BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-97-M1

"A Distributed Threads Package for Solaris 2.4+"

by

Charles G. Hoecker

A Distributed Threads Package for Solaris 2.4+

Charles G. Hoecker

Department of Computer Science
Brown University

September 1, 1996

Submitted in partial fulfillment of the requirements for the Degree of Master of Science in the Department of Computer Science at Brown University

[illegible]

Professor Thomas Doeppner
Advisor

A Distributed Threads Package for Solaris 2.4+

Charles G. Hoecker
Dept. of Computer Science
Brown University
cgh@cs.brown.edu

September 1, 1996

Abstract

Multithreaded programs have become a popular method of parallel programming, allowing concurrent tasks to be executed by a single process. Many multithreaded programs will run faster on a machine with *n* processors than they will on a machine with *n-1* processors and hence their performance is bounded above by the number of processors on the host machine. Moreover, multiprocessor hardware is expensive and many organizations instead have a numerous uniprocessor machines connected over a network. As these machines and networks have become more powerful, they have become a viable platform on which to execute distributed parallel programs. Recent operating system improvements have allowed us to develop a *distributed threads package* that simulates a "virtual process" over these networked machines, allowing multithreaded programs access to more processors than are physically on the host machine. While some multithreaded applications require the tight communication and synchronization latencies provided by multiprocessor hardware, many others perform well over the network configuration and our package.

I. Introduction

Parallel programs are written to improve the performance of solving a problem or accomplishing a task by increasing the computational speed, throughput, responsiveness or some combination of the three. They take advantage of the concurrency of multiple processors in a multiprocessor system or of separate systems connected by a network. Since multiprocessor hardware is expensive, the latter configuration is referred to as "cheap parallelism" and employed by a number of packages such as PVM and MPI. Programs written with these packages distribute processes across a set of machines with inter-process communication consisting of messages sent over a network.

A multithreaded program, however, runs as a single process on a given machine and consists of multiple threads of control that access the memory in the address space of the process. A multithreaded program will only achieve true concurrency when run on a multiprocessor machine where more than one thread can execute instructions at any given time by running on a separate processor. The concurrency of a multithreaded program, however, is limited by the number of physical processors on the machine. The only option a multithreaded programmer has to achieve more concurrency than is available on a single machine is to rewrite the program and use a distributed explicit message-passing package such as PVM or another distributed shared memory (DSM) system such as Treadmarks. Both options require substantial changes to the source code.

To provide the multithreaded programmer another option, we developed a distributed threads package that takes advantage of the resources of a set of networked computers. Our package provides distributed shared memory[i] and synchronization primitives including semaphores, mutexes, read-write locks, condition variables, and barriers and requires only minimal revisions of the source code. Currently our package works on the Sun Sparcstation platform running Solaris 2.4 and above.

II. Multithreaded Programming

First, it must be understood that multithreaded programs, and parallel programs in general, are very efficient and desirable in completing certain tasks but can also be slower and inefficient in completing others. In general, multithreaded programs are usually desired in one of two cases. The first is to perform two or more operations at the same time. For example, a word processor does not want the user to wait for a key to show up on the screen while it repaginates; it thus can have a thread waiting to process input while another performs background repagination. A database client program might have a thread waiting for a user query at all times, handing queries off to a set of slave threads that actually execute the requests. The database server might have a separate thread handle each incoming request so that newly arrived client requests do not have to wait behind some prior request that is blocked on I/O. Web browsers will have multiple threads fetching the various elements of a page. This way, a user might be able to read the text of a page while a title graphic or advertisement is still being loaded.

The second desirable use for parallel programs is performing computationally intensive tasks where the work for a problem can be divided amongst several threads or processes in a manner that the different threads or processes spend most of their time running independently and not synchronizing their efforts. A good example of this is a matrix multiplication program. In this case, the only operation that need be synchronized is the building of the solution matrix at the end of all computation as the threads do not need to synchronize the reading of the two input matrices as no thread modifies them. Many iterative algorithms, where the output of one iteration becomes the input of a subsequent iteration, require that all threads performing the first iteration finish before a subsequent iteration is performed. In some cases this synchronization happens infrequently and the threads spend most of their time working. In others, however, it occurs often in which case the cost of synchronization can outweigh the benefit of "parallelization."

