Migration of Threads Containing Pointers in Distributed Memory Systems

Saisanthosh Balakrishnan
Electrical and Computer Engineering
University of Wisconsin at Madison
Madison, WI 53706, USA
santhosh@cae.wisc.edu

Karthik Pattabiraman
Department of Information Technology
Sri Venkateswara College of Engineering
Sriperumbudur, Madras, India
karthik_p@vsnl.com

Abstract

Dynamic migration of lightweight threads support both data locality and load balancing. However, migrating threads that contain pointers referencing data in both the stack and heap, among heterogeneous systems remains an open problem. In this paper we describe the design of a system to migrate threads with pointers referencing both stack and heap data. The system adopts an object-based approach for maintaining consistency because this provides fine-grained sharing with low overhead. Also, it does not rely on the operating system's virtual memory interface to detect writes to shared data. As a result, threads can be migrated between processors in a heterogeneous distributed memory environment without making any modifications to the operating system and in a totally transparent way to the migrating thread.

1 Introduction

Lightweight threads, or user level threads, have become increasingly popular over the last few years due to their shorter context switch time and portable implementation. This is especially so for threads running in a distributed memory environment. Thread migration is the process of moving or migrating a thread from one processor to a remote processor. This provides dynamic load balancing and failure recovery in a multi-threaded environment.

One of the most difficult problems while migrating the state of a thread is dealing with pointers in the migrant thread. If the pointers refer to data in the thread's stack, then they will only remain correct if the stack is placed in the same memory location on the destination processor as on source processor. Providing this assurance can be very difficult and inefficient. A similar situation exists for pointers that reference data in the heap.

There are several "solutions" that have been proposed for this problem. Some systems do not allow the use of pointers in migratable threads, while others allow pointers to become undefined following migration [7] [6]. These "solutions" restrict the use of such common data structures such as multi-dimensional arrays, and are not practical.

Another solution is to perform allocations identically on all processors, reserving the memory locations for each thread and its associated data in case a thread must migrate [4] [5]. In this approach when a thread migrates, all its associated data must be stored in the same memory location on the destination processor as on the source processor. This results in severe memory restrictions on the system. Moreover,

the number of threads is limited by the memory capacity of a single processor, regardless of the total number of processors.

Our system explores a general approach which allows dynamic thread migration at arbitrary suspension points and direct-access pointers for both heap and stack data without the above restrictions [1]. The design keeps track of all pointers so that their values can be updated upon migration to reflect the new data locations. However, this approach requires the variables to reside in a contiguous block of memory and migration involves moving the entire heap of the thread to a different processor which can impose a huge overhead.

In this paper we outline a method to migrate only the pointer data which has actually been allocated. We do not require the allocation to be contiguous. The migration is done in a way totally transparent to the migrating thread at arbitrary suspension points. The nucleus of the paper is a system to allow sharing of data among threads independent of the processors the threads are running on. Our approach is similar to object based DSM's like Rthreads [2]. We show how this can greatly aid the easy migration of threads, while at the same time maintaining the consistency of shared data. This is the main technical contribution of our work.

We implement an object based approach to provide data sharing and consistency. Object based DSM's [3] rely on the compiler and runtime system to detect writes to shared data without invoking the operating system. No virtual memory operations are required. Thus object-based DSMs can be implemented using message-passing environments such as PVM and MPI which have a high degree of portability and support heterogeneous workstation clusters. Experimental studies in literature have shown that the object-based method has low average write latency and supports fine-grained sharing with low overhead. Also, it avoids many of the problems encountered with page based DSM's such as thrashing and false sharing.

In the following sections, we describe the design of a novel dynamic thread migration model in the presence of pointers to both heap and stack data. Since the focus of this paper is on pointer manipulation, we omit the other details of thread migration, such as selection of the thread to be migrated, selection of destination processor, and how the actual sending takes place.

The remainder of the paper is organized as follows: Section 2 provides a description of the system architecture and components. Section 3 details the algorithms used in the precompiler while Section 4 describes the runtime system. Section 5 summarizes the rationale behind the various design decisions we made and Section 6 provides a summary and scope for future work.

2 System Description

The entire system is built on top of an operating system independent communication library like MPI or PVM. At the minimum, the library should support point to point messages from one node to another. The system invokes the standard message passing primitives provided by the library to provide synchronization and sharing of data in a portable way. Figure 1 provides a conceptual view of the system.

The system consists of 2 components - A precompiler and a stub. The precompiler operates on the source program and inserts function calls at appropriate places in the code to convey information to the runtime system. The runtime system consists of a stub, which executes on a per processor basis. This takes care of the remote service requests arriving at the processor.

