SANDIA REPORT

SAND2015-1027
Unlimited Release
Printed October 2014

Using Discrete Event Simulation for Programming Model Exploration at Extreme-Scale: Macroscale Components for the Structural Simulation Toolkit (SST)

Jeremiah J. Wilke, Joseph P. Kenny

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2015-1027
Unlimited Release
Printed October 2014

Using Discrete Event Simulation for Programming Model Exploration at Extreme-Scale: Macroscale Components for the Structural Simulation Toolkit (SST)

Jeremiah J. Wilke and Joseph P. Kenny
Scalable Modeling and Analysis
Sandia National Laboratories
7011 East Ave
Livermore, CA 94550
{jjwilke,jpkenny}@sandia.gov

Abstract

Discrete event simulation provides a powerful mechanism for designing and testing new extreme-scale programming models for high-performance computing. Rather than debug, run, and wait for results on an actual system, design can first iterate through a simulator. This is particularly useful when test beds cannot be used, i.e. to explore hardware or scales that do not yet exist or are inaccessible. Here we detail the macroscale components of the structural simulation toolkit (SST). Instead of depending on trace replay or state machines, the simulator is architected to execute real code on real software stacks. Our particular user-space threading framework allows massive scales to be simulated even on small clusters. The link between the discrete event core and the threading framework allows interesting performance metrics like call graphs to be collected from a simulated run. Performance analysis via simulation can thus become an important phase in extreme-scale programming model and runtime system design via the SST macroscale components.

Acknowledgment

This work was supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing program and the DOE Office of Advanced Scientific Computing Research. SNL is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE NNSA under contract DE-AC04-94AL85000.

Contents

1 Introduction **9**

2 Related Work **11**

2.1 Simulators 11

2.2 Programming Models 12

2.3 Parallel Discrete Event Simulation 13

3 SST Macroscale Simulator Components **15**

3.1 Code stack 15

3.2 Performance Analysis 17

3.3 Global variables 18

4 PDES Results **21**

4.1 Experimental Setup 21

4.2 Weak-Scaling Experiment 22

5 SST Visualizations and Performance Metrics **25**

5.1 Call Graph 25

5.2 Fixed-time Quanta (FTQ) Chart 26

6 Summary **29**

References **30**

List of Figures

3.1 The SST Macroscale software/hardware stack. The application links to library APIs implemented on top of the SST operating system/hardware models. 16

4.1 Weak-scaling experiment for 33K-262K network endpoints. Both the total simulation (wall clock) time and the ratio of wall clock to virtual time are shown for each experiment. 22

5.1 Call graph visualization in KCachegrind/QCachegrind: full application window . . 25

5.2 Call graph visualization in KCachegrind/QCachegrind: zoomed in view of MPI function 26

5.3 Fixed time quanta (FTQ) showing compute/communication activity aggregated over all threads for MPI SPMD model. 27

5.4 Fixed time quanta (FTQ) showing compute/communication activity aggregated over all threads for asynchronous many-task programming model. 27

List of Tables

2.1 Simulators based on analytic or constitutive models of nework communication. 11

2.2 Structural simulators modeling individual network components and congestion. 12

Chapter 1

Introduction

A major branch of discrete event simulation explores application performance on high-performance computing platforms. Here we are concerned in particular with skeleton applications which replicate a full application's control flow. Only the minimal computation necessary to reproduce the communication pattern is executed rather than emulating instruction-by-instruction. Simulated (virtual time) therefore advances much faster than computation (wall clock) time for a single process. Ideally, several minutes of execution on a capability supercomputer could be executed within a few hours on a small cluster. Simulation can inform machine procurements, exploring how applications perform on speculative hardware before purchasing. While small test beds can be informative (particularly for simulator validation), many effects only manifest at extreme scales such as the rapidly shrinking mean-time-between failures (MTBF) as systems move towards exascale [1]. Conversely, simulation can inform software development. Before implementing and debugging every detail at extreme-scale, an application or runtime's performance can be evaluated via simulation.

Our particular HPC focus is science and engineering applications, most often partial differential equation (PDE) solvers. Notable examples include combustion (e.g. explicit Navier-Stokes solvers [2]), astrophysics (e.g. CASTRO [3]), or nuclear fusion (e.g. GTC [4]). The vast majority of applications are written in the communicating sequential processes (CSP) model. The CSP model maps well onto the off-line simulation mode, collecting traces and replaying them step-by-step in the simulator. To handle resilience and massive parallelism, however, new runtime systems and programming models are being explored [5, 6]. The models are much more dynamic, moving work and data between processes. They therefore do not map well to trace replay, instead demanding on-line simulation where a real application runs and generates events. We present here the macroscale components of the Structural Simulation Toolkit (SST) [7, 8] with emphasis on programming and execution model exploration.

The discrete event simulation philosophy for the SST macroscale components emphasizes coarse-grained approximations. We are here not primarily concerned with high-accuracy simulation of a fixed communication pattern. Rather we want to evaluate how program execution evolves for varying inputs, machine parameters, or even component failures. The runtime layer that manages placement of processes, tasks, and data should be executed exactly instruction-by-instruction. In contrast, the compute-intensive math kernels are not executed, instead advancing the virtual simulation time via a coarse-grained model based on, e.g. number of flops or bytes read.

