SAND2011-9146C

Coarse-Grain Simulation of Networks-on-Chip using SST/Macro

Gilbert Hendry
Sandia National Laboratories
Livermore, CA
ghendry@sandia.gov

Simon Hammond
Sandia National Laboratories
Albuquerque, NM
sdhammo@sandia.gov

Abstract—**Simulation is the foundation of current network-on-chip research methods, which enables researchers to use established knowledge of computer networks in the context of a future single many-core chip. However, current NoC simulation methods can either lack in accurate traffic characterization in the case of synthetic drivers, or be cumbersome to develop and run in the case of microarchitectural simulation. Coarse-grain simulation offers key benefits which can augment current simulation efforts by providing extremely efficient execution of real applications while retaining the ability to investigate key NoC features such as topology and routing. In addition, execution-driven simulation of skeleton applications provides accurate and efficient characterizations of important codes which can be easily ported across simulators. SST/Macro is introduced as a useful tool for a variety of demonstrated simulation scenarios which extend across the spectrum of NoC research.**

I. Introduction

Since the birth of the concept of the network-on-chip (NoC), simulation has played a large role in understanding design tradeoffs for hypothetical implementations, mainly because real instances of NoCs are relatively rare. In general, few widely-used or general-purpose chips today exhibit what most researchers in the NoC community would consider a real network connecting many cores together. One good example of the NoC is the Tilera TILE series, which has a network connecting many simple cores consisting of 5 independent specialized regularly-tiled meshes [1]. However, this chip, like many other system-on-a-chip (SoC) or embedded processors which have a real network-on-chip, are not used in general or high-performance computing, and have specialized implementations for which generic NoC research does not readily apply.

The absence of the abundance of the true NoC is in part due to the persistence (and importance) of the large base of shared-memory code which relies on low-latency coherence protocols, and core counts not yet exceeding the capabilities (per area and power) of bus-based, point-to-point, or crossbar implementations. However, future scaling predicts that a full network-on-chip is the likely solution to performance, power, area, manufacturability, and resilience challanges for many-core chips. For this reason, simulation is the major tool of NoC research today.

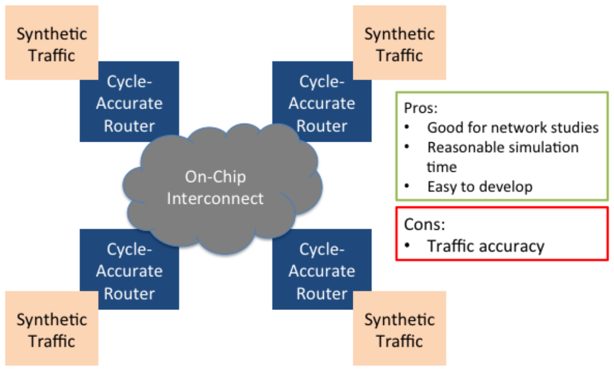
Most NoC research efforts which use simulation to focus on performance and power employ one of two methodologies to model a single many-core processor: a cycle-accurate router model driven by synthetic or traced traffic (shown in Figure 1a), or a cycle-accurate router model driven by microarchitectural simulation (shown in Figure 1b). Both of these approaches have advantages and disadvantages in regard to the proposed enhancements and the conclusions drawn from the experiments.

For many research efforts, synthetic traffic is generated which is meant to be characteristic of real application traffic, or possibly many applications. Common patterns include Random, Neighbor, Tornado (neighbors that are 2 hops away), and Hotspot (all-to-one). Though it is convenient and quick to use these patterns to test network architectural features, they are not enough to be confident about the performance of real applications. Indeed, designers of real commercial processors and the interconnects that go on them resort to large, complex simulations which run real industry benchmarks to test performance before they can make design and implementation decisions.

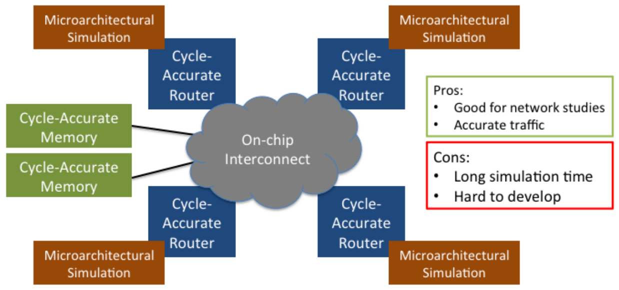
While industry simulators are almost never available to the public in the interest of maintaining competitive advantage, there are micro-architectural simulators which can be used for NoC research such as Garnet [2]. Using these detailed models enables the simulation of full application codes, often including the operating system, with fully-accurate processor and memory models. However, these simulations can be cumbersome to develop and maintain, take a long time to run, and may not necessarily be characteristic of real commercial hardware anyway.

