Novel Parallel-Programming Idioms

Jeremy H. Brown
(jhbrown@ai.mit.edu)

The Problem: Programming massively parallel MIMD systems is a difficult task. Although certain limited-communications applications have been successfully divided over networks of workstations, communications-intensive parallel applications have proven difficult to think about, write, and port. Issues of memory model, data locality, synchronization, and side-effect management make matters particularly thorny. The programming languages and libraries currently available simply do not provide the idioms and constructs necessary to program MIMD systems efficiently and reliably.

Motivation: In spite of the difficulties involved in programming MIMD systems, the need for ever-greater computational power is causing even the most mainstream computer systems available to sport two, four, and even more more processors. A recent announcement from IBM takes matters a step further: multiple processors are being placed together on a single chip. Clearly, there is a desperate need for tools that make utilizing multiprocessor systems easier and less bug-prone.

Previous Work: There are dozens of programming languages sporting constructs designed to assist with programming parallel systems. MIMD languages such as Java and Cilk[1] express parallelism via multi-threading and guard side-effects with locks and semaphores. SIMD languages such as HPFortran, *Lisp, and APL provide data-parallel primitives which are a thin veneer on standard vector-parallel operations. "Idealized" SIMD languages such as Paralation Lisp and NESL[2] express parallelism with parallel-apply operations performed over data collections (typically vectors); nested parallelism is allowed and does not prevent efficient compilation to non-nested, vector-parallel operations.

Finally, pH [3], a parallel dialect of Haskell, discovers parallelism through dataflow analysis. Parallel side-effects are moderated through two special data structures: I-structures, which are write-once slots, and M-structures, which require alternating writes and reads. Reads to empty I- or M-structures block until a value is written.

Approach: We are experimenting with a small set of programming constructs suitable for addition to arbitrary programming languages; our testbed language is Scheme, a dialect of Lisp. These constructs area aimed not only at conventional multiprocessor systems, but also at a massively parallel architecture our group is currently designing. A brief description of a few of the key programming constructs follow:

First, parallel-apply invokes a function on each element of a collection in parallel. Order of evaluation is unspecified except that the thread calling a parallel-apply blocks until all threads spawned by that call have completed. Parallel-apply may be nested.

We manage inter-thread side-effects using two new constructs: combiners and splitters. A combiner is a 2-to-1 function used to merge side-effects to the same location; a splitter is a 3-to-2 function used to split the merged result into independent values for each of the callers. Combiners and splitters allow arbitrary order of execution, with no risk of deadlock. On distributed-memory systems, combiners and splitters reduce bottlenecking at the side-effect target's node by performing their work at other locations in the processing network. Note that these operations map gracefully to the Ultracomputer[4], which performs combining operations in the communications network itself.

To use combiners and splitters, a thread specifies the side-effect target location T; a value V_s; a combiner C; and an (optional) splitter S. The effect of the invocation is divided into two cases. In the simple case, the value V_t is read out of T, the value $V_r = C(V_s, V_t)$ is computed, and V_r is written into T; if a splitter has been provided (i.e. a return value is solicited) V_r is returned. (In this case, the splitter isn't actually used.)

In the more complicated case, shown in Figure 1, multiple threads attempt to side-effect the same location, specifying the same C and S but different values; in this case, the system may (but is not required to) merge any pair of requests carrying values l and r into a new one carrying the value $m = C(l, r)$. This pairwise reduction may be done recursively, generating a depth $\log_2(N)$ reduction tree as shown in Figure 1a.

Once the tree has been completely built (i.e. a final value has been written into T), return values are generated as shown in Figure 1b by invoking S at each join-point to generate two values: $(R_l, R_r) = S(l, r, b)$, where l and r were the values initially combined at that join point, and b is the value returned from lower in the tree (the first b is the value from the actual target-variable.). R_l and R_r are each returned up one branch of the tree.

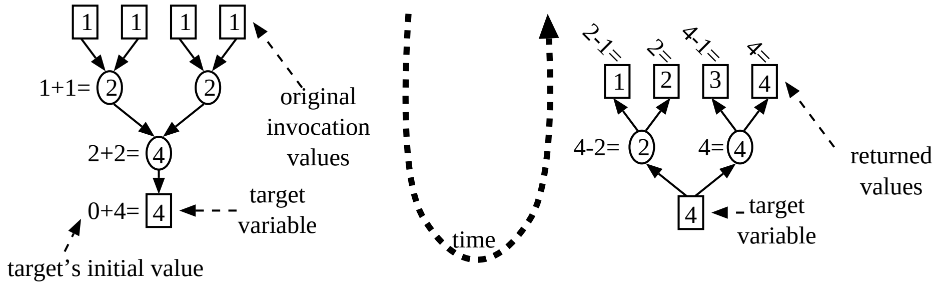
Difficulty: Naive implementation of the basic semantics of parallel-apply, combiners, and splitters is a straightforward task. The much subtler task is evaluating their expressiveness by selecting a representative set of applications and coding them to exploit these features.

Assuming positive evaluation, the next step is an aggressive implementation; developing the system infrastructure to efficiently detect combinable operations and merge them, etc. will be a significant undertaking.

Impact: In response to the increasing availability of multiprocessor systems, programmers are learning to coordinate small numbers of parallel threads using locks and semaphores, but managing large numbers of threads in this way remains difficult and error-prone. If our set of idioms proves generally useful and efficient, the development and debugging overhead of aggressively parallel code will be dramatically reduced.

Future Work: A Scheme-specific set of our parallel programming idioms has been defined; a naive implementation is in progress. A small number of representative applications have been selected, but all still need to be coded. An aggressive implementation of our idioms will be undertaken only after evaluation with the naive implementation assures us that they are defined correctly.

Finally, we are in the process of defining a set of simple hardware features that will make detecting mergeable combiner invocations particularly cheap in the custom system we are designing.



(a) Combining write with C(l,r) = l+r

(b) Subsequent splitting return with S(l,r,b) = (b-r,b)

Figure 1: Example Combining Write With Splitting Return

Research Support: This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Support for this research was provided in part by the Defense Advanced Research Projects Agency under Rome Labs Contract Number F30602-98-1-0172.

References:

[1] C. F. Joerg. The Cilk System for Parallel Multithreaded Computing. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January, 1996.

[2] G. E. Blelloch. NESL: A Nested Data-Parallel Language. CMU-CS-93-129, Carnegie Mellon University, April, 1993.

[3] R. S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J. Maessen and Y. Zhou. pH Language Reference Manual, Version 1.0 — preliminary. MIT LCS CSG Memo #369, Massachusetts Institute of Technology, January, 1995.

[4] S. R. Dickey and R. Kenner. Combining switches for the NYU Ultracomputer. Fourth Symposium on the Frontiers of Massively Parallel Computation IEEE Comput. Soc. Press. 1992, pp.521-3. Los Alamitos, CA, USA.