Of critical importance is the ratio of simulated virtual time (the time elapsed on the system being simulated) versus the wall-clock time elapsed on the machine actually executing the simulation. The SST macroscale components are aimed at simulating extreme-scale (e.g exascale) systems with 100K or more endpoints. To be useful we propose the general rule that one hour of wall-clock time should at least be capable of progressing one minute of virtual time to be useful, or roughly a 100:1 time dilation (at worst case 1000:1). The virtual time acceleration relative to physical wall-clock time relies primarily on coarse-grained approximations. If simulating 100K endpoints in serial on a single machine, coarse-grained approximations would have to accelerate virtual time by a factor of 1000:1 to achieve the time goal. In an extreme-case, a large matrix-vector product that requires 1 ms to actually execute might be simulated in 1 μs, leading to a 1000:1 time acceleration. However, even if leveraging coares-grained models for the compute-intensive kernels, we must still simulate the application runtime. If the runtime consumes 50 percent of active CPU cycles, our simulation strategy would at best achieve a 2:1 time acceleration. In contrast, as is often the case with MPI or even newer runtime systems like Legion [9], the runtime is often 5 percent or less overhead.

If combined with parallel discrete event simulation (PDES) [10], however, the coarse-grained requirements become much less severe. Even for 100-way PDES parallelism, coarse-grained approximations must now only accelerate application execution by a factor of 10:1. For PDES, the critical design choice now becomes a conservative or optimistic algorithm. While optimistic algorithms can be far more scalable, they require either checkpointing or reverse computation to account for time order violations. As described above, for our purposes, we wish to simulate real software stacks for programming model exploration. A great deal of application state exists, making checkpointing a serious computational bottleneck. Similarly, our goal here is to directly compile and run C/C++ codes, rather than building ad-hoc state machine representations of an application. Since arbitrary code can be executed, reverse computation becomes very difficult. A central question here is therefore whether conservative PDES is scalable enough to achieve our goals.

This paper therefore describes our SST macroscale simulator as a programming model/runtime development tool. We first describe related work in Section 2. In Section 3, we describe the simulator components and basic examples of directly compiling and linking C/C++ code into the simulator. Section 2.3 details our choice of PDES algorithm and how it relates to other conservative and optimistic PDES algorithms. Section 5 describes some of the visualizations and performance metrics like call graphs and fixed-time quanta (FTQ) histograms made possible by our particular simulator structure. Finally, we describe a coarse-grained machine model constructed to maximize PDES performance along with scaling results for an example problem.

Chapter 2

Related Work

2.1 Simulators

We have previously outlined the large range of related HPC work [11]. Broadly simulators can be classified based on the following:

- How network traffic is generated, either replaying an application trace (i.e. off-line/post-mortem) or running applications on-line.
- How network traffic is modeled: Simple analytic (i.e. latency/bandwidth) models or structural simulators with switches and queues to estimate congestion effects.
- How computation is modeled: simple constitutive models, performance counter convolution [12], to even detailed cycle-accurate simulation [13].

Features for some (but not all) literature examples are given in Tables 2.1 and 2.2, separated by whether the simulators are primarily analytic or structural. Analytic functions often ignore contention, but some formulas do incorporate congestion [14].

Simulator	Ref no.	Type	Compute	Language
LogGOPS	[14]	Trace/On-line	Model	DSL (GOAL)
BSIM	[15]	On-line	Model	Native
Mambo/Seshat	[13]	On-line	Cycle-Acc	Native
PSINS	[16]	Trace	Trace	n/a
MPI-SIM	[17]	On-line	Direct	Native
Dimemas	[18]	Trace	PerfCtr	n/a
WARPP	[19]	Trace	PerfCtr	Native

Table 2.1. Simulators based on analytic or constitutive models of nework communication.

On-line simulation often involves API emulation, e.g. with bindings for message-passing interface (MPI) [27] The simulator links to application code, intercepting function calls to estimate

Simulator	Ref no.	On/Off-line	Computation	Network	Language
BigSim	[20, 21]	Both	PerfCtr/Model	Packet	Native
SIMGRID	[22, 23]	Both	PerfCtr	Flow	Native
MARS	[24]	Trace	Time	Packet	n/a
MPI-NeTSim	[25]	On-line	Direct	Packet	Native
PACE	[26]	Both	Abstract	Abstract	DSL (CHIP^{3}S)
SST/macro	[7]	Both	PerfCtr/Model	Packet/Flow	Native

Table 2.2. Structural simulators modeling individual network components and congestion.

elapsed time. Off-line simulators instead perform post-mortem analysis of application traces. Traces are often time-dependent, collecting only time between communication events and are therefore essentially limited to simulating a single architecture. Time-independent traces instead collect architecture-agnostic hardware counters that allow replay on new architectures [18, 23].

The closest simulators to macroscale SST are BSIM, BigSim, and SIMGRID. BSIM also emphasizes coarse-grained modeling [15]. BigSim is structural, and large simulations incorporating congestion and statistical computation modeling have been performed [28]. Like SST, BigSim can be an on-line network emulator or on-line simulator estimating elapsed time without actually performing the computation [21]. SIMGRID is also designed primarily as an on-line simulator with various bindings (including MPI) [22].

2.2 Programming Models

The simulation emphasis here is programming model exploration, in particular many-task runtimes that break from the MPI communicating sequential processes (CSP) model. In particular, we demonstrate runtime emulation for a many-task programming model developed in the last year within the lab. Related models include Charm++ [29, 30], Concurrent Collections (CnC) [31], Legion [9], Uintah [32], or DaGuE [33]. The runtimes control placement of data and tasks, with the majority of compute-intensive kernels and large memory allocations contained within tasks. The programmer focuses on the scientific problem being solved, leaving details of task-scheduling to the runtime. The runtimes are usually based on messaging or transport layers like Cray uGNI [34], PAMI [35], SHMEM [36], or Gasnet [37]. For on-line simulation of task runtimes, simulators must provide API bindings matching these message layers.

