D2.2.1 – Simulation and modeling exascale technology

WP2: Underpinning and cross-cutting technologies

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M12
Submission date:	30/09/2012
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	UEDIN
Version:	2.0
Status	Final
Author(s):	Stephen Booth (UEDIN)
Reviewer(s)	Achim Basermann (DLR), Derek Groen (UCL)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	29/08/2012	First version of the deliverable	Stephen Booth (UEDIN)
1.0	31/08/2012	Version for review	Achim Basermann (DLR), Derek Groen (UCL)
2.0	14/09/2012	Addressed internal reviewers comments	Stephen Booth (UEDIN)

Table of Contents

Index of Figures

1 Executive Summary

Simulation and modeling are important tools in the development of exascale systems. There are very few other mechanisms for evaluating our designs for exascale hardware and software other than developing models of their behavior and simulating these models in a computer. The behavior of both hardware and software needs to be modeled.

In the early stages of the design process these models need to be quite simple and abstract. This allows us to develop and evolve our designs quickly and efficiently. If we attempt to use overly complex models in these early design stages then we will waste time and resources performing overly detailed simulations of design choices that will be abandoned before the final system is built.

Current thinking about exascale hardware design is that these designs will be highly constrained by system power consumption. To keep power consumption within acceptable levels exascale systems will need to utilize very high degrees of parallelism and the performance of their communication systems may also have to be limited. This implies that we should be using software models that explicitly capture the available parallelism and the communication requirements of an algorithm. One way of capturing this information is to consider modeling the parallelizable sections of the algorithm as directed acyclic graphs.

Application behavior can be simulated at a high level by simulating the communication pattern of the application. This allows the behavior of the application to be extrapolated to different (possibly theoretical) hardware platforms. This allows us to explore the behavior of applications on exascale hardware well before such hardware becomes available.

Various different approaches to application simulation exist. One particularly interesting approach is the use of simple skeleton applications to drive the simulation. These are lightweight simple codes intended to capture the essential behavior of larger much more complex applications. They provide a mechanism of exploring the behavior of new designs without the cost of first developing the design into a fully functional application. As this approach aims to capture the communication pattern rather than the details of the computational sections it provides a mechanism to develop a directed acyclic graph model into a form that can be simulated.

A number of simulation platforms exist that are suitable for this kind of simulation. These can give useful insights into the limitations that the network imposes on application performance. Many of these platforms are explicitly targeting the development of exascale systems. Though these simulation tools are useful they are fairly complex research tools and can be quite difficult to use.

2 Introduction

This report looks at the role of modeling and simulation in the development of exascale hardware and software.

Section 3 describes the motivation for using modeling and simulation in the development of exascale systems and the role these can play in the design process.

Section 4 discusses what types of model are appropriate for the current stage of exascale system development.

Section 5 investigates the various types of simulator that might be useful in this process and some of the available software packages.

Section 6 is a case study applying some of the tools and techniques discussed in this report to the FFT algorithm.

2.1 Glossary of Acronyms

DES	Discrete Event Simulation
PDES	Parallel Discrete Event Simulation
DAG	Directed Acyclic Graph
MPI	Message Passing Interface
OTF	Open Trace Format
FFT	Fast Fourier Transform

3 Motivation

3.1 The need for models and simulation

In order to achieve exascale science we will require Computers capable of running at exascale performance levels. We also require codes and system software that are capable of utilizing these machines effectively. Neither of these exists at the current time so will need to be designed and built over the next few years. Inevitably the development processes for hardware and software will have to take place in parallel. This introduces a potential problem, as the target hardware will not be available while the software is being developed. This makes it difficult to evaluate how the new codes will run on real exascale hardware. Similarly hardware designers will be trying to anticipate the requirements of software that does not yet exist. The normal approach to understanding the behavior of any system that cannot be observed directly is to build a model of the system and to use the model to predict the behavior of the system. Frequently the model is too complex to allow analytic calculations based on the model. In these cases we simulate the model in a computer and observe the behavior of the simulation. It therefore seems sensible to attempt to model and simulate the behavior of exascale hardware and software.

The other obvious way of approaching this problem is to assume only incremental changes in hardware and software. Software developers can assume that future machines will behave as larger versions of current systems and hardware designers can assume that future code requirements may be extrapolated from the behavior of current codes. Even when taking this conservative approach, simulation is still a very useful tool as it provides a mechanism to perform these extrapolations to systems larger than those currently available.

If we wish to consider more radical and disruptive changes in hardware architecture then software developers can use models of possible exascale hardware designs and can simulate code running on these designs in order to evaluate how these designs will impact the behavior of their codes.

