Issues in writing a Parallel Compiler starting from a Serial Compiler

Alexandros Tzannes,
Rajeev Barua,
George C. Caragea,
Uzi Vishkin

University of Maryland, College Park

Motivation

- “The Free Lunch is Over” [Herb Sutter]
- CPU clock speed stopped increasing
- Dual Cores (on chip) are now mainstream and Quad cores around the corner
- Intel has a 5 year roadmap for a 80-core teraflop processor

Motivation

- Parallel machines need to be programmed → Parallel compilers are needed
- Writing an optimizing compiler is hard.
- Parallelism cannot be implemented exclusively as a library (e.g., PThreads) [Boehm05]
- Can we use a serial compiler as the basis for a parallel compiler ?

Background

- Conflict:
  - Statement instances *p* and *q* conflict if they access the same memory location *m* and at least one of them is a write.
- Dependence:
  - if *p* always accesses *m* before *q* then we have a dependence and *q* depends on *p*.

Sequential Consistency

- Memory Consistency Model.
  - Definition
- Two levels:
  - Hardware
  - Software
- [Mark Hill] The programmer should program in SC semantics.

Our Focus

- Shared Memory SPMD PL with SC semantics
- XMTC: spawn statement:
  - spawn (low,high) { CODE }
  - create *high-low+1* threads with IDs in {low, low+1,…,high}
  - the threads are executed in any order at any speed and implicitly synchronize at the end of the spawn statement.
  - the TID can be accessed in CODE by means of the special symbol ‘$’.
  - No jumps across serial parallel boundaries.

Spawn Example

```
int A[100];
spawn (0, 99) {
    A[$] = $*$;
}
```

Why not…

- compile each thread as a serial program?[Midkiff90]
  - conflicts and dependencies make many serial optimizations inapplicable. (More on this later)

Adding the spawn statement

- Alternative 1:
    - Augment the internal representation with new types of nodes for parallel constructs.
    - Update all optimization passes to deal with new nodes.
- Alternative 2:
    - Insert placeholder nodes (e.g., by means of function calls) that will be expanded at the end of the compilation.

Illegal dataflow

```
int main (void) {
    int c=0;
    spawn (0,4) {
        increment c by 1 atomically;
    }
    … = c;
}
```

1

A solution: outlining

Outlining Example

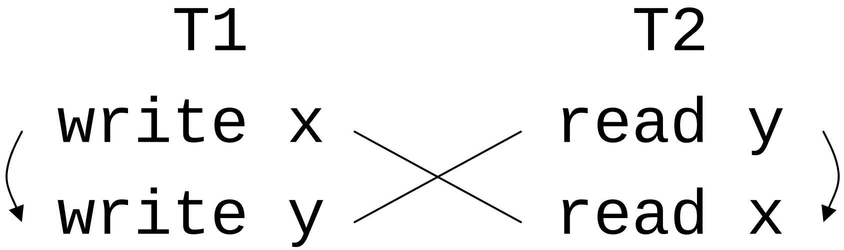
```
outlined_spawn (int *c) {
    spawn(0,4) {
        increment *c atomically by 1
    }
}
int main (void) {
    int c = 0;
    outlined_spawn(&c);
    … = c;
}
```

Outlining

- Outlined functions placed in a different file.
- Outlined Functions Might need arguments (by value or by reference).
- Global variables might need to be accessible
- Inlining cannot be done before all optimization passes are through. It is hard and not very rewarding.

Shared Variables

- [Midkiff90] Some serial optimizations become illegal for SC semantics :



T1	T2
`write x`	`read y`
`write y`	`read x`

1st Alternative

- Turn off all optimizations that can reorder memory accesses.
- But the we disallow register allocation.
  - Solution: declare shared vars as `volatile`
- If the parallel code is outlined to a separate file, optimizations need to be turned off only when compiling that file.

2^{nd} Alternative

- [Shasha88] It is enough to turn off illegal optimizations only on shared variables.
- Detect which statements contain shared variables and annotate them
  - This can be complicated if we want to be precise(ptr analysis, array footprint analysis)
- Update optimizations to honor the annotations.

3rd Alternative

- Do elaborate whole program dependence analysis for shared vars [Krishnamurthy95]
- Need to do shared var detection as in Alternative 2.
- Alternative 3 builds on Alternative 2 which builds on Alternative 1.

Stack allocation for parallel threads

- Dynamic Memory Allocation is considered inefficient for stack allocation
- Cactus Stacks are a popular data structure [Sardesai]
- Stack sharing techniques can be relevant [Middha]

Vectorize by Processor vs. by Thread

- The number of processors is a fixed constant.
- The number of threads can be unbounded.
- Ideally we would like to allocate the minimum of these two numbers for each spawn statement.

Vector of stack frames vs. vector of variables

- Vector of Frames:
  - 1 update of the stack pointer at the beginning of the parallel code, and one at the end.
- Vector of Vars:
  - no update of the stack pointer, but indirect access of all shared variables (overhead)

Function Calls in parallel threads

- Same issues with shared variables as with parallel code.
- Each function must be compiled for use in parallel or serial mode.
- If function does not have side effects and does not access shared vars → no complications in compilation

Functionality in Libraries

- We do not address issues that fall under the category of “Functionality in Libraries” such as:
  - Dynamic Memory Allocation
  - Synchronization
- There is rich literature on these topics.

Conclusions

- For our class of PLs (SPMD, SC, shared memory) we presented a methodology to:
  - Prevent Illegal dataflow and control-flow (Outlining)
  - Prevent Illegal optimizations on shred variables (3 incremental alternatives).
- We pointed out pitfalls and solutions for:
  - Stack Allocation
  - Function Calls

Questions ?

References

- [Herb Sutter] http://www.gotw.ca/publications/concurrency-ddj.htm
- [Boehm05] Boehm, H.J.: Threads cannot be implemented as a library. In PLDI `05. ACM Press (2005) 261-268
- [Mark Hill] Hill, M. D.: Multiprocessors should support simple memory – consistency models. Computer 31(8) (1998) 28-34.
- [Midkiff90] Midkiff, S.P., Padua, D.A.: Issues in the optimization of parallel programs. In ICPP (2). (1990) 105-113.
- [Shasha88] Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2) (1998) 282-312.
- [Sardesai] Sardesai, S., McLaughlin, D. Dasgupta, P.: Distributed cactus stacks: Runtime stack-sharing support for distributed parallel programs.
- [Middha] Middha, B., Simpson, M., Barua, R.: MTSS: Multi Task Stack Sharing for embedded systems.