2.3 Parallel Discrete Event Simulation

Parallel discrete event simulation (PDES) can be broadly split amongst conservative and optimistic strategies [10], although a more exact taxonomy has been given [38]. We refer to each worker (thread) in a PDES run as a logical process (LP). In general, LP here is synonymous with MPI rank, which produces some confusion as we often have a real MPI program simulating a virtual MPI program. While a given LP may schedule most events to itself, at least some events are scheduled to other LPs. In order to exploit parallelism, each LP must have its own event queue and will thus have its own notion of time. The core problem in PDES is that events sent between LPs cannot violate time ordering, i.e. events cannot be scheduled in the past. When LPs exchange events (in the current case, when messages are sent between nodes or switches), the event has a virtual time latency, e.g. it may require 100ns for a network packet to hop from switch to switch. This latency creates a time window or lookahead δ=100ns in which the LP is free to run events with no risk of time violation. Once LPs have synchronized on a given time, t, they are safe to run events in the time window $(t, t+\delta)$. Conservative strategies enforce strict time synchronizations amongst the LPs so there is no need for checkpointing or rollback. In many cases, the enforced synchronization can severely hamper parallelism as only a few (or one) LP may have events in the time window $(t, t+\delta)$. The efficiency of conservative strategies is highly dependent on the lookahead δ as larger values will allow much greater parallelism.

Optimistic strategies allow LPs to run events with little synchronization, allowing greater parallelism at the risk of time violations. When violations occur, state must be rolled back, for which a popular method is Time Warp [39]. The most basic rollback is checkpointing, saving application state at regular intervals to provide a fallback point. In reverse computation, only an event list is needed which can be reversed to undo state changes. Checkpointing induces a significant overhead even if no time violations occur. Reverse computation induces most of the overhead when violations occur, but does require extra storage for the event list. Detailed packet-level network simulations have been performed at very large scales with the ROSS simulator [40, 41].

Both checkpointing and reverse computation present special challenges for on-line simulation. There is a large amount (several TB) of application state such that checkpointing is likely not feasible. Reverse computation is also difficult since we want to allow arbitrary code to be executed on real software stacks. A simple, well-defined state machine is most amenable to reverse computation, e.g. a packet sent from a queue can be placed back in its original position to reverse the event. For arbitrary code, reverse computation is not trivial. Thus, despite potential for increased parallelism with optimistic PDES, we pursue a conservative strategy here.

Conservative approaches generally operate through event queues [42, 43]. Each LP maintains an event queue for every LP it communicates with which is associated with a unique virtual timestamp. As an LP receives events, the new events advance the queue timestamp. This scheme effectively exploits locality. Even if there are 100 LPs, each LP might only communicate with, e.g., 3 neighbors. Only local point-to-point communication is required to synchronize times rather than a global collective. This scheme requires an LP to regularly receive events on each queue, though. If no events are received, the queue time is never updated and an LP will block waiting for updates. If deadlock occurs, an LP must send so-called null messages to "request" a time update

(although methods avoiding null messages have been proposed [44]).

For our workloads, the event queue algorithm becomes highly inefficient. In many network topologies and models, each LP is connected to many other LPs, erasing the benefit of local communication. Furthermore, our simulator emphasizes coarse-grained compute and network models. Large time gaps often exist between consecutive events, often much larger than the lookahead. This leads to a pathological situation in which LPs are consumed sending null messages back and forth.

Here we instead use a global synchronization algorithm. While in some ways the simplest algorithm, it actually represents an optimization for workloads encountered with macroscale SST.

while $t < t_{termination}$ **do**
  Run all events until $t + \delta$
  Log event sends to $S = \{N_{events}^{LP0}, N_{bytes}^{LP0}, N_{events}^{LP1}, \ldots\}$
  MPI_ReduceScatter with SUM array S
  MPI_ReduceScatter returns N_{events}^{total}, N_{bytes}^{total}
  $N_{bytes}^{left} = N_{bytes}^{total}$
  for $msgNum < N_{events}^{total}$ **do**
    MPI_Recv(void*, MPI_ANY, N_{bytes}^{left})
    $N_{bytes}^{left} = N_{bytes}^{left} - msgSize$
  end for
  Determine t_{min}^{local}
  MPI_Allreduce(t_{min}^{global}, t_{min}^{local})
  $t = t_{min}^{global}$
end while

The simulation begins at $t = 0$ with lookahead δ determined from the network parameters. We consider a simulation with N_{lp} logical processes. Within the time window $(0, \delta)$ each LP will call MPI_Isend to deliver events to other LPs. At $t = \delta$, every LP performs two blocking MPI collectives. First, they perform a reduce-scatter with a sum function. Each LP tracks the total number of events and the total number of bytes sent to every other LP. This array of size $2N_{lp}$ is passed as input to the reduce-scatter. Every LP receives as input from the reduce-scatter two integers: the total number of messages and the total number of bytes received. The LP can then execute MPI_Recv for every incoming message. The source of each message is not important since a wildcard MPI_ANY can be specified. The size passed to MPI_Recv need not be exact - only greater than or equal. After completing the receive, both the actual source and size can be determined. Once all messages are sent or received, the LPs perform an MPI_Allreduce with a min function to determine the minimum virtual time. Once all LPs have agreed on a minimum virtual time, t_1, the cycle repeats for the new time window $(t_1, t_1 + \delta)$. Time synchronizations are pushed into highly-optimized collective operations, minimizing the communication overhead. Additionally there is no probing or polling for messages. Each LP determines exactly how many messages are pending and can execute receives without wasting cycles checking for new messages.