Similarly if we wish to consider more radical and disruptive changes in software, including application code and system software, then the hardware designers can use models of these possible exascale software stacks in order to evaluate the suitability of their designs for these new requirements.

If we want to consider radical and disruptive changes in all areas then is seems logical to use a co-design approach where the overall design space is explored as a coordinated activity using models of both hardware, applications and system software.

3.2 Abstract models and design

Any design activity proceeds via a series of candidate designs. Each design needs to be evaluated in some way to determine if the design is a good one or if it contains flaws or inefficiencies that need to be fixed in the next design iteration. This requires some model of the behavior of the design in order to perform the evaluation. Designs and models are therefore very closely related.

Designs (and their associated behavioral models) can exist at various levels of detail and complexity ranging from a single power point slide up to detailed design drawings and full source code listings. Even though these fully detailed designs will be necessary eventually, simple abstract and low-detailed designs have an important part to play in the design process. Frequently when a problem is discovered during the design process it is necessary to backtrack to an earlier design in order to pursue a different design choice. When this happens much of the design effort (and the modeling and simulation effort) expended on the abandoned design choice is wasted. However if the possible design choices can be explored and partially evaluated early in the design

process, using simple abstract designs and models, it might be possible to identify such problems at this early stage. If we can use relatively inexpensive abstract models to identify problems then the amount of effort being wasted will also be relatively low. If the same flaw is not detected until much later in the design process, when the design has become more detailed, then cost of fixing the problem will probably be substantially greater.

It is therefore useful to have a range of different models at different levels of abstraction. Simple lightweight and highly abstract models are needed to evaluate early designs. As the design process proceeds and designs become more detailed newer more detailed models will also need to be developed in order to evaluate them. When modeling software, these abstract early models need to capture the essential characteristics of the algorithm with more implementation specific details being added later in the process.

These need to be parameterized models. Parameters typically represent design choices such as the number of processors, clock frequency or the size of problem being simulated. They may also represent characteristics that would be derivable from a more detailed model but need to be added as an estimated parameter in simpler models.

If a design is expressed in a formal well-defined syntax it is often possible to automatically extract a behavioral model from the design. For example a simulation of an electronic circuit may be automatically generated from a VHDL circuit specification. Program source code can also be thought of as a fully specified design that the compiler uses to build an executable. Compiler based technology can also be used to extract a behavioral model of the software. Unfortunately such well-defined designs are typically only available quite late in the design process and the tools needed to extract the behavioral models are complex software products in their own right and will require significant effort to develop.

4 Models for the exascale

Because the space of possible models is very large the first thing we need to consider is what kind of models are most appropriate for the current stage of development of exascale systems.

Current exascale hardware designs are very abstract; the key details of current straw-man proposals for exascale systems can be conveyed in a handful of PowerPoint slides. Nevertheless simple models of these very rough designs have been sufficient to identify a number of critical constraints on how exascale systems will be built which in turn place constraints on the software designs and inform the types of corresponding model that we should be using to represent software. One key constraint on practical exascale hardware seems to be a limit on the total power consumption of the system. This implies that models of power consumption are as important as the performance models. There seems to be unanimous agreement that processor clock frequencies cannot be allowed to increase much beyond the current GHz frequencies so exascale performance can only be achieved for applications with an inherent parallelism of $O(10^9)$. In addition the power costs of data movement are also expected to become a significant part of the overall power budget so the communication capabilities of practical exascale systems will need to be limited to keep this power consumption in check. Models of exascale algorithms/software will therefore need to pay particular attention to potential parallelism and data movement patterns.

Most proposed exascale hardware designs assume multiple levels of hardware concurrency with a high degree of on-node parallelism (e.g. SIMD instruction sets and multiple cores connected by shared memory) as well as a high degree of parallelism between nodes (i.e. a large number of nodes). Some proposals assume almost as much parallelism within a node as between nodes.

Traditionally, algorithms are described in terms of their complexity O(n), O(n.log(n)) etc. This can be thought of as an extremely simple parameterized behavioral model that only considers the total number of operations to be performed. While this would be appropriate for computer architectures where the performance is primarily limited by the rate that instructions can be issued; it is far too simple for our purposes as it ignores both parallelism and communication. A more complex, but more informative, approach is to model algorithms as directed graphs, where the nodes of the graph represent computations and the edges of the graph represent data dependencies. While graphs containing cycles are required to represent a general algorithm, for example an iterative or time-stepping algorithm, the parallelizable sections of the algorithm can always be represented by Directed Acyclic Graphs (DAGs). Using such a representation makes the available parallelism explicit (the width of the graph) as well as capturing the essential communication pattern (see Figure 1 Example of a DAG representation of a computation).

