

SANDIA REPORT

SAND2016-6049
Unlimited Release
Printed

Abstract Machine Models and Proxy Architectures for Exascale Computing

Version 2.0

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.

Approved for public release; further dissemination unlimited.



Sandia National Laboratories

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

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

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>



Abstract Machine Models and Proxy Architectures for Exascale Computing

Version 2.0

J.A. Ang¹, R.F. Barrett¹, R.E. Benner¹, D. Burke², C. Chan², J. Cook¹, C.S. Daley², D. Donofrio², S.D. Hammond¹, K.S. Hemmert¹, R.J. Hoekstra¹, K. Ibrahim², S.M. Kelly¹, H. Le, V.J. Leung¹, G. Michelogiannakis², D.R. Resnick¹, A.F. Rodrigues¹, J. Shalf², D. Stark, D. Unat, N.J. Wright², G.R. Voskuilen¹ ¹

¹Sandia National Laboratories, P.O. Box 5800, Albuquerque, New Mexico
87185-MS 1319

²Lawrence Berkeley National Laboratory, Berkeley, California

Abstract

To achieve exascale computing, fundamental hardware architectures must change. The most significant consequence of this assertion is the impact on the scientific and engineering applications that run on current high performance computing (HPC) systems, many of which codify years of scientific domain knowledge and refinements for contemporary computer systems. In order to adapt to exascale architectures, developers must be able to reason about new hardware and determine what programming models and algorithms will provide the best blend of performance and energy efficiency into the future. While many details of the exascale architectures are undefined, an *abstract machine model* is designed to allow application developers to focus on the aspects of the machine that are important or relevant to performance and code structure. These models are intended as communication aids between application developers and hardware architects during the co-design process. We use the term *proxy architecture* to describe a parameterized version of an abstract machine model, with the parameters added to elucidate potential speeds and capacities of key hardware components. These more detailed architectural models are formulated to enable

discussion between the developers of analytic models and simulators and computer hardware architects. They allow for application performance analysis and hardware optimization opportunities. In this report our goal is to provide the application development community with a set of models that can help software developers prepare for exascale. In addition, through the use of proxy architectures, we can enable a more concrete exploration of how well new and evolving application codes map onto future architectures. This second version of the document addresses system scale considerations and provides a system-level abstract machine model with proxy architecture information.

Acknowledgment

Support for this work was provided by the Advanced Scientific Computing Research (ASCR) program and funded by the Director, Office of Science, of the U.S. Department of Energy. Lawrence Berkeley National Laboratory operates under Contract No. DE-AC02-05CH11231. 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.

Thanks to Prof. Bruce Jacob of the University of Maryland for input on NVRAM trends and to Mike Levenhagen for his participation in some of our discussions.

Contents

1	Introduction	13
2	Programming Considerations	17
	Data Movement/Coherence Model	17
	Hardware Performance Heterogeneity	17
	Increased Parallelism	18
	Distributed Memory Programming Models	19
	Point-to-point Transfer Mechanisms	20
	Active Messaging Mechanisms	21
	Collective Transfer Mechanisms	21
	Hardware Support for Transfer Mechanisms	21
	Communication Efficiency	22
3	Abstract Machine Models	24
	Overarching Abstract Machine Model	24
	Processor	24
	On-Chip/Package Memory	25
	Cache Locality/Topology	26
	Integrated Components	26
	Hardware Performance Heterogeneity	27
	Abstract Model Instantiations	27
	Homogeneous Many-core Processor Model	28
	Multicore CPU with Discrete Accelerators Model	29
	Integrated CPU and Accelerators Model	29

Heterogeneous Multicore Model	30
Abstract Models for Concept Exascale Architectures	31
Performance-Flexible Multicore-Accelerator-Memory Model	31
4 Memory System	34
Memory Drivers	34
Future Memory Abstractions	36
Physical Address Partitioned Memory System	37
Scratchpad Memory	37
Multi-Level Cached Memory System	38
3-D Stacked Memory Systems, Processing in Memory (PIM), and Processing Near Memory (PNM)	39
5 Proxy Architectures	40
Abstract Model of Future Computing Nodes	40
Detailed Processing Node models	41
Lightweight Cores and Processing Communication Runtime	42
Reference Proxy Architecture Instantiations	43
Homogeneous Manycore Model: Intel Sandy Bridge	43
Multicore CPU + Discrete Accelerators Model: Sandy Bridge with Discrete NVIDIA GPU Accelerators	44
Integrated CPU + Accelerators Model: AMD Fusion APU Llano	45
Proxy Parameters	45
Processor	46
6 System Scale Considerations	47
System Analysis of Advanced Workflows	47
Interconnect Model	47

Communication Model	48
Communication Cost Model	49
Interconnect Physical Characterization	50
Routing Transfer in Direct Interconnect	55
Endpoint Communication Resources	55
Interconnect Technologies and Impact on Application Development	55
7 System Abstract Machine Model and Proxy Architectures	57
Partition Model	58
System AMM Components	58
The Processor and Memory AMM System Components	58
The Interconnect Component	59
Nodes, Sub-Cabinets, and Cabinets	59
System Proxy Architectures	60
Processor	60
Memory	61
Node Architecture	61
System Network	62
System Organization	62
8 Conclusion	64
References	65

List of Figures

2.1	Distributed Memory Programming Models	19
2.2	Typical Transfer Efficiency Curve	22
3.1	Abstract Machine Model of an Exascale Node Architecture	25
3.2	Homogeneous Manycore Model	28
3.3	Multicore CPU + Discrete Accelerators Model (Acc: Accelerator)	29
3.4	Integrated CPU + Accelerators Model (Acc: Accelerator)	30
3.5	Heterogeneous Multicore Model	30
3.6	Homogeneous Multicore-Accelerator-Memory Model (Mem: Memory, Acc: Accelerator, Mov: Data Movement Engine)	31
4.1	Per-Node Bandwidth and Capacity of Various Memory Technologies	35
4.2	Memory Subsystem Layouts	37
5.1	Example Node Architecture using Processor AMM	41
5.2	Reference proxy architecture instantiation: Intel Sandy Bridge	43
5.3	Reference proxy architecture instantiation: Multicore CPU with Discrete GPU Accelerators	44
5.4	Reference proxy architecture instantiation: AMD APU Llano	45
6.1	Estimated Data Transfer Time, LogGP model	49
6.2	Direct interconnection Networks (Tree and 4D Hypercube)	51
6.3	Indirect Interconnect (butterfly) using Multiple Stage Switches	51
6.4	Dragonfly Interconnect	54
7.1	System AMM	63

List of Tables

4.1	Approximate Bandwidths and Capacities of Memory Subsystem	39
6.1	Current and Projected Communication Model Parameters	50
6.2	Link Technologies and Performance Characteristics	53
6.3	Switch Characteristics	54
6.4	Direct Network Characterization	54

1. Introduction

In this report we present an alternative view of industry’s exascale system hardware architectures. Instead of providing highly detailed models of each potential architecture, as may be presented by any individual vendor, we propose initially to utilize simplified or high-level, abstract models of machine components that allow an application developer to first reason about data structure placement in the memory system and the location where computational kernels run. Then using a system-scale abstract model that combines the machine component models into one that represents a system-level machine, programmers can reason about placement of the application across the entire system. These component abstract machine models (AMMs) in conjunction with the system abstract machine model (sAMM) will provide software developers with sufficient detail of architectural and system features so they may begin tailoring their codes for these new high performance machines and avoid pitfalls when creating new codes or porting existing codes to exascale machines. While more accurate models will permit greater optimization to the specific hardware, it is our view that a more general approach will address the more pressing issue of initial application porting and algorithm re-development that will be required for future computing systems. Once initial ports and algorithms have been formulated, further refinements on the models in this document can be used as a vehicle to optimize the application. These models offer the following benefits to the research community:

Simplified model Abstract models focus on the important high-level hardware components, which in turn affect code structure and algorithm performance – implementation specific details are omitted.

Enable community engagement Abstract machines are an important means of communicating to application developers about the nature of future computing systems so they can reason about how to restructure their codes and algorithms for those machines.

Enable design space exploration The AMM is the formalization of a particular parametric model for a class of machines that expresses the design space and what we believe is important in that space. Additionally, the purpose of each of the presented models is to abstractly represent many vendor specific hardware solutions, allowing the application developer to target multiple instantiations of the architecture with a single, high-level logical view of the machine.

Enable programming systems development A high-level representation of the machine also enables design of automated methods (runtime or compile time) to efficiently map an algorithm onto the underlying machine architecture.

In order to sufficiently reason about application and algorithm development on future exascale-class compute nodes, a suitable AMM [18] is required so that algorithms can be developed independent of specific hardware parameters. It is useful to think of the AMM as a way to simplify the myriad complex choices required to target a real machine and as a model in which application developers can frame their algorithms [19]. The AMM represents the subset of machine attributes

that will be important for code performance, enabling one to reason about power/performance trade-offs for different algorithm and execution model choices. We want an abstract model of the underlying hardware to be as simple as possible to focus on the durable cross-cutting abstractions that are apparent across machines of the same generation, and to represent long-term trends for future generations of computing systems. While there exist many potential machine attributes that could be included in our models, we instead take one of three actions to concentrate our models into more concise units:

- **Ignore it.** If ignoring the design choice has no significant consequences for the consumption of power or the provision of computational performance we choose to eliminate the feature from the model. We include architecture-specific instantiations of hardware features, such as specific single-instruction, multiple-data (SIMD)-vector widths, in this category since it is the presence of the functional unit that is important for reasoning with the model – not the specific capabilities of the functional unit itself. Such details are provided in the *proxy architecture* annotation of the model.
- **Abstract it.** If the specific details of the hardware design choice are well enough understood to provide an automated mechanism to optimize a layout or schedule, an abstracted form of the choice is made available (for example, register allocation has been successfully virtualized by modern compilers).
- **Expose it.** If there is no clear mechanism to automate decisions but there is a compelling need to include a hardware choice, we explicitly expose it in our abstract machine model deferring the decision of how best to utilize the hardware to the application programmer. For instance, the inclusion of multiple types of memory will require specific data structure placement by the programmer, which in turn implies a need for the programming model to also support data placement.

To obtain a more concrete instantiation of an AMM, we add detail to the abstraction by giving parameters to model components and assigning values to these parameters. For example, an abstract memory model simply defines the organization of the various components in a hierarchy (e.g., three levels with the first backed by the second, the second backed by the third). Each of these components (i.e., each level of the hierarchy) is characterized by parameters such as capacity and bandwidth that given a concrete value (e.g., capacity = 2GB, BW = 50GB/sec) defines the technology (e.g., HBM, DRAM, NVRAM) and becomes what we define as a proxy architecture. A fully parameterized abstract machine model with associated parameter values is a proxy architecture that defines a specific component implementation that can be subsequently simulated. Further analysis and optimization of the initial port of algorithms and applications to an AMM can be accomplished through the simulated proxy architecture.

In Chapter 2 we discuss issues to be considered from a programming perspective when studying the viability of abstract machine models. We provide some basic background information on how various hardware features will impact programming and ultimately application performance. As the HPC community drives toward achieving exascale, new metrics of energy efficiency, increased

concurrency, programmability, resilience, and data locality will play an increasing role in determining which hardware solutions are feasible and practical to utilize for production-class *in-silico* scientific research. From the application programmers' perspective, programming to each potential exascale hardware solution presents an unwelcome situation in which multiple rewrites of the application source may be required - in some cases demanding radical shifts in data structure layout or the re-design of key computational kernels. Put simply, many application development teams and supercomputing centers will be unable to afford the luxury of frequent application rewrites either due to the sheer cost of such an endeavor or the number and availability of programmers needed to undertake the activity.

In the face of a large number of hardware design constraints, industry has proposed solutions that cover a variety of possibilities. These solutions range from systems optimized for nodes comprising many ultra-low power processor cores executing at vast scale to achieve high aggregate performance throughput to large, powerful processor sockets that demand smaller scales but provide performance at much higher per-node power consumption. Each of these designs blends a set of novel technologies to address the challenges laid out. To assist developers in reasoning about these disparate ideas and solutions, we will present an overarching processor abstract model designed to capture many of the proposed ideas into a single, unified view in Chapter 3. Then we present a family of abstracted models that reflect the range of more specific architectural directions being pursued by contemporary CPU designers. Each of these processor models are presented in sufficient detail to support many uses, from application developers becoming familiar with the initial porting of their applications to hardware designers exploring the potential capabilities of future computing devices.

In Chapter 4, we discuss memory architectures. An example of how an abstract model may be applied to assist users in their transition from current machines can be seen in the memory systems of future machines. It is likely that due to a rising number of cores per socket the memory hierarchy will become further subdivided to maintain a reasonable amount of memory available per core. This subdivision will make trade-offs of capacity versus bandwidth at different levels, forcing programmers to manage vertical locality more explicitly than currently required. In addition, maintaining cache coherence constitutes a large percentage of on-chip data traffic. Future designs are leaning toward maintaining cache coherence, however the cost will be distance (or level in hierarchy) dependent.

Chapter 5 presents node-level abstract machine models and parameterized instantiations, or *proxy architectures*, of some abstract machine models that combine the CPU and memory models outlined in this report to represent simple compute nodes. Proxy architectures, both node-level and system-level, are an especially useful communication vehicle between hardware architects and performance analysts. Models can be developed based on the AMM descriptions and the parameter space identified in the associated proxy architecture. When a sufficient range of parameters is applied, we foresee that the models may be shared openly among users, academics, and researchers. A specific parameter set that closely relates to a point design will likely be proprietary and, therefore, only shared within the appropriate disclosure constraints. Simulations using these models will explore the ever-expanding definition of performance. In addition to the Holy Grail of minimizing application run times, power usage, and data movement, maximizing resiliency and programmer

productivity are also parts of the equation that must be considered in identifying usable exascale systems.

In Chapter 6 we discuss characteristics of application communication that must be considered in developing a system model. We first discuss the impacts of the choice of the node model on system characteristics such as the runtime and communication processing. The development of an interconnect model requires not only the physical definition of the interconnect itself, but other characteristics such as communication cost, routing, endpoint resources, and data transfer mechanisms. These are presented and discussed in the context of a system-scale interconnect model. We conclude this chapter by presenting an I/O and storage model that can be used in defining a system abstract machine model.

Chapter 7 presents a generalized system abstract machine model and system-level parameterized instantiations of some combinations of the node, interconnect, and I/O and storage models to form system proxy architectures. Point designs of existing and future Exascale systems comprise the proxy architecture space. The parameters for each of the respective component models provide enough detail to inform accurate simulations of these systems that can be used for more targeted application optimization.

Finally, we conclude our report in Chapter 8 and add a discussion of future work and important codesign and programmatic interfaces for other research and development areas in the overarching Department of Energy (DOE) exascale program.

2. Programming Considerations

Emerging architectures are bringing with them a shift in constraints that will require careful consideration for development of future exascale-class applications, particularly for those demanding an evolutionary approach to porting [4]. In the future we expect that new optimization goals will become commonplace, specifically that application developers will target the minimizing of data movement and the maximization of computational intensity for each piece of data loaded from memory, rather than a focusing on increasing the raw compute performance (FLOP/s) used by each processor. The optimization of data movement is becoming more complex as future architectures may have greater levels of Non-Uniform Memory Access (NUMA) and we anticipate an explosion of on-chip parallelism. These architectural shifts must be accompanied by changes in programming models that are more adept at preserving data locality, minimizing data movement, and expressing massive parallelism.

