Abstract

Thread migration is one approach to remote memory accesses on distributed memory parallel computers. In thread migration, threads of control migrate between processors to access data local to those processors, while conventional approaches tend to move data to the threads that need them. Migration approaches enhance spatial locality by making large address spaces local, but are less adept at exploiting temporal locality. Data-moving approaches, such as cached remote memory fetches or distributed shared memory, can use both types of locality.

We present experimental evaluation of thread migration’s ability to reduce the impact of remote array accesses across distributed-memory computers. Nomadic Threads uses compiler-generated fine-grain threads which either migrate to make data local or fetch cache lines, tolerating latency with multithreading. We compare these alternatives using various array access patterns.

1. Introduction

The execution time of many programs running on distributed memory parallel computers is often dominated by the time needed to access remote data. Many techniques to reduce and/or tolerate this remote memory access time have been proposed and evaluated with varying degrees of success. This paper examines the utility of migrating fine-grained computation to the various processing elements (PEs) in the system that contain the needed data. This migration approach is appealing because it has the potential to exploit the vast spatial locality provided by all the data in the memory of a PE.

Others have examined migration’s ability to exploit locality in dynamic data structures and with threaded extensions of conventional languages. This paper focuses on array access patterns for both parallel and sequential algorithms operating on distributed arrays. Because of our particular interest in array access patterns, we tended to use relatively simple algorithms and program components in order to simplify analysis of the results. The obvious next steps will include extending the work to more complex algorithms and programs that combine the array access patterns evaluated here. This paper provides guidance for future research into fine-grain migration and exposes strengths and weaknesses of the approach.

Section 2 briefly discusses some of the approaches to reduce and/or tolerate remote memory access times, several of which form the basis of our research. Section 3 introduces our Nomadic Threads migration approach and provides an architecture overview. Section 4 provides performance results of our migration approach and contrasts that behavior with cached remote memory accesses. Finally, section 0 summarizes our results.

2. Background

Large parallel computer systems have become increasingly popular during the last two decades, as problem size and complexity have grown and microprocessor speed and memory affordability have skyrocketed. Computational problems ranging from computational fluid dynamics, to weather prediction, to image manipulation and target recognition, to high-resolution ray tracing require considerable processing resources and, often, very large data sets. The programs that solve these problems generally have significant portions that can be executed concurrently on many PEs. While such a program’s work can be distributed amongst the PEs, these programs often need access to data stored on other PEs, thus remote data access is a major concern. Since such accesses require message sending over the interconnection network that joins the PEs, the latency is orders of magnitude greater than local memory accesses.

A number of approaches can be used to reduce the impact of remote memory accesses on programs. Compilation techniques [3, 11, 12] can often rearrange the data and program partitioning to significantly reduce the number of remote memory accesses needed, but many remote accesses often remain. Caching of remote memory accesses and fetching multi-element cache lines rather than a single remote element at a time can greatly reduce
the number of remote memory accesses required. Caching takes advantage of both temporal and spatial locality by making recently used elements and their neighbors available locally. Techniques such as multithreading [9, 13, 17], including the Threaded Abstract Machine (TAM) [8] which greatly influenced our work, and prefetching [15] reduce the effects of remote memory accesses by keeping the processor busy while remote memory access operations take place. Even with the assistance of these approaches, parallel programs on large distributed memory machines can still spend much of the time fetching remote data.

Several thread and computation migration approaches have been proposed to address this and other problems. Olden [7, 16] is very successful at exploiting the locality present in dynamic data structures, such as trees and lists. Other approaches, including Cilk [6], use migration to balance the load across the machine’s PEs. Finally, an early, hand-coded implementation of Nomadic Threads showed promise at exploiting locality [10].

3. Thread migration to exploit array locality

The Nomadic Threads architecture is based on a multithreaded approach, similar to TAM and others, in which threads run to completion based on available data and frames store inputs, results, and intermediate values. Since threads always run to completion once activated, they do not perform long-latency operations, such as remote memory accesses, alone. Instead, one thread will initiate the long-latency operation, but one or more other threads get activated by the completion of the operation, as shown in Figure 1. While the long latency operation is occurring, other available threads may run, thus keeping the processor busy doing useful work. More detailed descriptions of multithreaded execution models can be found in [5].

![Figure 1. Split-phase remote memory access](image)

![Figure 2. Remote memory access and migration compared](image)

Despite migration’s simplicity and appealing intuitiveness, there are several complications or factors that can hurt the performance of programs that use migration. First, migration can exploit spatial locality, but not temporal locality. As soon as an activation migrates,
those data items that were local are now remote, so temporal locality is not applicable. Since temporal locality dominates some algorithms, as shown in section 4.3, migration should not be used exclusively, but can be effective when combined with other approaches.