We use a global table *gtable* to map variables to processors. It provides a mapping between the variable name and the current processor it resides on. When a processor wants to read shared data, the owner of the data is first determined from the *gtable* and, if it is not local, a remote data access request must be made. Our *gtable* is similar to the one used in [2], but with one important difference :- In order to provide

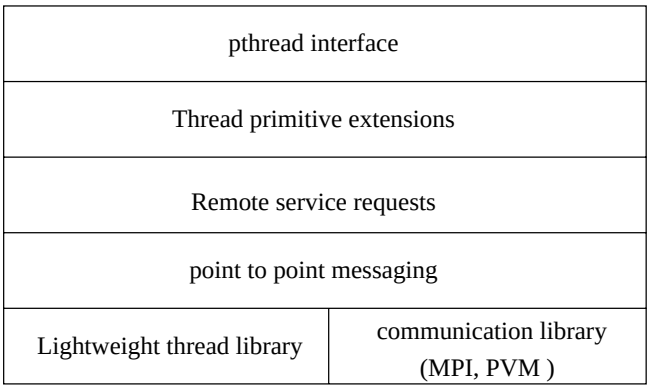

Figure 1: This is a conceptual view of the system. The upper layers consist of the pthreads library interface while the lower layers take care of the message passing and remote service requests

support for pointers, we add an extra field to the *gtable*. This holds the number of levels of indirection which must be applied to the pointer to access the data it points to. This field is initialized to zero for non-pointer data.

2.1 Precompiler

The system provides primitives for control and synchronization of remotely executed threads and specific primitives for remote access of scalar variables, arrays, structures and pointers. The synchronization is implemented to work between threads on different machines and ensure sequential consistency. Explicit remote read or write operations (provided by *read* and *write* function calls) are used to share data between the threads. All these are provided in the form of function calls in a runtime library written by us.

These function calls are inserted into the source code by a precompiler. The precompiler identifies the synchronization points in the program (see next section) and inserts the appropriate function calls at these points in the source program. The precompiler also identifies the variables used and their types and inserts function calls to enter this information in the *gtable* at runtime. Thus the *gtable* is used to communicate information from the precompiler to the runtime system. The use of a precompiler avoids the need to make any changes to the compiler which may render the system non-portable.

2.2 Stub

Most requests require some acknowledgment to be sent back to the requesting thread, such as value of a remote fetch (request for data from the address space of another thread) or the return value from a remote procedure call. We require a polling mechanism by which remote service requests can be checked. Since the main problem with the requests is that they arrive at a processor unannounced, we need a new thread, called the stub which is responsible for receiving all remote service requests. The stub repeatedly issues nonblocking receive requests for any remote service request message. When a request is received, the stub immediately assumes a higher scheduling priority and processes the request. The stub is also responsible for processing requests to maintain the consistency of the *gtable*.

The stub also takes care of the addressing issue. Each stub contains a mapping table indicates which thread is currently executing on which processor. All requests by a source thread to a target thread are

first directed to the local stub at the processor the source thread is running on. The stub then determines the processor at which the target thread is currently running on and sends the message to this processor. This means that when a thread needs to be migrated, it must send a message to all other stubs informing them of the processor it is going to be migrated to, so that the stubs may update their mapping table accordingly.

The runtime system merely refers to each thread by a globally unique integer and lets the stub determine which processor it must be mapped to.

3 Precompiler

The programmer first develops a local version of the distributed program using pthreads. This gives him the opportunity to test and debug the newly created code locally. He must then decide which tasks to distribute and which to process by local pthreads. By default, threads with low communication needs should be transformed to remote threads, by substituting the pthreads function calls with their distributed equivalents (provided by our library). After this the programmer runs our precompiler on the source program.

The precompiler extracts important information from the source program like type information, number of levels of indirection for pointer data and read/write operations on variables using this information, it inserts calls to our library functions at appropriate places, transparent to the programmer.

The following is a detailed description of how the precompiler processes the source program.

3.1 Handling of Shared Variables

All global variables in the pthreads program become shared variables. The precompiler creates a numerical identifier or label for every defined global variable of the program and combines these labels to an enumerated type. Second, it generates a *gtable* with one entry for global variable, the array is sorted corresponding to the enumerated type. The most important information in each entry is the size of the global variable, levels of indirection (in case of a pointer) and the processor that owns the variable (initially zero). The precompiler declares the shared variables not locally declared in the thread as local variables. At the time of creation of a thread, the parent's *gtable* is inherited by the new migrant thread. During runtime, the *gtable* is extracted from the header file and appropriate updates (eg. indicating size of the dynamically allocated variables) are made. An example *gtable* is shown in Table 3.1.

