SAND2017-12240C

Compiler-assisted Source-to-Source Skeletonization of Application Models for System Simulation

Jeremiah J. Wilke, Joseph P. Kenny, and Samuel Knight

Sandia National Laboratories,
7011 East Ave, Livermore, CA, USA
{jjwilke,jpkenny,sknigh}@sandia.gov

Abstract. Lightweight endpoint models for system-scale simulation can be challenging to develop. Today, endpoint models consist mainly of post-mortem trace replay, but these off-line simulations lack flexibility and may not be scalable. On-line simulations running so-called skeleton application code offer greater scalability and flexibility, but often need to be custom written. Auto-skeletonization of existing application source code via compiler tools would provide lightweight endpoint models with minimal development effort. However, these source-to-source transformations have only been narrowly explored with limited success. Using static analysis alone, it can be difficult or impossible to remove and model computation which is not needed to reproduce an application's network traffic pattern. We introduce an approach which uses a pragma language to provide compiler hints for auto-skeletonization. Skeletonization is data-centric, requiring large data structures unnecessary for preserving traffic patterns to be explicitly marked by application developers. The remaining process is guided by the compiler, automatically indicating where auto-skeletonization fails and where further hints are required. We describe the compiler toolchain and its usage, showing scalability beyond 100K endpoints for three example MPI applications (Lulesh, CoMD, and HPCG) using the Structural Simulation Toolkit (SST). Further,we validate weak and strong scaling curves, showing the flexibility of lightweight, on-line endpoint models.

1 Introduction

Simulations require application endpoint models to generate representative traffic or memory patterns. To achieve system scale, endpoint models need to be as lightweight as possible while still capturing the most important application features. Endpoint models can be generally classified as *off-line* or *on-line*. Off-line simulations replay post-mortem traces collected from an existing system. The traffic pattern is therefore fixed. On-line simulations instead run modified application codes to inject traffic into the simulator. These modified codes generally consist of either state machine models (motifs) [1] or skeleton applications [2].

Today endpoint models consist mainly of post-mortem trace replay. These off-line simulations lack flexibility and may not be scalable past 10s of thousands

of endpoints. Traces must be collected on a real system that is large enough to generate the desired scale, and the trace files may be very large (GB to TB). The traces are often only valid for the exact problem considered. Characterizing an application should ideally cover scaling behavior for a range of application inputs. Trace extrapolation can generate approximate large-scale traces from small-scale runs in some cases [3], but extrapolation presents validation concerns and is mainly useful for increasing scale, not tuning input parameters.

On-line simulations offer much greater scalability and flexibility since both scale and input parameters are tunable. On-line simulations, however, often rely on custom models written specifically for simulation, as done in state machine models [1]. Skeleton applications are regular source code with computation replaced with delay models. For message-passing applications (MPI), all MPI symbols are intercepted at the linking stage. MPI calls are then emitted to the simulator for modeling in place of a real network stack (Figure 1). While skeletons might be easy to derive from an existing application, they are less popular than trace replay because manually eliminating expensive computation currently requires significant human effort.

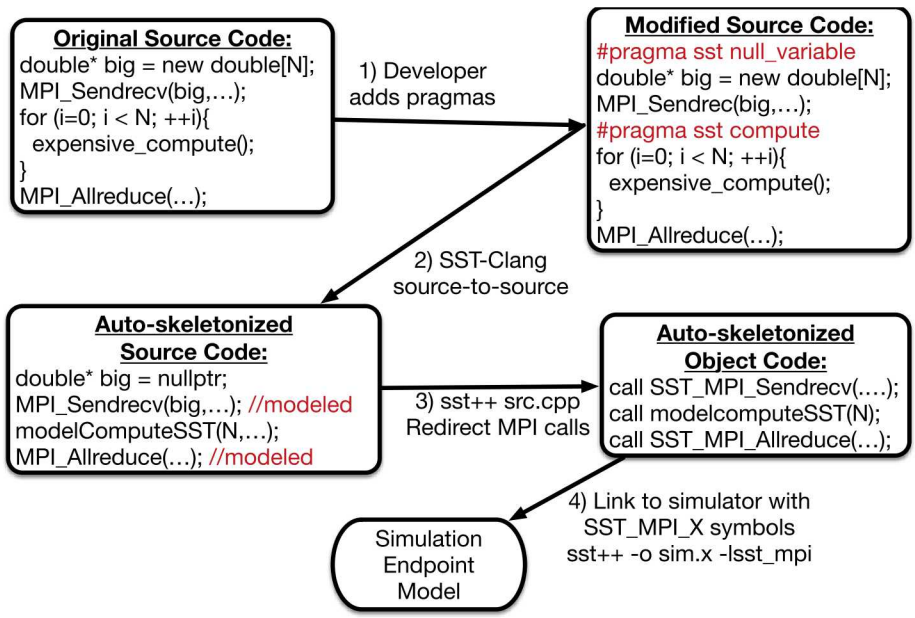


Fig. 1. Overview of skeletonization workflow for generating lightweight endpoint models from application source code. 1) Initial pragmas are added to direct skeletonization, 2) compiler completes source-to-source transformation with skeleton source code, 3) code is compiled with SST compiler wrapper on top of Clang/GCC, 4) MPI symbols linked to simulator.

Auto-skeletonization via source-to-source transformation of existing application source code would be a significant step for endpoint models. A single application source code that is usable for production and large-scale simulation improves 1) validation since the skeleton is directly derived from the real application, 2) co-design since any changes made in production immediately transfer to simulation and vice versa, and 3) ease of use since input parameters and system scales can be flexibly tuned. Auto-skeletonization through source-to-source transformations has been narrowly explored with limited success [2]. Distinguishing between computation blocks that can be safely elided and those that are required to reproduce the traffic pattern can be difficult or impossible at compile-time. An accurate traffic pattern further requires that skeletonization replace computation blocks with realistic delay models. Scalable skeletons also necessitate the removal of large memory allocations which would otherwise exceed the capacity of the simulator's hardware.

