The hosts that have only an enabled copy of \(a\), sorted by the number of the jobs of \(a\) in increasing order.
2. The hosts having an enabled copy of a and disabled copies of some other applications, sorted in the same way.

3. The hosts having an enabled copy of a and some other applications, sorted in the same way.

4. The hosts having a disabled copy of a and nothing else.

5. The hosts having a disabled copy of a and disabled copies of some other applications.

6. The hosts that do not have any copies.

7. The hosts having only disabled copies of other applications.

8. The hosts having enabled copies of other applications.

The master then asks the chosen host $H_2$ whether it can receive $n$ jobs. $H_2$ may refuse if someone else already offered it enough jobs, or if it has become underloaded recently, and it has no copy of a, but it has some copies of other applications. In the latter case it expects job offers from the masters of existing applications. If it remains underloaded for long enough without receiving such offers, it assumes that it will not receive jobs from the existing applications, and consequently it may agree to create a copy of an additional application.

If the underloaded host refuses, the master asks the next underloaded host. For a certain period of time the master keeps the information that the host refused to receive work, assuming the reasons for this do not frequently change. This allows the master to avoid sending useless work offers to such hosts at the next time it looks for an underloaded host.

If the selected underloaded host agrees to receive work, the master checks whether $H_1$ agrees to send the jobs out. The reason for this extra check is to avoid useless migrations in the case that several applications use $H_1$, and all the corresponding masters decide to send their jobs out of $H_1$. In this case, if there is no negotiation with $H_1$, these masters decisions may cause $H_1$ to become underloaded, and so immediately following the transfer of the jobs out of $H_1$ the master will start the process of collecting jobs back to $H_1$.

Consequently, the master asks $H_1$ if it can send $n$ jobs. The daemon on $H_1$ checks whether it still has sufficient number of jobs. If sending $n$ jobs will make it underloaded, it refuses; the master then cancels its offer to $H_2$. Otherwise it agrees; in this case the master sends $H_2$ a copy of a (or enables an existing one if needed) and sends $H_1$ a final request to send work. Upon receipt of this request the daemon on $H_1$ asks the Migration Server of a’s local copy to send $n$ jobs to $H_2$. The Migration Server selects the jobs with respect to their remote access history, striving to achieve maximal locality of DSM accesses. The general method for the selection process, as well as its implementation in millipede, is described in [Schuster and Shalev (Wolfovich), 1997]. Then the mgs transfers the selected jobs to $H_2$ and notifies the local daemon, which notifies the master. If the application is evicted from $H_1$, the local daemon also disables its copy.
5.2.2 Treating an underloaded host

When the master on $H_0$ receives a message from an underloaded host $H_1$, it looks for an overloaded host $H_2$ that has an enabled copy of $a$. If there exists such a host, the master tries to transfer the excess jobs from $H_2$ to $H_1$ in the same way as in the previous section.

Otherwise, all the hosts that are executing $a$ are in either underloaded or normal load state. The master checks if $H_1$ is running the application $a$. If it does, it tries to find another underloaded host $H_2$ that is executing $a$, attempting to merge the two copies into a single host (thus eliminating unnecessary communication between them). It decides which of the two hosts will receive the work from the other one, by using the same precedence order that was described in Section 5.2.1. The other host will evict $a$ to the chosen host. The rejection mechanism is used here too, meaning that both the sender and the receiver may reject the master request due to recent changes in their load state.

6 Performance evaluation

In this section we present the results of our experiments with the millipede system. We show that thread migration can be used to improve load balancing and to reduce the amount of communication. We executed the tests on six X86 Pentium workstations running the Windows-NT operating system, and connected by 100-Mbps Ethernet. The workstations have different amount of physical memory and different processor speed. The average latency of thread migration in this environment is 70ms, while the latency of a message of “zero” length (for example, an mjec message or a page lookup message) is 2ms.

6.1 TSP problem

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

In the parallel algorithm work is divided among threads in the following way. For each node $0 \rightarrow i$ its subtree is searched by $k$ threads; the sons of the node are evenly divided between the threads, so that each thread receives a set of initial paths of the form $0 \rightarrow i \rightarrow j$, where its mission is to search for the minimal path in sub-trees of these paths. Each thread performs a DFS-style search in each of its subtrees; the search is exhaustive in the worst case. In order to optimize the search, all threads use a shared variable to store the weight of the shortest path; a thread cuts off the search in a certain subtree if the weight of the
partial path at the root of that subtree is greater than the weight of the shortest path that was found so far.

Each thread uses a certain amount of dynamically allocated memory. This memory should be allocated in the DSM to make thread migration possible. Depending on the DSM allocations size, some of the threads might get their memory on the same pages, which is a typical example of false sharing. We compared three different variants. In the first variation, denoted NO-FS, false sharing is avoided by allocating more memory than necessary (the allocations are padded to precisely fit into a page). In the other two variants, \( k \) 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 other one, called OPTIMIZED-FS, uses optimizations for data locality by enabling the histories mechanism.

