LA-UR-16-?????

Automatic Parallelization and Transparent Fault Tolerance (project in progress)

Kei Davis, Dean Prichard,
***David Ringo*, Loren Anderson,**
and Jacob Marks

Trends in Functional Programming, June 8-10, 2016

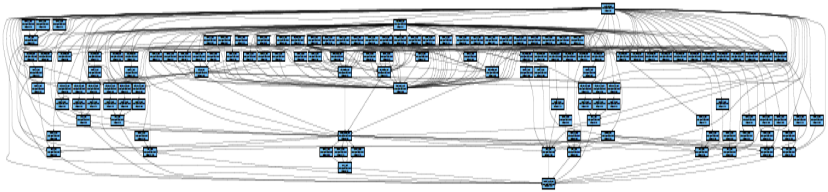
Scientific Computing in our Microcosm

Local evolution of scientific computing

- Serial Fortran programs
- MPI everywhere (inter- and intra-node)
- C, C++
- MPI+X, X is Pthreads, OpenMP, OpenCL, CUDA, etc.
- Parallel runtimes, e.g., Cilk++, Intel Threading Building Blocks, Stanford's Legion, etc.

Currently MPI+threading model dominates.

In Search of Automatic Parallelization *or at least automatic scheduling*


[1]

S3D task dependency, combustion chemistry calculation

Simplest interesting chemistry, task graph much larger with more complex reactants. Schedule by hand?

[1] Courtesy Stanford Legion project

Transparent Fault Tolerance?

Checkpoint:

1. Processes synchronize at predetermined point;
2. Stop the world;
3. Dump global state;
4. Resume computation.

Restart:

1. Observe that application has hung/crashed;
2. Identify last valid (complete) checkpoint image;
3. Re-launch application specifying checkpoint image.

General claim: “data movement costs more energy than computation.”
Fact 1: In C/R, idle processors may cost more energy than data movement.
Fact 2: C/R isn’t scaling.

Pure Functional Semantics

Now we have a ‘new’ generation of scientific programmers, aka computational scientists, who have some understanding of meaning and virtue of *pure functional*, and even dabble in Haskell programming.

In a multi-100,000-line program, do not temporarily alter the global speed-of-light ‘constant’ variable.

But, these are not computer scientists:

- Passing functions as arguments is familiar (since early Fortran);
- Implicit space leaks are confusing;
- Non-strict evaluation is largely irrelevant, strictness annotations are unintuitive.
- Unboxed primitives and arrays of the same (unboxed vector) are the primary data.

Claimed Trends

Trends in functional programming

- Appreciation of the virtues of strict-by-default semantics;
- Transparent fault tolerance.

Trends in scientific computing

- Implementation of pure functional computational concepts/components.
- Recognition that checkpoint/restart is increasingly intractable and unscalable.

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\Longrightarrow$
function arguments can be evaluated early, in parallel

- Strictness analysis
- Bang patterns
- Par/pseq, other specifications
- Speculative evaluation
- …
- *Strict(er) default semantics*

Similarly constructors.

Update: Strict `let` bindings.

Musings

Suppose we had a strict(-er) pure functional language implementation than Haskell?

- Could we keep a large number of hardware threads (e.g., up to 256) busy with *implicit* parallelism?
  - With what parallel efficiency?
    - With different scheduling strategies?
- How often do new functional programmers/old scientific programmers make essential use of non-strictness? *(very rarely)*
  - Laziness? *(never)*

Project

A *light-weight* implementation of a pure, higher-order, polymorphic, functional language and runtime system with which we can experiment with automatic parallelization strategies with varying degrees of language strictness, and secondarily, with mechanisms for transparent fault tolerance.

1. Haskell to STG (or Core) via GHC *(todo)*;
2. STG to C *(serial done, parallel in progress)*;
3. Fault tolerance *TBD*.

STG and Core are higher-order polymorphic functional languages in their own right.

Reinventing the wheel?

Why not use Manticore, MultiMLton, F#, multicore OCaml, etc., or even GHC in toto?

- Very light weight: easy to instrument, modify;
- Know exactly what is going on under the hood;
- There exist conduits for getting student interns up to speed inexpensively;
- We have a small cadre of scientists already interested in Haskell/GHC.

Technical hurdles: tail calling and unwinding the stack

Following SPJ/STG our control structure is a stack of continuations.

Each continuation contains an address to *go to next*, not *go back to*, so we want something like *goto* or *tail calling*.

SPJ identifies defines the term *code label* as something that can be

- used to name an arbitrary block of code;
- can be manipulated, i.e., stored for later use;
- can be used as the destination of a jump (not just call).

All the myriad ways in C

- setjmp/longjmp;
- giant switch;
- computed goto (gcc extension);
- trampoline, the simplest of which parameterless C functions return the address (function pointer) of where to go to next:

```
while (1) f = (*f)();
```

- implement an intermediate language, e.g., C--/Cmm, where tail call is primitive;
- generate assembly code, e.g., LLVM, directly.

Modern compilers to the rescue: sibling call

Clang/LLVM and GCC implement a restricted form of tail call, the *sibling* call, i.e, jump reusing the TOS frame. Here

```
void f() {
  ...
  g();
}
```

function `g` is jumped to; there is no (implicit) return following `g();`.

Critically, this also works for indirect calls to previously stored addresses.

```
    (getInfoPtr(stgCurVal.op)->entryCode)();
```

Sibling Call

The restrictions are due to the x86(-64)/*nix/C calling conventions—caller clean-up—and the possibility of pointer aliasing.

Sufficient formula for recent gcc and Clang/LLVM on Linux or MacOS X

- Call in leaf position of CFG;
- Caller and callee have the same type signature
- No addresses taken of formal params;
- -O3.

Obviously this is not portable.

Unwinding stacks

In one version of GHC, Harris et al. identify the need to unwind thread stacks for shared memory parallelism.

Again, we do not want to contemplate using, e.g., setjmp/longjmp.

Therefore, we do not want the C main stack or Pthreads stacks to grow.

Exception: calls to the runtime.

Again, sibling call saves us.

Here is a simple frame

Here is a simple frame

- Here is the first bullet in a standard bulleted list, as is customary for a presentation like this
  - It includes some sub-bullets
  - Here they are
- Here is another bullet
- And here is another, with some math:

$$e^{i\theta} = \cos\theta + i\sin\theta$$

Example frame

- Here is a bulleted list that sits alongside a graphic
- With a second bullet item
- And a third
  - It can have sub-bullets too
  - Like this

insert caption here