Quite simple, but nevertheless informative behavioral models, can then be built by modeling the hardware in terms of a network of computational elements and decomposing the nodes of the graph onto the compute elements. Time and energy costs can be assigned to computation on the nodes and messages passing through the network. Though still relatively simple this kind of model will quickly become too complex to easily extract performance predictions without simulating the model in a computer. This is particularly necessary when trying to understand contention for a shared resource.

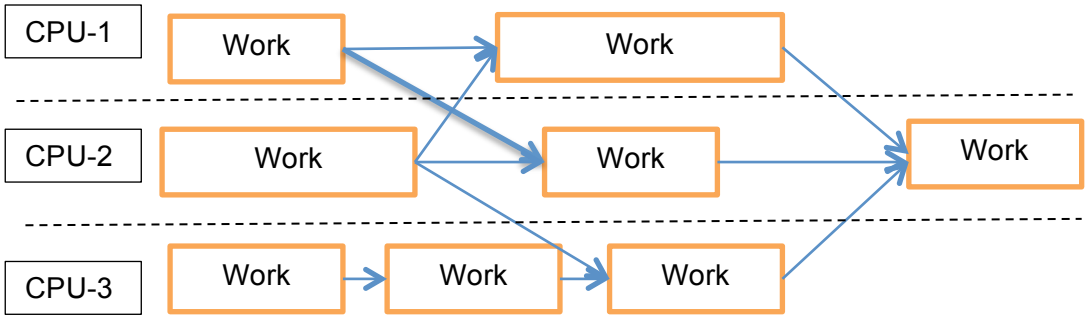

Figure 1 **Example of a DAG representation of a computation**

The DAG representation of an algorithm can also form the basis of a model of the on-node parallelism. The behavior of the memory system will be hard to evaluate in early abstract models as small changes in the cache architecture or the memory layout can have a significant impact on performance. This means that the hardware and software designs will have to be very detailed to allow this to be simulated to any accuracy and can therefore only be considered much later in the design process. Nevertheless a simple DAG representation of the algorithm does allow the degree of on-node concurrency in the hardware design to be compared with the amount of intrinsic parallelism available in the on-node fragment of the DAG. In the early stages of design it seems logical to concentrate on modeling and simulating the inter-node behavior of the system.

5 Simulation

While it might be possible to perform some design evaluations by direct analysis of the model there are many cases where this is too difficult and the model will need to be simulated in a computer. Historically there have been two main use-cases for simulators of HPC systems.

Highly detailed simulators have been used to emulate new machine architectures on existing hardware as part of the final design stages. These simulations typically emulated individual machine instructions running full applications compiled for the target architecture. These simulations run very slowly so there is always a practical limit on how much use can be made of detailed simulations of this type. Currently we are far too early in the design process for exascale systems to be able to build sufficiently detailed models to utilize simulations of this type. In addition a highly detailed simulation of an exascale system would itself be a challenging computational problem, possibly beyond current systems.

The second main use case is as a part of application performance analysis. These kinds of simulators typically work at a much lower level of detail and concentrate on simulating application communications. This makes them more suitable for our current purposes.

5.1 Trace driven simulation

Many HPC performance analysis packages such as Vampir [1] Paraver [2] Tau [3] and Scalasca [4] can generate application performance traces. These performance traces typically contain detailed information about every MPI message sent by the program as well as data from performance counters about the computational requirements of the code. This can then be used to construct a behavioral model of the application that can then be simulated using course-grain simulators such as Dimemas [5] and sst-macro [6] to predict the performance of the application on various different kinds of hardware (real or theoretical). These are equivalent to the DAG based models discusses in Section 4. The performance counter data is used to generate a model of the computations and the MPI message data is used to generate a model of the communications. The main use of this kind of simulation is to gain insight into the behavior of applications. For example the sensitivity of application performance to communication bandwidth and latency can be explored by performing a series of experiments varying the corresponding simulation parameters. This is a very powerful technique for modeling existing application codes as the models are automatically extracted from existing applications. However there are a number of limitations to this technique.

- Many of the important model parameters (such as the number of processors and the problem size) are represented as input parameters of the application being traced so it is only possible to generate a trace corresponding to an exascale problem running on a realistic number of processors by running the same problem on a lesser machine using significantly more MPI tasks than processors.
- Fully functional application codes are required to generate the traces so new algorithms and approaches need to be fully coded before they can be evaluated via simulation.
- The trace data can quickly become very large making it difficult to simulate large systems for long periods of time.

