

# Efficient Global Pointers With Spontaneous Process Migration

Koji Noguchi, Michael B. Dillencourt, Lubomir F. Bic  
University of California, Irvine  
bic@ics.uci.edu

## Abstract

*We present an approach to implementing and using global pointers in a distributed computing environment. The programmer is able to create pointer-based distributed data structures, which can then be used by sequential or parallel programs without having to differentiate between local and global pointers. Any reference to a remote address causes the process to either migrate to the remote host, where it continues its execution, or to perform a remote access operation. The decision is made automatically and fully transparently to the programmer. By using a hardware-supported memory checking mechanism, we avoid any overhead associated with the detection of remote references.*

## 1. Introduction

Pointers in C provide a powerful and flexible approach for designing dynamic data structures whose size and shape change at run-time, e.g. linked lists, trees, stacks, hashed tables, directed acyclic graphs (DAG), general graphs, etc. Unfortunately, many of the advantages of pointers must be abandoned when running on a loosely coupled cluster of computers. The reason is that few existing systems support global pointers, i.e., pointers that cross the memory boundaries of different hosts, and even systems that do support some form of global pointers impose serious restrictions on their use. The resulting lack of transparency prevents the design of distributed programs that match the elegance—and hence the resulting ease of development and maintenance—of sequential pointer-based programs.

This paper describes an approach to support a global distributed heap without having to differentiate in any way between local and global pointers. There are two key aspects that make this approach efficient at runtime. First, the system detects any remote reference automatically using a memory trap mechanisms supported by the hardware of most contemporary architectures [1]. Consequently, there is no overhead imposed on the use of pointers unless a remote

reference is actually encountered. Second, the system supports two ways of resolving a remote reference, depending on the program's behavior: If the computation following the remote reference is to continue on the remote host, the process automatically migrates to the remote host. On the other hand, if the process simply needs to read or write a single value on a remote host, the system avoids the overhead of migration by performing a remote read or write operation. The process uses a heuristic to determine whether it should migrate or perform a remote access operation. The heuristic is based on the process's past behavior at the same location and the decision is made automatically and fully transparently to the programmer.

## 2. The Programming Model

One of the key questions is how to handle pointers at the programming level. Specifically, does the programmer need to be aware of any distinction between local and global pointers, and, if so, what are the consequences for the development of distributed applications. To illustrate the problem, consider the simple tree traversal function (`treeAdd`) in Figure 1, which recursively adds the values of all nodes of a given tree (created previously using the function `treeAlloc`.) Assume now that the tree has been allocated in a distributed manner over a network of four processors, such that each processor 0 through 3 holds some portion of the tree, as illustrated in Figure 2. The question is: what, if anything, do we have to do to the function `treeAdd` to make it work with the now distributed data structure? There are two main issues to consider.

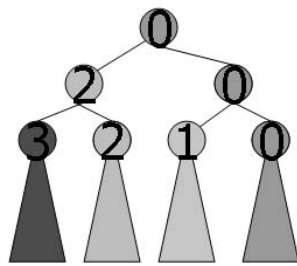
First, the program must be able to recognize a global reference. In the above example, the program starts processing the root of the tree on host 0. The right subtree (`t->right`) is a local reference but the left subtree (`t->left`) is remote: its target is on host 2. There are three possible approaches to this problem. One is to explicitly differentiate between local and remote pointers (by using a different programming notation for each type). The main drawback of this approach, taken in several existing systems, including Cid [7], Earth [3], Olden [10], or MCRL

```

1  main() {
2    tree * head; int result;
3    head = treeAlloc(...);
4    result = treeAdd(head);
5  }
6
7  int treeAdd(tree *t) {
8    if( t == NULL ) return 0;
9    else
10     return t->val + treeAdd(t->left)
11                + treeAdd(t->right);
12 }

```

**Figure 1. Sequential Traversal of Distributed Tree**



**Figure 2. Tree Structure**

[4], is that the original program must be significantly modified (manually or using compiler analysis [6] [10]) such that it always uses the correct reference—remote or local—based on the given mapping of the data structure. It also prevents the mapping of the data structure to be decided at runtime, which is a major limitation on dynamic pointer-based data structures.

Another approach is to detect remote references at runtime by inserting an explicit check into the code in front of every pointer. Unfortunately, this results in a serious performance penalty, as will be shown in Section 5.1.

