Semi-Automatic Extraction of Software Skeletons for Benchmarking Large-Scale Parallel Applications

Matthew Sottile, Amruth Dakshinamurthy, Gilbert Hendry, Damian Dechev

May 20, 2013

Motivating HPC problem

In High Performance Computing (HPC), we want to:

- Design parallel computers to run relevant applications well.
- Design parallel applications to run well on new computers.

Common design methods

- How did we do system design in the past?
  - Generic benchmark suites representing relevant applications.
  - Aim for highly generic metrics (e.g., Linpack performance).
- Design applications around generic programming models.
  - Message passing, threading, etc. . .
  - Independent of system nuances.
  - Defer nuance to parameters (or `#ifdef` blocks).
  - Pay a flock of programmers to adapt apps to new machines.

Why is this suboptimal?

- Apps tend to be too generic.
  - Fail to tune for the right parameters for new machines.
- Systems tend to be either too generic or too foreign.
  - Design for generic implementations of algorithms.
  - Miss nuances that appear in real codes.
  - Surprise app developers with new features.

Co-design hypothesis

Systems and apps might be better suited for each other if they were designed with knowledge of the other from the outset.

Successful approach in embedded systems community!

HPC co-design challenge

- Building machines is expensive.
  - E.g.: ORNL Jaguar cost $104 million.
- Simulation before construction to experiment with parameters.
- Simulations are expensive with respect to time, not dollars.
- Want to optimize simulation throughput:
  - Minimize time to results for experimenting with system designs.

How do we reduce simulation time yet preserve fidelity with respect to real applications?

Skeletons

What is a skeleton?

- A simplified program derived from a source application.
- Retains approximate performance characteristics of interest.
- Removes program logic that is orthogonal to performance properties to study.
- Suitable for use in simulation.
  - Assume simulator links with source code of skeleton program.

Skeletons

A full application has a number of relevant performance dimensions.

- Memory traversal pattern
- Disk I/O
- Floating point operations
- Branching patterns
- Network / message passing I/O

More than one skeleton for a given application is possible!

- This is one reason (semi-)automation is attractive.
- Rapid generation of skeletons lets people experiment more than they would have otherwise.

Making a skeleton

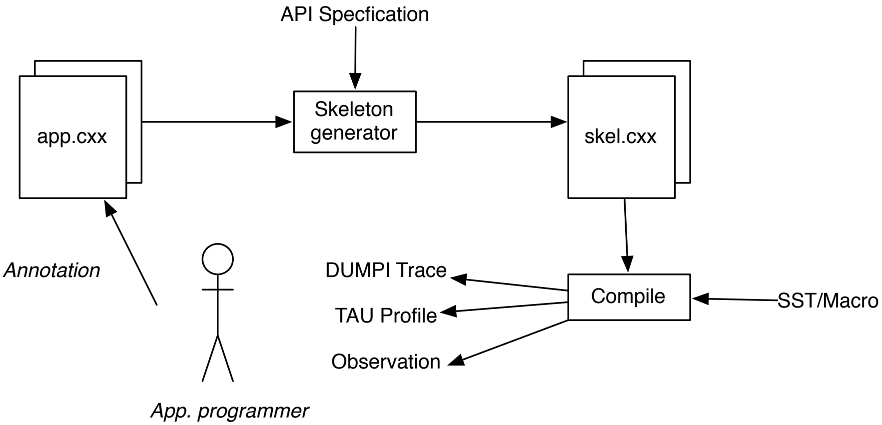


Figure : Iteration process

Methodology

- API specification
  - What functions make up API?
  - What data roles exist for the API?
  - Relation of data roles to function arguments.
- Static analysis
  - Dependency analysis.
- Code transformation
  - Slicing and outlining.
- Simulation via SST/Macro

Static analysis

- Need to identify code that *must* be preserved (API calls)
  - ...then identify code that must be preserved such that API calls still function.
- Label code based on how it interacts with the API.
  - Define "roles" for arguments.
    - Message payload, messaging topology, error codes, etc.
  - Allows finer control over how dependencies are treated.
- Start with static single assignment (SSA) form for code.
  - Common compiler technique.
  - Gives very regular structure that makes analysis easy.
  - ROSE provides bridge between AST and SSA form.

API specification

- Dependency types that are meaningful to roles within this API.
  - Payload, topology, tag, other
- Default dependency type for arguments not specified.
  - Essentially, things we aren't concerned with tracking.
- For each function:
  - What is its name?
  - How many arguments does it have?
  - For each argument, what dependency type does it have?

```
(MPI_Send   6  (payload 0 1 2) (topology 3) (tag 4) )
```

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest,  int tag,   MPI_Comm comm)
```

API specification : MPI subset

```
(api-spec MPI
  ( dep-types payload topology tag other )
  ( default_deptype other )

  ( (MPI_Init         2 )
    (MPI_Finalize     0 )
    (MPI_Abort        2 )

    (MPI_Comm_rank    2  (topology 1) )
    (MPI_Comm_size    2  (topology 1) )
    (MPI_Comm_split   4 )