Data Movement/Coherence Model

Programmers have relied on a large shared cache to virtualize data movement. As the number of cores per socket increase programmers will be faced with the need to explicitly manage data movement. In the best case, regional coherence domains will automatically manage memory between a subset of cores. Between these domains there may be a relaxed consistency model, but the burden will still fall on developers to efficiently and manually share data on-chip between these domains.

Non-uniform memory access issues are already prevalent in today's machines. With the high core counts (in the 100s or 1000s for GPUs), the issue will be more detrimental to performance because programmers can no longer assume execution units or the various memory components are equidistant. The importance of locality in these new architectures that utilize explicitly managed memory systems will drive the development of a more data-centric programming model and tools to allow programmers to more naturally express the data layout of their programs in memory.

Moreover, configuring local memory as cache and scratchpad memory will become an important tuning parameter. Recent CPU and GPU architectures allow a split of the on-chip storage between hardware-managed and software-managed memory. We expect that future systems will allow for even more flexible configuration in the split between scratchpad and cache.

Hardware Performance Heterogeneity

Current parallel programming paradigms such as the bulk-synchronous parallel (BSP) [28] model implicitly assume that the hardware is homogeneous in performance. During each phase

of computation, each worker thread is assigned an equal amount of work, after which they wait in a barrier for all other worker threads to complete. If all threads complete their work at the same time, this computational paradigm is extremely efficient. On the other hand, if some threads fail to complete their portion of the work, then large amounts of time could be wasted by many threads waiting at the barrier. As on-chip parallelism increases, the achieved performance of the cores on a chip will become less and less homogeneous, which will require adaptation by the programmer, programming model, domain decomposition and partitioning tools, and/or system runtime to maintain a high level of performance.

One source of increasing heterogeneity is the use of multiple types of cores on a chip such as thin and fat cores. As described in the section titled Hardware Performance Heterogeneity, largely parallel tasks will run more efficiently on throughput-optimized *thin cores*, while mostly serial tasks will run more efficiently on latency-optimized *fat cores*. The programming model and runtime must be aware of this distinction and help the programmer utilize the right set of resources for the particular set of tasks at hand so the hardware may be utilized in an efficient manner. The compiler may utilize the frequency scaling information to make static scheduling decisions, or the runtime may adapt based on how quickly it observes task imbalances run on the different cores.

Another source of performance heterogeneity is due to imbalanced workloads, which may create thermal hotspots on the chip that result in frequency throttling. Such throttling will temporarily impact the execution rate of subsets of cores on a chip. Furthermore, if exascale machines are subject to increased rates of hardware faults, then the associated mitigation techniques such as error correction or recovery could cause large delays to subsets of the threads or processes in an application. Network-on-chip (NoC) congestion resulting in starvation to some cores is also a source of performance heterogeneity.

In each of these cases, a strict BSP program formulation will result in all of the worker threads waiting for the affected threads to complete before moving on to the next phase of computation or communication. Future machines with higher degrees of performance heterogeneity are therefore likely to rely on runtime systems to dynamically adapt to changes in hardware performance or reliability. Current research strategies for programming to accommodate these performance variations include extending existing languages and parallel APIs, such as C++14, future C++ standards-based language level parallelism, and traditional runtimes including OpenMP, and developing alternative languages and task-parallel runtimes, such as Chapel [8], Legion [5], HPX [15], UPC++ [32], and the Open Community Runtime (OCR) [23]. For each approach, the solution will need to provide an efficient mapping with a low burden to the application programmer.

Increased Parallelism

With increasing core counts, more parallelism will need to be exposed by the applications to keep all of the cores on the chip busy. This could prove challenging if application developers have structured their algorithms in a way that limits the level of concurrency expressed to the programming model.

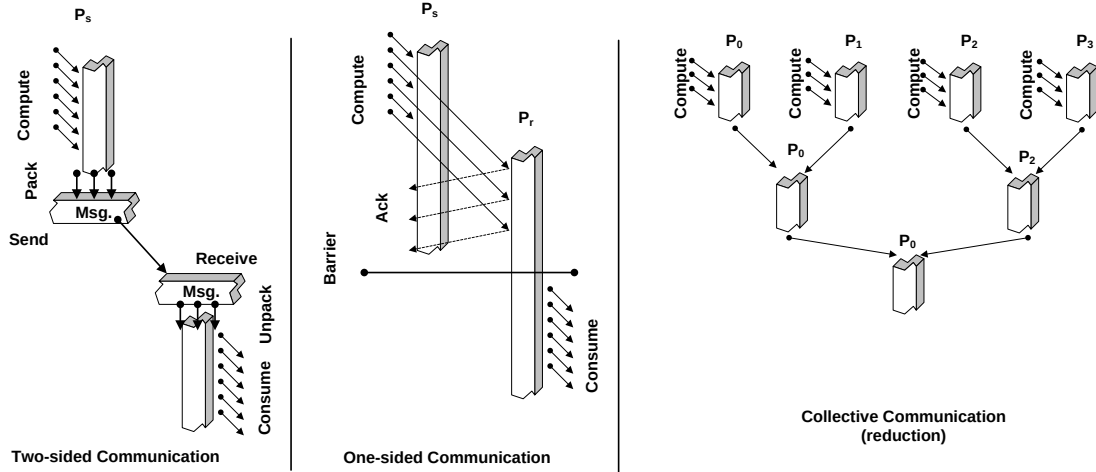


Figure 2.1. Distributed Memory Programming Models

There are four broad sources of parallelism: (1) Instruction-level parallelism between independent instructions, (2) Vectorization of an instruction over multiple data elements, (3) Thread-level parallelism between independent execution contexts/threads, and finally (4) Domain decomposition-level parallelism, which is typical of scientific applications designed to run over massively parallel nodes.

Increased concurrency will need to be exposed in the software stack to utilize large numbers of functional units and to hide higher latencies through hyper-threading. There is potential for future programming languages, compilers, and runtimes to help developers expose greater parallelism through data-centric programming models that allow automatic task decomposition and pipelining. These same systems that help reason about task and data dependencies for scheduling could also be used to manage data movement across the chip to increase access locality for energy and performance benefits described earlier.

Distributed Memory Programming Models

Distributed memory programming models provide mechanisms for data transfers and primitives for synchronization (satisfying dependencies). Two-sided models (send/receive) bundle the transfer mechanism with synchronizations, while one-sided models use global view of the memory to decouple transfers from synchronizations. Collective communication involves multiple processes exchanging data and possibly processing data while being in transit.

Programming models in distributed environments provide mechanisms for satisfying data dependencies between producers and consumers of data. Data transfers are the first of a two-step process to create such a dependency relation. The transfer mechanism can be characterized by the type of the destination and the number of participants in the communication transaction. The

type of the target could be a peer rank (as in two-sided paradigms), or a memory location in the local or a remote node (as in global address space (GAS) models). The initiator could be a single rank, typically in one-sided or two-sided mechanisms, or multiple ranks in collective communication. The second step in establishing a dependency relation involves a synchronization that the data is ready for consumption. Programming models provide different mechanisms to guarantee such ready notification. In two-sided mechanisms (based on send/receive) the synchronization is implicit in the matching of the receive. In one-sided transfers, the notification is decoupled from the transfer and can be done either using point-to-point signaling or collectively using a barrier.

Point-to-point Transfer Mechanisms

Point-to-point transfers are either one-sided or two-sided as discussed earlier. The distinction between the two transfer mechanisms includes the type of synchronization and efficiency of handling different message patterns. A two-sided model based on matching send and receive bundle transfer with synchronization is shown in Figure 2.1. One good fit for this model is when we have predictable communication with a small number of peers. A one-sided model is clearly a better option for random accesses. One-sided mechanisms typically decouple the transfer from the synchronization. A separate call for synchronization (flag or barrier) is needed for establishing a consistent state. Many patterns can be implemented using either mechanism. One-sided transfers are the core primitives to support partitioned global address space (PGAS) programming models, such as SHMEM [20] and UPC [7].

The two-sided model could be easier for memory consistency because the local memory within a node cannot be modified until being exposed for communication by a receive call. The one-sided mechanism offers more flexibility in programming, but requires special care in applying synchronization. Programming models such as MPI offers both flavors of programming.

In terms of performance, most optimized runtimes converge at a certain message size to the peak performance delivered by the hardware. The convergence point differs depending on the overhead of the software stack and the transfer mechanisms.

While ultimately, the source and destination of a transfer is user-level data, the runtime may require internal buffering to meet semantic guarantees, or optimize for performance. For instance, to provide non-blocking semantic or to reduce perceived application latency of a blocking transfer, the runtime may need to copy user data into runtime buffers. Additionally, the interconnect hardware may have faster protocols for certain segments of the memory (memory registered with the interconnect). The runtime may copy data from memory segments not registered with the Network Interface Chip (NIC) to registered memory for faster transfers. The use of internal memory for faster transfers is more common in two-sided transfers than one-sided.

Scalability in future systems may put constraints on the runtime buffers that can be used to accelerate transfers. Runtimes may use different buffer allocation strategies depending on the scale. As such, the application should expect the tuning strategy at extreme scale to differ from that at small scale. Applications may need to experiment with different transfer mechanisms for

the best performance.

Active Messaging Mechanisms

Active messages involve a remote procedure invocation at the remote node in addition to transferring data. They can provide a very flexible mechanism for implementing atomic operations, especially when hardware support is not provided. They can also be used in asynchronous programming models, where data arrival can trigger signaling a task to start. Hardware support for active messages in future systems could reduce the overhead of executing tasking models, thus opening new opportunities for low-overhead load-balancing execution. Programming models which use the directed acyclic task-graph abstraction to represent a program may benefit from hardware support for active messages.

Collective Transfer Mechanisms

Collective transfer involves multiple peer ranks in a single communication event. The group of ranks may involve all ranks within a job or a small subset. Collectives have an implicit synchronization, possible computation, in addition to the data transfer part. All ranks involved in the collective need to arrive at the collective for it to complete. While doing a collective reduction on data, an operator (mathematical or logical) is applied. Most runtimes try to optimize the data movement by applying the reduction operator in a staged fashion.

Hardware Support for Transfer Mechanisms

The efficiency of implementing a programming model typically relies on hardware support for implementing basic functionality. For instance, tracking the completion of a one-sided operation is typically needed to establish a consistent state. The interconnect can either provide a hardware mechanism for completion tracking or leave it up to the runtime to use a software protocol to ensure completion. Local completion from the initiator perspective allows reuse of the buffer used to initiate the communication. Another important class of primitives is the support of collectives by the interconnect in a totally offloaded fashion. This capability is currently provided by many high performance interconnects. Hardware acceleration of communication primitives can significantly reduce the overhead of establishing a consistent state. For example, hardware may support tag matching in two-sided communication.

Collectives and remote atomic operations are additional classes of communication primitives that benefit greatly from hardware support. For instance, remote atomics allow direct modification of a remote memory region and support efficient, irregular memory accesses in distributed machines.

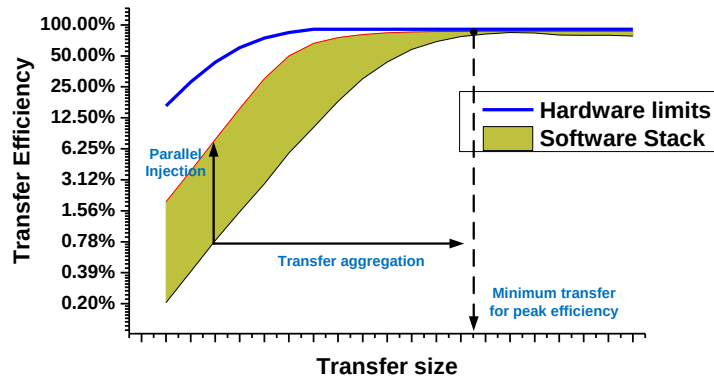


Figure 2.2. Typical Transfer Efficiency Curve

Communication Efficiency

In a distributed environment, a data transfer involves multiple overheads due to transfer packet format and because of latency introduced by executing the complex communication software stack. At the hardware level, additional header information is added to the packet payload for routing, error checking, etc. The header could exceed the payload size for word level transfers. The software stack needs to execute many steps to initiate, track progress and complete a transfer.

Figure 2.2 shows a typical transfer efficiency curve. Transfer efficiency is dependent on transfer size, the level of transfer parallelism, and software overheads. Achieving efficiency typically requires transfer aggregation, increasing transfer parallelism, and reducing software overheads. Hardware-level efficiency is typically observed at much smaller transfer size than transfer efficiency realized by the application. Depending on the software overhead, the minimum transfer size to observe near ideal transfer efficiency differs with the programming model and interface, and the interconnect support for small transfers.

The software overhead is a function of the complexity of the software stack and the capability of the processing cores. The use of lightweight cores is expected to result in reduced observed efficiency by the application. The application needs to either increase the transfer size or have concurrent injection of transfers to amortize such overheads.

From the application developer perspective, the application tuning efforts should follow the cost model for transfers provided by the system. For instance, the application developer should know the minimum transfer size that allows approaching the peak performance. However, for portability, this should be abstracted in some way. Applications should be designed to allow concurrent initiation of transfers. The application can be tuned to overlap initiation of transfers while data are being produced as long as a reasonable transfer granularity is used. Even if current run-times do not always provide fully non-blocking APIs, future systems are likely to provide this capability out of necessity.

Hardware support for aggregation at source and scatter at the destination could serve the purpose of replacing the software mechanism (by the application or the runtime). The caveat is that if supported it will be limited to the strided access cases.

3. Processor Abstract Machine Models for Algorithm Design

An AMM by definition and name is abstract in the sense that implementation details such as speeds and feeds are not represented in the model. Further, AMMs can represent system components such as processors, memories, and interconnects and can be subsequently combined to represent architectures at system scale. In this chapter, we present processor abstract machine models to be used in future node architectures. Note that the level of abstraction in these processors define models that comprise core, NOC, NIC, and accelerator components. The organization of these components defines the various processor AMMs.

For the models that follow in this chapter, we describe the basic node compute infrastructure with a view that the memory and interconnect associated with the node are orthogonal capabilities. To be clear, each of these design dimensions – memory and network interface – are orthogonal in the design of an exascale machine and will be addressed separately in subsequent chapters. We expect network interfaces to be integrated into the processor, but we define the specifics of the network topology in a distinct abstract model since very few algorithms are designed or optimized to a specific network topology. The network topology description could be important for the design of future applications and associated workflows.

Overarching Abstract Machine Model

We begin with a single model that highlights the anticipated key hardware architectural features that may support exascale computing. Figure 3.1 pictorially presents this as a single model, while the next subsections describe several emerging technology themes that characterize more specific hardware design choices by commercial vendors. In the Abstract Model Instantiations section, we describe the most plausible set of realizations of the single model that are viable candidates for future supercomputing architectures.