Chapter 3

SST Macroscale Simulator Components

Each virtual (simulated) process or kernel thread is modeled as an explicitly-managed user-space thread. It has its own stack and register variables, but is not scheduled as a full OS thread. For simulations involving many thousands of virtual processes, the OS scheduler would otherwise become quickly overwhelmed. Most events occur on the main simulation thread, which handles events and advances the virtual time. Any event which does not directly derive from the application (packets moving through the network, memory models) occurs here.

3.1 Code stack

The SST macroscale components and software stack are shown in Figure 3.1. An application is written with function calls to standard software APIs such as MPI or pThreads. Those calls are intercepted and passed to an SST implementation rather than the standard implementation. Network and compute activity is now modeled with SST rather than being executed on real hardware. For example, if executing an MPI_Send

```
#include <sstmpi.h>
int main(int argc, char** argv)
{
  MPI_Init(&argc,&argv);
  ...
  MPI_Send(...);
  ...
  MPI_Finalize();
  return 0;
}
```

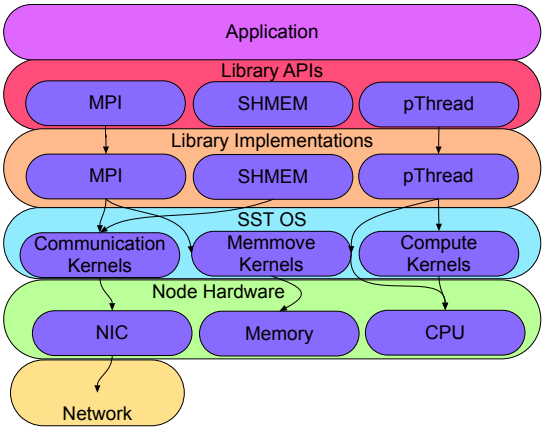

Figure 3.1. The SST Macroscale software/hardware stack. The application links to library APIs implemented on top of the SST operating system/hardware models.

In the header file `sstmpi.h`, we have

```
#define MPI_Send _SSTMAC_MPI_Send
#ifdef __cplusplus
extern "C"
#endif
int _SSTMAC_MPI_Send(...)
```

The SST header file links MPI calls to new symbols prefixed with SSTMAC to ensure that virtual (simulated) MPI calls do not conflict with real MPI calls in the parallel simulator core. The SST implementation of MPI looks like

```
extern "C"
int _SSTMAC_MPI_Send(...) {
  mpiapi* mpi = current_mpi();
  int rc = mpi->send(...);
  return rc;
}
```

The C function maps the current thread to an MPI object. This allows multiple (i.e. 1K-10K) virtual MPI ranks to coexist within a process. Rather than a global function modifying global state,

a member function modifies state local to the object. This corrects the global variable problem explained in Section 3.3. Inside the send function, we have the code:

```
int mpiapi::send(...) {
  ...
  //create object encapsulating msg
  mpi_message* msg = new mpi_message(...);
  //push to OS for sending
  os->execute_kernel(COMM_SEND, msg);
  //create a key to manage block/unblock
  key* mpi_key = new key(...);
  //associate the key with an MPI request
  mpi_request* mpi_req = new mpi_request(mpi_key);
  //block the thread until the send is complete
  os->block(mpi_key);
  ...
  return MPI_SUCCESS;
}
```

The operating system class passes the message to a network interface (NIC) to inject. The operating system must then block the application thread to advance time.

```
void os::block(key* k) {
  // associate the currently running thread with the key
  k->add_context(current_context);
  // get the main simulation thread off the thread queue
  thread_context* main_des_thread = thread_queue.top();
  // context switch back to the main simulation thread
  switch_context(current_context, main_des_thread);
  /* EXECUTION STOPS HERE */
  /* EXECUTION RESUMES HERE */
  ...
}
```

We get a context switch from the application thread to the main discrete event simulation (DES) thread. The DES thread now models the message being injected and any ACK notifications within the hardware. When the NIC generates an ACK, it passes the message back to the OS. The ACK now causes an unblock event to occur, returning back to the application context.

```
void os::handle(message* msg) {
  switch(msg->type())
  {
  case NIC_ACK:
    key* k = get_key(msg);
    unblock(k);
  ...
}
void os::unblock(key* k) {
  thread_context* blocked_context = k->get_context();
  switch_context(current_context, blocked_context);
}
```

3.2 Performance Analysis

The block/unblock mechanism is central to several performance metrics collected by SST. Every key gets extra state associated with it, identifying the type of event.

```
int mpiapi::send(...){
  key* mpi_key = new key(MPI);
  os->block(mpi_key);
}
```

In this case, the key is identified as MPI. In the block function, we can then put

```
void os::block(key* k){
  timestamp t_start = now();
  switch_context(...)
  /* EXECUTION STOPS HERE */
  /* EXECUTION RESUMES HERE */
  timestamp t_stop = now();
  timestamp delta_t = t_stop - t_start;
  // process the event that just occurred
  collect_stats(delta_t, k);
}
```

The simulator logs the current time as a stack variable before switching to the DES thread. When then send is complete and the application thread resumes, it can compute the time elapsed between the block and unblock. The OS object can now log the time spent in the event, which has been identified as MPI. This becomes useful in the fixed-time quanta (FTQ) charts demonstrated in Section 5.2. A function stack and backtrace (not explicitly shown) is also maintained for the active user-space thread. The simulator logs the virtual time spent with a backtrace allowing an application call graph to be generated (Section 5.1), analogous to profiling or sampling tools like kcachegrind. Although the discussion here focuses on MPI, the same discussion applies to any event that progresses virtual time (compute, memcpy, disk write).