One additional tool which could augment the current state of NoC simulation research is using execution-driven application models without resorting to microarchitectural simulation. As we will see in Section II, this can be achieved with the careful management of lightweight threads. This method allows us to fully model the characteristics of an application, including computation, control, and communication by writing or modifying codes in the same language and in the same way as real application codes are written. This also allows us to model key software libraries, such as MPI, in an efficient way to capture the way they operate but retaining fast, efficient simulation. In general, we refer to models which are more abstract than cycle-accurate but still describe the specific behavior of discrete software or hardware components as *coarse-grain* models.

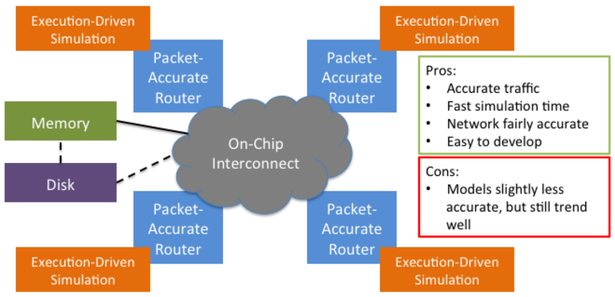
Figure 1c shows an illustration of coarse-grain modeling to



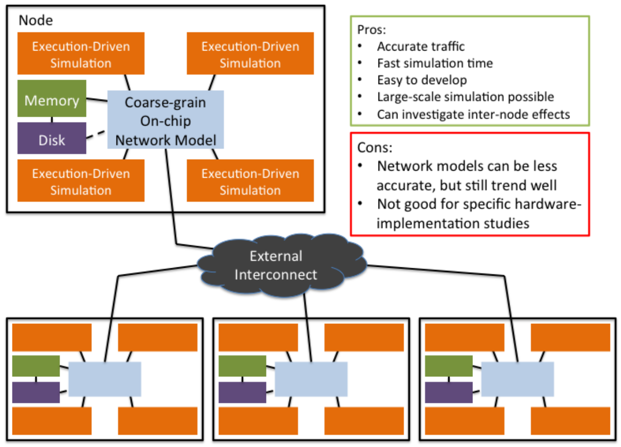
(a) Basic



(b) Microarchitectural



(c) Coarse-Grain


Node
Execution-Driven Simulation
Execution-Driven Simulation
Memory
Disk
Coarse-grain On-chip Network Model
Execution-Driven Simulation
Execution-Driven Simulation
External Interconnect
Pros:
Accurate traffic
Fast simulation time
Easy to develop
Large-scale simulation possible
Can investigate inter-node effects
Cons:
Network models can be less accurate, but still trend well
Not good for specific hardware-implementation studies

(d) Macro-scale

Figure 1: Simulations of networks-on-chip with varying degrees of accuracy, and the pros and cons of each.

compare to the previous methods. Modeling of memory and even disk is still fully possible, but in a more abstract way which can vary in degree. Coarse-grain modeling also enables the investigation of networks-on-chip in the context of large machines, such as supercomputers or datacenters, which is shown in Figure 1d.

This work is aimed at introducing coarse-grain simulation to NoC research as an efficient and effective tool by demonstrating various uses of SST/Macro, a simulator capable of running a large number of application threads which can closely approximate system performance and power. Coarse-grain simulation offers key benefits to NoC research which synergistically augment conventional cycle-accurate simulation research, including the use of real application codes, relatively accurate software and hardware models, and fast simulation times. Section II describes more about SST/Macro, and Sections III and IV perform some basic NoC experiments which are meant to demonstrate its various possible uses.

II. The SST Simulation Framework

The Structural Simulation Toolkit (SST) is a framework for building simulations by incorporating a variety of existing packages including processor, memory, disk, network, and software models [3]. SST/Macro is one branch of the SST project which is aimed exclusively at coarse-grain, macro-scale, and efficient simulations, and has previously been described and used in high performance computing-related studies [4], [5]. This section describes some of the more important parts of SST/Macro, including proxy applications, software models, hardware models, and validation.

A. *Managing State with Lightweight Threads*

One of the most novel implementation features of SST/Macro is its use of lightweight threads to save the state of an executing thread while it blocks so that simulation time can advance. This process is illustrated in Figure 2. The main discrete event simulation (DES) thread which processes general events initializes and starts application threads. These lightweight threads then can execute as normal code, but don't advance simulated time until they make calls into libraries, such as MPI. Libraries can register events and perform actions through a well-defined interface which accesses the hardware models and DES. Eventually, the library or application thread will block to let the DES process events and advance simulation time. When the library receives an event back from the DES, it decides when to unblock the application thread attached to it to continue execution.