5.2 Library driven simulation.

Library driven simulators attempt to address the large size of the trace data by coupling the application directly to the simulator rather than recording simulated trace data and simulating it at a later time. As the applications being simulated are typically large distributed memory codes, library driven simulators are typically implemented as

distributed Parallel Discrete Event Simulators (PDES) that are linked into the application. The simulator either replaces or is layered on top of the usual MPI library.

Using a distributed simulator also makes it easier to scale to very large simulations of very large numbers of virtual processes. Unfortunately distributed discrete event simulation is a significantly hard problem to implement and frequently very simple network models are used to improve the scaling behavior of the simulation.

Library driven simulation typically only simulates the network and relies on running the full application to provide information on the computational sections and to drive the network simulation. When extrapolating to very large systems this requires very large numbers of MPI tasks to be run on each node of the host system. Some of these simulation environments attempt to reduce the costs associated with this over-subscription by implementing the MPI tasks as threads instead of processes. This can make porting applications to these environments difficult for large and complex codes. Running full application codes might be expected to provide a perfect simulation of the computational portions of the code. In practice, competition for memory resources between MPI tasks on the same node will compromise the accuracy of the results.

5.3 Skeleton driven simulation.

The authors of the sst-macro simulator [6] have developed an approach that addresses many of the problems associated with the other two approaches. When using sst-macro the model can be defined using “skeleton applications”. These are very simplified code fragments intended only to generate the pattern of MPI calls corresponding to the behavior of much more complicated full applications but requiring significantly less resources to execute. Skeleton applications need to duplicate enough of the control logic from the full application to generate the correct communication and computation patterns but instead of actually implementing these operations library calls are used to drive the behavior of the simulation engine.

Skeleton driven simulation is closely related to library driven simulation except that the skeleton is not attempting to perform the actual calculation only to drive a simulation of its performance. Both the communication and computation parts of the application are simulated. This reduces the overall cost compared to adding a simulation overhead on top of running the actual application.

Despite being simpler than full application codes, representative skeleton applications nevertheless require significant effort to develop. The control and communication logic of the skeleton application may need to be very complex. In fact it needs to be of equivalent complexity to the control and communication logic of the target application. This is inevitable if the skeleton is to accurately generate a representative pattern of communications. On the other hand it should be possible to use a very simplified model of the computational parts of the application.

For existing applications it is frequently possible to use conditional compilation to allow a single set of source code to be compiled as either the full application or a representative skeleton. Even though this does not allow lightweight model development it does allow much larger simulations to be attempted than would be practical than using a trace file to drive the simulator. As an added advantage the performance characteristics of additional functionality being added to the code can be first explored by adding it in skeleton form. If the simulation results look promising this skeleton can then be extended to support the computational components needed to implement the new functionality.

Where a set of applications use very similar common communication patterns (for example boundary communication) then it might be possible to develop a single parameterized skeleton application that can represent any of the applications from the set.

5.4 Simulation tools

Simulation tools are highly complex pieces of software that take significant amounts of resources to develop. It therefore seems prudent to use existing simulation software where possible. This section is a survey of some of the existing software packages capable of simulating HPC applications.

5.4.1 DIMEMAS

Dimemas [5] is a trace driven simulator developed by the Barcelona Supercomputer Center and intended for application analysis. It can operate in a trace-to-trace mode where the output of the simulation is a perturbed trace file that can then be used as input to trace visualization tools. It is primarily intended to work with the BSC Paraver trace visualization tool but these traces can be converted to the Open Trace Format (OTF) for use with other tools such as Vampir.

The default network model for Dimemas is a fairly abstract linear model. Point to point communication is modeled using parameters such as latency, bandwidth and link contention. However contention between messages is modeled using a simple "shared-bus" model. It is also possible to add explicit point-to-point communication links between nodes that allows a more accurate representation of communication networks.

Separate (very simple) models are used to model collective communication giving a choice of constant/logarithmic/linear scaling of the fan-in/fan-out stages of each collective.

It is also possible to use the Dimemas replay engine to drive much more sophisticated network simulators such as the IBM Venus network simulator [7]. This simulator is in turn based on the commercial Omnest [8] network simulator.

Dimemas is promoted as a performance analysis tool rather than a research simulator so it comes with a GUI interface. Unfortunately this GUI only provides a mechanism to edit the complicated simulator configuration file so the tool remains difficult to use.