3.3 Global variables

Because user-space threads are central to simulating parallel programs (i.e. MPI ranks) as real software stacks, the use of global variables in application code becomes complicated. Consider:

```
int counter = 0;
int main(){
  MPI_Init()
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ++counter;
  printf("On rank=%d, counter=%d\n",
    rank, counter);
  MPI_Finalize();
}
```

Programs run as distinct processes with distinct address spaces each get their own set of global variables. Run as a real MPI code, each MPI rank would print 1. In SST, every virtual MPI rank now runs as a user-space thread and every MPI rank now shares global variables. Run in the simulator, consecutive MPI ranks now print 2,3,4,... Mechanisms are provided for simulating global variables, for example

```
sst_global_int counter = 0;
```

```
int main(){
 ...
```

Here `sst_global_int` is a C++ class with syntax overloading matching an int type. We are also exploring compiler-based solutions, i.e. build the abstract syntax tree, identify global variables, replace them with SST lookup functions, and build the refactored code. Global variables are currently the biggest usability stumbling block.

Chapter 4

PDES Results

4.1 Experimental Setup

A key step in any parallel-discrete event simulation is partitioning components amongst the logical processes (LPs). As alluded to before, the optimal partition should occur on high latency links to allow the largest lookahead. In most current systems, the injection latency (i.e. NIC into network) is largest at approximately 1 μs [45]. Links between switches within the network are much smaller, on the order of 100ns or less. Very fast programming model explorations can be achieved if network details are abstracted away. In many cases, the primary focus of the study is the intrinsic load-balancing or resilience capabilities of a programming model assuming a steady, reliable network. Thus, even though in some sense primitive, simple latency-bandwidth models can still be very useful as a first-phase study. In particular, LogGOPSim [14] operates primarily through simple analytic expressions and has been used successfully on a number of problems [46, 47]. If the exact details of network switches, buffers, and queues are abstracted away, the simulation can be partitioned on injection links, greatly increasing the lookahead. Here we explore a coarse-grained hardware system model with three bandwidth and three latency parameters.

- Memory bandwidth/latency
- Injection bandwidth/latency
- Network bandwidth/latency

Even though network congestion is essentially ignored, injection and memory congestion is accounted for - albeit in a coarse-grained way. Experiments were run on the Hopper Cray XE6 at the National Energy Research Scientific Computing Center (NERSC) using 16 cores per node with one logical process per core. Although Hopper at full scale is a petascale machine, here we use at most 16 nodes (256 cores).

4.2 Weak-Scaling Experiment

We ran a weak-scaling experiment, simulating increasing problem sizes from 32K-262K network endpoints (nodes) on an asynchronous many-task matrix kernel from Section 5.2. The runtime manages task and data through a distributed hash table (DHT). An affinity scheduler sends tasks to nodes containing the majority of the input data. An overlay network monitors node activity/-failures and migrates tasks from overloaded nodes to nodes without work. The simulation covers approximately 0.1-0.2 s of virtual time. The asynchronous runtime is multithreaded and we simulate one process per node with 5 threads per process. For the largest simulation, this corresponds to explicitly simulating over 1M threads. Although falling short of our initial goal of 100:1 wall clock/virtual time ratio, we were still able to complete an exascale class simulation (100K-200K endpoints) in only two minutes. While we are only simulating a single kernel within an application, we have still reached the point where application segments can be rapidly explored in sub-hour times. For rapid prototyping or initial studies, a 10 minute wait may be use more useful than waiting in a PBS queue for days. Most critically, the simulation here is running a real asynchronous runtime and software stack - not just a DSL or stripped-down state machine model of the application. The actual matrix computation is not being performed, however, instead being simulated by a coarse-grained compute model parameterized by matrix size. This coarse-graining in most cases is straightforward, easily justified by the ability to simulate 100K distinct thread stacks on a small cluster or large SMP machine.

