Semi-automatic extraction of program skeletons for parallel program performance analysis.

Matthew Sottile, Amruth Dakshinamurthy, Gilbert Hendry, Damian Dechev

April 8, 2013

Outline

- Problem statement
  - Co-design in HPC
  - What is the role of a program skeleton?
- Methodology
  - Static analysis
  - Skeletonization specification
  - Simulation
- Results
  - Preliminary results from SIGPADS paper
- Discussion
  - Ongoing work

Basic problem

In HPC, we want to:

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

Past methods: computers

- How did we do system design in the past?
  - Established benchmark suites for measuring both performance and programming models.
    - ★ NAS parallel benchmarks
    - ★ PARKBENCH suite
    - ★ Salishan/Cowichan problems
    - ★ etc. . .
  - Aim for highly generic metrics (e.g., Linpack performance).
  - Matrix methods; sorting; fast Fourier transform; multigrid; AMR; etc. . .
  - Microbenchmarks.

Past methods: applications

- Systems tend to have common flavors.
  - Shared memory parallelism: OpenMP.
  - Distribured memory parallelism: MPI.
  - Accelerators: CUDA or OpenCL, compiler directives.
- Design applications around these generic programming models.
  - Independent of system nuances.
  - Defer nuance to parameters (or `#ifdef` blocks).
  - Pay a flock of programmers to adapt apps to new machines as they arise.

Why is this suboptimal?

- Apps tend to be too generic.
  - Fail to tune for the right parameters for new machines.
  - Unsurprising: app writers don't know what they are!
- Systems tend to be either too generic or too foreign.
  - Design for generic implementations of algorithms.
  - Miss nuances that appear in real codes.
  - Provide high-payoff but strange features, foreign to developers.
- 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.

Co-design in other spaces

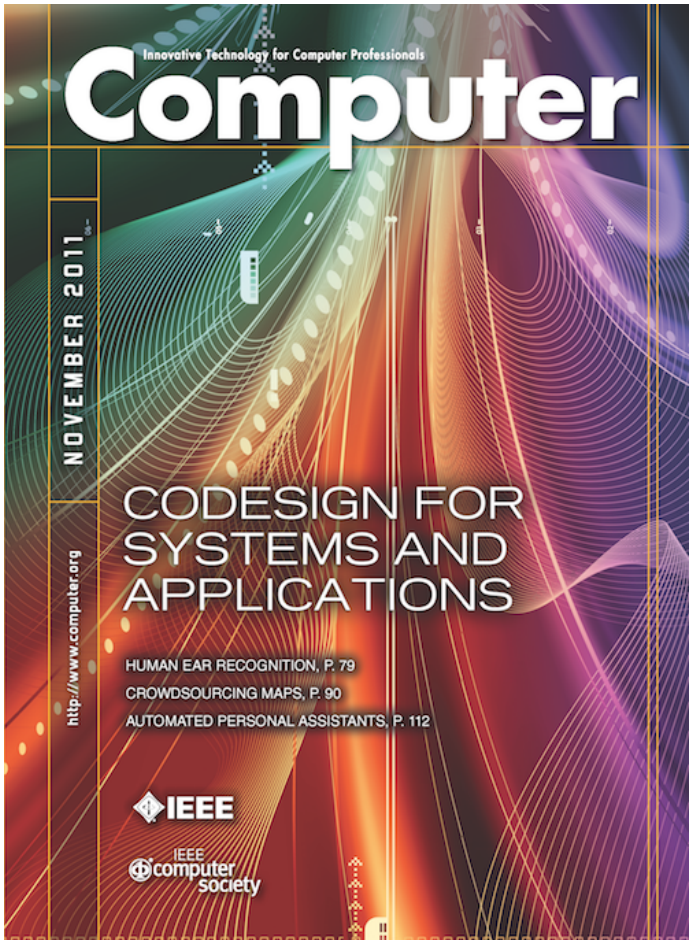
"Embedded computing is unique because it is a hardware-software co-design problem — the hardware and software must be designed together to make sure that the implementation not only functions properly but also meets performance, cost, and reliability goals."