Instead of completely automatic skeletonization, we propose augmenting source code with skeletonization hints via pragmas. While pragma augmentation is not fully automatic (Figure 1), developers with domain-specific knowledge can easily add pragmas, in contrast to the tedious and error-prone task of manually eliminating expensive compute regions and memory allocations. In addition, rather than making conservative assumptions or highly approximate guesses, a compiler can report specific line numbers where static analysis fails and hints are required. This creates an almost automatic workflow in which the compiler tells the domain experts exactly where pragmas are needed.

To achieve this workflow, we describe an approach for user-guided application skeletonization. We introduce a simple pragma language which provides hints to a skeletonizing compiler and demonstrate this approach with a toolchain based on the Clang compiler library. We provide data-centric pragmas to eliminate expensive memory allocations and execution-centric pragmas to eliminate expensive computation. We demonstrate the approach for three applications (Lulesh, CoMD, and HPCG) including validations of the generated traffic patterns. We show the scalability of the approach with simulations beyond 100K endpoints using the Structural Simulation Toolkit (SST) running on a single node. We show full weak and strong scaling curves that both validate the correctness of the generated skeletons and demonstrate the flexibility of on-line simulations.

2 Related Work

2.1 Simulators

Numerous simulators have been developed with various accuracy/cost tradeoffs. On-line simulators usually emulate an API such as MPI [4] and link to application code, intercepting function calls to estimate elapsed time. Off-line simulators, such as Tracer/CODES [5], use time differences between trace communication events to determine compute delay. Time-independent traces with architecture-independent hardware counters have also been used [6, 7] as well as

application-specific task specifications [8]. Xu explored auto-generating skeletons from traces [9] without requiring source code transformations.

For estimating communication time, simulators usually model individual hardware components or use a fixed analytic function to provide timings for the overall system. Analytic functions often use a simple delay model, and some formulas try to incorporate congestion [3]. Structural simulators simulate discrete events on each switch and link as messages traverse the network with varying degrees of accuracy. There are high levels of detail in Booksim [10], packet-level models in SST/macro [11] and CODES [5], and more coarse-grained models or flow models in BSIM [12] and SMPI [13]. There are numerous other simulators with various off-line/on-line modes and different degrees of accuracy; these include MPI-SIM [14], PSINS [15], WARPP [16], and MARS [17].

2.2 MPI Source-to-Source

There are several studies that involve either source-to-source transformations of MPI codes or semi-automatic construction of communication skeletons. Guo et al. used source-to-source transformations of MPI codes to improve communication overlap [18]. Preissl et al. used the ROSE compiler to perform general performance optimizations [19]. Strout et al. performed data-flow analysis of MPI programs, although mainly in the context of automatic differentiation [20].

Sottile et. al. explored source-to-source skeletonization of MPI codes using ROSE [2]. Skeletonization was guided mainly by configuration files with some supplementary pragmas. The tool performed a def-use analysis to delete all code that did not affect the parameters inside an MPI call. This skeletonization is brute force, deleting all code that does not affect MPI function parameters, but also conservative, preserving all code that affects MPI parameters no matter how expensive to execute. Compute modeling of removed code was not performed.

The auto-skeletonization approach from Sottile et al [2] followed a bottom-up procedure, deriving backwards all code that could affect MPI parameters. Our approach is top-down and data-centric, labeling large data structures that should not be allocated. It prioritizes expensive code and memory allocation removal over conserving MPI call parameters. This also has the consequence that the MPI parameters may not be exactly preserved, instead being estimated. While our approach may require more input, the compiler directs domain experts to the lines of code where hints are needed and provides a rich set of pragmas to support skeletonization.

3 SST/macro

SST/macro is a Structural Simulation Toolkit (SST) element library designed for scalable modeling of large (potentially exascale) systems. This is achieved through 1) lightweight endpoint models, 2) coarse-grained approximations, and 3) brute force parallelization. Analytic network models can also be used instead of packet models for increased scalability.

SST/macro *emulates* many virtual MPI processes running in a single physical process. The terms virtual and physical are critical here to distinguish between the simulated application code and the simulator code itself. Without skeletonization, SST/macro becomes an MPI emulator and the C/C++ code compiled and linked with the `sst++` compiler wrapper must execute as it would on a real platform. To achieve this, SST/macro must emulate three memory regions used by physical processes:

- Heap memory
- Stack memory
- Global variables

On a real system, the kernel enforces memory separation between MPI ranks. Virtual MPI ranks, which are encapsulated by lightweight threads, must synthetically enforce memory separation. Virtual ranks can allocate from a common heap without the risk of sharing private data. Stacks require more work; instead of each virtual MPI rank receiving its own kernel thread (pthread), they are explicitly-managed user-space threads (GNU pth, ucontext, Qthreads). Global variables are the most difficult because source-to-source transformation must convert all global variables to user-space thread-local variables. For certain architectures, it may be possible to swap the data segment pointer when context switching between user-space threads.

At some point, application ranks must transfer control to the simulator. This occurs when emulated bindings, e.g. a call to `MPI_Send`, are emitted to the simulator instead of an actual MPI implementation. This symbol interception requires both compile-time and link-time steps. The SST compiler wrapper includes preprocessors which redirects MPI calls to SST/macro functions: MPI is not the only layer where symbol interception can occur at any layer of a software stack (Figure 2). A full MPI software stack could run on a simulated uGNI or libfabrics layer.

```
1 #define MPI_Send(...) SSTMAC_MPI_Send(...)
```

The SST compiler then links against `-lsstmac` which provides the simulator's MPI bindings.