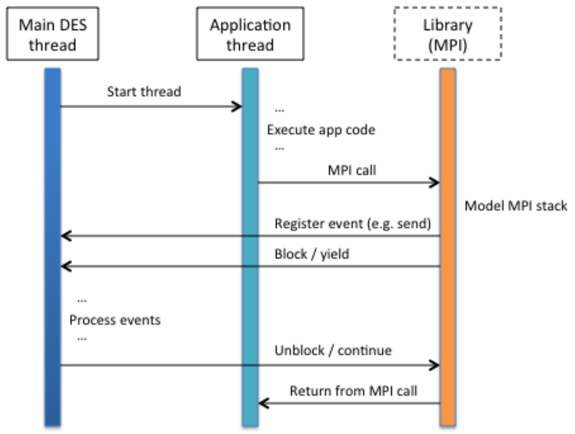


Figure 2: Lightweight thread execution in SST/Macro.

B. SST/Macro Software Model

SST/Macro 2.1, the latest version, has a full software stack model, shown in Figure 3, which contains a full coarse-grain model of an implementation of MPI. An MPI strategy layer allows us to experiment with different implementatons of collections, such as the barrier experiment performed later in Section IV. Each application thread has its own MPI queue where source and tag matching occur. These MPI queues feed to a single MPI server which muxes and demuxes incoming and outgoing MPI messages, and allows for intra-node communication via shared memory. For this paper, only one application thread is instantiated per core.

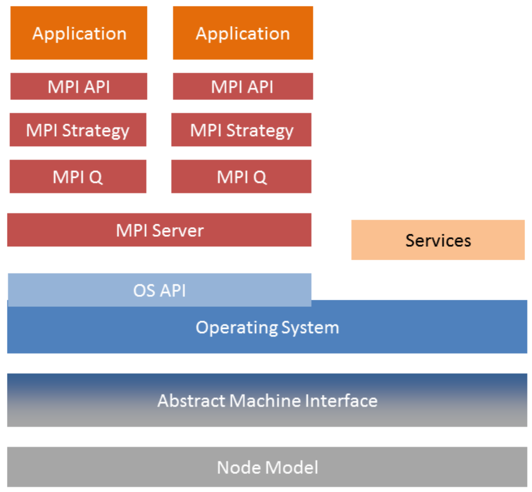


Figure 3: SST/macro software stack.

The MPI server, and other classes inherting from *Library*, have access to a limited set of operating system functions that allow executing specific operations through the hardware model. Services, such as application launchers, have more direct access to the operating system for accessing hardware addresses and thread management. An abstract machine interface (AMI) separates the software stack from the hardware models that implement the required functions: send, compute, etc. This allows us to easily port the software stack to other simulators or simulator backends. Currently, the SST/Macro software stack can run in its native backend, the SST/core backend [3], OMNeT++ [6], and SystemC.

C. Proxy Applications

Proxy applications are pieces of code which are meant to represent the characteristics of specific full applications, but are smaller, likely simpler, and easier (faster) to run than their parent codes. Using proxy applications is gaining support in high-performance computing as a way to make the base of extremely large and complex scientific codes which are of interest in chemistry, physics, astronomy, combustion, climate, and other domains accessible to simulation research in a timely fashion.

SST/Macro is meant to run *skeleton* applications, or applications that retain the communication and control information of the original code but abstract away any computation that is used to produce a real numerical result. This allows us to model an entire application at scale while looking at the features we are interested in, namely communication and its dynamic run-time behavior, without requiring massive computing resources to do so.

To illustrate a concrete example of the skeletonization process, Figure 4 shows a typical simple implementation of a distributed matrix multiplication using MPI which distributes one matrix and broadcasts the other among the processes, and collects the result. Figure 5 shows the same algorithm converted to a skeleton application to be run in SST/Macro. Note that the same control and communication structure is preserved, while the real work (multiplying the matrix) is taken out to speed up the simulation. In its place is inserted a call to a compute library, which models the computation as a few bulk processor and memory events. While in this example it might be reasonable in terms of computational power required to run the simulation to keep the real computation part of the code, extrapolating to larger, more complex problems it can be seen that removing the actual computation can potentially make the simulation extremely efficient without sacrificing too much accuracy. Note that the calls to MPI are specific to SST/Macro's MPI implementation, though future work will provide an interface identical to the real thing, facilitating the conversion of existing codes to skeletons.