Processor

It is likely that future exascale machines will feature heterogeneous nodes composed of a collection of more than a single type of processing element. The so-called *fat* cores that are found in many contemporary desktop and server processors are characterized by deep pipelines, multiple levels of the memory hierarchy, instruction-level parallelism and other architectural features that prioritize serial performance and tolerate expensive memory accesses. This class of core is often optimized to run a small number of hardware threads with an emphasis on efficient execution of system services, system runtime, or an operating system. The alternative type of core that we expect to see in conjunction with fat cores in future processors is a *thin* core that features a less

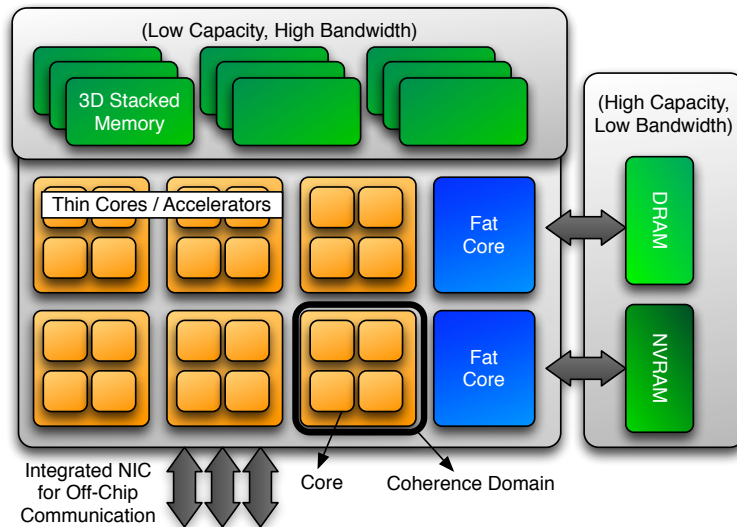


Figure 3.1. Abstract Machine Model of an Exascale Node Architecture

complex design in order to use less power and physical die space. By utilizing a much higher count of the thinner cores a processor will be able to provide high performance and energy efficiency if a greater degree of parallelism is available in the algorithm being executed.

Application programmers will therefore need to consider the uses of each class of core. A fat core will provide the highest performance and energy efficiency for algorithms where little parallelism is available or the code features complex branching schemes leading to thread divergence, while a thin core will provide the highest aggregate processor performance and energy efficiency where parallelism can be exploited, branching is minimized and memory access patterns are coalesced.

On-Chip/Package Memory

The need for more memory capacity and bandwidth is pushing node architectures to provide larger memories on or integrated into CPU packages. This memory can be formulated as a cache if it is fast enough or, alternatively, can be a new level of the memory system architecture. Additionally, scratchpad memories (SPMs) are an alternate way to ensure a low latency access to data. SPMs have been shown to be more energy-efficient, have faster access time, and take up less area than traditional hardware cache [24]. Going forward, on-chip/package SPMs will be more prevalent and programmers will be able to configure the on-chip/package memory as cache and/or scratchpad memory, allowing initial legacy runs of an application to utilize a cache-only configuration while application variants using scratchpad-memory are developed.

Cache Locality/Topology

A fundamental difference from today's processor/node architecture will be the loss of conventional approaches to provide processor-wide hardware cache coherence. This will be driven heavily by the higher power consumption required with increased parallelism and the greater expense in time required to check the various respective resources for cached copies of data. A number of existing studies provide a description of the challenges and costs associated with maintaining cache coherence: Schuchhardt et al. [24] and Kaxiras and Keramidas [17] provide quantitative evidence that cache coherency creates substantial additional on-chip traffic and suggest forms of hierarchical or dynamic directories to reduce traffic, but these approaches have limited scalability. Furthermore, Xu et al. [31] finds that hierarchical caching doesn't improve the probability of finding a cache line locally as much as one would hope – a conclusion also supported by a completely independent study by Ros et al. [22] that found conventional hardware coherence created too much long-distance communication (easily a problem for the scalability of future chips). We find that considerable recent work has followed along the lines of Choi et al.'s 2011 DeNovo approach [9], which argues that hybrid protocols, including self-invalidation of cache and flexible cache partitions, are better ideas and show some large improvements compared to hardware cache coherency with 64 cores and above.

Fast Forward is DOE's node-level hardware technology development program (<http://www.exascaleinitiative.org>). Due to the strong evidence in the literature and numerous independent architectural studies, we and a number of the Fast Forward vendors believe there is ample evidence that continuing to scale current hardware coherence protocols to manycore chips will come at a severe cost of power, performance and complexity. In a potential hybrid architecture, it is likely that the fat cores will retain the automatically managed memories now familiar to developers. However, scaling up coherence across hundreds or thousands of thin cores in a single node may be provided but will come at a high performance cost. As with other shifts in the node architecture this change in coherence scaling will require application developers to pay close attention to data locality.

Current multicore processors are connected in a relatively simple all-to-all or ring network. As core counts surpass the dozen or so cores we see on current processors these networks cease to scale and give rise to more sophisticated network topologies (e.g., mesh topologies). Unlike the current on-chip networks, these networks will stress the importance of locality and force the programmer to be aware of where data is located on-chip to achieve optimal performance.

Integrated Components

We have seen a continued trend towards greater integration. In general, integration of components on die or in package results in increased performance and decreased power. A recent example of physical integration can be seen in the integration of the network interface controller. The NIC is the gateway from the node to the system level network, and the NIC architecture can have a large impact on the efficient implementation of communication models. For large parallel systems, the

inter-node network is the dominant factor in determining how well an application will scale. Even at small scale, applications can spend a large portion of their time waiting for messages to arrive and reductions in bandwidth or failure to substantially improve latency over conventional methods can greatly exacerbate this problem. A custom NIC that integrates the network controller, and in some cases the messaging protocol, onto the chip to reduce power, is expected to also increase messaging throughput and communication performance [6, 27]. Although there is a risk of choosing a network interface that is not compatible with all the underlying data communication layers, applications that send small and frequent messages are likely to benefit from such integration.

Hardware Performance Heterogeneity

One important aspect of the AMM that is not directly reflected in the schematic of the AMM in Figure 3.1 is the potential for non-uniform execution rates across the many billions of computing elements in an exascale system. This *performance heterogeneity* will be manifested from chip-level all the way up to system-level. This aspect of the AMM is important because the HPC community has evolved a parallel computing infrastructure that is largely optimized for bulk-synchronous execution models. It implicitly assumes that every processing element is identical and operates at the same performance. However, a number of sources of performance heterogeneity may break the assumptions of uniformity that underpin our current bulk-synchronous models. Since the most energy-efficient floating point operations (FLOPs) are the ones you do not perform, there is increased interest in using adaptive and irregular algorithms to apply computation only where it is required, and also to reduce memory requirements. Even for systems with homogeneous computation on homogeneous cores, new fine-grained power management makes homogeneous cores look heterogeneous [25]. For example thermal throttling on Intel Sandy Bridge enables the core to opportunistically sprint to a higher clock frequency until it gets too hot, but the implementation cannot guarantee deterministic clock rates because chips heat up at different rates. Options for active (software mediated) power management might also create sources of performance non-uniformity [11]. In the future, non-uniformities in process technology will create non-uniform operating characteristics for cores on a chip multiprocessor [16]. Fault resilience will also introduce inhomogeneity in execution rates as even hardware error correction is not instantaneous, and software-based resilience will introduce even larger performance heterogeneity [29].

Therefore, even homogeneous hardware will look increasingly heterogeneous in future technology generations. Consequently, we can no longer depend on homogeneity, which may present challenges to bulk-synchronous execution models.

Abstract Model Instantiations

Given the overarching model, we now highlight and expand upon key elements that will make a difference in application performance. Note that in all of the models, a NIC is shown on the processor package with a dashed outline around it. This signifies that the NIC could potentially

be integrated on the processor package or may not be. Currently, not all processors have in-package NICs, but we expect in the future that many will, which is why it is shown this way in the models. In many of the models presented, an accelerator is shown either integrated into the processor package or in a separate device connected to the processor package. We define an accelerator as a specialized computational unit that is designed to speed computation and is used as an offload engine rather than a primary computational device. Devices such as GPUs and FPGAs are considered accelerators by this definition.

Homogeneous Many-core Processor Model

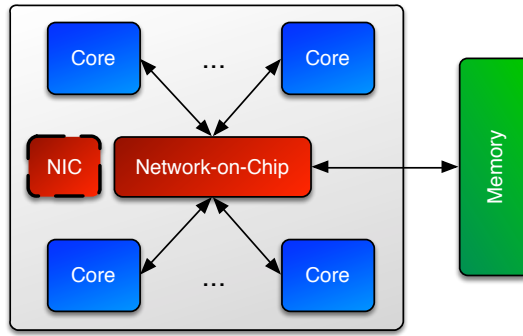


Figure 3.2. Homogeneous Manycore Model

In a homogeneous manycore node (Figure 3.2) a series of processor cores are connected via an on-chip network. Each core is symmetric in its performance capabilities and has an identical instruction set (ISA). The cores share a single address memory space and may have small, fast, local caches that operate with full coherency. We expect that the trend of creating individual clock and voltage domains on a per-core basis will continue allowing an application developer or system runtime to individually set performance or energy consumption limits on a per-core basis meaning that variability in runtime will be present on a per-core basis, not because of differences in the capabilities of each core but because of dynamic configuration. Optionally, the cores may implement several additional features depending on the performance targets including simultaneous multi-threading (SMT), instruction level parallelism (ILP), out-of-order instruction execution or SIMD (single instruction, multiple data) short-vector units.

Like a system-area interconnect, the on-chip network may “taper” and vary depending on the core pair and network topology. Similarly, the programmer and compiler will have to contend with network congestion and latency. Depending on the programming model, communication may be explicit or largely implicit (e.g. coherency traffic).

Multicore CPU with Discrete Accelerators Model

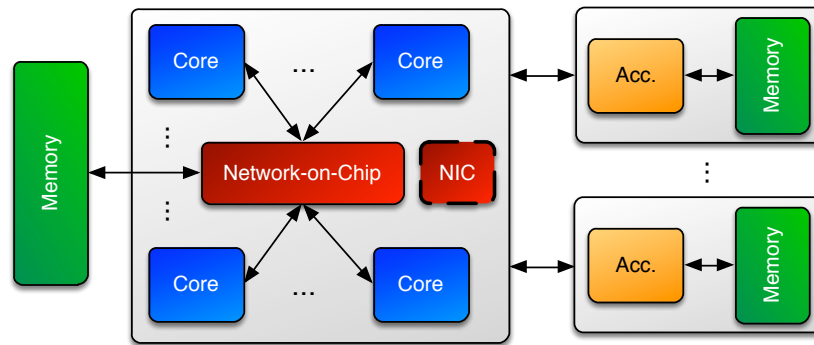


Figure 3.3. Multicore CPU + Discrete Accelerators Model (Acc: Accelerator)

In this model a homogeneous multi-core processor (Figure 3.3) is coupled with a series of discrete accelerators. The processor contains a set of homogeneous cores with symmetric processor capabilities that are connected with an on-chip network. Each core may optionally utilize multi-threading capabilities, on-core caches and per-core based power/frequency scaling. Each discrete accelerator is located in a separate device and features an accelerator processor that may be thought of as a throughput oriented core with vector processing capabilities. The accelerator has a local, high performance memory, which is physically separate from the main processor memory subsystem. To take advantage of the entire compute capability of the processor, the programmer has to utilize the accelerator cores, and the programming model may have to be accelerator-aware. Future implementations of this AMM will increase bus-connection bandwidth and decrease latency between processor and accelerator reflecting a trend towards increased logical integration.

Integrated CPU and Accelerators Model

An integrated processor and accelerator model (Figure 3.4) combines potentially many latency-optimized processor CPU cores with many accelerators in a single physical die, allowing for potential optimization to be added to the architecture for accelerator offloading. The important differentiating aspect of this model is a shared, single coherent memory address space is accessed through shared on-chip memory controllers. While this integration will greatly simplify the programming, latency optimized processors and accelerators will compete for memory bandwidth.

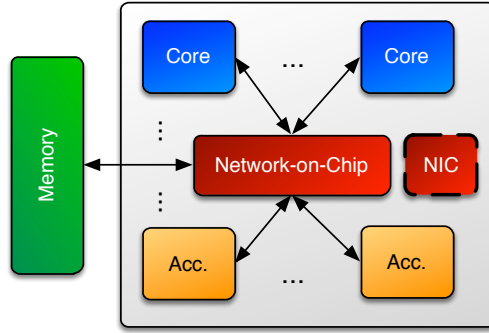


Figure 3.4. Integrated CPU + Accelerators Model (Acc: Accelerator)

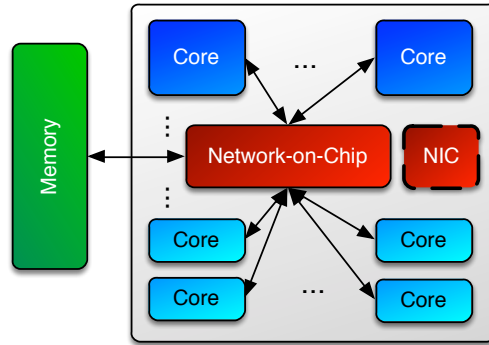


Figure 3.5. Heterogeneous Multicore Model

Heterogeneous Multicore Model

A heterogeneous multi-core architecture features potentially many different classes of processor cores integrated into a single die. All processor cores are connected via an on-chip network and share a single, coherent address space operated by a set of shared memory controllers. We envision that the cores may differ in ISA, performance capabilities, and design, with the core designers selecting a blend of multi-threading, on-chip cache structures, short SIMD vector operations, instruction-level parallelism and out-of-order/in-order execution. Thus, application performance on this architecture model will require exploiting different types and levels of parallelism. Figure 3.5 provides an overview image of this design for two classes of processor cores.

The main difference between the heterogeneous multi-core model and the previously discussed integrated multi-core CPU and accelerator model (Integrated CPU and Accelerators Model section) is one of programming concerns: in the heterogeneous multi-core model each processing element

is an independent processor core that can support complex branching and independent threaded, process or task-based execution. In the integrated multi-core with accelerator model, the accelerators will pay a higher performance cost for heavily divergent branching conditions and will require algorithms to be written for them using data-parallel techniques. While the distinction may appear subtle, these differences in basic hardware design will lead to significant variation in application performance and energy consumption profiles depending on the types of algorithm being executed, motivating the construction of two separate AMMs.

Abstract Models for Concept Exascale Architectures

The machine models presented in the Abstract Model Instantiations section represent relatively conservative predictions based on known vendor roadmaps and industry trends. However, with the advent of system on chip (SoC) design, future machines can be optimized to support our specific requirements and offer new methods of implementing on-chip memories, change how coherency and data sharing are done and implement new ways to support system latencies. This creates a much wider design space. In this section we present one possible concept for a customized system node.

Performance-Flexible Multicore-Accelerator-Memory Model

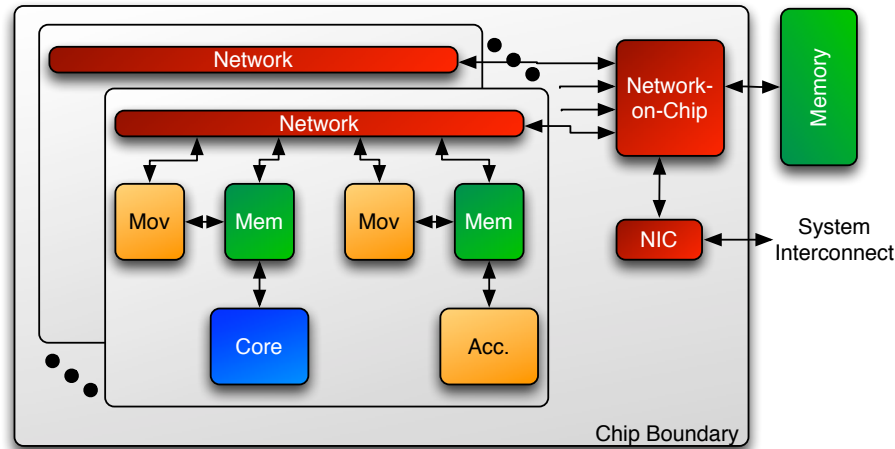


Figure 3.6. Homogeneous Multicore-Accelerator-Memory Model (Mem: Memory, Acc: Accelerator, Mov: Data Movement Engine)

The homogeneous multicore-accelerator-memory (MAM) model in Figure 3.6 is an aggressive design for a future processor based around new approaches to make general computation more

efficient. The focus in this design is on achieving higher percentages of peak performance by supporting execution through a greater variety of specialized function units and multi-threaded parallel execution within each core. The processor features many heterogeneous cores, a hierarchical internal network and an internal NIC with multiple network connections, allowing multiple memory channels and extremely high internal and external access to local and remote processor memories throughout the system. There are multiple components in the memory system: many internal memory blocks that are integrated into the CPU cores as well as main memory that is directly connected to nodes and is available to other nodes through a system's network.