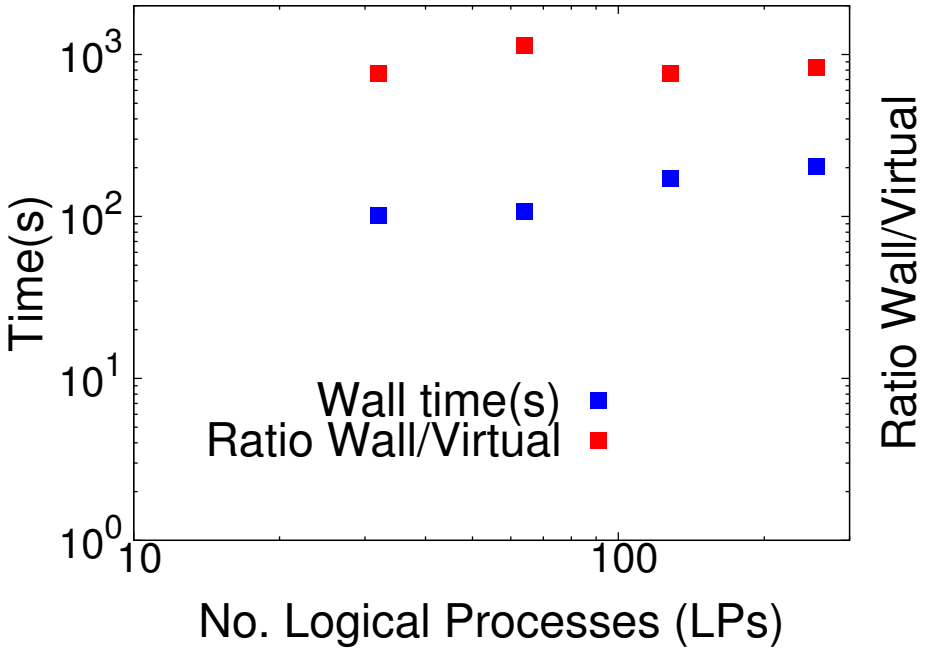

Figure 4.1. Weak-scaling experiment for 33K-262K network endpoints. Both the total simulation (wall clock) time and the ratio of wall clock to virtual time are shown for each experiment.

The curve in blue shows wall-clock time. The curve turns upward, indicating some scalability concerns. The "fixed constant" in the weak-scaling experiment is the total number of discrete events per logical process. Thus the number of events simulated per virtual process per second is decreasing for larger simulations. The application being simulated does not weak-scale, though, taking slightly longer for larger runs due to load-imbalance. Each simulation is therefore not covering the same amount of virtual time. If we normalize by plotting ratio of wall time to virtual time, we see a flatter curve. This demonstrates an expected feature of conservative PDES methods. If the same number of discrete events are spread over more virtual time, the simulation will be slower even though, nominally, it is doing the same amount of work. The ratio of wall/virtual time can be made to weak-scale, though.

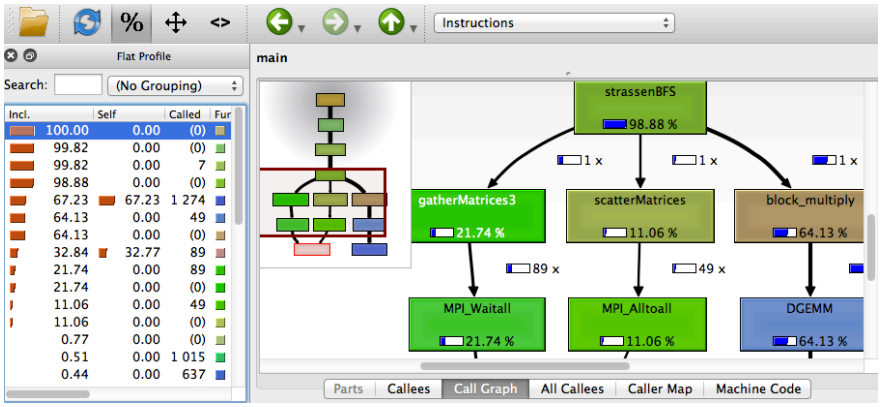

Figure 5.1. Call graph visualization in KCachegrind/Q-Cachegrind: full application window

Chapter 5

SST Visualizations and Performance Metrics

5.1 Call Graph

As discussed in Section 3, the block/unblock management of threads allows an application call graph to be generated. Here we demonstrate an example call graph for an experimental matrix-matrix multiplication code being explored with the simulator based on the Strassen algorithm [48]. Here the code is written in the standard MPI programming model. When SST runs, it creates a `callgrind.out` file readable by the KCachegrind/QCachegrind GUI (Figure 5.1). The list of all functions and time spent within the function are shown in the left panel. The call tree with time spent are shown in the right panel. From the initial overview, we see that 65 percent of the time is spent in the compute-intensive `block_multiply` function while the remaining time is spent in communication-intensive functions.

We can zoom into other areas of the call graph (Figure 5.2) for more details. Here we see that 33 percent of the application is actually spent in a busy poll loop within MPI. Some of the

contribution comes from an all-to-all collective while the rest comes from waiting on MPI requests in the `gatherMatrices` function.

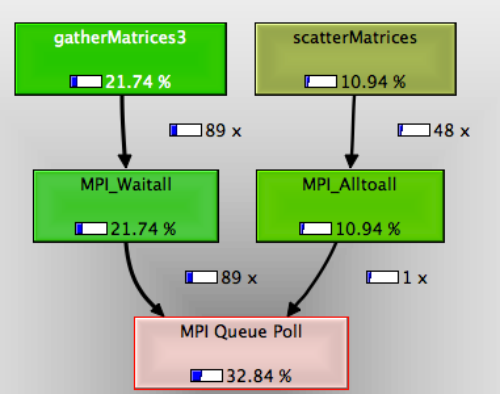

Figure 5.2. Call graph visualization in KCachegrind/Q-Cachegrind: zoomed in view of MPI function

5.2 Fixed-time Quanta (FTQ) Chart

As emphasized above, many programming models are focusing on asynchronous execution, data-flow, and overdecomposition, aiming to both overlap communication and rebalance load. The call graph only gives an aggregated picture, not showing progression over time. The fixed-time quanta (FTQ) provides a useful visualization of compute intensity (or generally any metric), showing what percentage of system cycles are spent on a given activity. Figures 5.3 and 5.4 show such an FTQ graph for a matrix-multiplication kernel, written in MPI and the asynchronous task runtime being explored within our group [49]. In both figures, green indicates communication overhead (i.e. bad) while blue indicates compute activity (i.e. good). In this simulation, we set one node to be degraded, running at half-speed (e.g. overheating, bad DIMM). As time progresses in the MPI code, communication overhead grows and grows. The MPI code is synchronous and does not rebalance. Thus all workers eventually go idle waiting on messages from the degraded node. In the many-task programming model, compute activity remains steady. Communication is a constant overhead, running on a dedicated thread. Because the many-task framework can rebalance and runs asynchronously, a constant stream of work is delivered to the nodes with only occasional patches of idle time.