For our experiments in Sections III, we are considering a number of scientific codes included in the SST/Macro distribution which have been reduced to *skeletons*. These mini applications all use MPI, the relative standard for existing distributed scientific codes. While MPI may not be the ultimate solution for inter-node communication in the far future, it is likely to persist in some form for many years. Knowing this, giving each core the ability to act as an MPI peer is not at all unreasonable. Regardless, the applications that we explore in this paper are listed below:

- **miniMD**: MiniMD is a molecular dynamics micro-application from the Mantevo project [7]. MiniMD, which is under 3000 lines of code, was created to investigate

```
int getRowCount(int rowsTotal, int mpiRank, int mpiSize) {
    return (rowsTotal / mpiSize) +
      (rowsTotal % mpiSize > mpiRank);
}

int main(int argc, char *argv[]) {
    int n = 0, n_ubound, n_local, n_sq, i;
    int mpiRank = 0, mpiSize = 1;
    double *A, *B, *C, t;
    int sizeSent, toBeSent;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpiRank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpiSize);

    /* Get n and broadcast it to all processes */
    if (!mpiRank)   n = atoi(argv[1]);

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    n_local  = getRowCount(n, mpiRank, mpiSize);
    n_ubound = n * n_local;
    n_sq     = n * n;

    A = (double*) malloc(sizeof(double)
      * (mpiRank ? n_ubound : n_sq));
    B = (double*) malloc(sizeof(double)
      * n_sq );
    C = (double*) malloc(sizeof(double)
      * (mpiRank ? n_ubound : n_sq));

    if (!mpiRank) { //Initialize A and B
        for (i=0; i<n_sq; i++) { A[i] = 1.0; B[i] = 1.0;
        }
    }

    if (!mpiRank) {  //Send A by splitting it row-wise
        sizeSent = n_ubound;
        for (i=1; i<mpiSize; i++) {
            toBeSent = n * getRowCount(n, i, mpiSize);
            MPI_Send(A + sizeSent, toBeSent, MPI_DOUBLE,
              i, TAG_INIT, MPI_COMM_WORLD);
            sizeSent += toBeSent;
        }
    }
    else { //Receive parts of A
        MPI_Recv(A, n_ubound, MPI_DOUBLE, 0, TAG_INIT,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Bcast(B, n*n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    for (i=0; i<n_ubound; i++) { //initialize C to zero
        C[i] = 0.0;
    }

    for (int i=0; i<n_local; i++) {
        for (int j=0; j<n; j++) {
            for (int k=0; k<n; k++) {
                C[i*n + j] += A[i*n + k] * B[k*n + j];
            }
        }
    }

    if (!mpiRank) {//Receive partial results from each slave
        sizeSent = n_ubound;
        for (i=1; i<mpiSize; i++) {
           toBeSent = n * getRowCount(n, i, mpiSize);
            MPI_Recv(C + sizeSent, toBeSent, MPI_DOUBLE, i,
              TAG_RESULT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            sizeSent += toBeSent;
        }
    }
    else { //Send partial results to master
        MPI_Send(C, n_ubound, MPI_DOUBLE, 0, TAG_RESULT,
          MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Figure 4: Original matrix multiply code

```
int getRowCount(int rowsTotal, int mpiRank, int mpiSize) {
    return (rowsTotal / mpiSize) +
      (rowsTotal % mpiSize > mpiRank);
}

int skeleton_main(int argc, char *argv[]) {
    long n_ubound, n_local, n_sq, i;
    double t;
    int sizeSent, sizeToBeSent;

    mpi()->init();
    sstmac::sw::mpicomm world = mpi()->comm_world();
    long mpiRank = world.rank().id;
    long mpiSize = world.size().id;

    mpi()->bcast(1, mpitype::mpi_int, mpiid(0), world);

    n_local = getRowCount(matrix_order_, mpiRank, mpiSize);
    n_ubound = matrix_order_ * n_local;
    n_sq = matrix_order_ * matrix_order_;

    mpitag itag(TAG_INIT);
    if (!mpiRank) {
        sizeSent = n_ubound;
        for (i = 1; i < mpiSize; i++) {
            sizeToBeSent = matrix_order_
                * getRowCount(matrix_order_, i, mpiSize);
            mpi()->send(sizeToBeSent, mpitype::mpi_double,
              mpiid(i), itag, world);
            sizeSent += sizeToBeSent;
          }
      }
    else { // Receive parts of A
        mpi()->recv(n_ubound, mpitype::mpi_double,
          mpiid(0), itag, world, mpirequest_t());
      }

    mpi()->bcast(matrix_order_ * matrix_order_,
      mpitype::mpi_double, mpiid(0), world);

    comp_lib_->double_mxm(n_local, 1, matrix_order_, 1);