The source codes for the Paraver/Dimemas tools are available under the LGPL. This package is a mature tool but is still being maintained with the most recent update July 2012.

5.4.2 BigSiM

BigSIM [9] is a family of simulator/emulator packages developed by the Department of Computer Science University of Illinois. It is explicitly targeting the development of future extreme scale applications.

This code does not provide support for general MPI programs but focuses on emulation and simulation of applications written using the Charm++ and AMPI libraries (also developed at UIUC). However within this limitation it seems to support a wide variety of features.

The BigSim Emulator allows large Charm++ or AMPI program runs to be emulated on much smaller machines, this allows debugging and testing and generation of tracefiles for performance simulation.

The BigSim Simulator is a parallel trace-driven discrete event simulator. It supports a variety of network models ranging from very simple latency models to complex models of the network fabric.

The BigNetSim package provides detailed network simulation models.

5.4.3 μπ/xSIM

These are similar but independent library driven simulator packages developed at Oak Ridge national laboratories. Though the xSIM simulator has been described in the literature [10] only the μπ [10] package appears to be available for download.

The μπ package uses a very simple latency bandwidth network model. The xSIM package seems to support a slightly more complex model where message latency is

adjusted based on the simulated network topology. xSIM also maps MPI processes onto threads to allow a higher number of MPI processes to be simulated. A clever linker based mechanism is used to prevent this implementation from breaking application codes that use global variables. It is not clear if either package is currently under active development.

5.4.4 SST/SST-macro

These are a family of simulation tools developed by the Sandia national Laboratories. SST stands for Structural Simulation Toolkit. The SST is a toolkit for performing simulation at various levels of detail up to full simulation of HPC systems including accurate models of both CPUs and networks. The toolkit is capable of integrating a number of existing simulators such as the DRAMSim2 package from the University of Maryland and the standard GeM5 processor simulator. The general approach taken by the toolkit is apparently to integrate existing simulator packages as components tied together as a distributed MPI application. It is explicitly targeting the design of Exascale systems and the simulations of power and thermal issues are key parts of the design.

SST/macro is the macro level simulator from the SST. It can be compiled and run independently from the rest of the SST and is a stand-alone simulation tool in its own right. SST/macro supports both skeleton driven and trace driven simulation. Trace driven simulation is supported by using a special skeleton application that reads and replays the events from a trace-file. SST/macro uses its own trace file format DUMPI and the simulator can be configured to output a modified trace as a result of the simulation. Tools are being developed to convert the DUMPI format into Open Trace Format (OTF) files that are readable by the majority of trace visualization tools such as Vampir and Paraver.

SST/macro has a relatively detailed network model as the network topology is explicitly modeled within the simulator. Within this framework a variety of different network models are available. Even the simplest of these models are quite detailed with messages modeled as data flows through the network with contention being handled by apportioning the available bandwidth on each link between the active data flows.

SST/macro is a research simulator and as such is fairly complex to use. This is mainly because the relatively detailed simulations require a complex machine description file to define the machine being simulated. However tools exist to automatically generate machine description files by querying the configuration of existing Cray XT systems. Machine description files corresponding to much larger systems can also be produced with a little more effort. Once such a configuration exists it is fairly easy to run additional experiments on the same simulated machine.

The skeleton driven simulation approach provides a good mechanism for exploring design choices without the overhead of developing a full application code. However this is a more complex mode of operation than trace driven simulation.

Unfortunately the trace driven mode of operation currently has some limitations. The trace generation library successfully captures all of the MPI calls in the application. However some of the more obscure auxiliary MPI routines such as MPI group operations are not currently handled by the skeleton application that replays the traces. This prevents trace based simulation of any application that uses the missing functions.

SST-macro is under active development by with frequent updates to the source code repository by multiple authors. While this is positive for the long term future of the package it does mean that the software is not easy to use and is lacking documentation in some areas. The latest release was made July 2012.

6 FFT Case study

As a case study we investigate modeling and simulation of an example problem. We selected the Fast Fourier Transform (FFT) as an example problem as an algorithm, important for many applications, that is known to be difficult to scale but nevertheless simple enough to be considered in detail.

The DAG representation of the FFT algorithm is the "butterfly" pattern (see Figure 2 DAG representation of a 2^4 FFT algorithm). As can clearly be seen from the DAG representation a 2^n FFT has a computational complexity of O(n log(n)). The algorithm has a potential parallelism of O(n) with good load-balance but is also a non-local algorithm requiring a high degree of data movement.