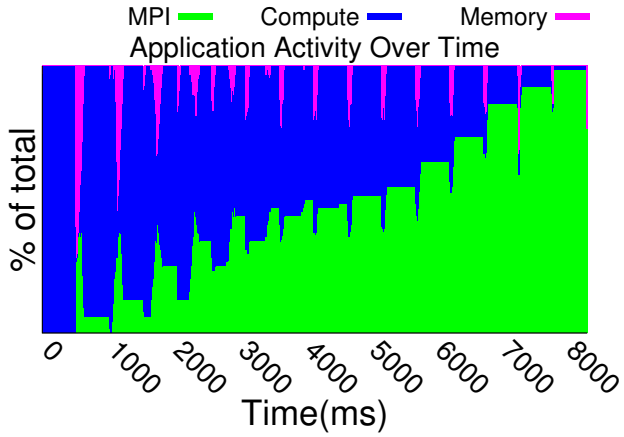

Figure 5.3. Fixed time quanta (FTQ) showing compute/communication activity aggregated over all threads for MPI SPMD model.

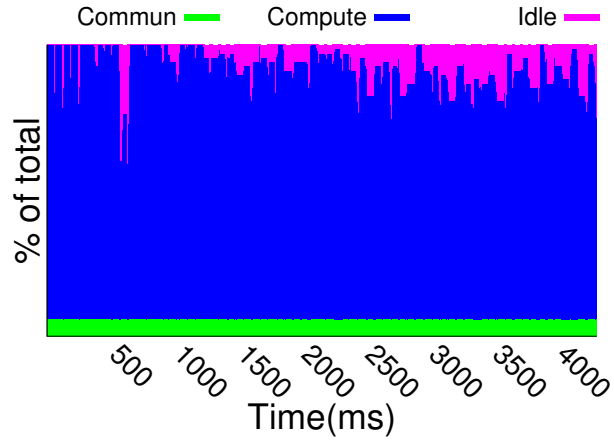

Figure 5.4. Fixed time quanta (FTQ) showing compute/communication activity aggregated over all threads for asynchronous many-task programming model.

Chapter 6

Summary

Here we have introduced macroscale components for the structural simulation toolkit (SST). The tools enable on-line architecture simulation for HPC applications, running real software stacks and application code. While broadly applicable, the design is particularly suited for programming model exploration and runtime system development. The complications associated with parallel discrete event simulation (PDES) are outlined. Through explicit management of user-space threads, a notion of virtual time is maintained for individual software stacks allowing metrics like call graphs and fixed time quanta to be collected as if from real application runs. PDES results show approximate weak-scaling, demonstrating that simulations involving 1.5 M explicit threads or more are feasible. Although we have fallen short of our original goal of 100:1 wall time to virtual time, further performance improvements are possible. The Cray XE6 results used MPI-only parallelism. SMP-aware optimizations should greatly decrease parallel overheads in future versions.

References

[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, B. Carlson, A. A. Chien, P. Diniz, C. Engelmann, R. Gupta, F. Johnson, J. Belak, P. Bose, F. Cappello, P. Coteus, N. A. Debardeleben, M. Erez, S. Fazzari, A. Geist, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "*Addressing Failures in Exascale Computing*," 2013.

[2] J. H. Chen, A. Choudhary, B. D. Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo, "Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D," *Comput. Sci. & Discovery*, vol. 2, 2009, p. 015001.

[3] A. S. Almgren, V. E. Beckner, J. B. Bell, M. S. Day, L. H. Howell, C. C. Joggerst, M. J. Lijewski, A. Nonaka, M. Singer, and M. Zingale, "CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity," *Astrophysical J.*, vol. 715, 2010, p. 1221.

[4] S. Ethier, W. M. Tang, and Z. Lin, "Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms," *J. Phys. Conf. Series*, vol. 16, 2005, p. 1.

[5] T. Heller, H. Kaiser, and K. Iglberger, "Application of the ParalleX execution model to stencil-based problems," *Comput. Sci.*, vol. 28, 2013, pp. 253–261.

[6] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, and Z. Weirong, "ParalleX: A Study of A New Parallel Computation Model," in *IPDPS '07: 21st International Parallel and Distributed Processing Symposium*, 2007, pp. 1–6.

[7] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny, "Using Simulation to Design Extreme-Scale Applications and Architectures: Programming Model Exploration," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 38, 2011, pp. 4–8.

[8] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo, "A Simulator for Large-Scale Parallel Computer Architectures," *Int. J. Distrib. Syst. Technologies*, vol. 1, 2010, pp. 57–73.

[9] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *SC '12: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[10] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, vol. 33, 1990, pp. 30–53.

[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] V. Subotic, J. C. Sancho, J. Labarta, and M. Valero, "A Simulation Framework to Automatically Analyze the Communication-Computation Overlap in Scientific Applications," in *CLUSTER 2010: IEEE International Conference on Cluster Computing*, 2010, pp. 275–283.