Each core implements a high order number of threads to hide latency and the capability to tradeoff the number of threads for greater performance per thread. Multiple threads in a core run at the same time and the implementation is such that the core is kept busy; for example if a thread is held waiting for a main-memory item, another thread is put into execution.

As stated above, each thread in a core is given a portion of local on-chip memory. That memory is originally private to each thread, though each core can choose to share portions of its space with other threads. Each portion has multiple pieces such that some pieces can be for caching and others for "*scratch space*" at the same time. Scratch space is memory that is used to store intermediate results and data placed there is not stored in main memory. This saves energy and reduces memory traffic. (Scratch data is saved if a job is rolled out.)

When a portion of its local, cached memory is shared with other threads, the sharing entities see only a single cache for that specific portion of the address space. This greatly reduces coherency issues. If the shared portion of local memory is scratch space, there is only a single copy of the data. In this latter case, coherency must be managed by software, or it can be done with atomic operations if appropriate.

There are also two different kinds of high-performance accelerators: vector units and move units (data movement engines) that are integral parts of each core. The vector acceleration units can execute arbitrary length vector instructions, unlike conventional cores which execute short SIMD instructions. The vector units are done such that multiple vector instructions can be running at the same time and execution performance is largely independent of the length of the vectors.

Multiple data movement engines are also added to the processor and vector units to provide general data movement, such as transposing multidimensional matrices. Both the vector and move accelerators can perform their functions largely independent of the thread processes or can be directly controlled by and interact directly with executing threads. Local scratch pads are included in the core and accelerators (shown in the block diagram as Mem-blocks) where applications can store data items at very low access latency. These local memories also provide a direct core-to-remote-core messaging capability where messages can be placed ready for processing. By providing separate memory blocks, vector units and processor cores can run independently.

Thread execution in each core is organized into blocks of time. An executing thread can have a single clock in an execution block or it can have multiple clocks. This enables the number of threads in execution and the execution power of threads to vary depending on application requirement. There can also be execution threads that are inactive but ready to execute when an executing

thread would be idle (waiting on some action to complete), or can be loaded or unloaded. Any thread seeing more than a one clock-block wait time is replaced with an inactive thread that is ready but is waiting for active execution time.

4. Memory System

In this chapter, we describe current and future memory technologies followed by our abstract models of future memory subsystems. For these models, it is important to note that we do not fully describe the coherency aspects of the various memory subsystems. These memory systems will likely differ from current coherency schemes and may be non-coherent software-based coherent, or hardware-supported coherent.

Memory Drivers

The current generation of main memory, DDR-4 is expected to be the basis of main memory for at least the next three to four years. But that memory cannot be used – at least not by itself – as the basis for the high-end and exascale systems envisioned here. As the DDR-4 standard pushes engineering to the limit, it is unclear if JEDEC (the standards body that supported the development and implementation of the DDR memory standards) will provide a DDR-5 standard. This may force system vendors to explore alternative technologies.

A promising alternative to DDR-5 is to provide a hybrid memory system that will integrate multiple types of different memory components with different sizes, bandwidths, and access methods. There are also efforts underway to use some very different DRAM parts to build an integrated memory subsystem; this memory has characteristics that are very different than DDR-4 technology such that power and energy would be reduced with respect to current memory. In addition, because the power is reduced, the capacity of memory can be greatly increased.

Consider a system that has two types of components in its memory system. This system will contain a fairly small number of parts that are mounted on top of or are in the same carrier as the CPU chip (e.g. one to eight memory parts with each part being a 3D stack of redesigned memory die). An example of a system with this memory system is Intel's Knight's Landing.

The bandwidth of these integrated memory parts will likely be in the low 100's of gigabytes-per-second each – much higher than any current memory parts or memory modules. But this increased bandwidth comes at a cost of lower capacity; therefore, this high-bandwidth memory alone will be unable to support any realistic applications and must be paired with a higher capacity, lower bandwidth memory. This higher capacity, lower bandwidth memory will likely be something like DDR-4, and will provide the majority of the system's memory capacity. Known memory technologies enable us to get memory capacity or memory bandwidth, but not both in the same device technology. This motivates the move towards a new organization for external memories. These trade-offs are diagrammed in Figure 4.1. Of course such a two-level structure raises co-design needs with respect to system software: compilers, libraries, and OS support, and other elements of the system software stack.

There are two new DRAM technologies that are coming to market that are aimed at getting

around some of the limitations of DDR DRAMs, though, as mentioned above, at higher per-bit costs.

High Bandwidth Memory (HBM) is a new JEDEC standard that has seen initial shipments. The parts are 3D stacks of DRAMs that are aimed at significantly higher bandwidths, but have limited size scalability. Hybrid Memory Cube (HMC) memory is a new internal memory organization and interface for 3D parts that offers somewhat higher bandwidths than HBM and has other features that will very likely be beneficial for large systems. Initial shipments have also been made.

There is also a new non-volatile (NV) technology that has been announced, but with little firm description and data at this point. 3D XPoint (from *Cross-Point*), is a non-volatile memory technology that is targeted to have a similar performance profile to present day DDR.

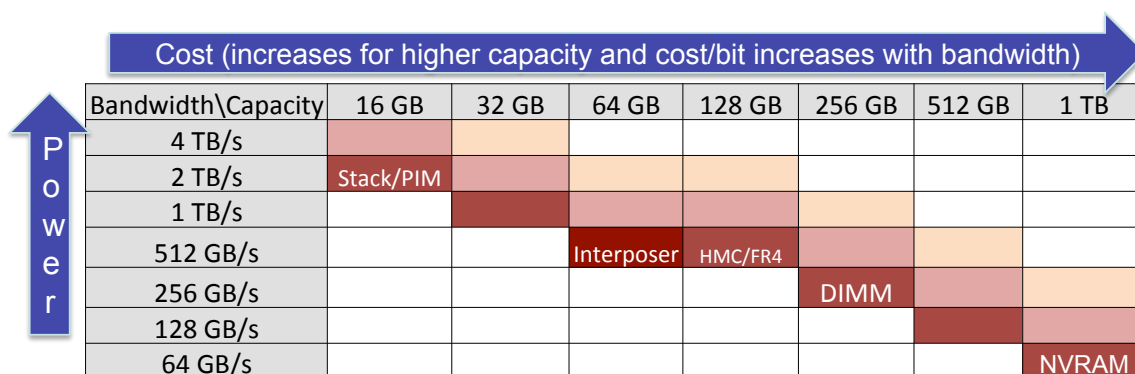


Figure 4.1. Per-Node Bandwidth and Capacity of Various Memory Technologies

New high-bandwidth components will come at significantly higher procurement cost. We envision analysis being performed to not only justify the increased cost of these components with respect to application performance, but also to determine the optimum ratio of high-bandwidth, low-capacity memory to low-bandwidth, high-capacity memory. Current market data shows the high-speed low-capacity stacked memory is consistently more expensive per bit of capacity than the higher-capacity lower-bandwidth DDR memories. This ratio is governed by both the cost of 3D technology and market dynamics.

This performance vs. cost analysis will also need to include relative energy consumption for each memory type. As an example, if DDR-4 is used as the basis for the majority of a system's memory capacity, then the total capacity in such a system will be significantly reduced when compared to a system utilizing DDR-4 in combination with technologies which are more energy efficient, such as NVRAM, and/or higher performance, such as HMC. This performance vs. cost vs. energy trade-off analysis does not have an immediate impact on the study here, but will affect choices that must be made at a system level such as: how much is additional memory worth with respect to system performance, total size on the computer room floor, and other facility considerations.

Finally, a system need not be restricted to two levels of memory and may have three or more levels with each level composed of a different memory technology, such as NVRAM, standard DRAM, 3D Stacked DRAM, or other memory technologies. As a result, future application analysis must account for complexities created by these multi-level memory systems. Despite the increased complexity, however, the performance benefits of such a system should greatly outweigh the additional burden in programming brought by multi-level memory; for instance, the amount of data movement will be reduced both for cache memory and scratch space resulting in reduced energy consumption and greater performance.

Further advantages can be demonstrated if, for example, NVRAM – of which most variants boast a lower energy cost per bit accessed compared to DRAM – is used as part of main memory, allowing for an increase in total memory capacity while decreasing total energy consumption. Additional research is needed to find the best ways for application programmers to use these changed and expanded capabilities.

Future Memory Abstractions

For each of the node architectures presented in Abstract Model Instantiations, the layout of memory within the node can usually be regarded as an orthogonal choice – that, it is possible for architectures to mix and match arrangements for compute and selection of memory components. Due to the explosive growth in thread count expected to be present in exascale machines, the total amount of memory per socket must increase accordingly, potentially in the range of two terabytes or more per node. As explained in the Memory Drivers section above, it is expected that the memory system will be made up of multiple types of memory that will trade capacity for bandwidth. In many ways this concept is not so unfamiliar to developers who currently optimize problem sizes to fit in local caches, etc. These new memory choices will present additional challenges to developers as they select correct working set sizes for their applications. As an initial first attempt to characterize likely memory sub-systems, we propose three categories/types of memory:

1. High-bandwidth-memory (HBM or HMC): A fast, but relatively small-capacity, high bandwidth memory technology based on new memory standards such as JEDEC’s high bandwidth memory (HBM) [14] or WideIO [13] standard, or Micron’s hybrid memory cube (HMC) technology [26].
2. Standard DRAM: A larger capacity category of slower DDR DRAM memory.
3. Non-volatile-memory: A very large but slower category of non-volatile based memory. However, 3D XPoint NV memory may have much higher bandwidth and ease of use.

As shown in Figure 4.2(a), there will be two principle approaches to architect these memory categories/types: (a) a physical address partitioning scheme in which the entire physical space is split into blocks allowing each memory category to be individually addressed, and (b) a system in

which faster memory categories are used to cache slower levels in the memory system. A third possible approach to constructing a memory system is to provide a blending of these two models with either user-defined or boot-time defined partitioning of the memory systems into partial cache and partial address space partitioned modes.

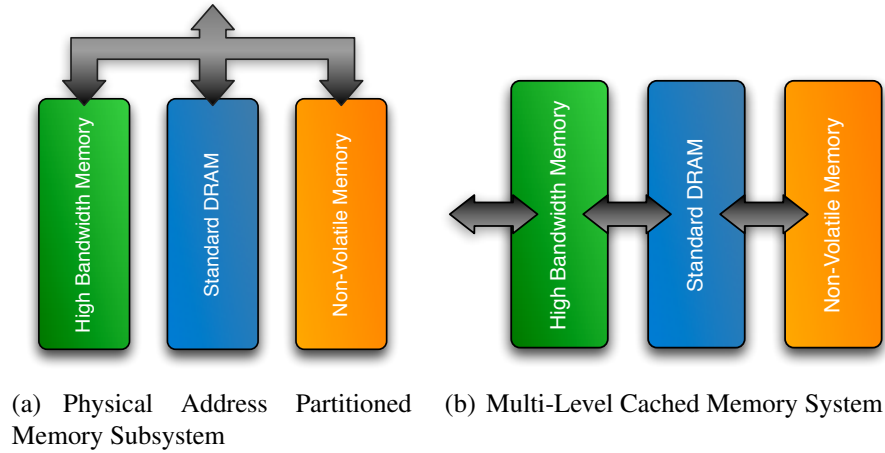


Figure 4.2. Memory Subsystem Layouts

Physical Address Partitioned Memory System

In a physical address partitioned memory system, the entire physical memory address space is split into discrete ranges of addresses for each category of memory (Figure 4.2(a)). This allows an operating system or runtime to decide on the location of a memory allocation by mapping the request to a specific physical address, either through a virtual memory map or through the generation of a pointer to a physical location. This system therefore allows for a series of specialized memory allocation routines to be provided to applications. An application developer can specifically request the class of memory at allocation time. We envision that an application developer will be able to request a specific policy should an allocation fail due to memory category exhaustion. Possible policies include allocation failure, resulting in an exception, or a dynamic shift in allocation target to the next slowest memory category. While the processor cores in this system may possess inclusive caches, it is not likely there will be hardware support for utilizing the faster memory categories for caching slower categories. This lack of hardware support may be overcome if an application developer or system runtime explicitly implements this caching behavior.

Scratchpad Memory

Historically, DDR-DRAM has dominated main memory. However, emerging memory technologies (NVRAM, 3D Stacked DRAM) may provide superior cost/performance/capacity trade-offs. Future main memory systems are likely to be comprised of multiple level memories (MLMs)

made up of different memory technologies. Emerging memory technologies such as High Bandwidth Memory (HBM) or Hybrid Memory Cubes (HMC) may provide an order of magnitude more bandwidth than conventional DRAM. However, the cost per bit of these new stacked DRAM technologies will probably be much higher than conventional DDR, so building an entire memory system from them may be prohibitively expensive. Certain types of NVRAM, such as NAND Flash is roughly 4-7 times less expensive than conventional DDR, but has much less bandwidth and much higher latency. If applications can be adapted to place seldom-used data in NVRAM, they may see minimal performance impact and achieve a major cost reduction. At the time of this report, the authors are awaiting public information about the Intel/Micron 3D XPoint NVRAM technology.

New programming and runtime techniques will have to be developed to take full advantage of MLM. This may include application driven *hints* about which portions of application data will be used the most or may require fundamental restructuring of the algorithm to take advantage of multiple levels of memory. Runtimes or operating systems may be able to provide automatic management of data, or they may require guidance from the application.

Preliminary analysis of miniapps indicates that a small percentage of an application's memory (5-20%) accounts for a disproportionate number of post-cache memory accesses (25-50%). Placing this frequently used data in a faster memory level may improve performance by 30-50%. However, due to the higher cost/bit of fast memory the growth in aggregate memory capacity of future machines may be limited if only a single level of fast memory is used.

Multi-Level Cached Memory System

An alternative memory model is that multiple categories are present in the node, but they are arranged to behave as large caches for slower levels of the memory hierarchy (Figure 4.2(b)). For instance, a high-bandwidth memory category is used as a caching mechanism for slower DDR or slower non-volatile memory. This would require hardware caching mechanisms to be added to the memory system and, in some cases, may permit an application developer or system runtime to select the cache replacement policy employed in the system. It is expected that this system will possess hardware support for the caching behavior between memory levels; however, a system lacking this hardware support could implement a system runtime that monitors memory accesses to implement an equivalent behavior in software.

Performance cost of using memory as multi-level cache is significant causing vendors to offer using certain levels in the memory hierarchy as a software managed cache.