In general, any task where the synchronization time would be a small fraction of the computation time is a good candidate for being performed in parallel while others should remain serial. It should be noted, too, that the development and debugging of multithreaded (and parallel) programs can be significantly more challenging than that of serial programs and this, too, should be a factor in deciding which method to use.

III. Performance

Network communication between the remote threads is responsible for retarding the performance of an application using our package and is the result of thread synchronization and memory segment transfer. Thus, in general, multithreaded applications whose threads largely run independent of one another – both in the amount of synchronization performed and the memory they access – perform better with our package than applications with threads whose execution and data access are tightly interwoven.

The following programs were used to test the package:

- **Quicksort** - Performs a quicksort of an array of integers. The data set is not large and threads constantly access each other's data. There is also a fair amount of synchronization.

- **Jacobi** - Performs a jacobi iteration on a matrix. At each iteration, every cell of the matrix becomes the average value of four cells adjacent to it and threads synchronize at a barrier at the end of each iteration. The matrix is divided up by rows evenly amongst the threads which run largely independent of each other. However, the first and last rows of each thread's set are shared by other threads which do not synchronize their access. This leads to a "ping-pong" effect where a page is rapidly transferred between one thread which is writing to it and another which is reading from it. Performance would improve if synchronization were added to avoid this situation.

- **Matrix** - Matrix multiplication. This program constantly calculates the matrix multiplication of A x B = C. At the end of each iteration, one thread substitutes different values for A requiring all threads to re-read the matrix. Like jacobi, the rows are divided evenly amongst the threads but as they do not read each other's data there is no "ping-pong" effect.

- **Shortest Path** - Shortest Path algorithm[1]. This program computes the shortest path between two vertices in a directed graph with weighted edges. The threads constantly read and write to the same memory locations and make many synchronization calls. There is so much synchronization involved that, in fact, the Solaris threads version runs significantly faster on a uniprocessor machine than a multiprocessor machine[2].

[1] - Written by Peng Dai (ped@cs.brown.edu). According to the author, the program is in an "experimental" stage.

[2] - Since only one thread is running at a time on a uniprocessor there is less interruption and, in this case, better performance. It is very likely that this program would run faster if there was only one thread.

Program	Threads	Data Size	Thread Completion Times (sec) Solaris	Distributed	Performance Gain / Loss
Shortest Path	4	-	2.50	507.00	-20180%
Quicksort	2	100000	0.84	9.20	-995%
	3	100000	0.72	12.40	-1622%
	4	100000	0.60	15.40	-2467%
	2	1000000	10.40	60.00	-477%
	3	1000000	7.80	102.00	-1208%
	4	1000000	7.80	132.00	-1592%
Jacobi	2	500x500	49.60	58.00	-17%
	3	500x500	35.40	64.20	-81%
	4	500x500	31.60	82.40	-161%
Matrix	2	500x500	141.00	139.80	1%
	3	500x500	136.00	93.00	46%
	4	500x500	75.80	70.40	8%
	8	500x500	-	37.00	281%
	4	1000x1000	1125.00	683.00	65%
	8	1000x1000	-	338.00	233%
	12	1000x1000	-	224.00	402%
	16	1000x1000	-	181.00	522%
	4	2000x2000	9020.00	5262.00	71%
	16	2000x2000	-	1279.00	605%

Figure 1 - Performance results. The first three programs highlight the pitfalls of using our package. The first two rely on tight communication and/or synchronization amongst the threads and do not perform well with the communication latency inherent in our implementation. Jacobi has sections where the threads rapidly swap pages of memory amongst themselves. The Matrix application using our package narrowly outperforms its Solaris threads counterpart for 4 and fewer threads but allows the program to be run with up to 16 threads. A Sun Sparcstation 10/404ZX (which has 4 processors) was used to gather the Solaris thread data while a network of Sun Sparcstation 10/41GX's connected over a 10 Mbs Ethernet line was used for the distributed thread data.

