Purely Functional Parallel Programming

Brian D. Sutton

March 10, 2003

Abstract

Implement a few parallel algorithms in the purely functional language Haskell. Provide anecdotal evidence on the state of parallel compilers for purely functional languages.

Consider the problem of parallelizing an existing sequential program. Three challenges come to mind.

1. Sections of code with few dependency relationships must be identified for separate threads.
2. Data integrity must be preserved in the face of concurrent threads.
3. Threads and data must be assigned to processors to provide efficient memory access.

These significant tasks suggest that the best solution may be to simply rewrite from scratch.

However, if the program is written in a purely functional programming language such as Haskell, then challenges 1 and 2 are no longer challenges! In a purely functional program, every statement is a mathematical equation—`pi = 3.14159; square x = x * x;`. In particular, no control flow constructs are available (besides recursion) and no variable re-assignments are allowed. Indeed, there is no traditional concept of control flow; many different execution paths are possible, bounded only by the data dependencies which are quite explicit in the code. Also, data integrity is not an issue, because of the property of *referential transparency*—an expression always has the same value, regardless of when it is evaluated. (This concept of "expression" assumes unique names for all variables.)

In short, the sequence of operations in an imperative program is replaced by a DAG of data dependencies in a purely functional program. And this same DAG reveals memory accesses.

Hence, *safely* parallelizing a purely functional program is trivial. If we view the DAG of data dependencies as a partial order, then any two incomparable expressions may be evaluated in parallel. Further, aliasing is not a problem; because of referential transparency, the programmer is free to copy data from one processor to another knowing that it will never be modified.

Efficiently parallelizing a purely functional program is another matter. From my reading, it is unclear just how effective implementations of purely functional programs have been. There are several proposals varying in evaluation strategies and in degree of the programmer's involvement in parallelization.

I propose to implement some small programs in GpH, the most mature programming environment available for Haskell. Then I will compare performance on one processor versus several processors. The results will be purely anecdotal.

Note that this project has wider implications than parallelizing existing code. Three immediately obvious applications follow.

1. Automatically parallelizing compilers.

2. Separating algorithms from implementation details. (Parallelization could be specified by annotations to sequential code.)

3. Coarse granularity, i.e. "big threads." In contrast, Matlab*P parallelizes at the library level. Calls to ScaLAPACK experience speed-up on multiple processors, but many Matlab*P programs are sequential at the highest level. A standard example of coarse granularity is a compiler with multiple stages, lex-yacc-bytecode-machine code. Theoretically, the output from one stage can be consumed by the next stage before all of the output is available. Purely functional languages may make this sort of producer-consumer parallelism easy to implement.