The third approach—the one taken in this paper—relies on the hardware mechanism provided by all contemporary systems for memory access protection. This triggers an exception whenever a pointer’s destination resides on a remote host and the corresponding signal handler takes the necessary steps to process the remote reference. (Section 3 presents the details of this mechanism.)

The second issue—what to do once a remote reference has been detected—can also be addressed in different ways. One approach is to follow the principles of Distributed Shared Memory [9]: transfer (or copy) the referenced data from the remote host to the local host and continue its processing locally. The problem with this approach is that it defeats the purpose of the data distribution and results in

poor performance. In the above example, host 0 would incrementally pull into its local memory all the other subtrees from hosts 1 through 3 in order to compute the `treeAdd` function.

The only practical approach in situations such as the above is to continue the processing on the remote host. This, unfortunately, is very difficult to achieve with message passing, which is the most popular approach to distributed computing today. It requires a complete rewrite of the original code.

A much better approach—the one relied on by our system—is to use processes capable of spontaneous migration: a process migrates automatically whenever it encounters a remote pointer. Thus a process simply follows any remote pointer to its target host and continues its execution there fully transparently to the user.

Based on the above observation, we provide a programming model based on the following three main characteristics:

```

1  int Par_treeAdd(tree *t, parLevel) {
2    my_sem *ptr_sem;
3    if (t == NULL) return 0;
4    if (parLevel <= 0) return treeAdd(t);
5    ptr_sem =
6     spm_malloc(local_id, sizeof(my_sem));
7    if(fork() ) { //parent
8     tleft = t->val +
9     Par_treeAdd(t->left, parLevel-1);
10     sem_wait(ptr_sem->semaphore);
11     tmp_ptr->val += tleft;
12   }
13   else { //child
14     ptr_sem->val =
15     Par_treeAdd(t->right, parLevel-1);
16     sem_post(ptr_sem->semaphore);
17     exit();
18   }
19   return ptr_sem->val;
20 }

```

**Figure 3. Parallel Traversal of Distributed Tree**

## 2.1. Location-aware Data Allocation

Appropriate distribution of the dynamic data structures is crucial to the performance of any application, because it determines the amount of process migration and of potential parallelism. We do not make any attempt to map dynamic data automatically. Rather, we rely on the programmer to tell the system where to initially allocate any new data item. This is accomplished by providing an alternate function to be used in place of the `UNIX malloc()`:

```
ptr = spm_malloc(daemon_id, size);
```

This function first migrates the calling process to the host specified by the additional parameter `daemon_id`. It then allocates the data item of the specified size on this host and returns a pointer (`ptr`) to this data item just like the conventional `malloc()` function. The main difference is that the pointer is globally valid, i.e., can be carried by the process to other hosts, stored there to build distributed data structures, and used just like a local pointer. It is important to emphasize that `spm_malloc()` is used for *all* data allocations, local and global. The programmer does not need to differentiate between two types of pointers.

## 2.2. Location-transparent Pointer Handling

Given that we do not differentiate between local and global pointers, a process automatically migrates to a new target host whenever it encounters a remote reference. To make this possible without the overhead of any software checks, we rely on the virtual address checking mechanism, which is implemented in hardware. This mechanism, combined with the spontaneous migration of processes at remote pointer dereference, is the main contribution of this paper and will be discussed in detail in Section 3.

## 2.3. Explicit Parallelism and Synchronization

Many applications can benefit greatly from distributing their data structures over multiple hosts even when execution remains strictly sequential. The performance improvements (as shown in Section 5.2) result from reduced paging as each host handles only a smaller portion of the large data structure [8].

Parallelism can of course increase performance further. For this we rely on the programmer to implement parallelism explicitly using `fork` statements. For synchronization, semaphores are used.

Figure 3 illustrates the basic principles by presenting a parallel version of the `treeAdd` function of Figure 1. The additional parameter `parLevel` determines the level of parallelism, i.e., the number of processes created to carry out the task. With `parLevel==0` the function simply degenerates to the sequential version, `treeAdd` (line 4). With `parLevel==1`, a single child process is forked (line 7), which processes the right tree branch (lines 14-15) while the parent processes the left tree branch (line 8-9). With `parLevel==2`, both the parent and the child processes each spawn a new child process, resulting in a total of four processes, each processing a different portion of the tree. Each parent process waits for its respective child using semaphores (lines 10 and 16), created by the parent on the current host.