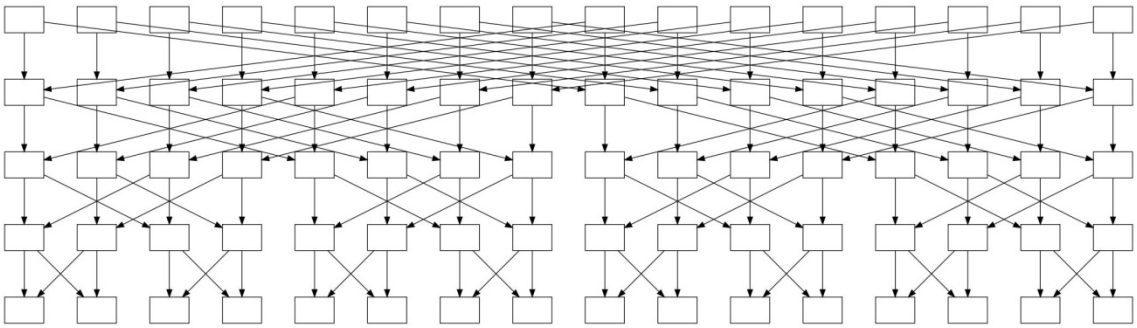

Figure 2 DAG representation of a 2^4 FFT algorithm

Even this simple model is sufficient for us to derive some analytic predictions about the use of the FFT algorithm on exascale systems. In section 4 we noted that current exascale proposals assume a level of parallelism of $O(10^9)$. Even allowing for a reasonable amount of instruction level parallelism within the computational sections this still means that FFT calculations smaller than say 512^3 will not expose enough parallelism to make full use of an exascale system. It is therefore important when designing exascale applications that use smaller FFTs to identify additional parallelism outside of the FFT itself that can be used to provide the additional required parallelism. This can be achieved by evaluating multiple independent FFTs in parallel or by identifying some other part of the application that can be executed simultaneously with the FFT.

In addition the number of data words being moved in each stage of the computation is of a similar magnitude to the number of floating point operations. For this algorithm the ability of the target platform to move data is at least as important as its ability to perform floating point operations. To a large extent optimizing the FFT algorithm consists of re-writing the DAG to minimize the amount of long distance communication, keeping as many levels of the algorithm as possible either within cache or within a node. However it is impossible to eliminate the long distance communication entirely. The best of the current "Petascale" systems have a floating point performance measured in Peta-flop/s but a bisection bandwidth that is only measured in Tera-bytes/s so communication costs dominate FFT performance on these systems.

The most common use of Fourier transforms in HPC applications are as multi-dimensional FFTs. The DAG representation of a multi-dimensional FFT is actually the same as that of a single large FFT with the same number of input points, the two computations only differ in terms of the phase factors applied in the computational steps. This tells us that any implementation strategy for a multi-dimensional FFT can be converted into a strategy for implementing large single dimension FFTs and vice-versa.

The most common implementation strategy for parallel multi-dimensional FFTs is to perform each dimension of the FFT in turn using node-local FFT implementations with a communication phase between each stage to transpose the data. In terms of the DAG representation this is a re-write of the graph to reduce the number of

communications that cross node boundaries. This is done by introducing an additional data redistribution step into the graph (see Figure 3 DAG representation of a (2^2 x 2^2) 2D FFT).