Simulation time advances when the application enters an emulated binding. The thread blocks (context switches), the simulator predicts a time delay, inserts another event into the event queue, and returns control to the main simulator thread. After simulation time advances, the main thread context switches back to the application thread.

4 Source-to-Source Compiler

4.1 Overview

The auto-skeletonizing compiler has three tasks:

- Remove large memory allocations that would prevent scalable simulation

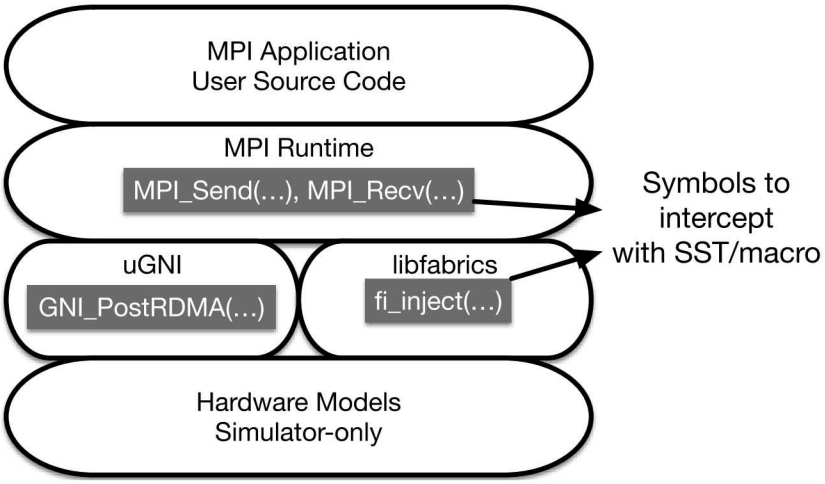


Fig. 2. Software stack diagram showing possible symbol interception points for SST/-macro varying from MPI calls directly in the application or at the network primitives themselves.

- Substitute compute-intensive kernels with an accurate delay estimate
- Redirect global variable accesses to thread-private memory

Global variable refactoring occurs automatically, but the remaining two require domain experts to insert pragmas.

4.2 Replacement Pragmas

pragma sst replace This pragma substitutes a variable or function call with a string. Consider an example from Lulesh.

```
1 #pragma sst replace volo 1
2    deltatime() = (Real_t(.5)*cbrt(volo(0)))/sqrt(Real_t(2.0)*einit);
```

The function call `volo(0)` is not valid in the skeleton since volumes are not actually computed. Here we could simply estimate that all cells have unit volume replacing `volo(0)` with `1`.

pragma sst init This pragma substitutes the right-hand side of an assignment operator. For example, in Lulesh:

```
1 #pragma sst init nullptr
2   destAddr = &domain.commDataSend[pmsg * maxPlaneComm] ;
```

The send buffer `domain.commDataSend` is not allocated in the skeleton. The pragma therefore sets `destAddr` to `nullptr`.

4.3 Data-Driven Pragmas

Marking large data structures that are not critical to control-flow as null types identifies large memory allocations to elide and provides hints to the compiler of compute blocks to avoid.

pragma sst null_variable This pragma decorates a variable declaration. An example can be seen in CoMD:

```
#pragma sst null_variable
   int* nAtoms;
```

In most cases, all operations involving the null variable are deleted. However, there may be cases where the compiler may decide deleting an operation cannot be done automatically since it may affect control flow, e.g., if the variable is used inside an if-statement. When this occurs, a compiler error is thrown flagging where the ambiguity occurs. Another pragma must then be applied to the conditional to tell the compiler how to proceed.

pragma sst null_type This applies to C++ class variable declarations. Memory allocations are eliminated, but specific member functions used for tracking type size may be kept. Consider an example from Lulesh:

```
#pragma sst null_type sstmac::vector size resize empty
   std::vector<Real_t> m_x ;  /* coordinates */
```

Here we wish to indicate the vector is "null" and should not actually allocate memory or allow array accesses. However, we still wish to track the vector size and whether it is empty. The first argument to the pragma is a new type name that implements the "alias" functionality. For `std::vector`, the SST compiler automatically provides an alias.

```
namespace sstmac {
class vector {
 public:
  void resize(unsigned long sz){
    size_ = sz;
  }
  unsigned long size() const {
    return size_;
  }
  template <class... Args> void push_back(Args... args){
    ++size_;
  }
  template <class... Args> void emplace_back(Args... args){
    ++size_;
  }
  bool empty() const {
    return size_ == 0;
  }
 private:
  unsigned long  size_;
};
}
```

Everywhere `std::vector` is substituted with the new type. The remaining arguments to the pragma are the list of functions we wish to mark as valid. In this case, even though the alias vector class provides more functions, we only allow `size`, `resize`, and `empty` to be called.

pragma sst branch_predict This pragma replaces a branch condition with a new expression. The `branch_predict` pragmas are necessary for branch conditions containing null variables because they are otherwise impossible to compute at runtime. Although most uses of this pragma will substitute `true` or `false` as the replacement, any arbitrary C++ boolean expression can be used as the replacement.

4.4 Compute Pragmas

pragma sst compute and pragma omp parallel These pragmas substitute the computations in decorated scoping block with a basic delay model. Delay modeling currently performs static analysis of memory accesses and arithmetic operations and passes these statistics to a coarse-grained processor model which generates a time estimate.

pragma sst loop_count If the `sst compute` or `omp parallel` pragma is applied to an outer loop with one or more inner loops, the compute model static analysis might fail. This occurs when the inner loop control flow depends on the actual execution. Any variables declared or modified *inside* the compute block are not valid to use in the compute estimate. Only variables in scope at the beginning of the outer loop are safe to use in compute modeling.