    mpitag rtag(TAG_RESULT);
    if (!mpiRank) {
        sizeSent = n_ubound;
        for (i = 1; i < mpiSize; i++) {
            sizeToBeSent = matrix_order_
                * getRowCount(matrix_order_, i, mpiSize);
            mpi()->recv(sizeToBeSent, mpitype::mpi_double,
            mpiid(i), rtag, world, const_mpistatus_t());
            sizeSent += sizeToBeSent;
          }
      }
    else { // Send partial results to master
        mpi()->send(n_ubound, mpitype::mpi_double,
          mpiid(0), rtag, world);
      }

    mpi()->finalize();
}
```

Figure 5: Matrix multiply as a skeleton

improving spatial-decomposition particle simulations as a simpler, but more accessible and easily built and executed version of LAMMPS [8], which is over 130k lines of code. Parameters to miniMD include problem size, atom density, temperature, timestep size, number of timesteps, and particle interaction cutoff distance.

- **mpi3d**: mpi3D represents a general class of applications by modeling the interactions between the faces, edges, and corners of spatially-decomposed 3-dimensional blocks each assigned to a process. The skeleton runs multiple iterations consisting of a send phase, receive phase, and compute phase. Non-blocking sends and receives are used to pipeline communication with computation.

- **LU**: LU [9] is an application-level benchmarking code supplied as part of the NAS Parallel Benchmark suite (NPB). Now in its third incarnation, LU has been adapted and rewritten in successive releases of the NPB to utilize parallel programming technologies such as OpenMP, HPF, MPI and Java. The algorithm solves a synthetic system of non-linear PDEs using a symmetric successive over-relaxation (SSOR) kernel employing a two-phase wavefront sweep through the 3-dimensional data domain. The reference implementation provides problem 'classes' which range from a small serial implementation through to multiple distributed nodes requiring multiple Tera-bytes of system memory. This paper focuses on the class B problem size in order to remain within acceptable runtime limits.
- **sweep3D**: Sweep3D [10] is a particle transport benchmark which was designed to be a compact and simplified representation of algorithms employed by the Los Alamos National Laboratory and other Department of Energy (DoE) HPC sites. For this reason, the benchmark version of the code has featured in recent large-scale DoE programme purchases including the most recent Sequoia and Ceilo machine procurements. The code employs a commmunication-optimized wavefront design pattern to solve a fixed 12-iteration multi-angle, multi-group Boltzmann transport problem.

D. SST/Macro Hardware Models

SST/Macro contains hardware models of whole processors, whole compute nodes, even whole networks which are aimed at large-scale high performance computing simulations. In this work, we will be using the simplecore model, which is an approximation of a single core of a many-core processor. It contains a instruction-processing unit, scratchpad memory for both instructions and data, a DMA engine for accessing external memory, and a point-to-point engine for directly sending to other cores.

E. SST Validation

Validation against real hardware is difficult given the lack of real chips which have the NoC architecture of the future commonly envisioned by the NoC research community. However, preliminary results aimed at validating SST/Macro's software stack model look promising by comparing a replayed MPI trace to the real execution. Figure 6 shows simulated execution time versus real execution time for a test application running on a large cluster. Note that while scaling up, SST/Macro stays within 5% error. While not validating NoC-like characteristics, it does show that the communication and computation primitives which are used as a part of the core model reflect real hardware operation. Future work will include cross-validation against micro-architectural simulators.