    (MPI_Send         6  (payload 0 1 2)
                         (topology 3)
                         (tag 4) )
  )
)
```

Dependency propagation

Given a subroutine:

- Identify all API calls in the API specification.
- For each argument to these calls, identify all code that the arguments depend upon prior to the call.
- For all identified dependencies, repeat, finding code they depend upon.
- Repeat until all code analyzed.

Static single assignment form makes this relatively straightforward.

Pragma guidance: `#pragma skel ...`

Pragmas provided to allow users control over skeletonizer.

- `preserve`, `remove`
  - explicit preservation or removal of code.
- `iterate atmost/atleast/exactly`
  - loop iteration count control.
- `initialize`
  - specify value for variable initialization.
  - basic scalar implementation - next steps will be to provide array value and size initializers.
- `branch`
  - branch true with given probability.
  - recognized, but not currently used.

SST/Macro Simulator

Our skeletons were used with the SST/Macro simulator from Sandia.

- Skeleton code + simulator implementation of MPI.
  - MPI calls modeled by simulator based on parameters that define machine characteristics.
  - Modifications to skeleton minimal : change MPI include, rename `main()` since simulator wants `main()`.
- DUMPI trace library used to record behavior of skeleton under simulation.
  - DUMPI provides analysis tools as well for comparing traces.

For more information, see: `http://sst.sandia.gov/`

Results with SST/Macro

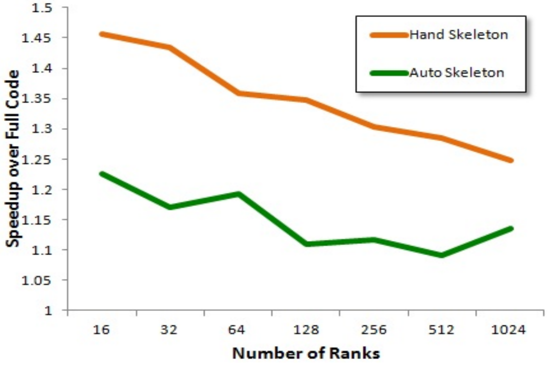
- Skeletons were examined by hand and executed under SST/Macro.
- These cases were small enough to allow for original app to be simulated.
  - Allow detailed comparison with skeleton.

Findings:

- Trace analysis with DUMPI showed equal bulk message count.
- Message counts between rank pairs also equal.
- Annotations were necessary to achieve good skeletons.

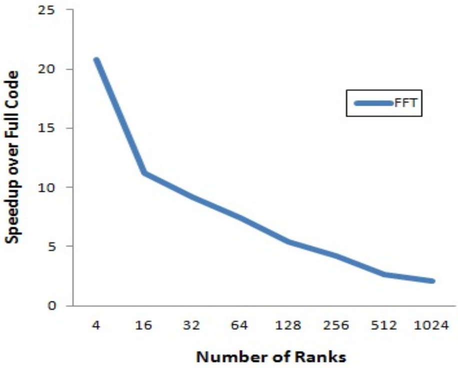
Three specific cases: HPCCG, 2D FFT, Jacobi.

HPCCG



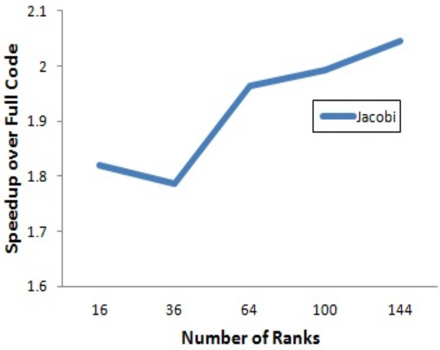
- Auto-generated skeleton exhibited speedup over base code.
- Hand-written skeleton outperformed auto-generated one.
- Auto-generated skeleton required minimal annotations.
  - Loop iteration counts and forced `preserve`.

FFT



- Code: Local FFT → Collective communication → Local FFT
- Removal of compute code = only collective operations remain
- Example of a skeleton for an app exhibiting *strong scaling*.
  - Fixed problem size: less work per CPU as CPU count increased.

Jacobi



- Simple stencil code: not much left when computations removed.
- Example of a skeleton for an app exhibiting *weak scaling*.
  - Fixed problem size per CPU: total problem size grows with more CPUs.
- Curve should be flat if averaged over many trials.

Auto-skeletonization challenges

- These were programs with static communication.
  - What about dynamic patterns?
  - How much can we leverage annotations here?
- What about asynchronous communications?
  - Difficult to validate with simulator.
- What about data dependent communications?

Ongoing work and opportunities for other researchers.

- Formal analysis of code to generate a replacement.
- Integration of trace information at skeletonization time.
  - Potential intersection with work by Subhlok from IPDPS 2008.

Wrapping up

- Code available with ROSE (new version this week!)
  - See `projects/extractMPISkeleton/` in ROSE distribution for details.
  - Contact : Matthew Sottile (`mjsottile@gmail.com`)
- Alternative skeletonizer underway:
  - Memory footprint skeletonizer.
  - Relevant for studying design decisions at the memory system level, versus the interconnect level that the MPI skeletons probe.

Thanks to the Lawrence Livermore National Laboratory and the US Dept. of Energy Office of Science, Advanced Scientific Computing Research for supporting this work.