Configuration	Bandwidth	Capacity
Single-Level HMC		
HMC (4 HMC “cubes”)	~640 GB/s	~16GB
Multi-Level DRAM		
HBM (4 stacks @ 128GB/s)	~512 GB/s	~16 GB
DDR (4 channels (8 DIMMs) @ 20GB/s)	~80 GB/s	~512 GB
NAND Flash		
NVRAM	10–20 GB/s	4 – 8× DRAM

† See notes in 3-D Stacked Memory Systems, Processing in Memory (PIM), and Processing Near Memory (PNM) section

Table 4.1: Approximate Bandwidths and Capacities of Memory Subsystem

3-D Stacked Memory Systems, Processing in Memory (PIM), and Processing Near Memory (PNM)

As mentioned above, a new technology that will emerge in the memory hierarchy is 3D-stacked memory. Table 4.1 shows bandwidth and capacity estimates of some of these future technologies. Note that the numbers in Table 4.1 are based on current state-of-the-art and may change based on future technology shifts and breakthrough developments. These 3D stacks of memory will have a logic layer at the base to handle read and write requests to the stack. Not only will there be multiple memory dies in a single memory component, but in some versions these memory dies will be mounted directly on CPU chips resulting in greater density with a reduced energy footprint. Additionally, processor-in-memory (PIM) functionality may emerge in conjunction with the stacked memory architectures that include logic layers at the base of the memory stacks. These PIM capabilities offer acceleration to many memory operations, such as atomics, gather-scatter, pointer chasing, search, and other memory bandwidth intensive operations. These accelerators can execute faster and more efficiently than general-purpose hardware. An SoC design flow creates an opportunity for many types of acceleration functions to improve application performance. These accelerators can make data movement more efficient by avoiding unnecessary copies, or by hiding or eliminating overhead in the memory system or a system’s interconnect network. However, how best to expose these operations to the programmer is still an active area of research.

Some 3D memory technologies, such as the HMC, allow memory parts or modules to be “chained” in different topologies. In contrast, DDR connects a small number of memory parts to a processor. Similarly, there are other standards for high-performance memory on the horizon, such as HBM that also only connect a single memory part to a single processor. “Chained” memory systems differ from DDR and HBM in their ability to support a very high memory capacity per-node. The limitations of per-node memory capacity when using chained systems will be dominated by dollar cost. While the relative dollar cost of stacked memory is expected to approach DDR it is expected to be more (\$ per bit) than DDR. In contrast, the power cost – Joules per accessed memory bit – is expected to be significantly less for 3D stacked memory when compared to DDR.

5. Node-Level Abstract Machine Models and Proxy Architectures for Exascale Computing

Proxy architecture models (PAMs) were introduced as a codesign counterpart to proxy applications in the DOE ASCAC report on the Top Ten Exascale Research Challenges [21]. This Computer Architecture Laboratory (CAL) AMM document separates the PAM concept into AMM and proxy architectures, but the intent is still to facilitate codesign and communication.

In this chapter we initially discuss node architectures and their possible configurations given the recent and potential future changes particularly in the way that memory and compute might be used in a node. For example, nodes may contain memory to be used for burst buffer or may be configured specifically for visualization. We continue by identifying approximate estimates for key parameters of interest to application developers at the node level. Many of these parameters can be used in conjunction with the AMM models described previously to obtain rough estimates of full node performance. These parameters are intended to support design-space exploration and should not be used for parameter- or hardware- specific optimization as, at this point in the development of Exascale architectures, the estimates may have considerable error. In particular, hardware vendors might not implement every entry in the tables provided in future systems; for example, some future processors may not include a Level-3 cache.

Abstract Model of Future Computing Nodes

Over the past several years, the complexity of a compute node has drastically increased as technology size decreases and power management improves. A compute node many years ago commonly comprised a single CPU, off-chip memory, and a NIC. With technology advances, nodes can now comprise multiple, multi-core and potentially heterogeneous CPUs with an on-package NIC, an on-chip network for CPU communication, and potentially both on-chip and off-chip memory. A typical node today comprises two CPU sockets, where a single CPU can be heterogeneous in that it may have traditional CPUs integrated with accelerators in a single die.

In Chapter 3, we presented several abstract machine models that can essentially be defined as the CPU of a node. If the node has multiple CPUs, then it simply implements two of the CPUs described by the abstract machine models presented in Chapter 3. Specifically, Figures 3.1, 3.2, 3.4, 3.5, and 3.6 can potentially be implemented as node CPUs. All of these models implement off-chip memory in the node.

Detailed Processing Node models

Processing nodes of the future could potentially have several different configurations combining compute, memory, and storage. Nodes may comprise:

1. homogeneous compute
2. heterogeneous compute (meaning a node could contain scientific/traditional compute in one socket and visualization or some specialized compute accelerator on another socket in the same node)
3. compute and high-capacity, low-bandwidth memory
4. compute and storage-class memory

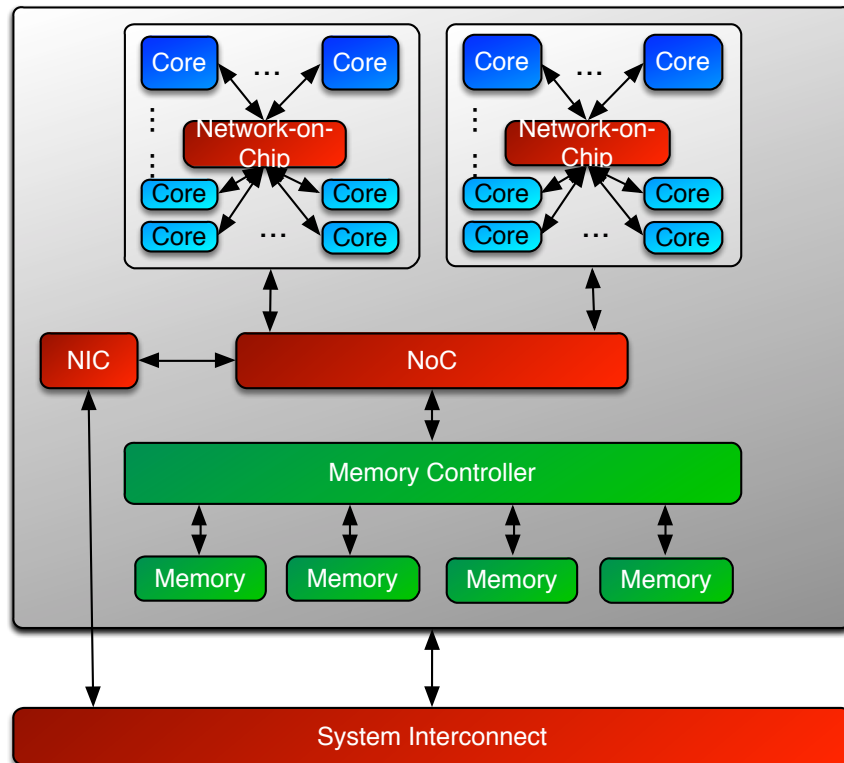


Figure 5.1. Example Node Architecture using Processor AMM

Figure 5.1 shows an example node architecture using the processor AMM from Figure 3.5. This is a dual-socket node with the heterogeneous multicore processor AMM as the CPU. Note that the memory component could potentially be any of the technologies listed in Table 4.1. Also

note that we define a node as a network endpoint, meaning everything on the other side of the network.

In addition to the node shown in Figure 5.1, the processor AMMs from Chapter 3 and the abstracted memory components outlined in Chapter 4 can be combined to implement various types of nodes such as visualization, fat and lightweight compute, or potentially storage nodes. The specific type of node used is specifically described in a proxy node architecture (Proxy Parameters) and can be subsequently used in a larger system proxy model (Chapter 7).

Lightweight Cores and Processing Communication Runtime

Lightweight cores are increasingly popular in modern system designs due to their power efficiency. Such efficiency is achieved by reducing the cache sizes, limiting out-of-order processing, increased use of vector processing, etc. While these architectural changes could be leveraged in optimized scientific codes, they pose a challenge to communication runtime implementations. Runtime software is typically control-flow intensive and has complex dependency between instructions, thus could benefit greatly from out-of-order and speculative execution. Using lightweight cores to execute runtimes (initiate, progress, and complete transfers) is likely to incur more cycles (processing overhead). This issue requires careful attention from runtime and application developers alike. To efficiently use systems with lightweight cores the communication runtime may have one of the following strategies:

- Use a specialized heavyweight core to manage the interconnect activity. In this case, few injection points are anticipated per node. This eases the resource problem at the runtime level because injection resources do not need to scale with the number of cores.
- Use lightweight cores, especially if they are the only compute resources available, to process the communication traffic to the interconnect. In this case, parallel transfer processing could amortize the software overhead. Scalability of runtime resources with core count is likely to be a challenge. Sharing runtime resources between cores could have a negative impact on performance because sharing could cause serialization in accessing the interconnect.
- Adopt a hybrid runtime with restricted functionalities (and low overhead) for the lightweight cores and a full-functionality runtime for the complex cores. This model is a possible research direction, but is not adopted by current programming models.
- Part of the runtime functionalities are offloaded to specialized hardware. As such, lightweight cores could efficiently execute complex programming model APIs.

A challenge that runtime developers face is the need to ensure that the memory requirements do not excessively impact the memory available to the application program. Concurrent initiation using lightweight cores could increase the memory requirement for interconnect injection points (endpoints). Runtime designs should avoid linearly scaling memory requirements with the endpoint count.

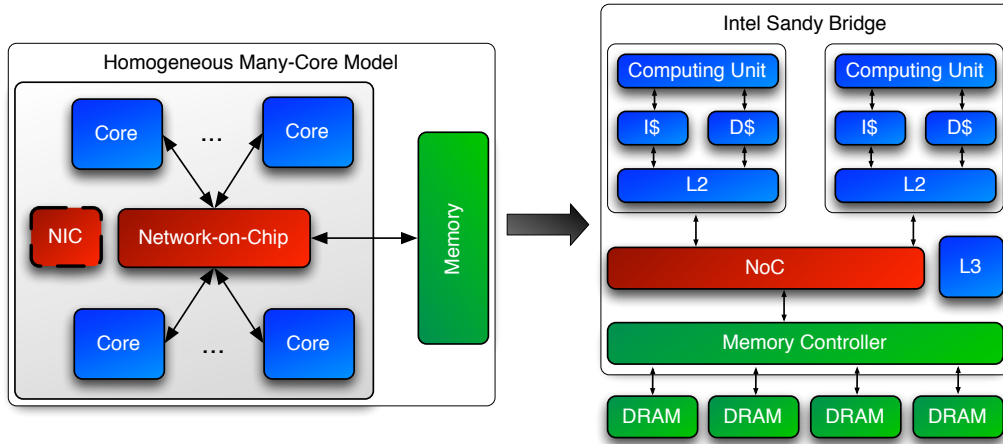


Figure 5.2. Reference proxy architecture instantiation: Intel Sandy Bridge

Some of the discussed strategies could be abstracted away by the runtime; others will require the application to restructure its communication pattern. Runtimes could, through communication assist, parallelize the transfers, aggregate small transfers, control the level of parallel injection, or manage asynchronous transfers. The application may need to control parallel injection of transfers or hint which transfers could be progressed independently, for instance through issuing non-blocking transfers.

Reference Proxy Architecture Instantiations

Some expanded AMMs in the Abstract Model Instantiations section have their roots in existing advanced technology processors. One of these could be a harbinger of what an exascale processor may look like. We provide their proxy architecture information here for reference.

Homogeneous Manycore Model: Intel Sandy Bridge

An example of the homogeneous manycore processor model (in the Homogeneous Many-core Processor Model section) is the Intel Sandy Bridge (shown in Figure 5.2), a 64-bit, multi-core, dual-threaded, four issue, out-of-order microprocessor. Each core has 32KB of L1 data cache, 32KB of instruction cache and 256KB of L2 cache. The processor comprises up to eight cores, a large shared L3 cache, four DDR3 memory controllers, and 32 lanes of PCI-Express (PCIe); there is a 32-Byte ring-based on-chip interconnect between cores.

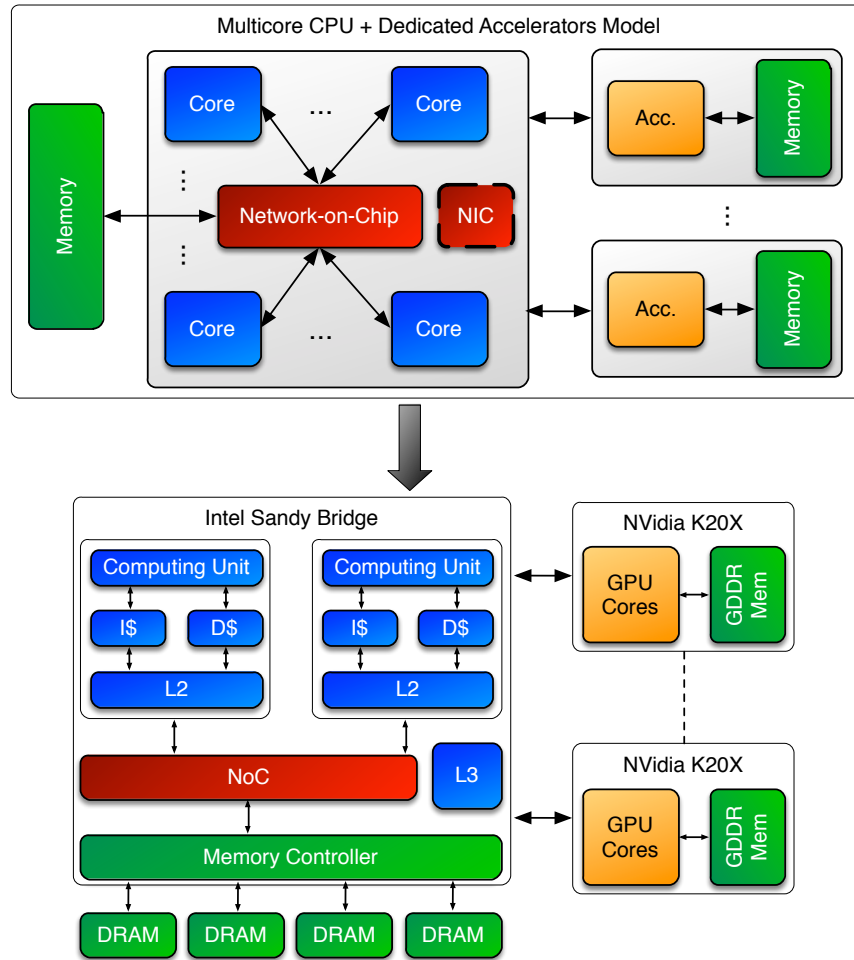


Figure 5.3. Reference proxy architecture instantiation: Multicore CPU with Discrete GPU Accelerators

Multicore CPU + Discrete Accelerators Model: Sandy Bridge with Discrete NVIDIA GPU Accelerators

An example of the multicore CPU + discrete accelerators model (Multicore CPU with Discrete Accelerators Model) is a system utilizing Intel’s Sandy Bridge multi-core processor with NVIDIA GPU-based accelerators (shown in Figure 5.3). Each node comprises a dual-socket Xeon host-processor (Sandy Bridge E5-2660 2.66GHz) for primary compute with two NVIDIA K20X Kepler GPU cards. These GPU cards connect to the primary compute package via a PCIe Gen2x16 interface. Each GPU card implements 2, 688 processor cores and 6GB of 384-bit GDDR-5 memory with a peak bandwidth of 250 GB/s.