In addition, due to the way computation and data are partitioned across the PEs of the system, each array element has a particular home PE where it must be stored. The writer returns property is the property of migration that specifies that store operations on array elements must occur on the home PE where that element is assigned based on the array partitioning scheme. If the thread of control that will perform such a store operation resides on another PE because of migration or for any reason, it (or a designee) must migrate to the home PE of the array element to perform the write. A remote store operation, if provided by the runtime system, may also be used to store the array element, thus acting as a designee for the thread.

The writer returns property is somewhat analogous to the owner computes rule, in which all the data needed for the computation of a result on a PE must be fetched to that PE. Caching and distributed shared memory approaches can overcome this issue because data can be replicated throughout the system, but thread migration moves the program, not the data, so migration can be significantly affected.

4. Experimental results

In order to evaluate the effectiveness of migration at exploiting available spatial locality and to compare it to caching remote memory accesses, we ran numerous benchmarks using varying numbers of PEs on a 128-PE CM5, the largest machine we were able to access at the time. The benchmarks were generally simple programs, because for this phase of our investigations, we were concentrating on categorizing performance on different array access patterns. Future research will involve more interesting benchmarks and significantly larger data sets on modern cluster computers rather than the relatively underpowered CM5.

4.1. Basis for comparison

The baseline memory access approach against which we compared our migratory approach is a software-based, pseudo-associative, caching remote memory access (CRMA) scheme. Our CRMA mechanism is implemented in the Nomadic Threads runtime system and co-exists with the migration mechanism, so both approaches can be used in a program. Significant effort was made to make certain that the caching implementation is as efficient as the migration scheme, so migration does not have an unfair advantage. Both approaches use the CMAML active message library [1], and after optimization, cache line fetches are actually somewhat faster on the CM5 than migration because fewer messages are needed and overhead is reduced.

The benchmark programs are written in the SISAL language [14] and compiled to an intermediate form, IF1 [18], by the Lawrence Livermore Optimizing SISAL Compiler. Our compiler then generates parallel, scalable C code from the IF1 graphs. The resulting C code is not machine-specific, since it uses the relatively portable Nomadic Threads runtime system to perform data accesses, thread spawning, and migration. The resulting C code is compiled with the GCC compiler on the CM5 and linked with the Nomadic Threads runtime system libraries and the CMMD message passing libraries.

The choice of whether to use migration or CRMA or a combination of the two for remote memory accesses can be forced via compiler flags or determined automatically by the compiler via program structure analysis and a set of heuristics. For most of our benchmarks, we used compiler flags to force the use of each mechanism for comparison. In one case, described below, we used the compiler’s combination approach for one of the alternatives to take advantage of differing access patterns in separate parts of the program.

We were also able to evaluate the latency tolerance ability of our multithreading approach and increased parallelism by varying the number of threads working on the portion of the problem assigned to each PE. This allowed us to determine whether the program was bound by the CPU capacity or the network throughput and overhead.

4.2. Affine access patterns

The first class of access patterns we studied are affine array accesses, because they are very common and also quite easy to understand and evaluate. Many of our affine access pattern benchmarks used simple linear array indexing, though some used more complex patterns, including strides of various sizes. Most of our benchmarks exhibited some degree of spatial locality, and a few had temporal locality as well. In general, spatial locality dominated these benchmarks, so they were useful to evaluate how well migration could do in good conditions.

We expected migration to perform well in cases where a large amount of work could be accomplished with local data between migrations. Conversely, we anticipated that migration would perform poorly when migrations were needed too often, particularly when the writer-returns property dominated the program’s operation. We were right on both counts, though we underestimated how much of a penalty the writer returns property could be in the worst cases.
Figure 3 shows the performance of a representative set of affine benchmarks with performance normalized to CRMA performance for each machine size. The figure shows the relative performance of both CRMA and migration in the cases when 16 PEs were used and when 64 PEs were used by the programs. In the matrix transpose benchmark (Transpose), migration did quite poorly because of the writer returns property. Much of the time, the activation required two migrations to fetch a remote element and then return it to the destination PE. CRMA required at most one cache line fetch per element and is able to take advantage of spatial locality if the cache is large enough, thus giving caching a distinct advantage. The other benchmark in which migration did quite poorly simply reverses a distributed array, which, like the matrix transpose, requires two migrations per array element. CRMA does particularly well at exploiting spatial locality in this simple benchmark, thus compounding migration’s defeat.

The other two benchmarks shown in the figure are more suited to migration, because a significant amount of local work can take place before a migration is needed. The RevProd benchmark performs a dot product on two arrays organized in such a way that most of the elements are not local to the PE where the results are needed. Our migratory activations send themselves to the appropriate remote PEs and perform the dot product multiplication and summing there before returning with the result. The CRMA approach requires fetching the remote elements a cache line at a time, which is still very efficient and takes advantage of spatial locality, but requires more message transfers than migration does.