# 3. Implementing Spontaneous Process Migration

## 3.1. System Architecture

Figure 4 illustrates the basic architecture to support spontaneous process migration with global pointers. The lowest level is the *physical network*, which is a cluster of PCs or workstations communicating with each other using the standard TCP/IP protocol.

The next level is the network of *daemon processes*, one on each participating host. The purpose of the daemons is to isolate the application programs from the details of the physical network topology and to provide the infrastructure for process migration. The daemons are similar to those of other systems that supports explicit strong process migration, e.g. [2]. One of the most important aspects of migration is to keep the overhead low. We eliminate the main source of overhead—the necessity to migrate code—by loading a copy of the application code on every node participating in the computation. Thus only local variables, the stack, and the local heap need to be transmitted. This additional overhead is very small relative to the cost of sending a message.

The third level of the architecture is the *global shared heap*, which is described in detail in Section 3.2.

The fourth level of the architecture comprises the application processes. Each such process can run on any of the hosts. A migration occurs automatically whenever a process accesses a remote pointer, which triggers an exception. The corresponding signal handler issues a migration request to the local daemon, which then negotiates the transfer of the process with the corresponding target daemon.

## 3.2. Global Shared Heap

The daemons jointly implement the *global shared heap*, which is a collection of pointer-based data structures created dynamically by the application processes in a distributed manner. Any process can allocate space on a global shared heap through the `spm_malloc` library function presented in Section 2. Pointers to any portion of the global heap data may be stored locally or on remote hosts, thus allowing the dynamic creation of arbitrary data structures distributed over the different hosts.

To implement the global shared heap, each daemon reserves the virtual address space for the *entire* heap, i.e., the space distributed over all daemons. As shown in Figure 5, this virtual space is allocated between the stack and the local heap area using the `mmap` call. This reserved space is split into blocks, each of which is assigned to a different daemon (0 through `n`). Note that any daemon maps the virtual memory to physical memory only for memory blocks that

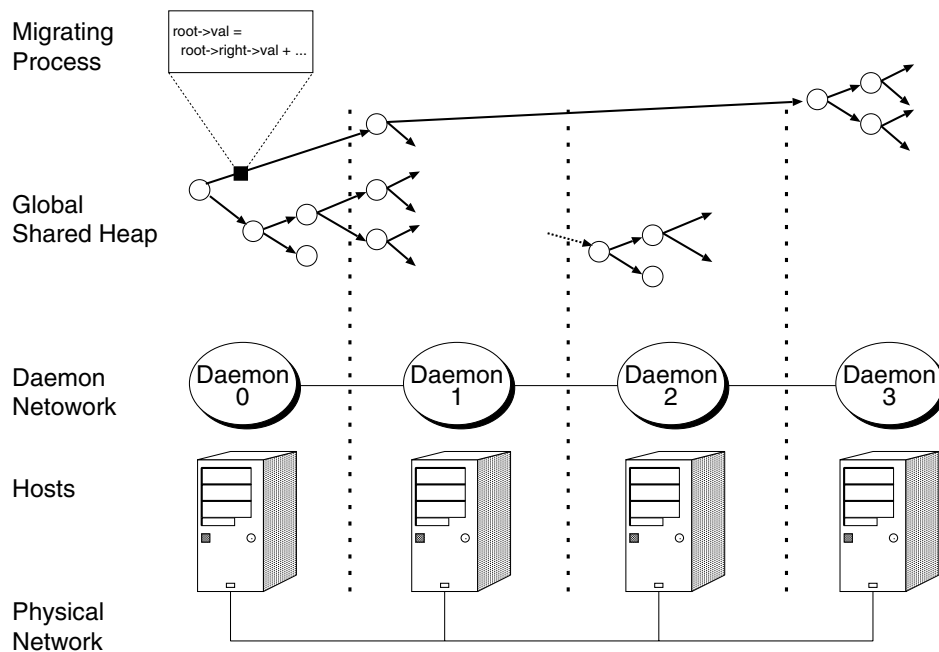


Figure 4. Execution Model

are actually assigned; the remaining space is not mapped to any physical memory, thus no physical memory is wasted.

