LA-UR-16-24056

Automatic Parallelization and Transparent Fault Tolerance

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

Scientific Computing in our Microcosm

Local evolution of scientific computing

- Serial Fortran programs
  Simple, imperative.

Scientific Computing in our Microcosm

Local evolution of scientific computing

- Serial Fortran programs
- C
  Similar. More pleasant syntax (Free form source code!)

Scientific Computing in our Microcosm

Local evolution of scientific computing

- Serial Fortran programs
- C
- Vendor communication libraries
  Non-portable, but parallel

Scientific Computing in our Microcosm

Local evolution of scientific computing

- Serial Fortran programs
- C
- Vendor communication libraries
- MPI everywhere (inter- and intra-node)
  De facto standard emerges to solve portability

Scientific Computing in our Microcosm

Local evolution of scientific computing

- Serial Fortran programs
- C
- Vendor communication libraries
- MPI everywhere (inter- and intra-node)
- C++
  Object oriented goodness (and more!)

Scientific Computing in our Microcosm

Local evolution of scientific computing

- Serial Fortran programs
- C
- Vendor communication libraries
- MPI everywhere (inter- and intra-node)
- C++
- MPI+{Pthreads, OpenMP, OpenCL, CUDA, etc}
  We augment MPI now with other explicit parallel models

Scientific Computing in our Microcosm

Local evolution of scientific computing

- Serial Fortran programs
- C
- Vendor communication libraries
- MPI everywhere (inter- and intra-node)
- C++
- MPI+{Pthreads, OpenMP, OpenCL, CUDA, etc}
- Parallel runtimes, e.g., Cilk Plus, Intel Threading Building Blocks, Stanford's Legion, etc.
  Automated scheduling for irregularly shaped problems.

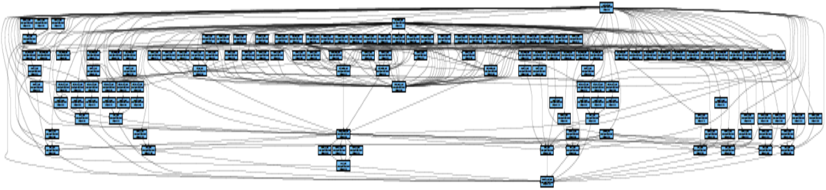
Scientific Computing in our Microcosm

Local evolution of scientific computing

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

Currently MPI+threading model dominates.

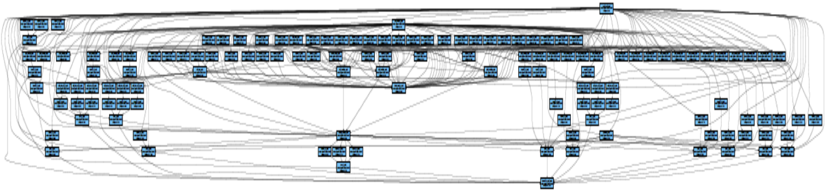
In Search of Automatic Parallelization
or at least automatic scheduling


[1]

S3D task dependency, combustion chemistry calculation

[1] Courtesy Stanford Legion project

In Search of Automatic Parallelization
or at least automatic scheduling

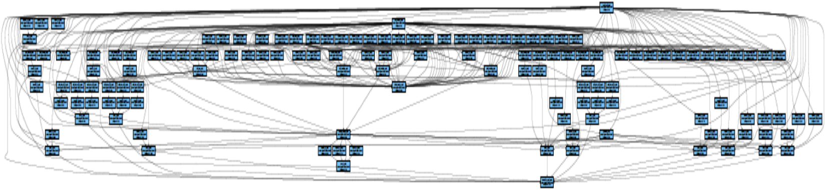
 [1]

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

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?

This comes for free in the pure functional world

[1] Courtesy Stanford Legion project

Transparent Fault Tolerance?

Processor clock speeds stopped growing a decade ago

Transparent Fault Tolerance?

Processor clock speeds stopped growing a decade ago

Solution? More processors.

Transparent Fault Tolerance?

Processor clock speeds stopped growing a decade ago

Solution? More processors.

But, with more processors comes smaller Mean Time to Failure.

Transparent Fault Tolerance?

Processor clock speeds stopped growing a decade ago

Solution? More processors.

But, with more processors comes smaller Mean Time to Failure.

Obviously, we can't just hope to get lucky on a months-long calculation

Transparent Fault Tolerance?

Checkpoint:

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

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.

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.

Inherent problems:

- Idle processors burn tight energy budgets.
- MTTF is shrinking and C/R isn't scaling.