The matrix multiplication benchmark (MatMult) is the most interesting of the benchmarks in the figure, because it requires the most work and has properties that make it well-suited to migration. Before the multiplication operation, the benchmark program transposes one of the matrices to align the columns which are the source vectors for the inner-loop dot product operations. These vectors are still distributed across all the PEs of the machine, but they are aligned so that the corresponding source elements for each dot product step are on the same PE. As we saw above, migration does poorly on the transpose step, so in this case, we allowed the compiler heuristics to choose CRMA for the transpose operation followed by migration for the multiplication itself. As we can see from the figure, when there is a large amount of spatial locality, as in the 16-PE case, migration is substantially faster than CRMA, but our small matrices hurt us with 64 PEs, for as locality decreased, so did migration’s advantage. We ran a separate test with a much larger matrix and found migration was more than four times faster on 128 PEs. Obviously, a blocking matrix multiply algorithm would benefit the CRMA case and shrink migration’s advantage, so this will be a topic for upcoming study.

4.3. Stencil access patterns

The next set of access patterns we examined used 2- or 3-dimensional stencil patterns of access around a point in the source arrays defined by functions of loop indices. Examples of such problems that we used in our benchmarking included image manipulation and rotation as well as a Finite Difference Time Domain (FDTD) electromagnetic field algorithm. The image processing benchmarks used 2-D stencils on 2-D arrays, while the FDTD benchmark used 3-D stencils on 3-D arrays.

Because the stencil access patterns exhibit significant spatial locality, we expected migration to do very well on these benchmarks. In general, programs using migration to access remote data did scale well and performed as expected. It turned out, however, that since each stencil is generally very near the previous one, and there is often significant overlap, a great deal of temporal locality is available as well. Since migration can only exploit spatial locality, CRMA generally did better on the stencil benchmarks because it combines spatial and temporal locality.

Figure 4 shows the performance of the stencil benchmarks normalized to the CRMA results. Here we see that CRMA is faster for these benchmarks when both 16 and 64 PEs were used. In both the image manipulation algorithms (smooth rotation and relaxation), CRMA is up
to twice as fast as migration, while adding the third dimension in FDTD pushes CRMA to three times migration’s performance. In general, the temporal locality helped CRMA and the writer returns property hurt migration. Since this pattern is so pronounced and significant, it is likely that migration will very rarely be the best choice for algorithms with stencil access patterns unless the amount of work done with the local data is far greater than that of these benchmarks.

4.4. Sequential access patterns

A surprisingly interesting category of benchmarks we evaluated is the sequential traversal of array data. Though much of our effort involves enhancing parallel performance and scalability, many parallel programs still have some sequential parts. Such sequential code could be caused by data dependences or other reasons, but they can significantly slow execution time. Amdahl’s law [4] shows us how even small sequential bottlenecks can substantially limit the speedup achieved on a parallel computer. When the sequential code must access data that is distributed across the PEs of a parallel computer, the problem is further compounded and performance may decrease as the machine grows.

We ran two very different sequential benchmarks and the results were sufficiently encouraging to warrant future study. The first benchmark sequentially traversed a 2-D array in order. Figure 5 shows that migration can exploit the vast spatial locality very efficiently, while CRMA fetches the data it needs to the PE running the sequential program. Prefetching could help the CRMA case, but may not be able to keep ahead of the program sufficiently to hide the remote memory access latency. The very flat curve for migration as we increase the number of PEs used is a terrific result for a sequential program, and this algorithm is similar to an image traversal stage seen in one of the Data Intensive benchmarks [2].

The other sequential benchmark simulated cyclic access to an array with varying stride values. Figure 6 shows the performance when the access stride is one 128th of the array size. This means that with fewer than 128 PEs, several accesses will occur per PE, which is exploitable by migration, thus the relatively flat timing until high numbers of PEs. When we reach 128 PEs, there is no locality to exploit by either migration or CRMA, so each remote access requires either a migration or a cache line fetch. Since cache line fetches are more efficient on our CM5 implementation, CRMA is somewhat faster than migration in this extreme case. If migration used fewer messages, such as on a Beowulf cluster, it would be faster than cache line fetches in this case as well.
5. Conclusion

This paper provides an assessment of the strengths and weaknesses of thread (and activation) migration to exploit spatial locality in remote memory accesses for arrays. We showed that migration does indeed take advantage of spatial locality and can perform large amounts of work by migrating threads to a PE containing data they need. We also showed that the writer returns property, where threads must migrate to a specific PE to write their results, can significantly hurt performance if such migration must occur frequently. Our experimental results on a CM5 demonstrated that migration was superior to cached remote memory access for some access patterns, but slower for others, including those that use stencils. We also examined some sequential algorithms where migration must occur frequently. Our experimental results, can significantly hurt performance if such migration must occur frequently. Our experimental results on a CM5 demonstrated that migration was superior to cached remote memory access for some access patterns, but slower for others, including those that use stencils. We also examined some sequential algorithms where migration was very effective at finding and using even minimal spatial locality. This class of problems, among others, warrants substantial attention in future research.

6. References