Variable Name	Size	Levels of Indirection	Source Processor
i	4	0	1
data	40	1	2
p	4	2	5
m	100	0	3

Table 1: example "global" table (gtable)

```
int i;
...
thread_main () {
    // global variables declared as local variables
    read ("i", &i);
    i = i + 10;
    write ("i");
}
```

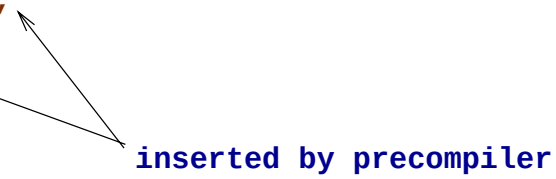

Figure 2: precompiler steps for normal variable

```
int **p = 5;
...
thread_main () {
    // global variables declared as local variables
    **p = 4;
    write ("p");
}
```

Figure 3: precompiler steps for pointers

3.2 Inserting `Read` and `Write`

Then the precompiler analyzes the pthreads program and replaces each pthreads function by the equivalent function provided by our library. Next remote read and write marking functions are inserted. A read marking function is inserted preceding each occurrence of a global variable as rvalue, and a write marking function succeeds each lvalue use of a global variable in the pthreads program. An explicit read/write function is called with an identifying label as parameter. The label is used as an index to the gtable array. Examples with normal variables, pointers and arrays are shown in Figures 2, 3 and 4 respectively.

3.3 Handling of Functions

A mapping table is created for parameters passed by call by reference. This table is used by the precompiler to identify the global variables and to insert `read` and `write` calls are appropriate locations. Also, the scope of the precompiler now includes the thread and also the function. If the parameters are passed by call by value then no calls are inserted for reads or writes for that variable. See Figure 5 for an example.

4 Runtime system

The following section details how the runtime system provides sharing of data with *read* and *write* calls inserted by the precompiler.

In the case of a *write* call on a variable, the thread sends a write invalidate message for that variable to all the other processors. The stub on the each of the other processors, upon receiving the invalidate

```
int **data, *sum;
...
for (int i = 0; i < rows; i++) {
    thread_create (i);
}
...

thread_main (int i) {
    //global variables declared as local variables

    int j, sum;
    read ("data", &data);
    for (j = 0; j < columns; j++)
         sum = data [i][j];
    sum [i] = sum;
    write ("sum", &sum);
}
```

Figure 4: precompiler steps for arrays

message, marks the *gtable* entries for that variable in the threads executing in its processor as invalid. It also updates the owner of that variable in all the thread's *gtable* to be the requesting thread. The requesting thread also modifies its *gtable* entry to identify itself as the owner of the variable.

A *read* call on a variable first locates the owner of the variable from the *gtable*. If the owner is not the calling thread's processor, it sends a remote read request to the thread which owns the variable. The stub at this processor, upon receiving this request, reads the value of the variable from the local address space of the owning thread and sends this value back to the requesting thread. The requesting thread updates the value of the global variable, which is then locally allocated and cached until it is invalidated by a `write` to the variable.

Special mechanisms are needed to relocate pointers and arrays. These are now described in detail:

4.1 Pointers

The gtable entry for the pointer contains the maximum levels of indirection. We assume that all reads and writes, to any level of indirection, relocates the pointer by recursively allocating memory on the requesting thread's heap and storing the final value. This is done transparently to the compiler so that memory accesses are processor independent. Although, reads to any level of indirection of the pointer is allowed, writes to intermediate levels of indirection of a shared pointer variable may not work.

When a pointer variable needs to be sent to another thread, the data value it points to is dereferenced using the indirection levels specified in the *gtable*. This value is alone sent to the destination thread. The destination processor receives the data and copies it into its local memory. It then adjusts the pointer to point to this new location of the data in the thread's address space, according to the number of levels of indirection for that pointer variable specified in its copy of *gtable*.