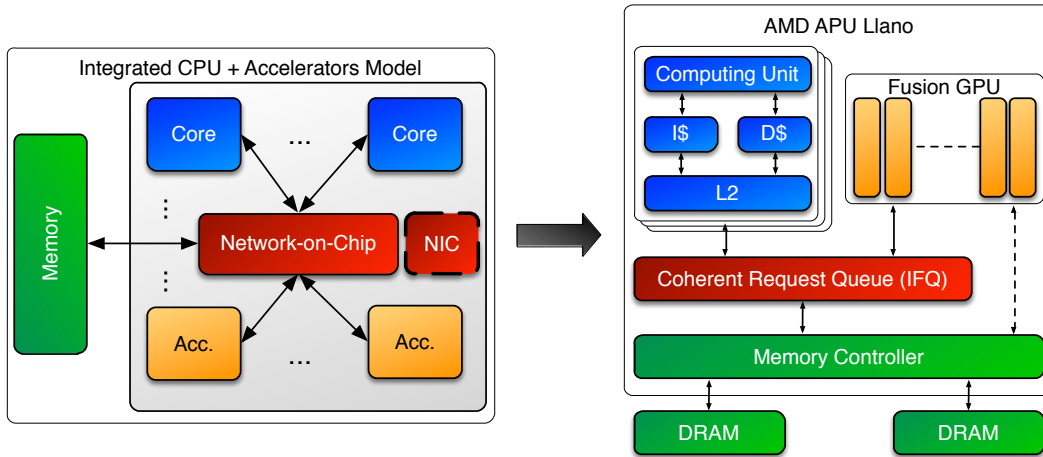


Figure 5.4. Reference proxy architecture instantiation: AMD APU Llano

Integrated CPU + Accelerators Model: AMD Fusion APU Llano

An example of an integrated CPU and accelerators AMM (Integrated CPU and Accelerators Model) is the AMD Fusion APU Llano shown in Figure 5.4. The Llano architecture implements four x86 CPU cores for general-purpose processing, an integrated GPU, I/O, memory controller, and memory. Each CPU core has its own L1 and L2 caches. The unified memory architecture (UMA) implemented in Llano allows CPU processors and GPU accelerators to share a common memory space. The GPUs also have a dedicated non-coherent interface to the memory controller (shown with a dotted line) for commands and data. Therefore, the memory controller has to arbitrate between coherent (ordered) and non-coherent accesses to memory. Note that the CPU is not intended to read from GPU memory.

Proxy Parameters

The following is a condensed list of parameters that focuses on the key metrics concerning application developers. This list allows developers and hardware architects to tune any AMMs to define common baselines. A more complete list of proxy architecture design parameters, although still not exhaustive, can be found in the System Proxy Architectures section and will continue to grow as needed. Since this list is for all AMMs presented in this document, not all parameters are expected to be applicable to every AMM. In fact, we expect that for each AMM only a subset of this list of parameters will be used for architecture tuning. Likewise, not all parameters are useful for application developers, such as bandwidth of each level of the cache structure.

Processor

Parameter	Expected Range	Notes
Bandwidths (GB/s)		
Chip ↔ Mem-ory	60–100	To off-chip DDR DRAM
Chip ↔ Mem-ory	600–1200	To on-chip DRAM
Chip ↔ Mem-ory	600–1200	To off-chip HMC-like
Capacities		
L1 Cache	8KB–128KB	Per Core
L2 Cache	256KB–2MB	Per Core
L3 Cache	64MB–128MB	Likely to be shared amongst groups of cores/accelerators
L4 Cache	2GB–4GB	Not on all systems, likely to be off-package embedded-DRAM
Memory System Parameters		
Cache block size	64–128B	
Number of cache levels	2–4	
Coherency do-mains	1–8	per chip
Network-On-Chip		
Bandwidth	8–64 GB/s	per core
Latency	1–10 ns	neighbor cores
Latency	10–50 ns	cross chip
Core Parameters		
SIMD/vector width	4–8 DP FP	
Dispatch, is-sue, execution widths	2–8	Simple processors may have limited dual dispatch (i.e. 1 general purpose instruction and 1 memory)
Atomic opera-tions		Possible implementations include: Simple or trans-actional

6. System Scale Considerations

Along with the changes exascale brings to the node architectures, come system level changes that will be driven by the increasing scale of production supercomputers. As node counts potentially push into the greater than 100,000 regime, system level factors will have greater impact on application workflows. Application developers will need to understand the performance implications on their workflows of new system-level architectures, such as communication costs and integration of storage on-platform. While the unprecedented scale of many exascale systems will be a challenge, new innovations will provide new opportunities for optimizing mission workflows. This chapter briefly discusses workflow analysis in and follows with a discussion of the interconnect and communication model considerations.

System Analysis of Advanced Workflows

One of the major changes that all current and planned DOE HPC systems are addressing are new workflows that remove the file system as the data buffer or intermediary between the HPC system as the source of modeling and simulation data, and the data analysis and visualization resources that historically sat next to the HPC system but were distinct, separate computing resources. This change is driven by the *cost* of data movement in terms of both energy and time, and the need to leverage the processing capability of future supercomputers to also perform post-processing on the data in either in-situ or in-transit strategies. This same infrastructure may be useful for problem set-up functions, e.g. mesh generation; or application development, e.g. in-situ analysis with performance monitoring tools or advanced debuggers.

We envision that a system abstract machine model (sAMM) and our associated system proxy architectures will help application developers, system software developers and system architects analyze and reason about how new workflows will create requirements for runtime connectivity between HPC applications running in the compute partition and data analysis or visualization jobs running in a concurrent NVRAM data analytic partition. Another scenario could be the need for a runtime network connection between the HPC application running in a compute partition and a concurrent job running application performance analysis tools.

Interconnect Model

In the exascale era, application performance can easily be limited by network bandwidth because of the cost and power consumption of high-radix routers as well as the optical and electrical channels. In addition to performance, procurement cost will be a large barrier as the cost of not only high-radix routers but also the cables (both optical and electrical) that increase in cost significantly with cable length. For instance, copper cables cost approximately 50% more for just 3

meters compared to 1 meter, and optical cables cost almost twice as much for 30 meters compared to 5 meters. In addition, high-radix routers impose a cost per port that is comparable to the cable connecting to that port. These costs have remained relatively constant over the last four years and are not projected to decrease significantly in the short term.

An important constraint on bandwidth growth in large-scale machines is the limitation in the total number of pins available on a board or connector. We are approaching the point where the physical size of the edge of a switch circuit board (constrained by the dimensions of a standard-sized rack as well as card manufacturing viability) starts to pose a bandwidth-limiting factor, because only so many (e.g.) QSFP cages or board pins can fit on the card edge. In addition, the power cost of moving data from or to remote caches, memories, and disks can far exceed the available power budget, thus limiting available bandwidth. DesignForward collaborators estimate that fitting an exascale system network into a 5MW budget in order to meet DOE's 20MWs for the entire system is a significant challenge and will require cross-layer innovations, even after using smart algorithms and state of the art hardware technology. This is partly due to the large amount of data movement expected, but also due to the energy consumption of off-chip communication hardware such as channels, circuit boards, and connectors. To exemplify the point, high performance computing (intra-machine) networks can consume over 20% of total system power even under simple low-communication benchmarks.

Communication Model

The performance of an application developed for a distributed memory environment is influenced by the efficiency of moving data. Orchestration of data transfers for optimal performance may even influence how computations are structured. The interconnect model could be viewed as a simple cost model for the communication. Another view of the interconnect could be based on the physical characterization of hardware components. We will discuss both views, how they relate to each other, and how they interact with programming model primitives.

Communication could be decomposed into a data transfer mechanism and a synchronization mechanism to establish a transfer completion or the consistency of data. Parallel programming models provide multiple mechanisms to achieve data transfer and synchronization. The architectural support for transfers and synchronization typically plays a major role in the efficiency of the programming model in implementing a computational pattern. Some of the programming model primitives are supported at the endpoint of the interconnect through special architectural features in the network interface.

This section presents a simple way of thinking about the interconnect model, followed by a physical characterization. We then discuss the distributed programming model's interaction with the interconnect.

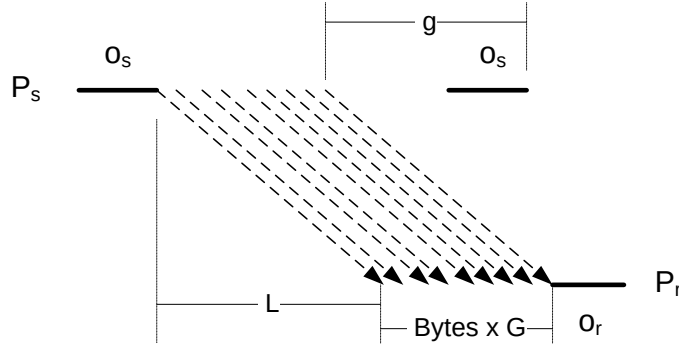


Figure 6.1. Estimated Data Transfer Time, LogGP model

Communication Cost Model

The communication cost model for a transfer is instrumental in structuring a communication and computational pattern. One of the widely accepted models is the LogGP model [3]. This model uses latency L , overhead o , and gap g parameters to estimate the cost of a transfer. The overhead is typically incurred by the runtime software to prepare the transfer, or process it upon arrival. The g gap is the minimum inter arrival time between transfer requests that the hardware can sustain. The latency L is the minimum (uncontended) transfer time between the nodes. The reciprocal of G , time per byte for large transfers, is the bandwidth available between the compute nodes. Figure 6.1 summarizes the model parameters and their influence on the data transfer time. The total transfer time is estimated as the sum of software overheads at the sender (o_s) and the receiver (o_r), the latency L , and transfer size scaled by the inverse of the bandwidth G . In general, the communication could be estimated as $o_s + L + B \times G + o_r$, where B is the transfer size, and the overhead o is being split into two components o_s at the sender and o_r at the receiver.

The parameters in this model are influenced by the interconnect architecture and software implementing the distributed programming model. The interconnect model is described in Interconnect Physical Characterization, while the discussion of the distributed programming model is provided in the Distributed Memory Programming Models section.

The transfer efficiency starts with being limited by the overhead (i.e., message rate) for small transfers. As the transfer size increases, the efficiency increases, but the observed efficiency by the application depends on the software overhead of the programming model runtime. With large transfer granularity the full hardware capabilities can be observed by the application. At the application layer, improving the efficiency of transfers typically involves either aggregation of data or parallel injection. The aggregation could also be provided by the runtime software through, for instance, MPI sparse data types or scatter/gather operations.

Table 6.1 summarizes some of the observed performance values on current systems and projected values in future systems. These measurements are either based on micro benchmarks such as OSU [2], or logGP performance model parameter estimator [1]. While the latency for processing a

	Current^a	Projected
Latency	1-8 μ s ^b	0.3-2 μ s
Bandwidth at the endpoint	8-25GB/s	25-100GB/s ^c
Software Overhead	1-2 μ s	< 1 μ s ^d

^aThe cost of processing a communication request is influenced by the complexity associated with the input parameters.

^bLarge transfer latency is typically observed with compute accelerator technologies attached through PCIe links.

^cThe bandwidth observed by a node could be impacted by the level of contention in the interconnect.

^dThis value might increase if today's software stacks are processed by lightweight cores.

Table 6.1: Current and Projected Communication Model Parameters

simple request is currently around 1 μ s for many MPI APIs, this value could increase significantly depending on architectural setting. For instance, transfers targeting accelerator memory such as GPUs that are traversing a PCIe link, could add 6-7 μ s to the latency. The software stack processing of the software stack could be influenced by the complexity to process a request. For instance, MPI request involving user supplied sparse data types could increase the software latency. Internal memory registration with interconnect hardware could also add multiple microseconds to the processing latency given that it typically involves an expensive operating system call.

The industry keeps introducing techniques to reduce such latency barriers, including tighter integration of accelerator memory, leveraging special direct access support in the PCIe technology, etc.

Overhead reduction trends are typically associated with improvements in CPU performance and offload of communication to the NIC processing engine. Software stacks are growing in complexity due to new features in addition to portability and productivity constraints. Recently, improvement of processor speed has stalled. Instead, many power-efficient architectures use simpler processors that may potentially increase the overhead of processing transfers. A complex API provided by the programming model that can not be assisted by a hardware acceleration mechanism is very likely to run slower in future systems. For more on the impact of the core design in processing transfers, refer to the discussion in Lightweight Cores and Processing Communication Runtime.

Interconnect Physical Characterization

The interconnection network types can be categorized into shared medium, indirect interconnect and direct interconnect. Shared medium connections (bus, rings, etc) are suitable for small scale systems. Most of the scalable interconnects used in supercomputers today are either direct, indirect, or some hierarchical hybrid or combination of these (discussed subsequently). Direct interconnect refers to networks connecting compute nodes directly, meaning every switch has a direct connection to a compute node. Examples of directly connected networks are k-ary n-cube [10], meshes, trees, etc. For instance, Cray XE6 uses a 3D torus interconnect, and IBM BlueGene/Q uses a 5D Torus interconnect. One could think of these direct interconnects as graphs where vertices are the compute nodes and links are the connection between vertices. Figure 6.2 shows two

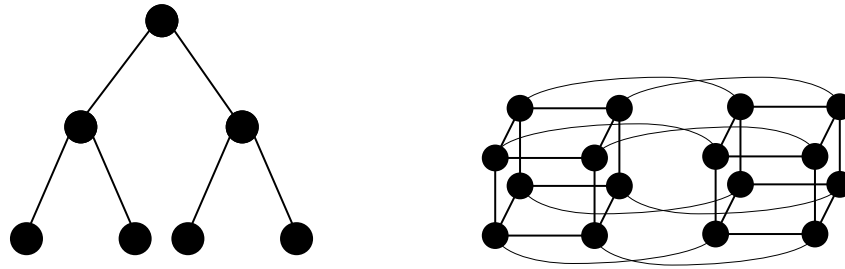


Figure 6.2. Direct interconnection Networks (Tree and 4D Hypercube)

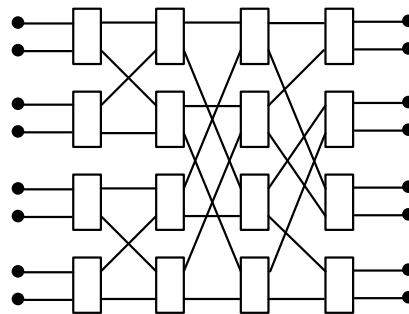


Figure 6.3. Indirect Interconnect (butterfly) using Multiple Stage Switches

examples (tree and 4D hypercube) of directly connected interconnects. Indirect interconnection networks typically use multistage interconnection switches. Examples of indirect interconnects include cross bars, Clos networks, and butterfly (shown in Figure 6.3). Typically compute nodes are connected to a subset of the switches at the edge of the interconnect.

Note that some levels of the interconnect may be within the compute chip. For instance, we may have a shared bus (a ring, or even a 2D torus) on-chip, and have another type of interconnect between nodes. A node may also have multiple chips connected with a special transport. We could also have an indirect network at one level of the hierarchy, shared medium on another level, and direct connection on a third level. The same topology, for instance a tree, could be used in both direct and indirect interconnects.

At the application level, we typically care about the diameter of the network (which affects the latency L parameter in LogGP model), the topology of the interconnect (which impacts the number of neighbors directly reachable from a node), and the bisection bandwidth (which dictates our share of the interconnect bandwidth under highly contended communication). Effectively, the topology of the interconnect makes the parameters in the LogGP model variables depending on the pair of nodes communicating together. Specifically, the latency L between nodes is not constant for all pairs of nodes. For instance, a pair of neighboring nodes will have lower latency than nodes with many hops in-between. The G parameter is influenced by the communication pattern and the run scale. For instance, the observed bandwidth could become a small fraction of the bandwidth available at the endpoint under high contention, for instance when all nodes need to communicate with each others simultaneously. A typical metric to characterize network bandwidth under high contention is the bisection bandwidth (the bandwidth between equal partitions of the system nodes with the narrowest cross section).

Latency typically impacts small transfers. Directly reachable neighbors define how to structure communication to do operations such as multicast in a minimal number of steps. The bisection bandwidth typically limits the performance in all-to-all communication. Obviously, the manner in which the interconnect is used results in varying latency (parameter L in LogGP) and bandwidth. In communication intensive applications, some developers put special care into rank placement in the interconnect to maximize the performance. This can be done manually or through special placement tools [12].