Whenever a new process is created, its initialization function maps its shared pages to coincide with the shared pages of the current daemon. When a process migrates from one daemon to another, the mapping of the global shared heap changes accordingly before resuming execution. Thus processes can access the global shared space on any daemon transparently without any special function calls. Accessing the portion of the global shared heap assigned to the current daemon is carried out without any interruption. Accessing the space assigned to a remote daemon is detected automatically as described next.

### 3.3. Detecting a Remote Pointer Dereference

The key to efficient transparent migration is to avoid any software checks to detect remote references. Our solution is to use the UNIX virtual address locking mechanism (*mprotect*) supported in hardware [1]. We simply lock all virtual memory pages on every daemon other than those belonging to that daemon. Consequently, any remote reference is an attempt to access a locked page and causes a segmentation fault signal (SIGSEGV). This is caught by our signal handler (registered in the process initialization function). This decides whether to migrate the process to the corresponding remote host or perform a remote access operation, and it carries out the appropriate action. Since the checking of

all pointer references is done in hardware, there is no additional overhead for local pointers.

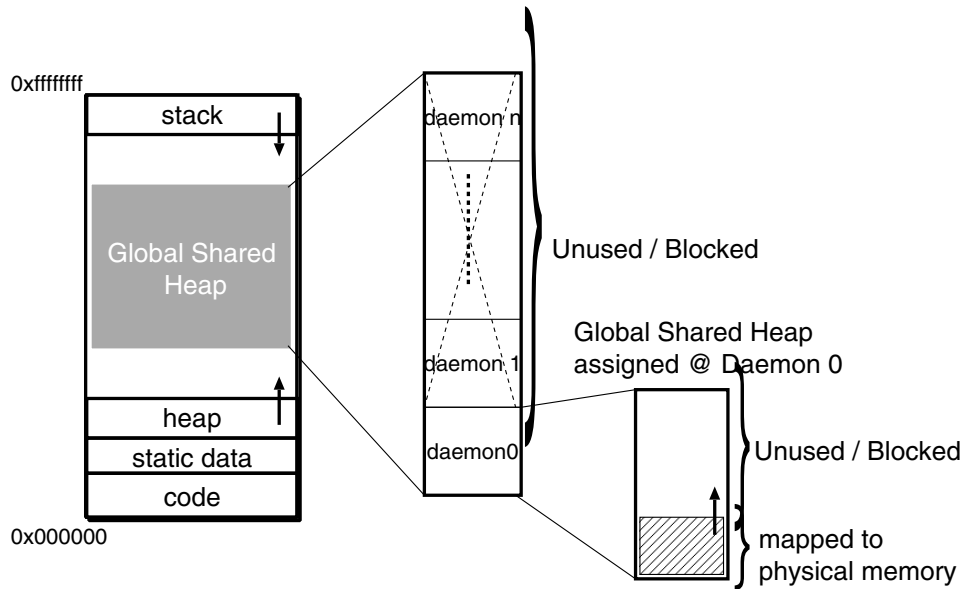
## 4. Remote Read/Write Operations

In section 2, we discussed the advantage of process migration over message passing. At the same time, we need to avoid the overhead of process migration whenever possible. A typical situation where process migration is undesirable is when a process needs to access (read or write) a single value on a remote host but to continue executing on the current host. Without the ability to perform a remote read or write operation, the process must hop to the target host, read or write the desired value, and immediately hop back.

To eliminate this problem, we introduce a remote read/write operation that a process can use to access a word of data on a remote host. The decision on whether to migrate or to use remote data access is performed by the system at runtime using a heuristic, which considers the program's past behavior. The decision and the corresponding action taken are all fully transparent to the user.

To accomplish this, a log is associated with every process, which records the process's past behavior. This is then used to predict its future behavior. Each log entry corresponds to a code address that referenced a remote pointer and thus caused a segmentation fault (Section 3.3). The following information is maintained for each entry:

- The code address that caused the segmentation fault, including the type of operation (read or write)



**Figure 5. Global Shared Heap**

- The time the process spent computing since it migrated to the current host
- The id of the daemon from which the process last migrated
- The id of the daemon holding the remote reference that caused the current fault
- A counter to keep track how many times the process migrated due to a fault at the current code address

Figure 6 shows the extended version of the signal handler, which uses and maintains the above logged information. When invoked, the handler first checks if the faulting code address is already registered for remote read/write (line 4). If so then the requested data is accessed through a remote operation (line 5), instead of forcing the process to migrate. The process then resumes execution on the same host.