The multiprocessor results show that the quicksort, jacobi, and matrix programs perform quicker with more threads; the quicksort program's execution improves 28%, jacobi's 36%, and matrix's 46% when the number of threads is increased from 2 to 4. The matrix program scales slightly better with the distributed threads package: 49% when the number of threads is increased from 2 to 4, and 74% when the number is increased from 2 to 8. Furthermore, with the distributed threads package, we were able to use up to 16 threads to solve the 1000x1000 problem which took 27% of the time it took 4 threads with our package and 16% of the time it took 4 threads on our 4 processor machine. Since the matrix multiplication requires only a small amount of synchronization amongst the threads, it exhibits an almost linear speedup with respect to the number of threads used. Its main performance penalty is the writing and dissemination of new values to one of the input matrices, which is 4 megabytes in size for the 1000x1000 and 12 megabytes for the 2000x2000.

What the performance results also illustrate is that a multithreaded program which runs faster on a multiprocessor machine when more threads are used will not necessarily scale as well with our package. While the quicksort and jacobi programs' performance improved with additional threads, its performance worsened with additional threads with our package. This was due to the increased latency penalty that comes with the additional inter-thread communication. When the data sets were larger, and threads spent more of their time on computation, the performance with our package improved. Overall, though, these programs require, as some other multithreaded programs do, very little communication latency to run well.

The matrix multiplication results reveal another strength of our package which is illustrated by the large disparity between matrix program's results with the larger data sets when run on a single multiprocessor machine and the results when the work is distributed across numerous machines. Whereas the 500x500 matrix times between the two platforms for 4 and fewer threads were competitive between the two platforms, the times for the 1000x1000 and 2000x2000 matrices are much larger for the multiprocessor. This is due to the amount of memory required to process the larger data sets. When the data is distributed across numerous machines, each cooperating machine needs to access a smaller portion of the virtual process memory than a single machine needs to. Providing access to this larger portion requires that the virtual memory mechanism of the single machine swap memory to and from disk which hampers performance of the process significantly.

Overall, the results appear to suggest that the common "real world" multithreaded program will not benefit from using this package but this is not necessarily the case. First, as just mentioned, the package will allow multithreaded programs with high memory requirements to avoid the virtual memory swapping and perhaps improve performance. Second, many multithreaded programs, such as the jacobi iteration tested above, can be slightly altered to reduce the effects of network latency and improve performance. The jacobi program would add mutexes for the rows shared by threads so that a thread modifying a page would not be interrupted by another trying to read it. This alteration would allow the jacobi program to perform well with the package.

Additionally, it must be kept in mind that the test hardware used is not all that powerful. The 10 Mbs Ethernet network connecting the host machines in our tests is somewhat slow and 100 Mbs connections are becoming abundant. A faster network connection would drastically improve performance by reducing the inter-thread communication latencies and bottlenecks.

IV. Internal Implementation

The package works by creating new threads in their own processes on remote machines rather than in the processes that spawn them. For instance, a `pthread_create(3T)` call forks the calling process, invokes the `rsh(1)` command to spawn the process on a remote machine, the remote process `mmap(2)`'s in the global shared memory and jumps to the thread procedure with a passed argument. Each remote process contains two threads: one that executes the user's procedures and another that responds to incoming communication from other threads. This second "serving" thread is spawned separately at process initialization.