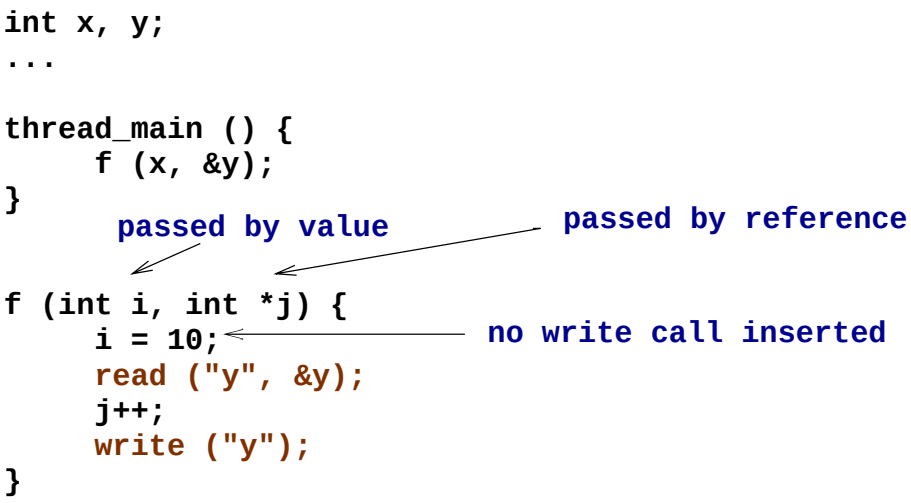

Figure 5: precompiler steps for function calls

4.2 Arrays

The system supports sharing of n dimensional arrays by relocating the entire array to the requesting thread. It is essential that we move the entire array for every reference as compilers assume that an n-dimensional array is stored in contiguous locations. In order to ensure this, the programmer has to use our function $nmalloc(n_1, n_2, \ldots, n_m)$. Reads or Writes to any of element of an array, relocates the complete array to the requesting thread. The requesting thread , on receiving the contents of the array, allocates the array in its local memory using the *nmalloc* function and copies the contents to this location. In the case of large arrays, the user can avoid the overhead in migrating the entire array by allocating the higher dimensions manually. This provides fine grained sharing of the array, but at the cost of increasing the number of entries in the *gtable* for the array.

5 Design Issues

This section summarizes the various design decisions we made and the tradeoffs involved. The major motivating factor at all times was to keep the system as portable as possible, without compromising seriously on efficiency.

The first and probably most important decision was to allow source level modifications to the program. The alternative was to modify the code segment. But the latter option would make the system operating system and machine dependent. Further it would depend upon the way the compiler handled pointers. So we decided to go in for the former option by providing a precompiler and a runtime library. The precompiler's function is to insert calls to our library functions at appropriate places, transparent to the programmer. This avoids the need to introduce language additions and restrictions which may complicate the programmer's task. This solution allows the large existing body of parallel programs written using pthreads to be ported to our system without any modifications and merely by recompiling them.

The second choice we made was to treat entire arrays as one chunk and move them between threads on every access to the array between threads. As threads tend to access array elements in groups, rather than make isolated references to the array, this reduces the number of messages passed significantly. However

this comes at the cost of the overhead of moving the entire array between threads, but the latency of communication is incurred only once. Further it reduces the amount of bookkeeping we need to do, to keep track of individual array elements, thereby reducing the size of the gtable significantly. It is a simple task to split arrays into smaller chunks based on the level of indirection. But this comes at the cost of maintaining individual gtable entries for each level of indirection.

Another design decision confronting us was the approach to take in maintaining the consistency of shared data. We decided to adopt the more widely used write-invalidate protocol as opposed to the write-update protocol. Our decision was motivated by the experience of similar object based DSMs. We are currently evaluating the effects of this decision on system performance.

6 Conclusion

Despite the usefulness of dynamic thread migration, supporting pointers during thread migration has received little attention. The paper introduces a novel approach which provides support for thread migration in the presence of pointers. We have outlined the steps required to ensure that all pointers are updated following a migration. This includes pointers that are located in either the stack or heap and which reference data in either stack or heap. The system is designed with portability and efficiency in mind.

Future work will improve both the efficiency and functionality of our design, and compare it with existing systems that offer dynamic thread migration. We also plan to devise a runtime interface that is data structure intelligent, and migrates structures such as linked lists, trees etc. efficiently.

References

[1] D. Cronk, P. Haines, P. Mehrotra: Thread migration in the presence of pointers, *Proc. of the 13th International Conference on Systems Science*, 1997, Volume 1, pp. 292 – 298

[2] B. Dreier, M. Zahn and T. Ungerer: Parallel and Distributed programming with Pthreads and Rthreads, *Proc. of 3rd International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998, pp. 34 – 40

[3] M. Zekauskas, W. Sawdon, and B. Bershad: Software write detection for a distributed memory. In *First Symposium on Operating Systems Design and Implementations*, 1994.

[4] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Little,eld: The Amber system - Parallel programming on a network of multiprocessors. In *ACM Symposium on Operating System Principles*, December 1989.

[5] J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole: Adaptive load migration systems for PVM. In *Proceedings of Super-computing '94*, pages 390-399, Washington D.C., November 1994. ACM/IEEE.

[6] R. Namyst and J.F. Mehaut: PM2 - Parallel Multi-threaded Machine. *In Proceedings of ParCo '95*, September 1995.

[7] E. Mascarenhas and V. Rego: Ariadne - Architecture of a portable thread system supporting mobile processes. Technical Report CSD-TR-95-017, Purdue University, March 1995.