Wayne H. Wolf, Proceedings of the IEEE, Vol. 82, No. 7, July 1994.

Co-design in HPC

This design approach has been gaining traction in the HPC recently.

Recommended reading: IEEE Computer, Vol. 44 Issue 11 (2011).



HPC co-design challenge

- Building machines is expensive.
  - E.g.: ORNL Jaguar cost $104 million.
- Simulations of machines before construction is necessary to experiment with parameters.
  - CPU designers have known and done this for years.
  - Embedded community as well (see previous quote).
- Simulations are expensive with respect to time, not dollars.
- Want to optimize simulation throughput:
  - Minimize time to results for experimenting with system designs.

HPC co-design simulation bottleneck

- Applications are the issue.
- Simulators for big parallel machines consume real applications.
  - SST/Macro is the simulator of choice for our co-design work.
- Running full applications under simulation will take far too long.
  - These apps already run for a long time on bare metal.

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.

Example code snippet : pre-skeletonization

```
MPI_Init(argc, argv);
initialize(x,N);

do {
    for (int i=1;i<N-1;i++) {
        y[i] = (x[i-1]+x[i]+x[i+1]) / 3.0;
        err = err + fabs(x[i]-y[i]);
    }

    if (rank < nproc) {
        MPI_Recv(...); MPI_Send(...);
    } else {
        MPI_Send(...); MPI_Recv(...);
    }

    MPI_Allreduce(err,...);
} while (err > epsilon);

MPI_Finalize();
```

Example code snippet : post-skeletonization

```
int iters = 0;

MPI_Init(argc, argv);

do {
    if (rank < nproc) {
        MPI_Recv(...); MPI_Send(...);
    } else {
        MPI_Send(...); MPI_Recv(...);
    }

    MPI_Allreduce(err,...);
    iters++;
} while (err > epsilon && iters < MIN_ITERS);

MPI_Finalize();
```

Computations removed, MPI retained, loop controls added to compensate for now-bogus error computation.

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

Often system designers only need a subset of characteristics accurately modeled in simulation.

- Interconnect designer likely doesn't need to know much about expected FP load on CPU.

Skeletons are not unique

Very important observation.

- Program $\Rightarrow$ Skeleton is not one-to-one.
- Many possible skeletons from a single program.

This is one reason (semi-)automation is attractive.

- Rapid generation of skeletons lets people experiment more than they would have otherwise.

Making a skeleton

What goes into skeleton generation?

- Static code analysis.
- User guidance to augment analysis.
  - Overriding analysis results manually.
  - Filling gaps where analysis falls short.
- Specification of "skeletonization points".
- Code transformation and generation.

Skeletonization is an iterative process.