All synchronization provided by the package, including the coordination of the memory access, is accomplished by an underlying "token" exchange amongst the various thread processes. This exchange is performed by the package and is completely transparent to the user. In this scheme, each process maintains a table of tokens, each of which a given thread may have *read*, *write*, or *no* access to. The package allocates a new token for each synchronization primitive (e.g., a mutex or semaphore) and shared memory segment used by the threads. The following example illustrates the library's actions with mutex calls:

What the user does	**What the library does**

`{`
`  ...`	

`  mutex_t *my_mutex = new`
`    mutex_t;`
`  mutex_init(my_mutex,`
`    USYNC_THREAD, 0);`
	Allocate a new token for 'my_mutex'. Give this thread *write* access to the token.

`  mutex_lock(my_mutex);`	Check thread's access to token corresponding to 'my_mutex'. If the access is not *write*, locate the thread with *write* access, request *write* access from that thread, and wait for access to be granted. Once we have *write* access, mark the token as having a writer so that no other thread may gain *write* access until `mutex_unlock()` is called.

`  mutex_unlock(my_mutex);`
`  ...`
`}`
	Remove the writer from the token. Check to see if any threads are waiting for access. If so, give *write* access to the waiting thread and make our access *none*.

When a thread requires a different type of access to one of its tokens, it makes a TCP/IP connection[3] to the token's "probable owner" (the thread with *write* access) and requests the new access. If granted, the calling (client) thread will change its local access for the token and depending upon the request the serving thread may need to change its own access for the token. In the event that the serving thread no longer has *write* access to the token (it had already given it to another thread), it tells the client to try the thread that the server had already given the *write* access to. The client thread then tries again with this new "probable owner." In the worst case, this token-chasing will reach the true owner of a token in N - 1 tries[ii], though usually there is at most one or two. Figure 2 shows another example, this time with shared memory access.

[3] - TCP was chosen over UDP for simplicity. Since the protocol ensures reliable data transfer quickly at the networking level, our package did not have to. Also, Solaris' TCP/IP implementation is very fast so performance cost is minimal. Also for faster performance, the new connection remains open for the remainder of the execution for subsequent requests for a given thread.

- `#include "dthread.h"` in the source file containing `main()`
- `#include "dthread.h"` in all source files calling `malloc()` and `free()`
- A call must be inserted at the beginning of `main()` to `init_dthread(int argc, char *argv[])`, where `argc` and `argv` are the arguments passed to `main()`.
- The path of the distributed threads package lib/ directory must be added to the linker library path flags.
- Our thread library `libdthread.a`, `libsocket.a`, and `libnsl.a` must be statically linked to.

Once a multithreaded program is recompiled with the distributed threads package, it may be run with additional command-line options. Arguments to the threads package are separated from other program arguments by a double dash "--". Following the double-dash may be one or more of the following optional arguments:

-s [size]	The amount of shared memory available to the package, in bytes. If `size` is smaller than one system page (4096 or 4K), `size` is interpreted as the number of megabytes. The default is 10 megabytes.
-f [filename]	Specify the shared memory file. When specified, the package will create a single shared memory file which all threads private map into their address space. Without this flag, the package will create a file on each system's /tmp directory. This is desirable if not much swap space is available as the file mapping effectively reserves swap space twice when the package creates a mapped file in the swap space.
-m [filename]	Specify the file containing the list of machines to be used by the package. By default, the package will use the `.machines` file in the directory specified by the HOME environment variable. The file contains a carriage-return-separated list of machine names.

For example:

```
% foo [foo arg] [foo arg] -- -s 20 -m /usr/local/etc/machines.lst
```

A file containing a list of machines is required by the package. When a new thread is created, a process is spawned onto the next machine in the list that has not been previously used. Processes are spawned onto machines via the "rsh" command and the user must be allowed to execute the command without supplying a password.

[i] - K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems*, Vol. 7, No. 4, Nov. 1989, pp. 321-359.
[ii] - Ibid, p. 335. *N* is the number of threads.
[iii] - P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. "Treadmarks: Distributed shared memory on standard workstations and operating systems," *Proceedings of the 1994 Winter Usenix Conference*, pages 118-120, January 1994.