When the static analysis fails, a corresponding compiler error is thrown. This usually requires giving a loop count hint. Consider the example from HPCG:

```
1 #pragma omp parallel for
2   for (local_int_t i=0; i< localNumberOfRows; i++) {
3     int cur_nnz = nonzerosInRow[i];
4    #pragma sst loop_count 27
5     for (int j=0; j<cur_nnz; j++) mtxIndL[i][j] = mtxIndG[i][j];
6   }
```

The static analysis fails on `cur_nnz`. However, that value is almost always 27. Thus we can safely tell the compiler to just assume a given loop count.

pragma sst branch_predict When `branch_predict` appears inside a marked compute block, the argument must be a value between 0 and 1 (or true/false). The value informs the static analyzer of the proportion of branches taken. Because all program logic must be estimated in the skeleton,`branch_predict` must be used inside such a block when the branch condition is dependent on variables declared in the same scope. Consider an example from CoMD:

```
1 #pragma sst branch_predict areaFraction
2   if(r2 <= rCut2 && r2 > 0.0){
```

Inside this compute block, a computation's occurrence depends on whether a particle distance is less than a cutoff. Based on the way CoMD constructs unit cells and halo regions, we can estimate the ratio of the particles (and therefore computations) that are expected within the cutoff.

4.5 Source-to-Source

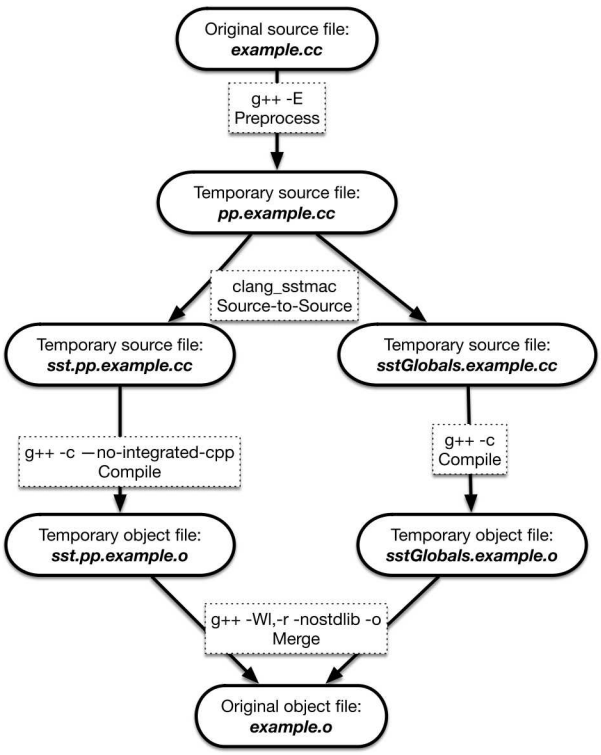


Fig. 3. Source-to-source transformation workflow for SST compiler. These steps occur automatically through the SST compiler wrapper. For C source files, g++ can be swapped with gcc. The choice of underlying compiler is arbitrary and could be gcc, icc, or another compiler.

No changes to an existing make system are required to auto-skeletonize. SST/macro provides compiler wrappers `sstcc` and `sst++` similar to MPI. Figure 3 shows the compiler auto-skeletonization workflow. Remapping global variables requires static registration of special C++ variables, which requires merging a temporary C++ object file into the original target object file (see Figure 3).

The source-to-source translation occurs through analysis of the Clang abstract syntax tree (AST). The AST is accessed by registering custom Clang

frontend actions and the AST visitor interface. New pragmas can be easily registered with the Clang preprocessor. At this time, Fortran codes are not supported due to limitations in the Clang fontend. The source-to-source can work on most C++ template code with no difficulties since the Clang frontend provides a visitable AST node for each implicitly instantiated template function. Type-dependent compute models must be constructed for compute pragmas inside template code. This has not yet been implemented, but is not required for any of the current skeleton applications.

5 Methodology

Lulesh 2.0.3 was downloaded from the Lawrence co-design center [21]. CoMD reference version 1.1 was downloaded from the Mantevo project [22]. HPCG reference version 3.0 was download from the HPCG benchmark site [23]. The Clang 4.0 frontend was used for source-to-source transformations and skeleton compilation. Each application takes a basic set of 3-6 input parameters defining either the total problem size or the problem size per MPI rank. Lulesh is an explicit shock hydrodnamics code with communication dominated by nearest-neighbor halo exchanges. CoMD is a lattice-based molecular dynamics code with communication dominated by cell exchanges of neighboring atoms. HPCG is a conjugate-gradient solver with multigrid preconditioning.

Results were generated using the version 7.2 development branch of the macroscale element library of the Structural Simulation Toolkit (SST) [24] available on GitHub [25]. An analytic delay model similar to LogP [3] was used for the network. The processor model also used a simple analytic delay model

$$\Delta T = \beta B + \gamma_f F + \gamma_i I \quad . \tag{1}$$

Here B is the number of bytes used, F is the number of floating-point operations, and I is the number of integer operations. β is the inverse bandwidth in seconds/bytes and γ is the inverse frequency in seconds/operation for floating point and integer arithmetic.

Timings were collected on a Xeon E7-8870 at 2.4 GHz. STREAM benchmarks showed ∼6 GB/s of single-core memory throughput. For our processor delay model we therefore use $\beta = 0.16$ s/GB and $\gamma_i = \gamma_f = 0.42$ s/Op.

6 Results and Discussion

6.1 Weak- and Strong-scaling Spider Plots

The skeleton applications should behave as much as possible like the full application running on a real system. As initial validation, the simulated results should generate correct weak and strong scaling curves for the application. Using contention-free analytic processor and network models in the simulator, scaling should be ideal. Figures 4-6 show spider splots for the skeleton applications with mixed weak and strong scaling curves. Given weak and strong scaling inputs, the simulator clearly produces the correct behavior.