Comprehensive coverage of this topic is beyond the scope of this report. For sake of conciseness, we will focus on a high level description of the components influencing the interconnect performance. We typically focus on three attributes of the interconnect architecture:

- **Link technology:** Multiple technologies could be used to provide connection between nodes (or switches). Table 6.2 summarizes some of these technologies. Generally speaking copper-based links provide a cost effective solution for small to medium range connections, while fiber optic-based links are preferred for their near constant latency attributes at relatively longer distance.
- **Switch capabilities:** Interconnect switches manage how connections are established between nodes or other switches. Circuit switching techniques establish a dedicated connection

Type	Distance Range	Bandwidth ^a	Latency
Copper	few meters	100s Gb/s	tens to hundreds of nano seconds
Fiber optics	tens of meters	100s Gb/s	tens of nano seconds ^b

^aThe actual bandwidth depends on the packet overhead and the supported error checking mechanisms.

^bThe overall latency could be higher for fiber-optics compared with copper for short distance due to the conversion latency between electrical and optical signals.

Table 6.2: Link Technologies and Performance Characteristics

across all hops in the path between two nodes prior to starting the communication and the connection persists as long as it is needed. A more commonly used technique in direct networks is packet switching, which does not entail persistent dedication of resources, thus improves the total system throughput. Another alternative is to use cut-through packet switching, which continually forwards partial transfers as they arrive. In HPC systems, switches typically use packet switching to maximize the throughput of the system (see Table 6.3 for some examples). Multiple switching techniques may be used by the same switch depending on the transfer size.

The radix of the switch (port count) controls the diameter of the interconnects. A high radix router can be used to reduce the number of hops between nodes. There are tradeoffs between increasing the switch radix and the latency of processing requests. The total latency between nodes is the sum of switching and link latencies multiplied by the number of hops.

- **Connection pattern (topology):** The topology influences most of the transfer cost model parameters. In Table 6.4, we summarize the characteristics of different interconnection networks (assuming link bandwidth is constant for all links b ; the system has P processors and routers has port count of r).
- **Network interfaces:** Many of the programming model functionalities are provided by the NIC interface, including the support of atomic, direct memory access (DMA), etc.

Interconnect architectural trends show direct interconnects are likely to continue dominating the future of supercomputing. The diameter of interconnects is decreasing in recent systems as we move from 3D to 5D torus or dragonfly interconnects. High-radix low-diameter interconnects, such as the dragonfly shown in Figure 6.4, are likely to dominate future HPC systems. Nodes are split into groups, with full connectivity within a group. Groups are then fully connected. The number of routers between two nodes is at most three, assuming minimal routing. The bisection bandwidth is typically higher in a dragonfly and a high dimension torus interconnect. The latency per hop is also small (and almost constant across a range of distances). All these positive trends in current interconnect designs could still be outpaced by the rate of producing data, especially as accelerator technologies are used within compute nodes.

	Switching technique	Port count (Radix)	Unidirectional bandwidth per port
Cray Gemini	packet	48	4.68 GB/s
Cray Aries	packet	48	5.25 GB/s
IBM BGQ	packet	10	1.8 GB/s
Mellanox Infiniband	packet	8-648	5-12 GB/s

Table 6.3: Switch Characteristics

	Class	Diameter	Link Count	Hop Range	Bisection Bandwidth	Example System
Bus	Shared	1	P	1	b	
Ring	Shared	$P/2$	P	$1 \rightarrow P/2$	$2 \times b$	
mesh (2D)	Direct	$2 \times (\sqrt{P} - 1)$	$O(\sqrt{P})$	$1 \rightarrow 2 \times \sqrt{P}$	$\sqrt{P}$	Intel Pragon
Torus (3D)	Direct	$O(\sqrt[3]{P})$	$O(P \times \sqrt[3]{P})$	$1 \rightarrow 3 \times \sqrt[3]{P}/2$	$2 \times \sqrt[2/3]{P}$	Cray XE, Bluegene/P
Torus (5D)	Direct	$O(\sqrt[5]{P})$	$O(P \times \sqrt[5]{P})$	$1 \rightarrow 5 \times \sqrt[5]{P}/2$	$O(\sqrt[4/5]{P})$	Bluegene/Q ^a
Torus (6D)	Direct	$O(\sqrt[6]{P})$	$O(P \times \sqrt[6]{P})$	$1 \rightarrow 6 \times \sqrt[6]{P}/2$	$O(\sqrt[5/6]{P})$	K-Computer ^b
Hypercube	Direct	$O(\log_2(P))$	$O(P \times \log_2(P))$	$1 \rightarrow \log_2(P)$	$P/2$	nCube/2
Tree	Indirect	$O(\log_r(P))$	$O(\log_r(P))$	$1 \rightarrow 2 \times \log_r(P)$	$O(1)$	
Fat tree	Indirect	$O(\log_r(P))$	$O(\log_r(P))$	$1 \rightarrow 2 \times \log_r(P)$	$O(P/2)$	Stampede
DragonFly	Hybrid	$O(1)$	$O(P^2)$	$1 \rightarrow 3$	$O(P)$	Cray XC
Fully connect	direct	1	P^2	1	$b \times P^2$	

^aThe size of the the fifth dimension in BGQ is fixed to two.

^bK computer has 3D torus impeded within another 3D torus.

Table 6.4: Direct Network Characterization

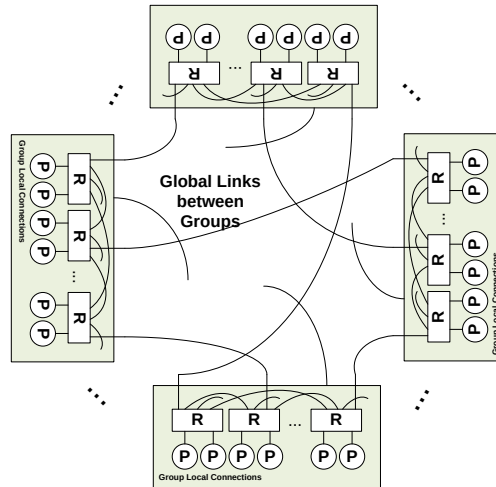


Figure 6.4. Dragonfly Interconnect

Routing Transfer in Direct Interconnect

Commonly, an application rank needs to communicate with multiple peers, or have multiple transfers between peers. Because not all nodes are directly connected, a transfer needs to travel multiple hops before reaching the destination. Additionally, a pair of nodes may have multiple routes for data transfers. Ideally, all possible routes are leveraged to relieve congestion or to bypass faulty links. This requires a class of routing protocols to allow un-ordered delivery between a pair of nodes, called adaptive routing protocols.

A critical question to performance is whether the application is requiring any ordering between these transfers. A common scenario for the need for ordering is when these transfers are used to express synchronization or there is a dependency relation. Programming models typically try to choose a balance between the programming simplicity of ordering guarantees and the performance advantage of unordered delivery. Applications, which are conscious of these tradeoffs, are likely to achieve the best performance on modern systems.

Endpoint Communication Resources

An HPC compute node used to have a single or few processors. Currently, the level of concurrency in core count has increased significantly. The injection of traffic to the network typically requires memory resources for the network interface and the runtime system. These resources, called a communication context or injection endpoint, could be a bottleneck in the process of injecting traffic. Having multiple dedicated resources (contexts) allows parallel initiation of transfers. At the software layer, this implies less frequent synchronization and coordination in communication. On the other hand if part of these resources scale linearly with the number of job endpoints, the memory requirements become a major concern. As such, an application may choose to have few communication endpoints, shared by the compute cores. Some runtime could successfully reduce the requirement of creating these endpoints, thus yielding a scalable solution.

For transfer sizes that are not dominated by overhead, splitting a large transfer between a pair of ranks into multiple transfers allows the interconnect logic to route these transfers across different network routes. Consequently, the interconnect could improve throughput and have more flexibility in congestion management.

Interconnect Technologies and Impact on Application Development

The performance of an application developed for a distributed environment relies heavily on the performance of transferring data between compute nodes. As such, observing the trends in interconnect performance could influence the application development strategy. The potential for improvement of bandwidth is likely to be through a higher degree of parallelism in the topology (higher radix), rather than by increasing the single link bandwidth. This means that application developers may need to consider having concurrent transfers to improve the utilization of the in-

terconnect. Having a low-diameter network means that the latency of the hardware is significantly reduced between the furthest nodes in the interconnect. The implication of this reduction is that an application could observe the full interconnect capabilities at small transfer size. Moreover, the software overhead of communication runtimes needs to be reduced for the application to fully leverage this improvement. The improvement in bandwidth is also likely to scale at a lower rate compared with the computational power of the nodes and their ability to produce data. This implies that applications may need to consider algorithms and computational patterns that reduce the amount of data transfer between nodes. Some performance optimization, such as task placement, could become trickier to leverage though beneficial to performance. The use of high-radix low-diameter interconnects routing will make the latency difference between compute nodes small. Additionally some of the routing protocols that aim to reduce congestion, for instance through adaptive or valiant routing, could use non-minimal routing between compute nodes. This means that node proximity may not necessarily translate to few traversed hops per transfer.

7. System Abstract Machine Model and Proxy Architectures

A system-level abstract machine model (sAMM) comprises components (nodes) and their relative physical layout at a level of abstraction that is appropriate to enable programmers to reason about application implementation on future conceptual Exascale systems. System AMMs also provide OS and runtime developers a framework in which to reason about system software implementation issues before the actual system is built and transitioned to production. In prior chapters, we presented processor and memory AMMs. Our system AMMs will utilize these node-level models in nodes or as collection of nodes with similar functions and capabilities, then organize these nodes into sub-cabinets and cabinets to define a system.

An sAMM should encapsulate available system components (e.g., processor, memory, storage, interconnect, I/O) and their organization (nodes, sub-cabinets, cabinets and their interconnection network fabric) in a way that fundamentally enables understanding of latency domains within and across the system. For example, for application and OS/runtime development, one needs to know which resources are available in a node, which resources are available “close by” or within some local domain associated with some latency and which components are outside a locality and, therefore, have a higher latency. Additionally, a system AMM should be robust enough to represent a variety of contemporary and future systems.

The node-level components (e.g., processor component, memory component, etc.) that comprise an AMM are defined by the various component AMMs (e.g., processor AMMs, memory AMMs, interconnect AMMs) presented in previous chapters. Proxy architectures for these component AMMs are defined as in Chapter 5. The sAMM specifies the number and organization of these component AMMs for nodes, sub-cabinets, and cabinets and how the cabinets are connected to form a system. The resulting latency domains are associated with the structure of the system organization. Therefore, proxy architectures for our sAMMs will simply specify the number and organization of nodes that populate sub-cabinets and cabinets, the total number of cabinets, then the organization of these cabinets across latency domains (i.e. how cabinets are organized and connected across the system). The description of how latency domains are organized and connected can also be provided by specifying the topology of the interconnect fabric as a part of the sAMM proxy architecture.

Given system components, systems may be organized in numerous ways. Therefore, we present a robust system AMM that can represent (through sAMM proxy architectures) past, current, emerging and future HPC systems. Note that what we are presenting here is a model, not a proxy system architecture. In the System Proxy Architectures section, we add specificity to nodes and their organization, we specify the number of nodes per sub-cabinet and cabinet, total number of cabinets and the interconnection network fabric topology of these cabinets to the system AMM to realize system proxy architectures. Our goal is to describe system proxy architectures in conjunction with the corresponding component proxy architectures with enough detail to allow application developers to reason about how their application will map to a system architecture, and

potentially for system architecture designers to simulate/analyze the system.

Partition Model

The partitioned system design as presented in [30] describes an organization in which system services are partitioned separately from compute resources. This not only refers to hardware resources, but to software components as well (e.g., OS facilities). We adopt this partition model in our system AMM and picture a system as an integrated set of nodes that are organized into partitions that are defined by their capabilities. Within a service partition, these resources may be described as Login nodes, I/O nodes, compilation nodes, system management nodes, etc. In contrast, a compute partition may make up the majority of the system and there may be new classes of nodes such as burst buffer or non-volatile memory nodes. The partition model will also provide a structure for analyzing the implications of advanced workflows in which analysis and visualization partitions may require new integration (via the interconnection network fabric) to all or part of the compute partition while a job is still running.

System AMM Components

Our system abstract machine model (sAMM) comprises all of the AMMs that were previously presented (processor, memory, I/O and Storage, and interconnect). The sAMM hierarchically organizes (in nodes, sub-cabinets, cabinets) these component AMMs into a system model as shown in Figure 7.1. Note again that this is an abstracted model and, therefore, the particular components that comprise a node, the particular node configurations that comprise a sub-cabinet and cabinet will specify a particular system. We will present some specific sAMM proxy architectures for existing and future systems in the following sections. Several proxy architectures can be defined for a single sAMM and are presented in System Proxy Architectures. In the following sections, we first define the hierarchical building blocks of the system AMM, namely the node, sub-cabinet, and cabinet. We then describe each of the component AMMs that are used to build the sAMM and any caveats associated with these components. Finally, we discuss how component AMMs can be organized at the node, sub-cabinet and cabinet levels to model various systems.

The Processor and Memory AMM System Components

The processor AMM component shown in Figure 7.1 can be any of the processor AMMs shown in Figures 3.1 - 3.5. Specifying compute nodes that comprise different types of processor cores is done at the Processor AMM level in that this is characterized in the actual processor model. The Processor AMM in Figure 3.1 is a model that has fat and thin processor cores; Figures 3.4 and 3.5 are multicore models that have cores of varying compute capability. It is imaginable that one could combine two distinct Processor AMMs with different compute capabilities into a single

node, particularly if standard compute and compute for visualization are combined in the same node. It is conceivable that this might utilize two different processor configurations.

The memory component in a system AMM can be any of the memory technologies presented in the Memory System section. If a processor component has on-package memory, it is specified in the processor AMM (e.g., Figure 3.1). Notice that the low capacity, high bandwidth memory shown in Figure 3.1 is on-package, but there is also off-package DRAM and NVRAM. In this document, we picture all memory in the same way whether it sits on package or off. The type of memory technology (e.g., DRAM, NVRAM, PCM, etc.) is reflected in the latency and bandwidth specifications that are defined for a particular proxy architecture. Our only distinction is if they are directly accessible or indirectly through a hierarchy.

The Interconnect Component

The interconnect component is essentially as outlined in Tables 6.2–6.4. We have abstracted the parameters shown in these tables to those that are of primary importance to an application program when reasoning about porting an application to future hardware and understanding its performance. This table of abstracted interconnect characteristics is presented in System Proxy Architectures below.

Nodes, Sub-Cabinets, and Cabinets

In the system AMM of Figure 7.1, *Locality 0* contains an unspecified number of *cabinets*, which are shown as layered boxes, in which some number of *nodes* are contained. A compute node, as shown in the figure, contains a processor (*Processor AMM*, that may or may not contain in-package memory), off-package memory (*memory AMM*), and *X*, which could be more off-package memory potentially used as burst buffer, storage memory, viz/specialized compute, or potentially another Processor AMM that implements a different type of core/compute architecture. Because nodes are essentially comprised of various component AMMs, many possible node architectures can be represented. This sAMM is representative of many contemporary and future systems of interest. These systems will be defined by proxy system architectures in the System Proxy Architectures section.

A *sub-cabinet* is simply a portion of a cabinet, as one would expect. It might be the case that part of the cabinet is populated with certain types of nodes (e.g., compute), another portion with another type of node (e.g., I/O). Although we do not picture this in Figure 7.1, some systems could potentially be implemented this way. The number and type of nodes per sub-cabinet and/or cabinet and the total number of cabinets is a parameter to be specified in the proxy system architectures.