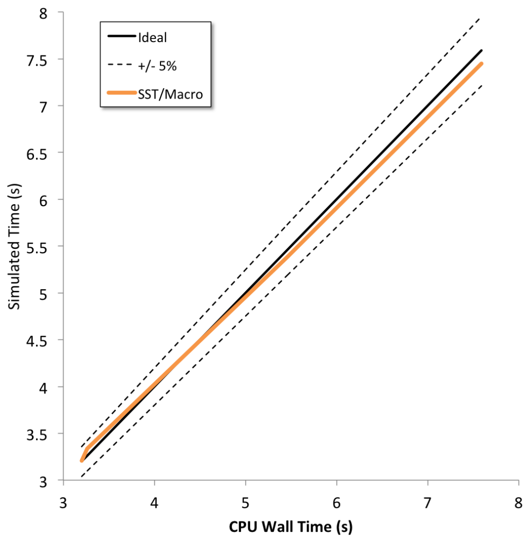


Figure 6: SST/macro validation running AMG.

Table I: Default Simulation Network Parameters

Parameter	Value
Clock Freq.	1 GHz
Input Buffer	8 kB
Ouput Buffer	16 kB
MTU	256 B
Virtual Channels	2
Arbitration Lat.	3 ns
X-Bar Lat.	2 ns
X-Bar BW	2 Gbps
Link BW	1 Gbps

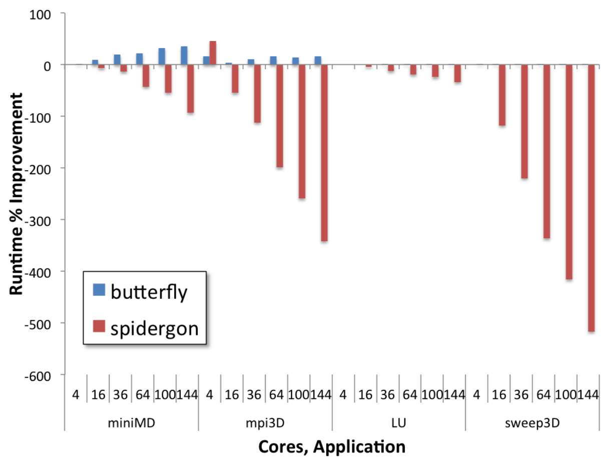
III. Experiment 1: A NoC Network Study

To demonstrate a baseline of simulator functionality in regard to NoC research, in this experiment we will look at performing some of the more classical hardware-design tradeoff NoC studies with SST/Macro using it's packet-level router model (Coarse-Grain from Figure 1c). Table I lists the baseline parameters which are the default unless otherwise mentioned.

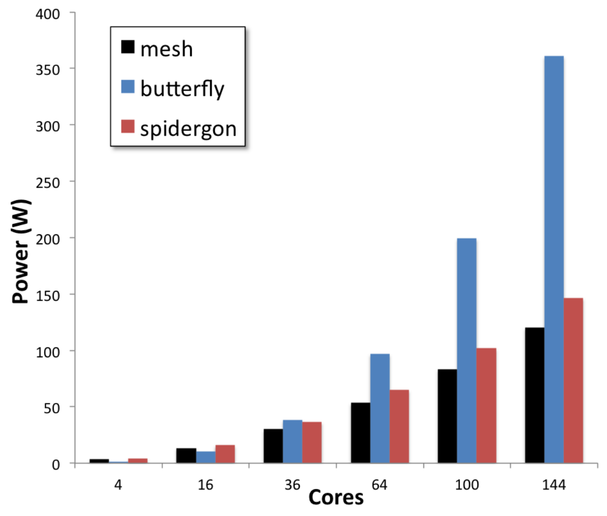
A. Scaling and Topology

Strong scaling, or scaling the number of processing units while keeping the amount of total work to be done *fixed*, is often employed as one way of characterizing the behavior of an application. In terms of hardware, it can also be viewed as a way of characterizing how the network responds to a growing number of endpoints in terms of power and area. The main characteristic of a NoC we will explore here is topology in the presence of a strongly-scaled application, measuring both power and performance.

The question of topology is a relatively straight-forward matter: what interconnection of routers yields the best performance for the lowest area and power cost. Router radix is a major factor in this, which affects the amount of buffering required and the degree of the crossbars. Often, a simple mesh is considered the baseline because of its attractive regularity, reducing cost in design, layout, and test. In this study, we will compare the mesh to the Spidergon [11] and Flattened Butterfly [12] topologies.



(a) % Improvement in runtime over mesh



(b) Power

Figure 7: Results from experiment 1-A: effects of scaling and topology on power and application performance.

Figure 7a shows the improvement in application runtime of the two networks in question over the baseline mesh topology. As network size grows, the Flattened Butterfly shows significant improvement for miniMD and mpi3D because of the two-hop latency between any points in the network, whereas the Spidergon suffers significantly even with more buffering, probably because of the grid-like nature of the data layout in the applications and the lack of north-south connections in the network topology. For LU and sweep3D, the Flattened Butterfly does not show significant improvement over the mesh. Looking at power in Figure 7b, any performance gained with Flattened Butterfly comes at the price of significantly more power due to the buffering, crossbar, and link scaling required with the higher-radix routers even though it was given smaller buffers. Spidergon, given more buffering to mitigate the severe contention caused by low-radix switches and limited connectivity exhibits slightly more power than the baseline mesh topology.

B. Dispersive Routing

Dispersive routing is a dynamic routing technique which refers to packets of the same message headed for the same destination taking separate routes through the network to spread, and ideally balance, the load. In this experiment, we will test both packet-level and message-level dispersive routing in a mesh topology.

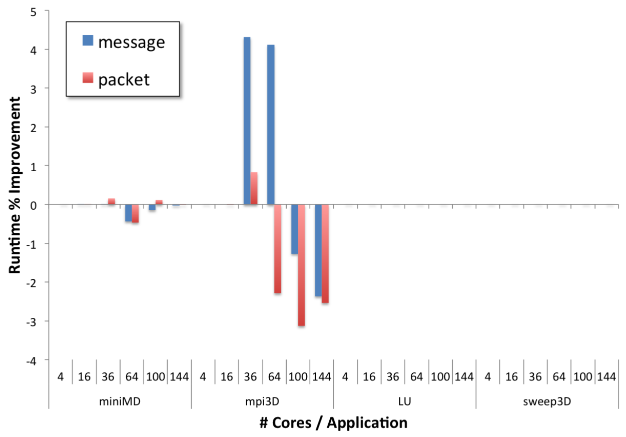


Figure 8: % Improvement of dispersive routing.

Figure 8 shows the improvement over standard X-Y routing for different size networks running different applications. We see that for small networks, the routing policy does not have a significant effect on performance. For most cases, dispersive routing has a slightly negative impact, possibly creating head-of-line blocking in buffers under already light loads. In a few cases dispersive routing is slightly better, possibly caused by the nature of the application's map onto the network. Interestingly, LU and sweep3D are not affected at all due to the nearest-neighbor and straight-line nature of their communication patterns, which takes routing path out of the performance.