6.1.1 Improving Load Balancing

Uniform input. We show now that migrating threads can improve performance even in cases that are trivially parallelizable. We compare execution time of the NO-FS variant of the TSP for two different scheduling strategies: static and dynamic. The TSP application receives a uniform input, i.e., all paths have almost the same length. Therefore all the threads have about the same amount of work, and also there is almost no communication between them. The static policy is a round-robin strategy: when \( n \) threads are to be created in a system of \( m \) machines, \( n/m \) sequential threads are created on each host. The dynamic policy is our load sharing strategy. Since the system is not uniform, thread migration improves performance by about 30% as one can see in Figure 5.

Evidently, one can suggest to improve the round-robin strategy by dividing the threads between the machines according to their respective performance. This might help, assuming that the expected execution time on each host can be accurately predicted. However, such prediction is only approximate, even if all threads have exactly the same amount of work.
The reason is that the prediction depends, in addition to the number of processors and their speed, on the amount of available physical memory, and on the behavior of other processes that are using the machines. Certainly, no improvements to the prediction help if the amount of work in the threads is not known in advance. As an example, we examine below an extreme case that cannot be treated by the static policy.

**Unpredictable computation amount.** Here we compare the static and dynamic policies applied to the TSP with extremely non-uniform input: the paths that are searched by the first six threads (among a total of 36 created threads) all have the same length, while the other paths begin with heavy edges, so the threads that search them terminate almost immediately due to pruning. Thus, when the static policy is applied in the system with at most six machines all jobs that do not terminate immediately will be scheduled to run on the same machine, while all other machines will become idle shortly after initiation. In contrast, the dynamic load sharing keeps all the machines utilized. Figure 6 shows that with the static policy there is no speedup at all, while the dynamic policy provides speedup close to linear.

### 6.1.2 Optimizing locality

Improper placement of communicating threads can impose huge communication overhead and significantly increase execution time. Table 7 summarizes the results of running the TSP algorithm on six machines with different numbers of threads contending for the same page. The table shows a dramatic reduction in the network traffic when the optimizations for locality are applied: the number of DSM-related messages, which reflects on the \( \text{miss-ratio} \) (ratio of remote to total number of accesses), drops by a factor of 30 to 40! Note that the number of extra messages that are added by the locality optimization mechanism itself is negligible compared to the improvement in the number of DSM-related messages.

### 7 Discussion

In this work we have shown that thread migration is one of the major capabilities of DSM systems. Systems that do not support migration of threads (or processes) may suffer bad performance due to load imbalance and due to improper placement of threads in the distributed environment.

Implementing preempted thread migration is not a simple task. Several approaches for implementing thread migration were introduced in the literature but generally they are inappropriate for most of the operating systems, and in some cases they are even incorrect. Because the portability issue in DSM systems is very important, it is vital to employ a solution that is guaranteed to be portable to various operating systems.

In this paper we proposed a correct system design for thread migration. We described the common approaches and discussed their advantages and flaws. Our proposed solution is the less demanding from the underlying operating system and thus is the most appropriate to be implemented on large varieties of operating systems, thus making the DSM system portable to many platforms.
In order to prove our design, we implemented it on the MILLIPEDE DSM system, under Windows-NT. The MILLIPEDE system allows better utilization of a network of single-processor and multiprocessor machines. It provides a simple interface for multithreaded concurrent programming on such a network, so that the network is viewed by the application as a single multiprocessor machine with shared memory.

Transparent thread migration is used in MILLIPEDE to provide dynamic load sharing while decreasing communication overhead by improving the locality of data accesses in an application-transparent way. In addition, migration is used to capture idle machines, and to provide user ownership on his personal machine by evicting remote threads and data from a machine when its user starts using it.

References


Figure 5. NO-FS TSP with uniform threads in non-uniform environment for static and dynamic scheduling.

Figure 6. NO-FS TSP with static and dynamic scheduling for extremely non-uniform input
Table 7. This table contains statistics regarding applying locality optimization to the TSP application running on 6 hosts with false sharing for different $k$ (number of jobs contending for a page). Applying locality optimizations decreases dramatically the number of DSM messages (page lookup and transfer). The added overhead imposed by ping-pong treatment mechanism and increased number of thread migrations is negligible.

<table>
<thead>
<tr>
<th>$k$</th>
<th>Optimized?</th>
<th>Number of Messages</th>
<th>Ping-pong Treatment Messages</th>
<th>Thread Migrations</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Yes</td>
<td>5100</td>
<td>290</td>
<td>68</td>
<td>645</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>176120</td>
<td>0</td>
<td>23</td>
<td>1020</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>4080</td>
<td>279</td>
<td>87</td>
<td>620</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>160460</td>
<td>0</td>
<td>32</td>
<td>1514</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>5060</td>
<td>343</td>
<td>99</td>
<td>690</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>155540</td>
<td>0</td>
<td>44</td>
<td>1515</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>6160</td>
<td>443</td>
<td>139</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>162505</td>
<td>0</td>
<td>55</td>
<td>1442</td>
</tr>
</tbody>
</table>