Iteration

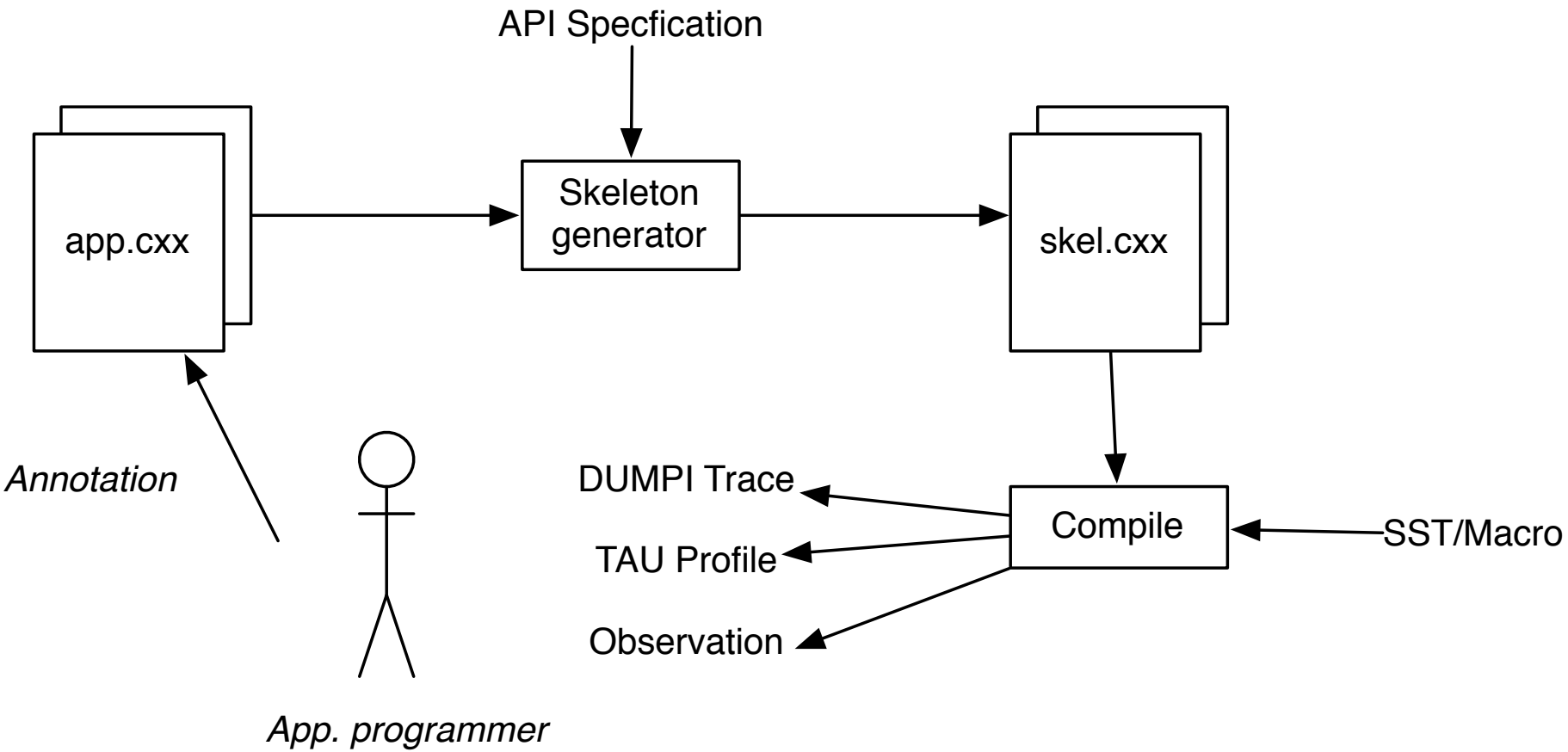


Figure : Iteration process

Challenges

- Skeleton concept is relatively straightforward.
  - Implementation of generator is not.
  - Dealing with full languages (ANSI C, C++, Fortran).
    - ⋆ We still break parts of the compiler infrastructure with production code.
- Heuristics to guide skeletonizer are not obvious.
  - E.g., `while` loop transformation to maintain expected iteration count.
- Working with real examples made necessary annotations clear.
- Understanding what constitutes a "correct" skeleton.
  - Still an ongoing research topic for codes with dynamic behavior.

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

Tools

- ROSE compiler framework from LLNL.
- Why ROSE?
  - Source-to-source: skeletons resemble original code.
  - Aids in understanding meaning of skeleton relative to original code.
  - Allows skeleton to be shared when cross-compilers unavailable.
- Other compiler frameworks provide similar analysis functionality
  - . . . but without source-to-source view.
  - The result is code that has been lowered too far.

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.

