Towards a Sequentially Consistent Memory Model for PGAS Languages

Amir Kamil Jimmy Su Katherine Yelick
Computer Science Division, University of California, Berkeley
{*kamil,jimmysu,yelick*}*@cs.berkeley.edu*

August 1, 2006

A key question in the design of partitioned global address space (PGAS) languages is the memory consistency model: In what order are memory operations performed by one thread observed on another? For simplicity, one would like the operations to appear in the order specified in the original program, i.e., any reorderings performed by the compiler or hardware should not be observable. In practice, parallel language designers have been reluctant to use such a strong *sequentially consistent* semantics because memory operations are often overlapped and reordered for performance reasons, and requiring that all threads see the same memory order requires the insertion of expensive memory fence instructions. In UPC, the programmer controls the memory consistency model using strict and relaxed accesses, but in practice, most codes are written with all accesses relaxed, even though the semantics of relaxed accesses are not well understood by many users. Titanium adopts Java's original weak memory model, which is also quite subtle, and in Co-Array Fortran, Fortress, and Chapel, the memory model is not clearly specified.

Such complex or underspecified models are very confusing to many programmers, and they tend to prefer a simpler and less error-prone specification such as sequential consistency. However, providing sequential consistency is nontrivial, requiring memory fences to be placed around conflicting memory accesses to prevent them from being reordered explicitly by the compiler or implicitly by the hardware. Two memory accesses *conflict* if they can access the same location, they can occur concurrently, and at least one of them is a write. These fences can adversely affect runtime performance, since they prohibit some optimizations and also incur a significant runtime cost in PGAS settings. In order to minimize the cost of sequential consistency, the number of required fences, and therefore the set of conflicting accesses, must be minimized. In combination with concurrency analysis, pointer analysis can be used to precisely determine this set.

In this presentation we introduce a pointer analysis that is designed for a distributed setting, and single-program multiple-data (SPMD) languages in particular. The analysis takes into account a hierarchical machine model, in which individual threads are grouped into multiple physical address spaces, all of which constitute the whole program, as shown in Figure 1. In this model, a pointer may refer to data only within a single thread, to data associated with any threads within a physical address space (e.g. an SMP node), or to any thread in the machine. A pointer has a designated *width* restricting the memory it can refer to: a pointer of width w on thread T can only reference memory in the subtree of the machine rooted at level w and

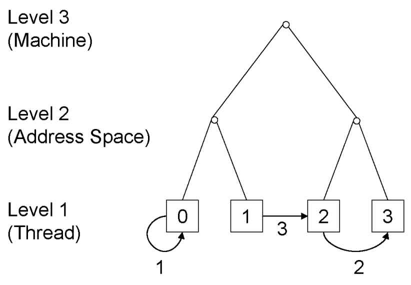

Figure 1: A hierarchical grouping of threads into physical address spaces and an overall machine.

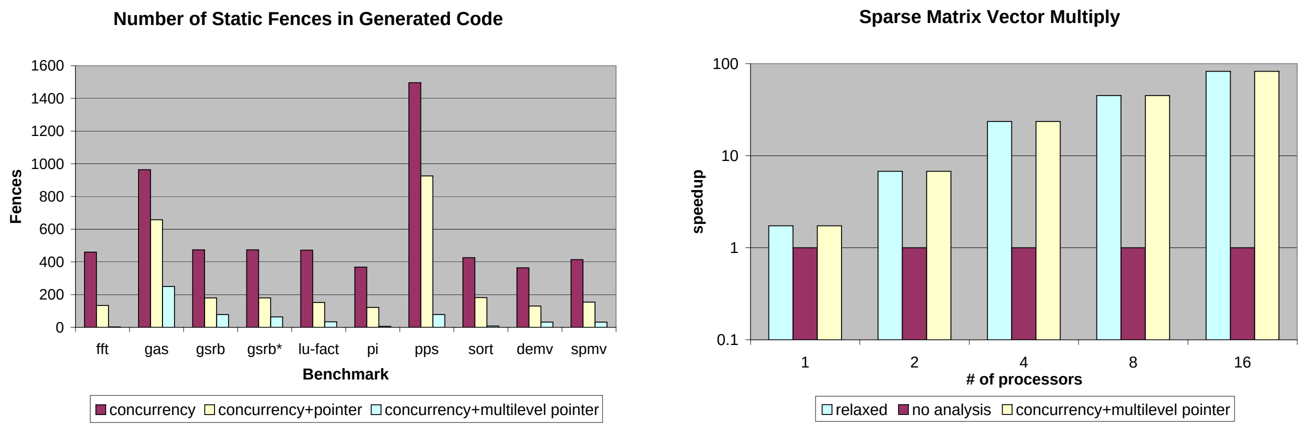

Figure 2: Fence results for some Titanium programs, and a performance comparison using one of the programs. The fence results use various levels of analysis: concurrency analysis only, concurrency analysis with single-level pointer analysis, and concurrency analysis with multilevel pointer analysis. The performance results compare a relaxed memory version of the code with sequentially consistent versions. Speedup is over the sequentially consistent version using no analysis.

containing T.

The pointer analysis examines the program once, taking advantage of the SPMD nature of the program to compute the result for all threads simultaneously. Like other pointer analyses, the analysis associates allocation sites with *abstract locations*. In addition, abstract locations in the analysis have a corresponding width, signifying which machines the corresponding memory can reside on. Three language constructs are particularly important to the analysis:

- **Allocation sites**: Allocations occur locally, so the resulting abstract location must have width 1.
- **Broadcasts**: A broadcast operation can communicate a memory address from one thread to the rest. The memory location can reside in a different physical address space than some of the destination threads, so the resulting abstract locations must have width 3.
- **Assignments**: Assignments have global effects, so the analysis must take into account the result of an assignment on each thread.

The analysis also handles other operations in a SPMD language, some of which can produce abstract locations of width 2.

We present the results of implementing our analysis in the Titanium language, a high performance parallel language based on Java, and its application to memory model enforcement. The results show that the pointer analysis greatly reduces the number of memory fences required to enforce sequential consistency, as shown in the left of Figure 2. We also compare relaxed and sequentially consistent versions of a sparse matrix vector multiply kernel written in Titanium. In order to obtain good performance, the compiler must be able to perform reordering optimizations on the code. Without the concurrency and pointer analyses, the inserted fences prevent the optimizations from being applied under the sequentially consistent model, resulting in much poorer performance than under the relaxed model. However, as shown in the right of Figure 2, the analyses reduce the set of fences required to obtain sequential consistency, allowing the compiler to reorder operations and match the performance of the relaxed version of the program.

Additional work is needed to determine whether our analysis can be applied to a broader class of applications, optimizations, and languages. Applications that use the global address space without awareness of partitioning may prove more difficult to analyze. Optimizations other than those we examine may prove more susceptible to false positives in the analysis. Finally, although our pointer analysis is applicable to a broad class of languages, the benchmark results rely on a concurrency analysis based on Titanium's textually aligned barriers. Nevertheless, we believe that these results show that sequentially consistent memory models may be possible for shared memory languages, even in a PGAS setting with its high costs of enforcing ordering on remote accesses.