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.

Inherent problems:

- Idle processors burn tight energy budgets.
- MTTF is shrinking and C/R isn't scaling.

With referential transparency, this is a much smaller problem

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.

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.

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

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.

They understand:
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.
- 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.
  GHC Strict and StrictData for HPC

Claimed Trends

Trends in functional programming

- Appreciation of the virtues of strict-by-default semantics.
- Transparent fault tolerance.
  Stewart's HdpH-RS

Claimed Trends

Trends in functional programming

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

Trends in scientific computing

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.
  Legion is “pure with respect to *X*”
  Also adding task-restarting for fault-tolerance

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\Longrightarrow$

function arguments can be evaluated early, in parallel

How can we achieve this?

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\Longrightarrow$

function arguments can be evaluated early, in parallel

How can we achieve this?

- Strictness analysis

  Can be inconclusive.

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\Longrightarrow$

function arguments can be evaluated early, in parallel

How can we achieve this?

- Strictness analysis
- Bang patterns in bindings

  Prone to error. Hard to read.

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\Longrightarrow$

function arguments can be evaluated early, in parallel

How can we achieve this?

- Strictness analysis
- Bang patterns in bindings
- Par/pseq, other specifications on expressions

  Ditto.

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\Longrightarrow$
  function arguments can be evaluated early, in parallel
  How can we achieve this?

- Strictness analysis
- Bang patterns in bindings
- Par/pseq, other specifications on expressions
- Speculative evaluation
  Difficult to get right without annotation.

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\implies$

function arguments can be evaluated early, in parallel

How can we achieve this?

- Strictness analysis
- Bang patterns in bindings
- Par/pseq, other specifications on expressions
- Speculative evaluation
- …

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\implies$
function arguments can be evaluated early, in parallel
How can we achieve this?

- Strictness analysis
- Bang patterns in bindings
- Par/pseq, other specifications on expressions
- Speculative evaluation
- ...
- *Strict(er) default semantics*

Non-strictness/laziness Anathema to Parallelism

(Semantic) function strictness $\Longrightarrow$
function arguments can be evaluated early, in parallel
How can we achieve this?

- Strictness analysis
- Bang patterns in bindings
- Par/pseq, other specifications on expressions
- 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?

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.

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

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)*;
   Fairly faithful to STG papers

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. Mini-Haskell front end (done);
   Close to ML, syntactically

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. Mini-Haskell front end (done);
4. Fault tolerance *TBD*.

Reinventing the wheel?

Why not use Manticore, MultiMLton, F#, multicore OCaml, SAC etc.,

Reinventing the wheel?

Why not use Manticore, MultiMLton, F#, multicore OCaml, SAC etc.,

- We have a small cadre of scientists already interested in Haskell/GHC;

Reinventing the wheel?

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

- We have a small cadre of scientists already interested in Haskell/GHC;
- Very light weight: easy to instrument, modify;

Reinventing the wheel?

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

- We have a small cadre of scientists already interested in Haskell/GHC;
- Very light weight: easy to instrument, modify;
- Know exactly what is going on under the hood;

Reinventing the wheel?

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

- We have a small cadre of scientists already interested in Haskell/GHC;
- 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.

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;
  Significant overhead and bookkeeping

All the myriad ways in C

- setjmp/longjmp;
- giant switch;
  Ugly hack, not really linkable

All the myriad ways in C

- setjmp/longjmp;
- giant switch;
- computed goto (gcc extension);
  Limited to gcc

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)();
```

  Our initial implementation used this approach

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;

  Fun, but out of scope

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.
  Ambitious. Haskell LLVM bindings are unreliable

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 Mac OS 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.

Our Future

- GHC tie-in and parallel implementation (hopefully) this summer

Our Future

- GHC tie-in and parallel implementation (hopefully) this summer
- Experimentation with non-trivial code

Our Future

- GHC tie-in and parallel implementation (hopefully) this summer
- Experimentation with non-trivial code
- Fault tolerance sometime thereafter

Acknowledgments

Funding for this project has been provided by the DOE NNSA LANL Laboratory Directed Research and Development program award 20150845ER; the National Science Foundation Science, Technology, Engineering, and Mathematics Talent Expansion Program (NSF STEP) program for undergraduate students; and the Department of Energy Science Undergraduate Laboratory Internship (DOE SULI) program.

Los Alamos National Laboratory is managed and operated by Los Alamos National Security, LLC (LANS), under contract number DE-AC52-06NA25396 for the Department of Energys National Nuclear Security Administration (NNSA).

Thank you

Questions?