Static single assignment form

- All variables are defined (assigned to) exactly once.
  - A new variable is created for each definition.
- Define-use chains are created in which each use of a variable can immediately determine the most recent definition.
  - This is key for this work: API calls are *uses* that we start from.
  - Code roles labeled based on traversing def-use chains back.
- Special statements (*phi* functions) are introduced to resolve branches in def-use chains due to conditionals, loops, etc. . .

```
// PRE-SSA
x = 1
x = x + 1
y = 4
z = x + y
```

```
// POST-SSA
x0 = 1
x1 = x0 + 1
y0 = 4
z0 = x1 + y0
```

API specification

- APIs define functions that we want to preserve.
  - MPI, HDF5, STDIO, etc.
  - Application specific set of functions for a large app that represent an interesting API to study.
- Each API function is specified with a role for each argument.
- API specification includes definition of roles for that API.
  - Skeleton generator designed to be independent of any specific API.

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) )
  )
)
```

API specification detail

- 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)
```

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.

Two-pass approach

- Two passes are sufficient.
  - **Pass 1**: Analyze each function to generate a signature indicating whether or not API calls occur in it, and if so, what role the function arguments take on as a result. All functions that are called are recorded as well.
  - **Pass 2**: Reconstruct the full program call graph from the signatures. Compute transitive closure to determine all functions reachable from each.

Caveats

- Function pointers treated as if they were API calls
- C++ methods require additional step of inheritance hierarchy analysis
  - Identify virtual functions where a derived class may invoke API functions

Outlining vs removal

Outlining preserves code, but separates skeleton from "fillet". Allows us to further analyze code that was removed from skeleton for possible replacement vs. wholesale removal.

```
// original
void foo() {
  x = 4;
  y = 5;
  z = 6;
}
```

```
// outlined
void foo() {
  foo_outlined();
}

void foo_outlined() {
  x = 4;
  y = 5;
  z = 6;
}
```

```
// removed
void foo() {
}
```

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/`

Test cases

We developed the skeletonizer with a number of test cases.

- Sandia Mantevo suite
  - HPCCG, Mini-MD, Mini-FE
- NAS Parallel Benchmarks
- Small MPI examples
  - 2D FFT, Jacobi iteration

Results for three presented here.