System Proxy Architectures

The following is a more complete list of parameters to allow application developers and hardware architects to tune any AMMs to their desire. The list is not exhaustive and will continue to grow as needed. Since this list is for all AMMs presented in this document, not all parameters are expected to be applicable to every AMM. In fact, we expect that for each AMM only a subset of this list of parameters will be used for architecture tuning. Likewise, not all parameters are useful for application developers, such as bandwidth of each level of the cache structure.

Processor

Parameter	Expected Range	Notes
Bandwidths (GB/s)		
Chip ↔ Memory	60–100	To off-chip DDR DRAM
Chip ↔ Memory	600–1200	To on-chip DRAM
Chip ↔ Memory	600–1200	To off-chip HMC-like
Core ↔ L1 Cache	O(100) – O(1000)	
Core ↔ L2 Cache	O(100)	
Core ↔ L3 Cache	Varies by NoC Bandwidth	
Capacities		
L1 Cache	8KB–128KB	Per Core
L2 Cache	256KB–2MB	Per Core
L3 Cache	64MB–128MB	Likely to be shared amongst groups of cores/accelerators
L4 Cache	2GB–4GB	Not on all systems, likely to be off-package embedded-DRAM
Memory System Parameters		
Cache block size	64-128B	
Number of cache levels	2–4	
Coherency domains	1–8	per chip
Network-On-Chip		
Topology	Options include: Mesh, fat-tree, hierarchical ring, cross-bar, etc. Still active research area.	
Bandwidth	8-64 GB/s	per core
Latency	1–10 ns	neighbor cores
Latency	10–50 ns	cross chip
Core Parameters		
Number of cores	64-256	per chip
Threads per core	2–64	
Thread Switching	Possible policies include: Each cycle, on long-latency event, time quanta	
SIMD/vector width	4–8 DP FP	
Dispatch, issue, execution widths	2–8	Simple processors may have limited dual dispatch (i.e. 1 general purpose instruction and 1 memory)
Max references outstanding	8–128	per core
Atomic operations		Possible implementations include: Simple or transactional

Memory

Parameter	Range	Notes
Capacities		
In-package DRAM Memory	32–64 GB	
Off-Chip DRAM Memory	~512GB–2TB	Memory capacity may be much larger, at the cost of fewer nodes. See notes in Section 4
Off-Chip NVRAM Memory	~2TB–16TB	
Capabilities		
Extended Memory Semantics	None, Full/Empty Bits, Transactional, data movement, specialized compute, general compute.	
In-Package Memory Technologies/Levels		
HBM/Wide-IO	~16GB per stack, ~ 200GB/s per stack. Can be stacked directly on processor. Cannot be “chained.”	
Off-Package Memory Technologies/Levels		
HMC	~16GB per stack, ~240GB/sec per stack. Can be “chained” to increase capacity. Multiple access sizes.	
DDR-4	~64GB per DIMM, 20GB/s per channel, up to 2 DIMMs per channel. Optimized for cacheline sized access.	
NVRAM	~4-8× capacity of DRAM, 10–20GB/s. Requires KB sized access. Highly assymetric access latency.	

Node Architecture

	Processor Cores	Cores/NUMA Region	Gflop/s per Proc Core	# Threads per Core	Processor SIMD Vectors (Units x Width)	Accelerator Cores	Acc Memory BW (GB/s)	Acc Count per Node
Homogeneous M.C. Opt1	256	64	64	2	8x16	None	None	None
Homogeneous M.C. Opt2	64	64	250	4	2x16	None	None	None
Discrete Acc. Opt1	32	32	250	8	2x16	O(1000)	O(1000)	4
Discrete Acc. Opt2	128	64	64	4	8x16	O(1000)	O(1000)	16
Integrated Acc. Opt1	32	32	64	8	2x16	O(1000)	O(1000)	Integrated
Integrated Acc. Opt2	128	64	16	4	8x16	O(1000)	O(1000)	Integrated
Heterogeneous M.C. Opt1	16 / 192	16	250	8 / 1	8x16 / 2x8	None	None	None
Heterogeneous M.C. Opt2	32 / 128	32	64	4 / 1	8x16 / 2x8	None	None	None
Concept Opt1	128		50		12x1	128	O(1000)	Integrated
Concept Opt2	128		64		12x1	128	O(1000)	Integrated

Note that *Opt1* and *Opt2* represent possible proxy options for the abstract machine model. *M.C.*: multi-core, *Acc.*: Accelerator, *BW.*: bandwidth, *Proc.*: processor, For models with ac-

celerators and cores, C denotes to FLOP/s from the CPU cores and A denotes to FLOP/s from Accelerators.

System Network

Parameter	Range	Notes
Topology	Possible implementations include: Torus, fat-tree, hierarchical ring, or Dragonfly	
Bisection Bandwidth		1/8 to 1/2 of injection bandwidth
Injection BW	100GB/s - 400GB/s	per node
Messaging Rate	250MMsg/s	Two-sided communications
Message Processing Rate	1BMsg/s	One-side communications
Latency	500-1500ns	Two-side communications nearest neighbor
Latency	400-600ns	One-sided communications nearest neighbor
Latency	3-5 μ s	Cross-machine

System Organization

Parameter	Range	Notes
Total Node Count	33K - 125K	
Nodes per Rack		
# Viz nodes		Must define locality in which nodes reside
# Compute nodes		Must define locality in which nodes reside
# Login nodes		

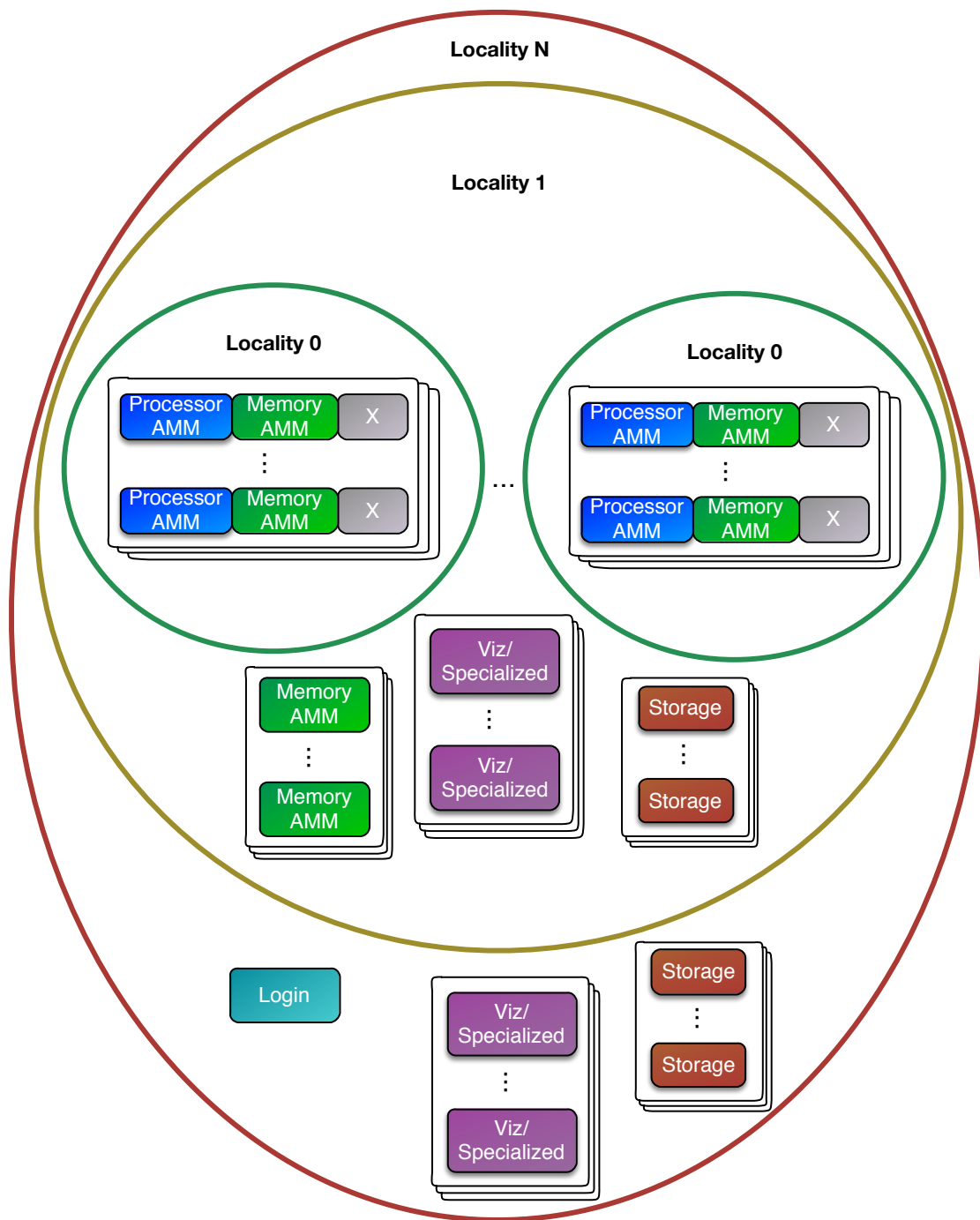


Figure 7.1. System AMM

8. Conclusion

Advanced memory systems, complex on-chip networks, heterogeneous systems and new programming models are just some of the challenges facing users of an exascale machine [21]. The AMMs presented in this document are meant to capture the design trends in exascale systems and provide application software developers with a simplified yet sufficiently detailed view of the computer architecture. This document also introduces and describes proxy architectures, a parameterized instantiation of a machine model. Proxy architectures are meant to be a communication vehicle between hardware architects and system software developers or application performance analysts.

Communication and coordination: The Computer Architecture Laboratory (CAL) has an important role in facilitating communication between laboratory open research projects and hardware architecture research and development by DOE's Fast Forward and Design Forward companies that is usually proprietary. CAL supports this communication by developing AMMs and non-proprietary, open proxy architectures. These proxy architectures and AMMs can be used to provide applications, algorithms and co-design projects with a target model of the hardware, node, and system architecture to guide their software development efforts. An important characteristic of these proxy architecture models is their ability to also capture lower level architectural parameters to provide a non-proprietary description of advanced architecture concepts for quantitative analysis and design space exploration with our architectural simulation tools. The development of proxy architectures and AMMs is intended to allow open exchange of advanced hardware architecture concepts without disclosing intellectual property.

Metrics for success: The key metric for success will be the use of AMMs at the abstract high level for guiding application development, and proxy architectures at the more detailed lower level for guiding system software and architectural simulation efforts. The successful outcome is co-designed hardware architectures and associated software.

AMM and proxy architecture evolution: Co-design is a multi-disciplinary endeavor. It is an activity that bridges many different technical communities, and as we have seen with proxy applications, having common terminology and touch points is extremely valuable in fostering multi-disciplinary collaborations. In this spirit, the CAL project is contributing this initial set of common definitions for AMMs and their associated proxy architectures to the DOE exascale program. As with proxy applications, our AMMs and Proxy Architectures are expected to evolve through the co-design process.

References

- [1] Netgauge loggps (logp, loggp) measurement. Web. <http://htor.inf.ethz.ch/research/netgauge/loggp/>.
- [2] Ohio state (osu) micro-benchmarks. Web. <http://mvapich.cse.ohio-state.edu/performance/>.
- [3] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.
- [4] R.F. Barrett, S.D. Hammond, C.T. Vaughan, D.W. Doerfler, M.A. Heroux, J.P. Luitjens, and D. Roweth. Navigating an Evolutionary Fast Path to Exascale. In *High Performance Computing, Networking, Storage and Analysis (SC12)*, pages 355–365. IEEE, 2012.
- [5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012.
- [6] R. Brightwell and K.D. Underwood. An Analysis of NIC Resource Usage for Offloading MPI. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 183–, April 2004.
- [7] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.
- [8] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
- [9] Byn Choi et al. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, Washington, DC, USA, 2011. IEEE Computer Society.
- [10] W.J. Dally. Performance analysis of k-ary n-cube interconnection networks. *Computers, IEEE Transactions on*, 39(6):775–785, Jun 1990.
- [11] Jack Dongarra et al. The International Exascale Software Project Roadmap. *IJHPCA*, 25(1):3–60, 2011.
- [12] Todd Gamblin. Rubik Task Mapping Generation Tool, 2015.
- [13] JEDEC. Wide I/O Single Data Rate. JESC 229, JEDEC, December 2011. <http://www.jedec.org/standards-documents/docs/jesd229>.

- [14] JEDEC. High Bandwidth Memory (HBM) DRAM. JESC 235, JEDEC, October 2013. <http://www.jedec.org/standards-documents/docs/jesd235>.
- [15] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX: An Advanced Parallel Execution Model for Scaling Impaired Applications. In *International Conference on Parallel Processing Workshops*, pages 394–401. IACC, 2009.
- [16] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design: opportunities and challenges. In *DAC*, pages 1153–1158, 2012.
- [17] S. Kaxiras and G. Keramidas. Sarc coherence: Scaling directory cache coherence in performance and power. *Micro, IEEE*, 30(5):54–65, Sept 2010.
- [18] Martha Kim. Scaling Theory and Machine Abstractions. <http://www.cs.columbia.edu/~martha/courses/4130/au13/pdfs/scaling-theory.pdf>, Sept 2013.
- [19] Peter M. Kogge and John Shalf. Exascale computing trends: Adjusting to the "new normal" for computer architecture. *Computing in Science and Engineering*, 15(6):16–26, 2013.
- [20] Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. Openshmem - toward a unified rma model. *Encyclopedia of Parallel Computing*, pages 1379–1391, 2011.
- [21] Robert Lucas et. al. Top Ten Exascale Research Challenges, DOE ASCAC Subcommittee Report, February 2014.
- [22] A. Ros, B. Cuesta, M.E. Gomez, A. Robles, and J. Duato. Cache miss characterization in hierarchical large-scale cache-coherent systems. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 691–696, July 2012.
- [23] V. Sarkar, B. Chapman, W. Gropp, and R. Knauserhase. Building an Open Community Runtime (OCR) framework for Exascale Systems, 2012.
- [24] M. Schuchhardt, A. Das, N. Hardavellas, G. Memik, and A. Choudhary. The impact of dynamic directories on multicore interconnects. *Computer*, 46(10):32–39, October 2013.
- [25] John Shalf, Sudip S. Dosanjh, and John Morrison. Exascale computing technology challenges. In *VECPAR*, pages 1–25, 2010.
- [26] Micron Technology. Hybrid Memory Cube Specification, 2014.
- [27] K.D. Underwood, K.S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A Hardware Acceleration Unit for MPI Queue Processing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 96b–96b, April 2005.
- [28] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

- [29] Brian van Straalen, John Shalf, Terry J. Ligocki, Noel Keen, and Woo-Sun Yang. Scalability challenges for massively parallel amr applications. In *IPDPS*, pages 1–12, 2009.
- [30] D.E. Womble, S.S. Dosanjh, B. Hendrickson, M.A. Heroux, S.J. Plimpton, and J.L. Tomkins. Massively parallel computing: A sandia perspective. *Parallel Computing*, 25(13-14):1853–1876, 1999.
- [31] Yi Xu, Yu Du, Youtao Zhang, and Jun Yang. A composite and scalable cache coherence protocol for large scale cmps. In *Proceedings of the International Conference on Supercomputing*, ICS ’11, pages 285–294, New York, NY, USA, 2011. ACM.
- [32] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. Upc++: a pgas extension for c++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114. IEEE, 2014.

DISTRIBUTION:

1 MS 0899 Technical Library, 9536 (electronic copy)