If the given code address has not yet been registered for remote read/write, the handler checks if it should be (lines 8-14). If the process is to migrate back to the daemon from which it just came (line 8), and if the computation on the current daemon has been very short (line 9: THRESHOLD1=200ms in the current implementation), then the process is a candidate for a remote read/write. The handler counts how many times the above behavior has been observed (line 11). When that number exceeds a given threshold (line 12: THRESHOLD2=1 for write access and 2 for read access), the address is registered for remote read/write (line 13). As a result, the process will not migrate the next time the same code address causes a fault.

## 5. Performance Evaluation

	No SW check	Pointer Range check	Pointer id check
Time	82.90 sec	108.61 sec	147.84 sec

**Table 1. Overhead of Explicit pointer checking: traversing 600,000 nodes for 2000 iterations**

The performance results presented in this section were all obtained by running on Sun-Blade-100 502MHz 256 MB memory machines.

### 5.1. Overhead of pointer checking

The objective of this section is to demonstrate how significant the savings are when the overhead of checking for remote references is done in hardware as opposed to software.

Table 1 shows the time it took to traverse a link list of 600,000 nodes 2000 times. We intentionally made the linked list short in order to avoid paging. The first column shows the speed of the sequential C code with no software checks. The second column shows the time when a range check is inserted before every pointer dereference. In the third experiment, the pointer is changed to a structure that holds the memory address and the daemon\_id. Before every pointer dereference, the code checks the daemon\_id of the pointer.

```

1  signal_handler(){
2      code_addr: machine code address that caused the SIGSEGV
3
4      if( code_addr is registered for remote read/write ){
5          handle_remote_readwrite();
6          return;
7      }
8      if( previous source daemon == current_target_daemon
9          && computation time < THRESHOLD1 )
10     {
11         increment the log counter for code_addr
12         if( counter > THRESHOLD2 )
13             register code_addr for future remote read/write
14     }
15     handle_migration();
16 }

```

**Figure 6. Signal Handler with migration heuristic**

We observe that even simple range checking can slow down the process up to 30%. Having extra information associated with the pointer is even more costly; we see the slow down of 78%. Our system uses the hardware memory check mechanism, which does not add any runtime overhead. Consequently, our runtime is the same as in the first column: 82.90 seconds.

## 5.2. Speedup Using Data Distribution and Parallelism

Table 2 shows the results of executing the tree traversal programs of Figures 1 and 3. The tree is very large, consisting of 24 levels and requiring 400 Mbytes of space. It is distributed evenly over 1, 2, 4, or 8 hosts, which correspond to the four columns of the table.

24 levels (400Mbytes)		Number of Daemons (Hosts)			
		1	2	4	8
Number of Active Processes	1	584.17	245.08	24.00	24.15
	2	670.51	131.67	13.94	12.69
	4	710.14	114.64	7.75	7.71
	8	2326.95	118.31	9.07	<b>5.93</b>

**Table 2. TreeAdd time with TCP/IP process migration**

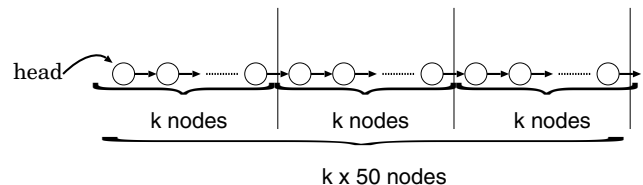
The first row contains the results from a purely sequential traversal of the entire tree (i.e., using the code in Figure 1.) The results show a super-linear speedup for the first three values. The reason is that paging is dramatically reduced as each host needs to handle progressively smaller portions of the tree. Once paging is completely eliminated, there is no more speedup. In our case, the collective memory of 4 hosts is sufficient to hold the entire tree. Hence

there is no benefit derived from mapping the tree on 8 hosts; on the contrary, the runtime is slightly higher (24.15), which is the result of the additional migrations of the process.

The second to fourth rows show the results obtained by task parallelism, i.e., by running the program in Figure 3. As expected, adding parallel processes on a single host (first column) makes the performance worse—there is no physical parallelism yet the concurrent processes need to cooperate with each other and compete for the single processor.