Results with SST/Macro

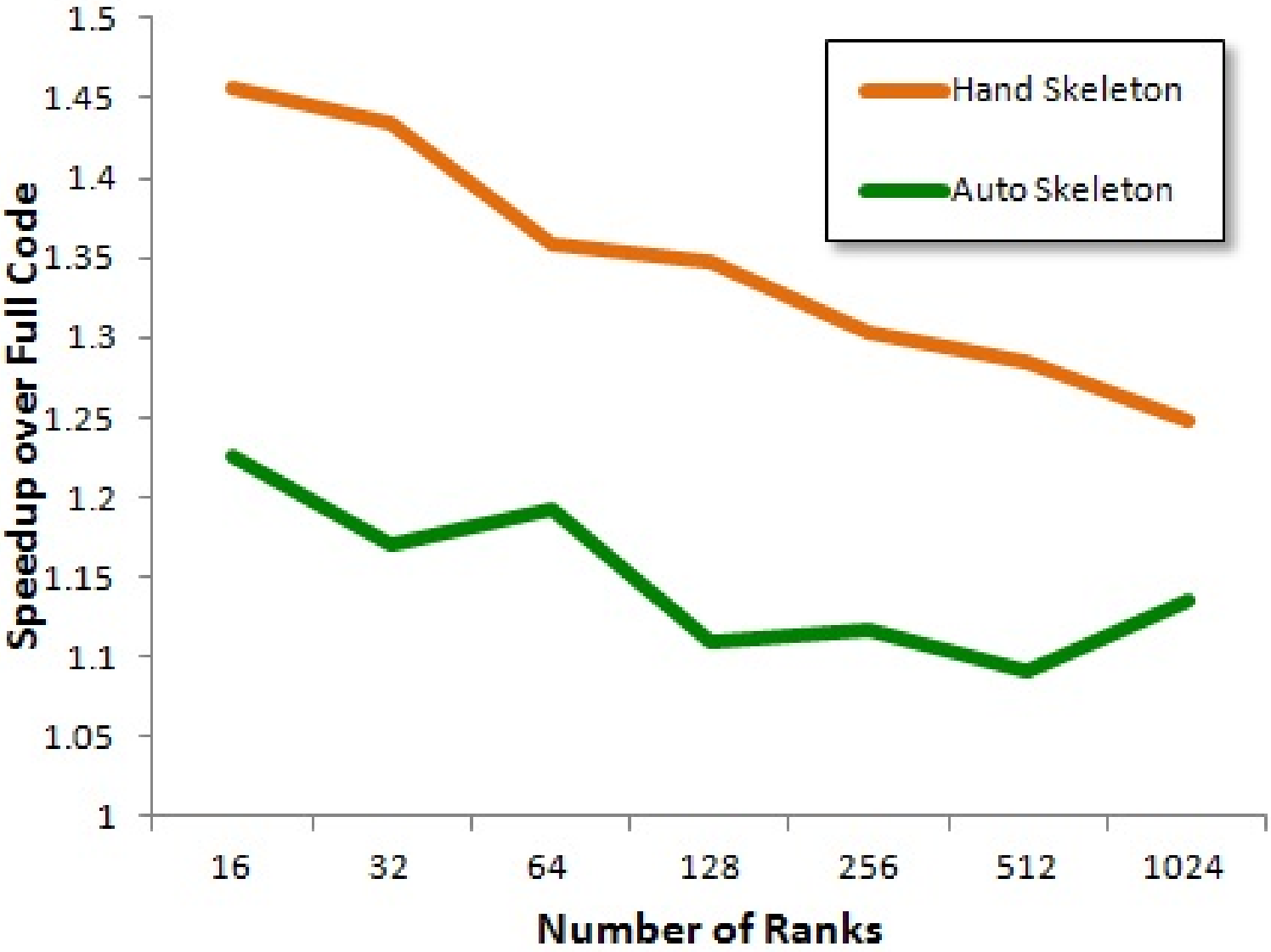
- Skeletons were examined by hand and executed under SST/Macro.
- These cases were small enough to allow for original app to be simulated.
  - Allow detailed comparison with skeleton.

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.