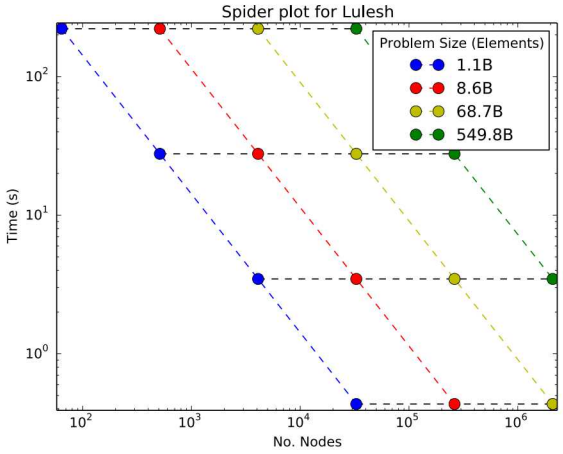


Fig. 4. Spider plot showing weak and strong scaling curves for simulated Lulesh up to 2M network endpoints. Weak and strong scaling inputs to the skeleton application exactly generate the expected weak and strong scaling behavior.

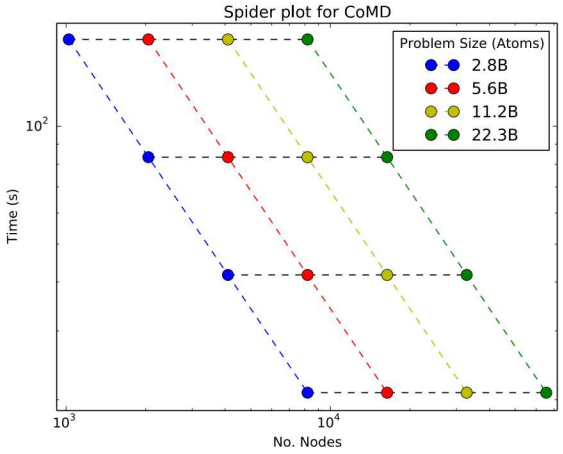


Fig. 5. Spider plot showing weak and strong scaling curves for simulated CoMD up to 65K network endpoints. Weak and strong scaling inputs to the skeleton application exactly generate the expected weak and strong scaling behavior.

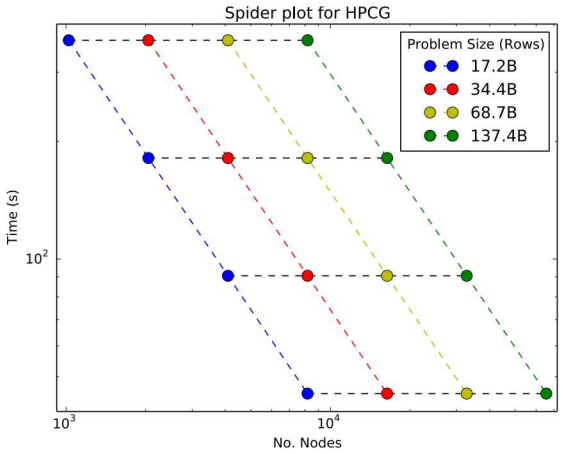
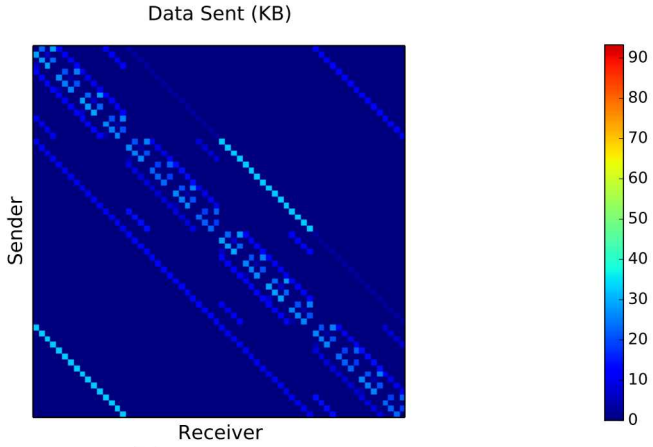


Fig. 6. Spider plot showing weak and strong scaling curves for simulated HPCG up to 65K network endpoints. Weak and strong scaling inputs to the skeleton application exactly generate the expected weak and strong scaling behavior.

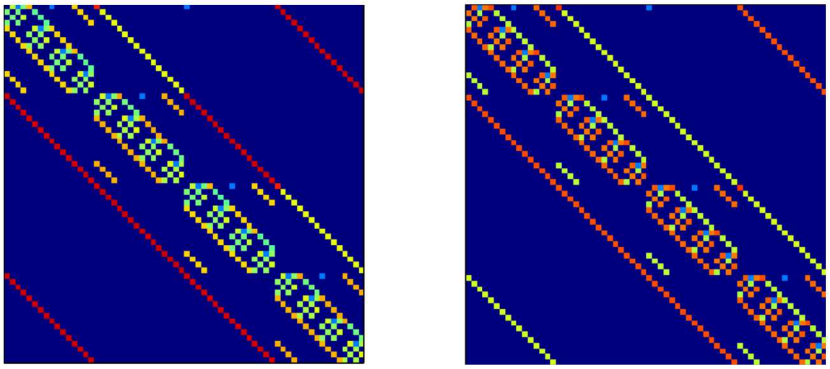
6.2 Traffic Pattern Validation

Beyond validating the timing behavior of the skeleton applications, we want to ensure that the generated traffic patterns are correct. Lulesh and HPCG exactly generate the correct number, size, and sequence of MPI messages (and thus generate no interesting figures). CoMD is more complicated. The skeleton application creates an approximate traffic pattern based on the average number of atoms per cell. No computation is ever performed on individual atoms. The exact CoMD traffic pattern is partially data-dependent, however. Certain border cells can have partial occupancy with fewer atoms than interior cells. This affect is not accounted for, but could be with extra pragmas and domain expertise. The approximations involved create a minor discrepancy between the exact traffic pattern and the skeleton traffic pattern (Figure 7).