With two hosts, performance almost doubles when two processes are available. A moderate additional improvement is achieved with four processes, but eight processes already create too much overhead to be useful. This general trend continues in the third and fourth columns. With four hosts the ideal number of processes is four; with eight hosts, the ideal number of processes is eight.

It is worth noting that the greatest speedup was achieved by simply distributing the tree over four hosts, i.e., without any parallelism; the execution time dropped from 584.17 to 24.00. Parallelism improved the performance further but only from 24.00 to 5.93. This is still a very respectable improvement but requires the non-trivial programming effort of changing the sequential program into a parallel one.



**Figure 7. Testing remote read/write using linked list**

### 5.3. Using Remote Read/Write

To test the performance of remote read/write, we devised two simple functions, shown in Figures 8 and 9. Both of these scan a linked list distributed over 50 hosts as shown in (Figure 7). That is, each group of  $k$  adjacent nodes is mapped to a different host.

The expression `prev->val = cur->val + 10` (line 7 of Figure 8) assigns the updated value of the current node to the previous node. Without remote access, the process must perform three migrations each time it crosses a host boundary: first it hops to the new host to get `cur->val`; next it hops back to update `prev->val`; finally, it hops again to the new host to continue the processing of the next node. This behavior is a prime candidate for a remote write. After the process crosses the host boundary, it does not hop back to update the previous pointer but sends a remote write request instead. This saves two hops at the cost of a single remote write.

The two curves in Figure 10 labeled “without remote write” and “with remote write” show the performance improvement attributed directly to the remote write access. When  $k$  is small, the execution time was reduced from about 5 seconds to 2 seconds—a 60% reduction. As  $k$  increases, more time is spent on computation than on migration and hence the speedup decreases. However, even with 100,000 nodes per host the execution time still decreases by approximately 25%.

```
1 test_remotewrite()
2 {
3     get_timer(&st_timer);
4     prev = head; ptr = head->next;
5     while(ptr != NULL)
6     {
7         prev->val = cur->val + 10;
8     }
9     get_timer(&en_timer);
10 }
```

**Figure 8. Code to benefit from remote write**

The function in Figure 9 was designed to test remote reads. The statement `cur->val = cur->val * prev->val - cur->val` (line 7) updates the current value using the value of the previous node. Without remote access, this results in three hops as in the previous case; that is, after migrating to the new host, the process must hop back to get the previous value. This can obviously be accomplished more efficiently using a remote read, which the system automatically chooses to use.

```
1 test_remoteread()
2 {
3     get_timer(&st_timer);
4     prev = head; ptr = head->next;
5     while(ptr != NULL)
6     {
7         cur->val = cur->val * prev->val
8                 - cur->val;
9     }
10    get_timer(&en_timer);
11 }
```

**Figure 9. Code to benefit from remote read**

The two curves in the same Figure 10 labeled “without remote read” and “with remote read” show the performance improvement attributed directly to the remote read access. The execution time of this program without remote read is almost identical to the time of the program used to test remote writes. The reason is that, while the actual computation is different, both programs perform three migrations at each host boundary. The execution time of the program with remote reads is slightly worse than the time of the program with remote writes. This is because remote writes can overlap the communication with computation, whereas remote reads always need to wait for the remote data to arrive.

It is worth emphasizing that the decision to use remote reads/writes instead of migrations was made automatically by the system using the mechanism and the heuristic described in Section 4. Thus the observed performance improvements were achieved without any intervention of the programmer.

## 6. Conclusion

We presented an approach to implementing and using global pointers in a distributed computing environment. Our system gives the programmer the tools necessary to create pointer-based distributed data structures. Such data structures can then be used by sequential or parallel programs without having to differentiate between local and global pointers. Any reference pointing to the memory of a remote host causes the process to automatically and fully transparently handle the remote reference. Depending on the process’s past behavior at the faulting location, it may decide to migrate to the remote host, where it continues its execution, or it may perform a remote read or write operation while remaining on the current host. The detection of the remote reference as well as the decision on whether to migrate or perform a remote access operation are all carried out automatically at runtime and are transparent to the