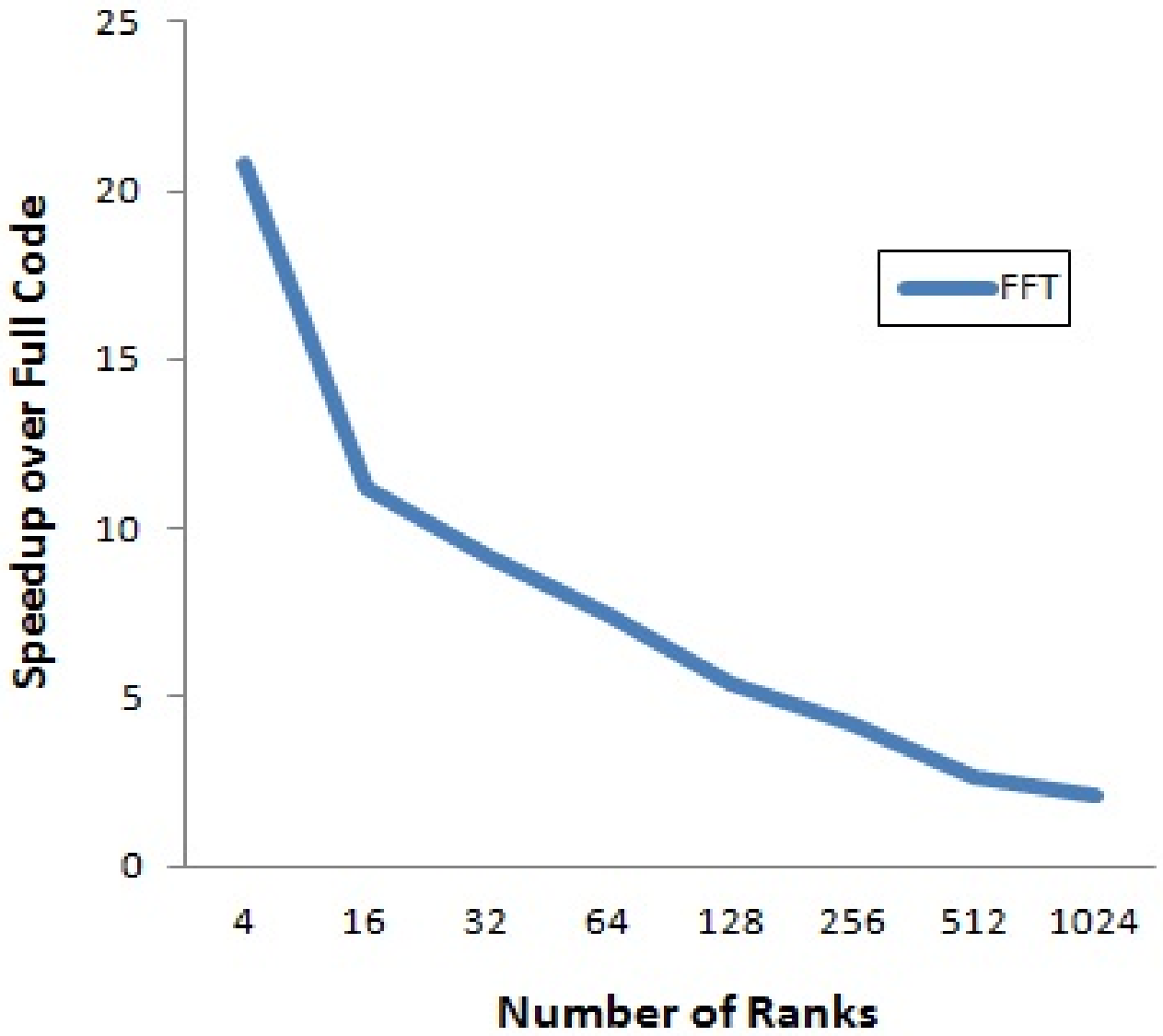
HPCCG


Hand Skeleton
Auto Skeleton
Speedup over Full Code
1.5
1.45
1.4
1.35
1.3
1.25
1.2
1.15
1.1
1.05
1
16
32
64
128
256
512
1024
Number of Ranks

HPCCG: Discussion

- Auto-generated skeleton exhibited speedup over base code.
- Hand-written skeleton outperformed auto-generated one.
- Auto-generated skeleton required minimal annotations
  - Most were `preserve` directives due to interprocedural dependency analysis being missing at the time.
  - One annotation to ensure iteration counts were realistic.