IV. Experiment 2: Programming a Single Node

One of the strongest features of SST/Macro is its modeling of the software stack, including the complete implementation of the key components of the MPI library. In this experiment, we will investigate changes to the software stack which is uniquely easy and accessible in SST/Macro.

A. Programming Models

With the end of frequency scaling and increasing parallelism in general-purpose and embedded chips alike, researchers are looking to new programming and execution models which will help domain scientists exploit algorithm concurrency. In this study, we will compare a matrix multiplication implemented three different ways: a simple MPI implementation, a MPI systolic array implementation, and an actor model implementation.

- Simple MPI
  As the baseline, we will consider the matrix multiplication from Figure 5 implemented by distributing the rows of one matrix among the MPI ranks, broadcasting the entire second matrix, and collecting the results back to a root core. This represents a naive solution, because the broadcast of one of the matrices is relatively undesirable due to a large amount of replication of the second matrix.
- MPI Systolic Array
  Our systolic matrix multiply skeleton application is based on Cannon's 2D algorithm as described by Golub and Van Loan [13], and Gupta and Kumar [14]. An MPI gather is performed after the computation is done to collect to a single core.
- Actor Model
  The actor model implementation in SST/macro is a framework for creating dynamic applications which assign work to computor threads asynchronously. Though not suited for every problem, and not necessarily optimal for an application running on a system, actor model applications can be good at adapting to run-time variances, such as a degraded clock frequency, because of the asynchronous nature of the computation. Our matrix multiply implementation assigns blocks of the matrix to physical cores and shares them with peers as needed. Once a computor is done, it can notify neighboring cores that it is able to accept work to balance the load.

Figure 10 shows the results of running the parallel matrix multiply implementations with different sized matrices on an 8×8 mesh topology. The systolic array implementation does consistently better than the simple broadcast method, as expected. Interestingly, the actor model consistently performs better than the MPI solutions but only if data prefetching is turned off, which is aimed at pipelining computation with communication but presumably creates network congestion affecting the main flow of traffic.

B. MPI Collective Implementation

Often, different implementations of MPI collectives will be better suited to different systems based on factors like application indexing and network topology. In this study, we will look at different implementations of an MPI barrier on different topologies. Barriers are often used to separate logical computation phases of an application, and can be important depending on the size of the application. We will look at three implementations:

- Linear
  The naive implementation where all cores send a message to one root core, who then broadcasts a response when it receives every core's notification. This requires $2\times(P-1)$ messages, where P is the number of processes.
- Ring
  A token is passed to sequentially-numbered processes (cores) starting at 0 until it is received back, where it is passed back to release processes from the barrier. This requires $2 \times P$ messages, but may reduce network contention by using mostly nearest-neighbor communication.
- Hensgen
  This implementation is based on the work in [15], which uses $\text{Log}_2(P)$ steps (and messages). In step k, process i sends to process $(i+2^k)\ \%\ P$ and receives from process $(i-2^k+P)\ \%\ P$.