This discrepancy is critical in distinguishing our current approach from previous auto-skeletonization work [2]. In previous work, every MPI call had to be exactly preserved. The compiler worked bottom-up from each MPI call to decide what code was necessary. For CoMD, the traffic pattern depends on individual atom computations and thus no skeletonization could occur. In our top-down approach, the large data structures containing atoms are marked null. This then generates compiler warnings or errors where computation depends on individual atoms. Approximations are then introduced with pragmas to estimate the number of atoms being exchanged. This generates an efficient skeleton with approximately correct MPI calls. Although some point-to-point sends are incorrect by 10-30% in Figure 7, the traffic pattern as a whole is preserved.



(a) Difference



(b) Full Application

(c) Skeleton

Fig. 7. Traffic matrix plots (spyplot) showing the (a) difference between skeleton and full application, (b) full application, and (c) skeleton application.

6.3 Simulator Memory and Wall Time

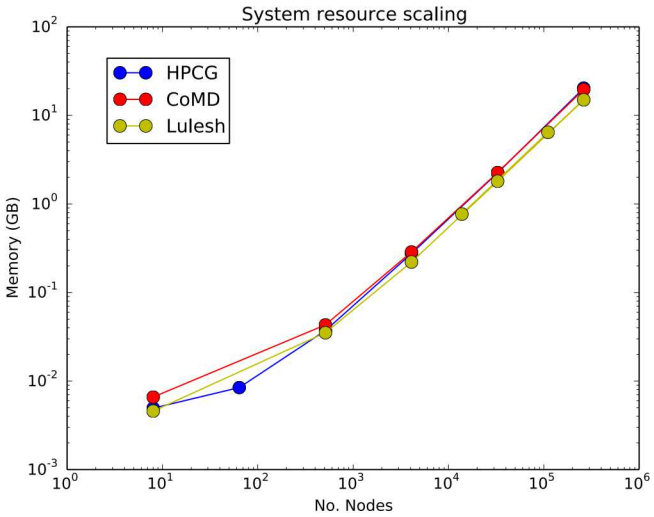


Fig. 8. Maximum memory usage (GB) of simulator for increasing scales of Lulesh, CoMD, and HPCG skeleton applications.

Beyond validating that the simulator produces correct scaling and traffic results, we want to ensure that skeleton application is consuming minimal amounts of memory and executing quickly. Performance results for memory usage and timings are shown in Figures 8 and 9. Even past 100K endpoints, memory usage is only 20GB for all apps. We note that the total memory usage for each skeleton app is almost the same. The dominant memory cost is actually the user-space thread stacks used by the simulator and not heap-allocated variables within the skeleton apps. Even for large runs (>100K endpoints), the simulations finish within a few hours on a single core, showing that no significant computation is occurring within each application.

6.4 Number of Pragmas

To indicate the amount of work required to skeletonize each application, we count the number of each type of pragma used in Table 1. As previously discussed, in our top-down approach skeletonization begins with inserting `null_type` pragmas to label data structures that should not be allocated. The remaining skeletonization is fairly automatic, with the compiler noting exactly where additional hints are required. For CoMD and HPCG, the number of pragmas is not significantly more than the number of pre-existing `omp parallel` pragmas. Lulesh

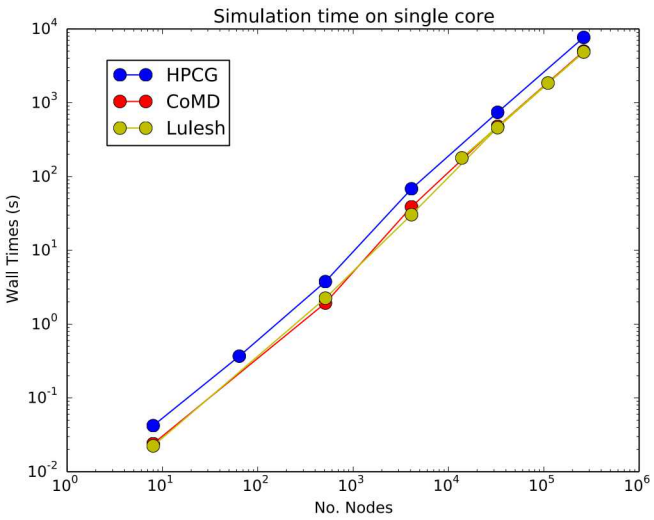


Fig. 9. Total simulation wall time for increasing scales of Lulesh, CoMD, and HPCG skeleton applications.

pragma	Lulesh	CoMD	HPCG
null_type/null_variable	48	8	18
replace/init	70	19	14
compute	76	8	13
omp parallel	30	15	17
branch_predict	10	2	0
loop_count	0	18	2

Table 1. Number of pragmas in each of the considered example applications.

has more data structures, but the process is still very quick for someone with domain knowledge of the application. The whole process in general should be a few hours, not days or weeks to write an entirely new skeleton app.

6.5 Compute Model Accuracy

Although the main goal of the current work is not to demonstrate an accurate compute model, we asses what accuracy can be obtained by simply counting operations in the source code and using the analytic delay estimate from (1). Figures 10-12 compare actual timings to the SST estimates. Actual timings will be highly-compiler dependent and thus we show a min-max range for -O3 and -O0 optimizations.