[13] E. A. Leòn, R. Riesen, A. B. Maccabe, and P. G. Bridges, "Instruction-Level Simulation of a Cluster at Scale," in *SC '09: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2009, pp. 1–12.

[14] 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.

[15] 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.

[16] 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.

[17] V. S. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou, "Compiler-optimized Simulation of Large-Scale Applications on High Performance Architectures," *J. Parallel Distrib. Comput.*, vol. 62, 2002, pp. 393–426.

[18] 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.

[19] 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.

[20] G. Zheng, K. Gunavardhan, and L. V. Kale, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *IPDPS '04: 18th International Parallel and Distributed Processing Symposium*, 2004, pp. 26–30.

[21] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalè, "Simulation-Based Performance Prediction for Large Parallel Machines," *Int. J. Parallel Program.*, vol. 33, 2005, pp. 183–207.

[22] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: A Generic Framework for Large-Scale Distributed Experiments," in *ICCMS 2008: 10th International Conference on Computer Modeling and Simulation*, 2008, pp. 126–131.

[23] F. Desprez, G. S. Markomanolis, and F. Suter, "Improving the Accuracy and Efficiency of Time-Independent Trace Replay," in *PMBS '12: 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, 2012.

[24] W. E. Denzel, J. Li, P. Walker, and Y. Jin, "A Framework for End-to-End Simulation of High-Performance Computing Systems," in *SIMUTools '08: Simulation Tools and Techniques for Communications, Networks and Systems*, 2008, pp. 1–10.

[25] B. Penoff, A. Wagner, M. Tuxen, and I. Rungeler, "MPI-NeTSim: A Network Simulation Module for MPI," in *ICPADS '09: 15th International Conference on Parallel and Distributed Systems*, 2009, pp. 464–471.

[26] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox, "Pace–A Toolset for the Performance Prediction of Parallel and Distributed Systems," *Int. J. High Perform. Comput. Appl.*, vol. 14, 2000, pp. 228–251.

[27] e. a. W Gropp, *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface*. Cambridge, MA: The MIT Press, 1999.

[28] A. Bhatele, N. Jain, W. D. Gropp, and L. V. Kale, "Avoiding Hot-spots on Two-level Direct Networks," in *SC '11: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[29] S. Kumar, Y. Sun, and L. V. Kale, "Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q," in *IPDPS 2013: 27th International Parallel and Distributed Processing Symposium*, 2013, pp. 689–699.

[30] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *OOPSLA 1993: 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993, pp. 91–108.

[31] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems," in *IPDPS '10: 24th International Parallel and Distributed Processing Symposium*, 2010, pp. 1–12.

[32] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland, "Large Scale Parallel Solution of Incompressible Flow Problems Using Uintah and Hypre," in *CCGrid 2013: 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2013, pp. 458–465.

[33] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," *Parallel Comput.*, vol. 38, 2012, pp. 37–51.

[34] S. Yanhua, Z. Gengbin, L. V. Kale, T. R. Jones, and R. Olson, "A uGNI-based Asynchronous Message-Driven Runtime System for Cray Supercomputers with Gemini Interconnect," in *IPDPS '12: 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 751–62.

[35] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, C. Dong, and B. Steinmacher-Burrow, "PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer," in *IPDPS '12: 26h International Parallel and Distributed Processing Symposium*, 2012, pp. 763–773.

[36] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS Community," in *Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3.

[37] D. Bonachea, "GASNet Specification, v1.1 U.C. Berkeley Tech Report (UCB/CSD-02-1207)," , 2002.

[38] J. P. F. Reynolds, "A Spectrum of Options for Parallel Simulation," in *WSC '88: 20th Conference on Winter Simulation*, 1988, pp. 325–332.

[39] D. Jefferson and H. Sowizral, "*Fast Concurrent Simulation using the Time Warp Mechanism: Part I*," 1982.

[40] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: A High-Performance, Low Memory, Modular Time Warp System," in *PADS 2000: 14th Workshop on Parallel and Distributed Simulation*, 2000, pp. 53–60.

[41] L. Ning and C. D. Carothers, "Modeling Billion-Node Torus Networks Using Massively Parallel Discrete-Event Simulation," in *PADS 2011: IEEE Workshop on Principles of Advanced and Distributed Simulation*, 2011, pp. 1–8.

[42] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, 1979, pp. 440–452.

[43] R. E. Bryant, "*Simulation of Packet Commnunication Architecture Computer Systems*," 1977.

[44] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Commun. ACM*, vol. 24, 1981, pp. 198–206.

[45] C. Dong, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker, "The IBM Blue Gene/Q Interconnection Fabric," *IEEE Micro*, vol. 32, 2012, pp. 32–43.

[46] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *SC '10: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[47] K. B. Ferreira, S. Levy, and P. G. Bridges, "*SAND REPORT: A Simulation Infrastructure for Examining the Performance of Resilience Strategies at Scale*," 2013.

[48] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *SPAA 2012: 24th ACM Symposium on Parallelism in Algorithms and Architectures*, 2012, pp. 193–204.

[49] J. J. Wilke, "Coordination Languages and MPI Perturbation Theory: The FOX Tuple-Space Framework for Resilience," in *PDSEC 2014: Parallel and Distributed Scientific and Engineering Computing Workshop at IPDPS*, 2014.

Sandia National Laboratories