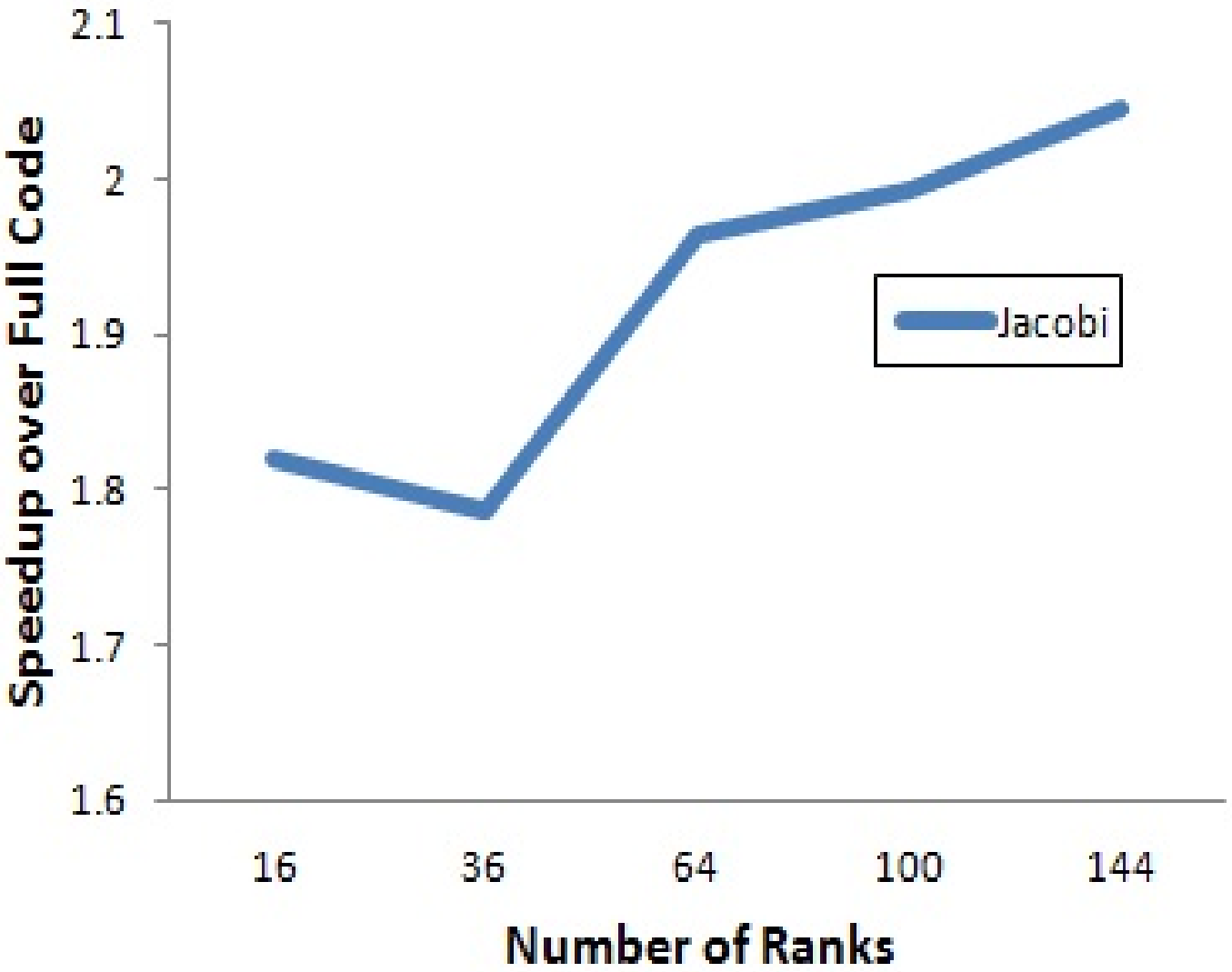
FFT


25
20
15
10
5
0
Speedup over Full Code
FFT
4
16
32
64
128
256
512
1024
Number of Ranks

FFT: Discussion

- Code structure was essentially:
  - Compute local FFTs without communications.
  - Execute four collective operations.
  - Compute more FFTs to complete full FFT.
- Removal of compute code caused the skeleton to turn into:
  - Execute four collective operations.
- Example of a skeleton for an app exhibiting *strong scaling*.
  - Fixed problem size, so less work per processor as processor count increased.
  - This is apparent in the speedup vs nproc data.

Jacobi


2.1
2
1.9
1.8
1.7
1.6
Speedup over Full Code
16
36
64
100
144
Number of Ranks
Jacobi

Jacobi: Discussion

- Simple stencil code.
- Example of a skeleton for an app exhibiting *weak scaling*.
  - Problem size per processor fixed.
  - Total problem size grows with processor count.
- Original code less sophisticated than HPCCG, so like FFT, removal of computation noticable.
- Speedup really a function of problem size per CPU.
  - Curve shape should stay the same, just move up as problem size grows.

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.

More info

- Code available with ROSE.
  - See `projects/extractMPISkeleton/` in ROSE distribution for details.
- 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.
- Interested in people with small apps who want to try to generate skeletons to kick the tires of the tool.
- Thanks to LLNL/DOE ASCR for supporting this work.