CoMD (Figure 11 is dominated by a single compute kernel: `force`. The force kernel is dominated by compute rather than memory and the estimated delay is surprisingly accurate. For comparison, Figure 11 shows timing estimates from a much smaller kernel. The velocity kernel is much more memory-bound, making

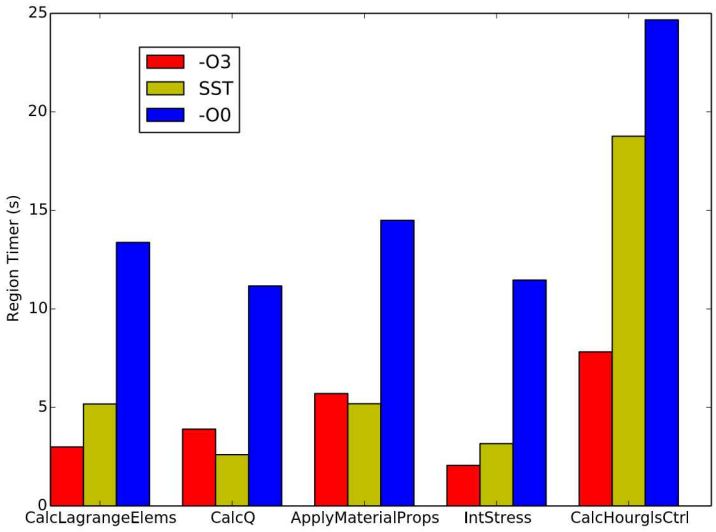


Fig. 10. Individual region timers for different kernels in Lulesh. The SST compute model is based only on counting floating point and integer operations in source code rather than actual instructions in assembly. The SST delay estimates are compared to two different levels of compiler optimization.

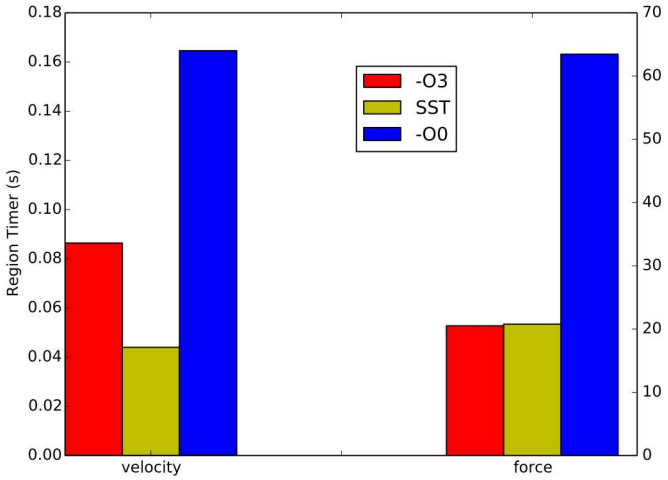


Fig. 11. Individual region timers for different kernels in CoMD. The SST compute model is based only on counting floating point and integer operations in source code rather than actual instructions in assembly. The SST delay estimates are compared to two different levels of compiler optimization.

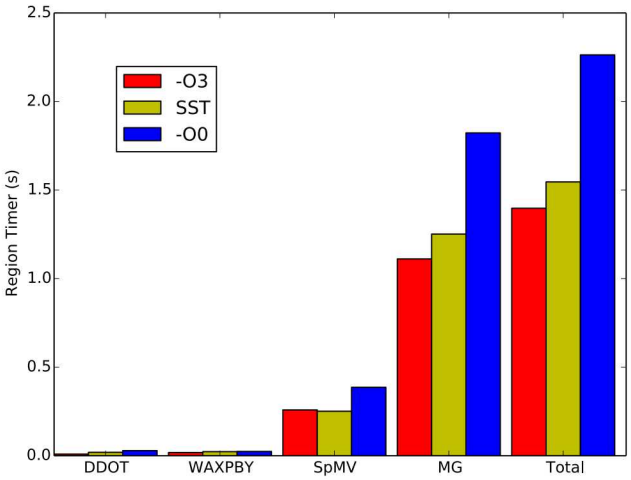


Fig. 12. Individual region timers for different kernels in HPCG. The SST compute model is based only on counting floating point and integer operations in source code rather than actual instructions in assembly. The SST delay estimates are compared to two different levels of compiler optimization.

it harder to estimate. In this case, the SST compiler actually undercounts the number of memory accesses leading to the discrepancy.

For HPCG (Figure 12, SST slightly overestimates relative to O3 but is bracketed above by O0. Lulesh (Figure 10 shows similar results to HPCG with O3 and O0 bracketing the SST estimates. SST does underestimate certain kernels even with O3. More study is required to determine if 1) the static analysis is undercounting operations or 2) the basic delay model fails to account for certain contention effects. For CalcQ, the square root function is used extensively. Because the square root code is not available to the compiler, it cannot estimate the number of flops and omits them. This leads to the observed underestimate. Additional pragmas could be added that allow flop/byte estimates to be given for such functions.

7 Future Work

The two most critical areas for improvement are compute model generation and skeletonization of more dynamic applications. While compute models are generated simply from source code operations (abstract syntax tree), more sophisticated models could be generated from either LLVM IR (intermediate representation) or even assembly. IR can be easily generated from the AST using LLVM code generators, allowing compute models for loops or other nodes in the AST to be based on IR. More sophisticated static analysis, particularly for

nested loops, might also involve polyhedral techniques to better estimate cache traffic or computational intensity [26].

Dynamic applications, e.g. adaptive mesh refinement (AMR), pose new challenges to the skeletonization process. HPCG, Lulesh, and CoMD have semi-static traffic patterns which are dominated by the structure of the computation, and not the values stored in the data structures. For an AMR code, the traffic pattern between refinements will similarly be "static" and compatible with skeletonization. For a library like BoxLib [27], the data structures defining the box sizes and nesting are modest in size compared to the actual element or field data contained within each box. The actual refinement computation must be replaced with an approximate model. Previous studies have used coarse-grained box traces [8]. In contrast to MPI communication traces, the box traces are flexible and can be immediately used for strong scaling studies. An alternative is to base box refinements on known properties of the input problem. For MiniAMR [22], refinement is driven by objects pushed through the mesh, which could be approximated by an analytic function yielding an inexpensive estimate of refinement. Since refinement calculation itself is limited to a small portion of the application, the compiler-derived skeleton application can still be based almost entirely on the original source code.