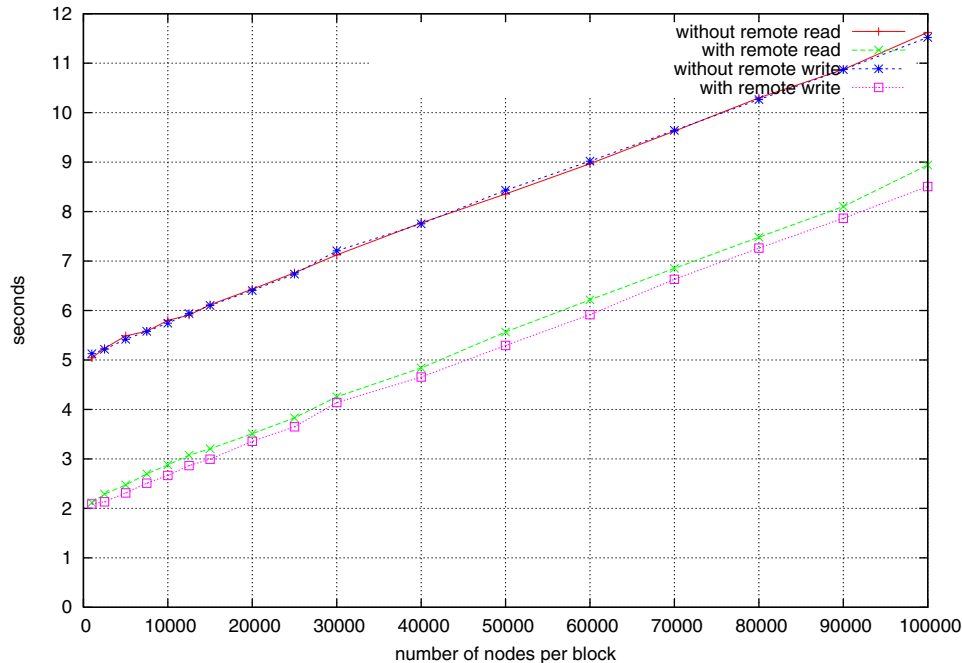


Figure 10. Result from remote read/write

programmer.

By using a hardware-supported memory checking mechanism, we were able to avoid any additional overhead associated with the detection of remote references. The optional remote read/write operations further reduce the overhead attributed to migration. With these mechanisms, we were able to demonstrate that distributing pointer-based data structure can lead to dramatic improvements in performance for both sequential and parallel programs.

To further improve performance for a wider variety of applications, our system implements an additional mechanism, which allows daemons to move entire pages among themselves, thus changing the initial mapping of the global heap. This is useful when repeated remote reads or writes to the same page are received. The details of this further enhancement may be found in [5] and will be published in a subsequent paper.

## References

- [1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 96–107, New York, NY, 1991. ACM Press.
- [2] M. Fukuda, L. F. Bic, M. B. Dillencourt, and J. M. Cahill. Messages versus messengers in distributed programming.

*Journal of Parallel and Distributed Computing*, 57(2):188–211, 1999.

- [3] L. J. Hendren, X. Tang, Y. Zhu, G. R. Gao, X. Xue, H. Cai, and P. Ouellet. Compiling c for the earth multithreaded architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, page 12. IEEE Computer Society, 1996.
- [4] W. C.-Y. Hsieh, M. F. Kaashoek, and W. E. Weihl. Dynamic computation migration in DSM systems. In *Proceedings of Supercomputing '96*, Nov. 1996.
- [5] K. Knoguchi. *Spontaneous Process Migration with Global Pointers*. PhD thesis, Dept. of Computer Science, University of California, Irvine, 2006.
- [6] B. Liblit and A. Aiken. Type systems for distributed data structures. In *Symposium on Principles of Programming Languages*, pages 199–213, 2000.
- [7] R. S. Nikhil. Cid : A parallel, shared-memory c for distributed-memory machines. In *7th International Workshop on Languages and Compilers for Parallel Computing*, pages 376–390, Ithaca, NY, August 1994. Springer-Verlag.
- [8] L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai. Distributed sequential computing using mobile code: Moving computation to data. In *Proceedings, Int'l Conf. on Parallel Processing (ICPP 2001)*, Sept 2001.
- [9] J. Protic, M. Tomasevic, and V. Milutinovic. *Distributed Shared Memory: Concepts and Systems*, volume Summer 1996. IEEE Computer Society Press, 1996.
- [10] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.