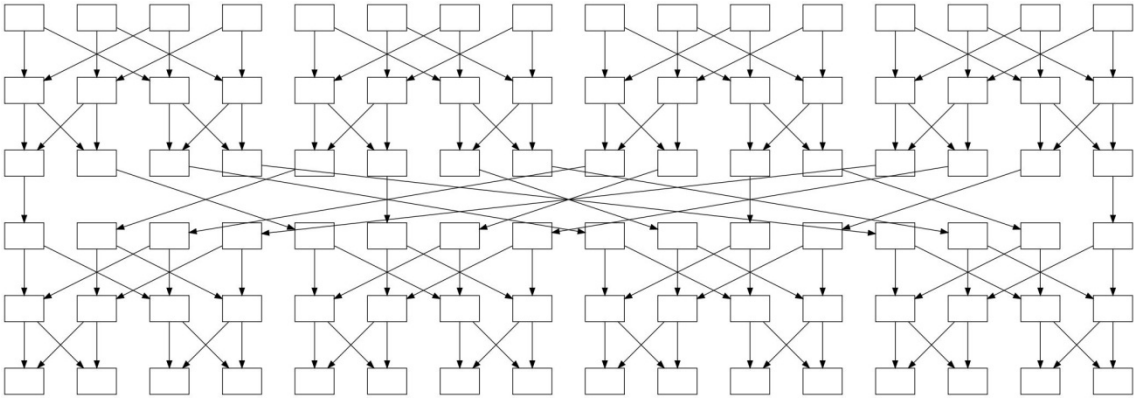

Figure 3 DAG representation of a (2^2 x 2^2) 2D FFT

Where the data can be equally distributed across processors this communication step is equivalent to a MPI **MPI_Alltoall** collective operation. If the data cannot be equally distributed this corresponds to an **MPI_Alltoallv** operation. When using only a single communication step this approach limits the available parallelism to the width of one dimension of the FFT. However additional parallelism can be introduced by decomposing the data in more than one of its dimensions or by using the equivalence between the 1D and 2D DAG representations to split up the 1D operations. This produces an overall communication pattern that consists of a series of all-to-all collective communications of size $\mathbf{p_i}$ where $N_{nodes} = \prod_i p_i$. As the computation performed by each node of the DAG is quite small the time to execute these communication steps will dominate performance at large processor counts. Because optimized single node FFT libraries are highly efficient the same can be true even for the course grain data decompositions commonly used in current HPC applications.

Where we have an analytic model of the performance of an all-to-all collective operation this is sufficient to build an analytic model of parallel FFT performance. While analytic models are an important part of the design evaluation process and should not be needlessly neglected in favor of simulation the performance of real distributed FFT operations have frequently been observed to be limited by data contention in the network. Data contention is very difficult to address analytically so we need to resort to simulation to investigate this.

We chose to investigate this problem using a simple FFT benchmark written using the MPI communication library. The benchmark was configured to perform a 256^3 FFT and to attempt all possible 1 and 2 dimensional decompositions that result in a balanced decomposition and performs 10 cycles of forward/backward FFTs for each of the decompositions.

It quickly became apparent that many of the available simulation technologies were not suitable for this task.

The BigSIM package is targeted at the Charm++ communication package not MPI.

The μπ package is only capable of simulating point to point communication and currently has no support for the MPI_Alltoall collective operation used by the benchmark. Though it would be possible to re-write the benchmark to use point to point communication the network model is very simple (having no concept of network topology) and would not be able to capture network contention.

The basic Dimemas simulator does support MPI_Alltoall but only through a simple analytic model for collective communications so simulation via this tool would provide no additional insights beyond an analytic analysis. The Dimemas/Venus simulation

environment would not have been much more capable but this was not available for evaluation.

The SSTmacro simulator proved to be a good fit for this problem. It uses a fairly detailed network model that could be capable of modeling network contention. It also provides good support for MPI_Alltoall

A trace of the benchmark was generated using 128 processor nodes of the Hector XE6 system. S single MPI-process/thread was used per node. In this configuration there are 8 possible 1 and 2 dimensional decompositions that result in a balanced decomposition, all of which were run within the benchmark.

This trace was then re-simulated using sst-macro configured to simulate an approximation of the Hector system. This simulation was then re-run multiple times varying the **network_bandwidth_link** and **packetswitch_bandwidth_n2r** parameters in the network model. The results of these simulations are shown in Figure 4 Simulated performance of FFT benchmark.