8 Conclusions

This work presents a compiler-assisted approach to generating skeleton applications directly from existing application source code. Having on-line endpoint models for system-level simulation improves validation, scalability, and flexibility of the simulation. Validation of the generated skeletons is demonstrated by analyzing weak and strong scaling behavior of the skeleton apps on an ideal system. Traffic patterns from skeletons are also compared to the parent application pattern. The simple compute models generated from static analysis here are surprisingly accurate, but could easily be replaced by more sophisticated models.

9 Acknowledgments

This worked was performed at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

1. T. Groves, R. E. Grant, S. Hemmer, S. Hammond, M. Levenhagen, and D. C. Arnold, "(SAI) Stalled, Active and Idle: Characterizing Power and Performance

of Large-Scale Dragonfly Networks," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 50–59.
2. M. Sottile, A. Dakshinamurthy, G. Hendry, and D. Dechev, "Semi-automatic extraction of software skeletons for benchmarking large-scale parallel applications," in *PADS 2013: ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2013, pp. 1–10.
3. T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim: Simulating Large-Scale Applications in theLogGOPS Model," in *HPDC '10: 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 597–604.
4. W. Gropp, E. L. Lusk, and A. Skjellum, *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface.* Cambridge, MA: The MIT Press, 1999.
5. N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, "Evaluating HPC Networks via Simulation of Parallel Workloads," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 154–165.
6. A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A Framework for Performance Modeling and Prediction," in *SC '02: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2002, pp. 1–17.
7. F. Desprez, G. Markomanolis, M. Quinson, and F. Suter, "Assessing the Performance of MPI Applications Through Time-Independent Trace Replay," in *PSTI '11: Second International Workshop on Parallel Software Tools and Tool Infrastructures*, 2011.
8. C. P. Chan, J. D. Bachan, J. P. Kenny, J. J. Wilke, V. E. Beckner, A. S. Almgren, and J. B. Bell, "Topology-aware performance optimization and modeling of adaptive mesh refinement codes for exascale," in *Communication Optimizations in HPC (COMHPC), International Workshop on.* IEEE, 2016, pp. 17–28.
9. Q. Xu, "*Automatic construction of coordinated performance skeletons*," 2007, p. 84.
10. N. Jiang, D. U. Becker, G. Michelogiannakis, J. D. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate Network-on-Chip simulator," in *ISPASS*, 2013, pp. 86–96.
11. J. J. Wilke, K. Sargsyan, J. P. Kenny, B. Debusschere, H. N. Najm, and G. Hendry, "Validation and Uncertainty Assessment of Extreme-Scale HPC Simulation through Bayesian Inference," in *EuroPar 2013: 19th International Euro-Par Conference on Parallel Processing*, vol. 8097, 2013, pp. 41–52.
12. R. Susukita, H. Ando, M. Aoyagi, H. Honda, Y. Inadomi, K. Inoue, S. Ishizuki, Y. Kimura, H. Komatsu, M. Kurokawa, K. J. Murakami, H. Shibamura, S. Yamamura, and Y. Yunqing, "Performance Prediction of Large-Scale Parallel System and Application Using Macro-Level Simulation," in *SC '08: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008.
13. A. Degomme, A. Legrand, G. S. Markomanolis, M. Quinson, M. Stillwell, and F. Suter, "Simulating MPI Applications: The SMPI Approach," *IEEE Transactions on Parallel Distrib. Syst.*, vol. 28, pp. 2387–2400, 2017.
14. S. Prakash and R. L. Bagrodia, "MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs," in *30th Conference on Winter Simulation*, 1998, pp. 467–474.
15. M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely, "PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications," in *EuroPar 2009: 15th International Euro-Par Conference on Parallel Processing*, 2009, pp. 135–148.

16. S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama, "WARPP: A Toolkit for Simulating High-Performance Parallel Scientific Codes," in *SIMUTools '09: 2nd International Conference on Simulation Tools and Techniques*, 2009, pp. 1–10.
17. G. Kathareios, C. Minkenberg, B. Prisacari, G. Rodriguez, and T. Hoefler, "Cost-effective diameter-two topologies: analysis and evaluation," in *SC '15: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–11.
18. J. Guo, Q. Yi, J. Meng, J. Zhang, and P. Balaji, "Compiler-Assisted Overlapping of Communication and Computation in MPI Applications," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 60–69.
19. R. Preissl, M. Schulz, D. Kranzlmüller, B. R. Supinski, and D. J. Quinlan, "Using MPI Communication Patterns to Guide Source Code Transformations," in *Proceedings of the 8th international conference on Computational Science, Part III*, 2008, pp. 253–260.
20. M. M. Strout, B. Kreaseck, and P. D. Hovland, "Data-Flow Analysis for MPI Programs," in *ICPP '06: International Conference on Parallel Processing*, 2006, pp. 175–184.
21. "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)," https://codesign.llnl.gov/lulesh.php.
22. "The Mantevo Project," https://mantevo.org/packages/.
23. "HPCG Benchmark," http://www.hpcg-benchmark.org/software/index.html.
24. A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The Structural Simulation Toolkit," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 37–42, 2011.
25. "SST/macro GitHub Repository," https://github.com/sstsimulator/sst-macro.
26. W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan, "Static and dynamic frequency scaling on multicore cpus," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 51:1–51:26, Dec. 2016.
27. W. Zhang, A. S. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An AMR software framework," *CoRR*, vol. abs/1604.03570, 2016.