Figure 11 shows the different barrier implementations running on different sized topologies, executing 100 barriers consecutively. Though topology plays a small part, the Hensgen implementation clearly scales the best with network size because of the more efficient synchronization strategy.

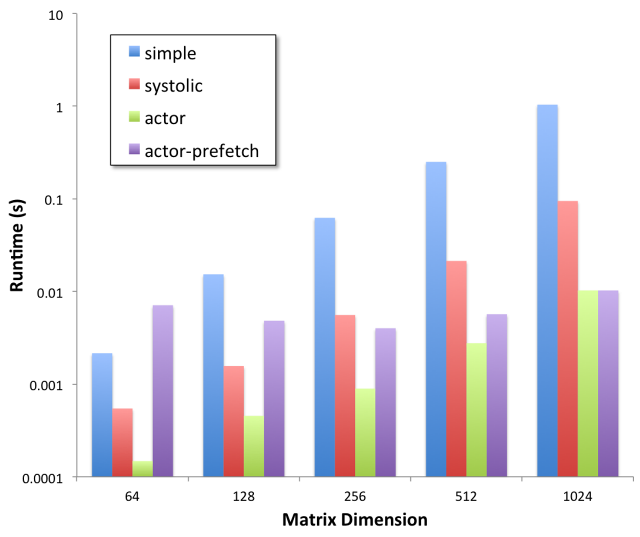


Figure 9: Comparison of matrix multiply implementations.

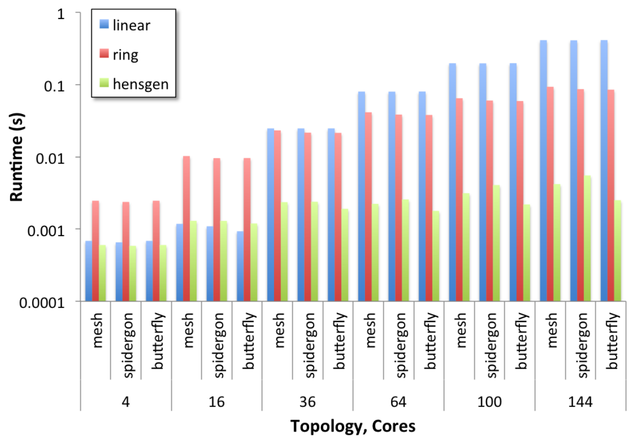


Figure 10: Comparison of MPI barrier implementations on different topologies.

V. Conclusion

Coarse-grain modeling was introduced as a means of quickly investigating design tradeoffs in both hardware and software for running real applications on future network-on-chip architectures. Proxy applications, or simplified representations of large complex codes, were transformed into skeleton applications which only retain the communication and control parts of the code, while abstracting and modeling the computation parts. Using these skeletons, we were able to demonstrate the use of SST/Macro by investigating network topology and routing, as well as software implementation features such as barrier algorithm and programming implementation. All simulations presented completed in under 5 minutes, demonstrating the quick experimental turn-around of the simulation method. Though the details of the simulations performed are not themselves important, introducing SST/Macro to the NoC community is pivotal in forming a common ground though a scientific application and benchmarking base.

Acknowledgment

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *Micro, IEEE*, vol. 27, no. 5, pp. 15 –31, sept.-oct. 2007.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, april 2009, pp. 33 –42.

[3] 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," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 37–42, March 2011.

[4] 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." *IJDST*, vol. 1, no. 2, pp. 57–73, 2010.

[5] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny, "Using simulation to design extremescale applications and architectures: programming model exploration," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 4–8, March 2011.

[6] A. Varga, "The omnet++ discrete event simulation system," *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001.

[7] M. A. Heroux *et al.*, "Improving performance via mini-applications," Sandia National Labs, Tech. Rep. SAND2009-5574, September 2009. [Online]. Available: https://software.sandia.gov/mantevo

[8] "LAMMPS molecular dynamics simulator," 2009. [Online]. Available: http://lammps.sandia.gov/index.html

[9] M. Yarrow and R. D. Wijngaart, "Communication improvement for the lu nas parallel benchmark: A model for efficient parallel relaxation schemes." NASA Ames Research Center, Tech. Rep. NAS- 97-032, November 1997.

[10] A. Hoisie, H. Lubeck, and H. Wasserman, "Performance and scalability analysis of teraflop-scale parellel architectures using multidimentional wavefront applications," *Int. Journal of High Performance Computing Applications*, vol. 14, no. 4, p. 330346, 2000.

[11] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, "Spidergon: a novel on-chip communication network," in *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, nov. 2004, p. 15.

[12] J. Kim, J. Balfour, and W. Dally, "Flattened butterfly topology for on-chip networks," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 172–182.

[13] G. Golub and C. Loan, *Matrix computations*, ser. Johns Hopkins studies in the mathematical sciences. Johns Hopkins University Press, 1996.

[14] A. Gupta and V. Kumar, "Scalability of parallel algorithms for matrix multiplication," in *in Proc. of Int. Conf. on Parallel Processing*, 1991, pp. 115–123.

[15] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Program.*, vol. 17, pp. 1–17, February 1988.