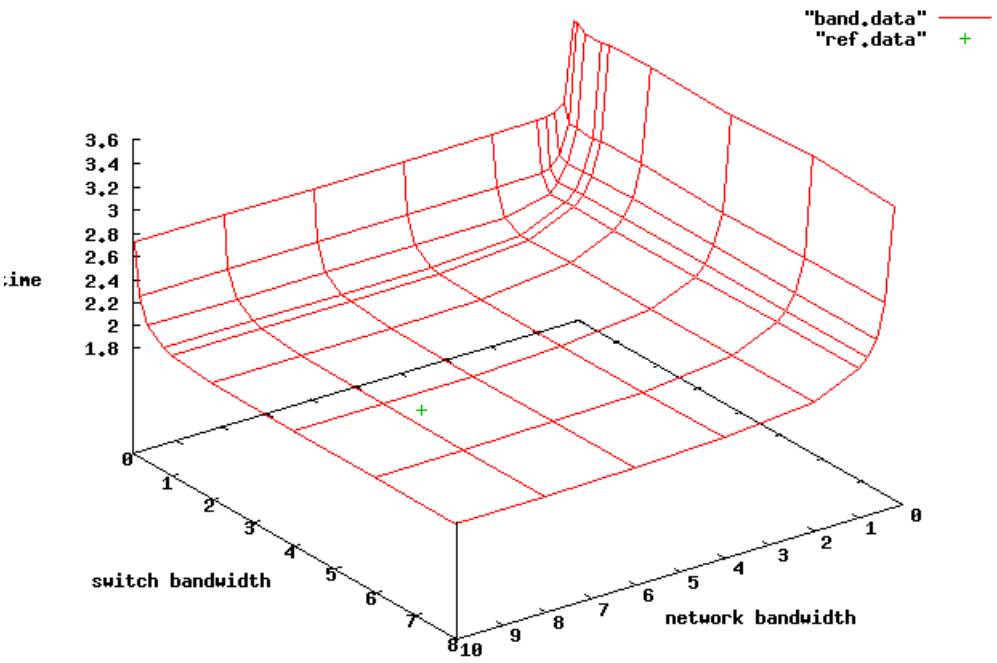

Figure 4 Simulated performance of FFT benchmark

The “ref.data” point shows the simulation corresponding to the network parameters of the original XE6 system. The simulation shows that for this example, when only a single core per node is used this point is within the region that is not limited by either of the network parameters.

6.1 Conclusions from the case-study

In the case of the FFT the DAG representation of the algorithm does form a useful model of the algorithm that can provide insights into the fundamental limits on possible parallel implementations. Even very simple and regular operations like the FFT can have quite complex DAG structures with many possible mappings onto compute nodes. The only practical way of generating these is via a program. This can naturally be mapped onto the skeleton driven approach to simulation.

All of the available simulation packages have limitations in terms of the operations they support and the sophistication of their network models.

The sst-macro simulator seems able to capture useful details about the interaction between the application and the network. This is primarily intended for skeleton driven

simulation and the support for trace driven simulation is still somewhat limited. Though the network model is quite detailed it is quite difficult to extract useful statistics from trace driven simulations in the current version of the code. Though the **dumpi** trace library successfully generates traces for all MPI calls not all of these are fully supported by the trace reader.

7 Conclusions

As part of the development process of exascale systems it is important that we develop behavioral models of our designs in order to be able to evaluate their effectiveness. Models of the software components of the systems are at least as important as models of the hardware.

These models have to evolve together with the designs, becoming more detailed and more complex as the designs become more detailed and more complex.

As the designs/models become more complex we need to simulate the models in order to evaluate the designs.

The energy consumption of exascale systems is at least as important as their performance so our models will need to capture both aspects of their behavior.

Available parallelism and intrinsic communication requirements are two of the key aspects of an algorithm that we should try to capture in our models. Modeling an algorithm using directed acyclic graphs seems to be a useful and informative way of capturing this information.

The skeleton driven approach to simulation appears to provide a route to performing simulations of exascale systems at relatively modest cost in terms of development time and simulations resources. It also seems to be a practical way of progressing from a DAG based model to a model that is capable of being simulated.

8 References

[1] vampir, “vampir,” 2012. [Online]. Available: http://www.vampir.eu/. [Accessed 12 07 2012].

[2] BSC, “Paraver,” 2012. [Online]. Available: http://www.bsc.es/computer-sciences/performance-tools/paraver/. [Accessed 12 07 2012].

[3] University of Oregon, “Tau,” [Online]. Available: http://www.cs.uoregon.edu/Research/tau/home.php. [Accessed 12 07 2012].

[4] fz-juelich, “scalasca,” [Online]. Available: http://www.scalasca.org/start.html. [Accessed 12 07 2012].

[5] BSC, “Dimemas,” [Online]. Available: http://www.bsc.es/computer-sciences/performance-tools/dimemas. [Accessed 12 07 2012].

[6] Sandia National Laboratories, “SST macro,” [Online]. Available: http://sst.sandia.gov/about_sstmacro.html. [Accessed 12 07 2012].

[7] “Venus - Interconnection Network Simulation,” IBM, [Online]. Available: http://researcher.watson.ibm.com/researcher/view_project.php?id=1071. [Accessed 23 August 2012].

[8] “Omnest,” [Online]. Available: http://www.omnest.com/. [Accessed 23 August 2012].

[9] UIUC, “Runtime Systems and Tools: BigSim - Simulating PetaFLOPS Supercomputers,” [Online]. Available: http://charm.cs.uiuc.edu/research/bigsim. [Accessed 14 09 2012].

[10] S. Bohm, “xSim: The extreme-scale simulator,” in *High Performance Computing and Simulation*, 2011.

[11] K. Perumalla, “µπ,” 08 06 2009. [Online]. Available: http://www.ornl.gov/~2ip/mupi/index.htm. [Accessed 14 09 2012].