SANDIA REPORT

SAND2008-3205
Unlimited Release
Printed May 2008

Summary of Multi-Core Hardware and Programming Model Investigations

Kevin Pedretti, Suzanne Kelly, Michael Levenhagen

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

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

Sandia National Laboratories

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

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

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

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

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

SAND2008-3205
Unlimited Release
Printed May 2008

Summary of Multi-Core Hardware and Programming Model Investigations

Kevin Pedretti, Suzanne Kelly, and Michael Levenhagen
Scalable System Software Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
{ktpedre,smkelly,mjleven}@sandia.gov

Abstract

This report summarizes our investigations into multi-core processors and programming models for parallel scientific applications. The motivation for this study was to better understand the landscape of multi-core hardware, future trends, and the implications on system software for capability supercomputers. The results of this study are being used as input into the design of a new open-source light-weight kernel operating system being targeted at future capability supercomputers made up of multi-core processors. A goal of this effort is to create an agile system that is able to adapt to and efficiently support whatever multi-core hardware and programming models gain acceptance by the community.

Acknowledgment

This work was funded by Sandia's Laboratory Directed Research and Development (LDRD) program. The authors thank Neil Pundit for his review of the manuscript and feedback. The description of the BEC (Bundle Exchange Compute) programming model was provided by Zhaofang Wen.

Contents

List of Figures

List of Tables

Nomenclature

ALP Architecture-Level Parallelism. Processors that contain a heterogeneous mixture of core architectures exhibit ALP. As more cores are added to a single processor, it can be beneficial from a power and area standpoint to provide some heavy cores oriented towards single thread performance and other simpler cores oriented towards highly parallel workloads. Another example of ALP is integration of specialized cores, such as graphics processing units, and general-purpose cores on the same chip.

CLP Core-Level Parallelism. CLP occurs when a single processor core provides support for multiple hardware threads. Hardware threads differ from OS-managed software threads in that they have much lower overhead and can be switched between one other (usually) on a cycle-by-cycle basis. This allows another thread to execute when the current thread stalls (e.g., on a memory operation), thus making more efficient use of the processor's resources. CLP is a technique for increasing memory-level parallelism (MLP).

ILP Instruction-Level Parallelism. Superscalar processors are capable of executing more than one instruction each clock cycle. ILP is a form of implicit parallelism that is usually identified by the hardware automatically (e.g., via out-of-order execution) or by the compiler (e.g., scheduling instructions for multiple-issue, VLIW). The programmer does not usually need to explicitly deal with ILP.

MLP Memory-Level Parallelism. Modern memory subsystems have very high latency with respect to processor speeds, and the gap is increasing. MLP occurs when multiple simultaneous requests are made to the memory subsystem. This allows pipelining to occur, thus hiding the latency. Architectural capabilities such as out-of-order execution, hardware multi-threading, and STREAM processing are all aimed at increasing MLP.

SIMD Single Instruction Multiple Data. SIMD parallelism occurs when a single instruction operates on multiple pieces of data. Vector processors provide SIMD parallelism—a single instruction operates on entire vectors of data at a time. Many modern commodity processors support SIMD parallelism through multi-word wide registers and instruction set extensions. Many compilers attempt to automatically detect SIMD parallelism and exploit it, however programmers often have to explicitly structure their code to get the maximum benefit.

SMT Symmetric Multi-Threading. SMT is a form of CLP (core-level parallelism). In an SMT processor, instructions from multiple hardware threads can be issued in the same clock cycle to different execution units of a superscalar processor. This is in contrast to fine-grained multi-threading where only instructions from one thread may execute

each clock cycle. SMT is useful on very wide superscalar processors (e.g., the Power6) where it is unlikely that a single thread will use all of the available execution units.

TLP Task-Level Parallelism. Multi-core processors support running more than one context of execution (e.g., a process, a thread) at the same time, one per core. Programmers must explicitly structure their application to break the problem into multiple pieces in order to utilize TLP. This is in contrast to ILP, which requires no action on the programmers part. Scientific applications for distributed memory supercomputers have very high levels of TLP.

Chapter 1

Introduction

This report summarizes our investigations into multi-core processors and programming models for parallel scientific applications. The motivation for this study was to better understand the landscape of multi-core hardware, future trends, and the implications on system software for capability supercomputers. The information gathered by this study is being used as input into the design of a new open-source light-weight kernel operating system targeted at future capability supercomputers made up of multi-core processors.

Figure 1.1 predicts the number of processor cores in a future exa-scale supercomputer based on data for past ASC capability systems. In the past, single core performance has increased with Moore's Law (green line). Going forward, single processor core performance is not expected to increase significantly, leading to an exponential increase in the number of cores needed to achieve the desired performance level (blue line). This poses challenges for system software in the areas of scalability, resiliency and programmability. The shear scale of future systems necessitate much improved resiliency, or else the majority of an application's runtime will be spent in overhead due to checkpoints and restarts [32]. Programming models are another area requiring attention. Not only will future exa-scale systems have vastly more processor cores than today's systems, they are expected to contain heterogeneous processors and much more complex memory hierarchies. Each compute node essentially becomes a massively parallel processor of its own—e.g., an Intel TFLOPS on a chip. This is motivating renewed interest in two-level or more parallel programming models that seek to exploit intra-node or even intra-chip locality. Future operating systems for capability systems must provide the functionality necessary to address these issues.

The remainder of this report is organized as follows: Chapter 2 examines the characteristics of several current and future multi-core processors. Chapter 3 examines programming models for capability systems and identifies the system software requirements of each. Finally, Chapter 4 includes closing remarks.

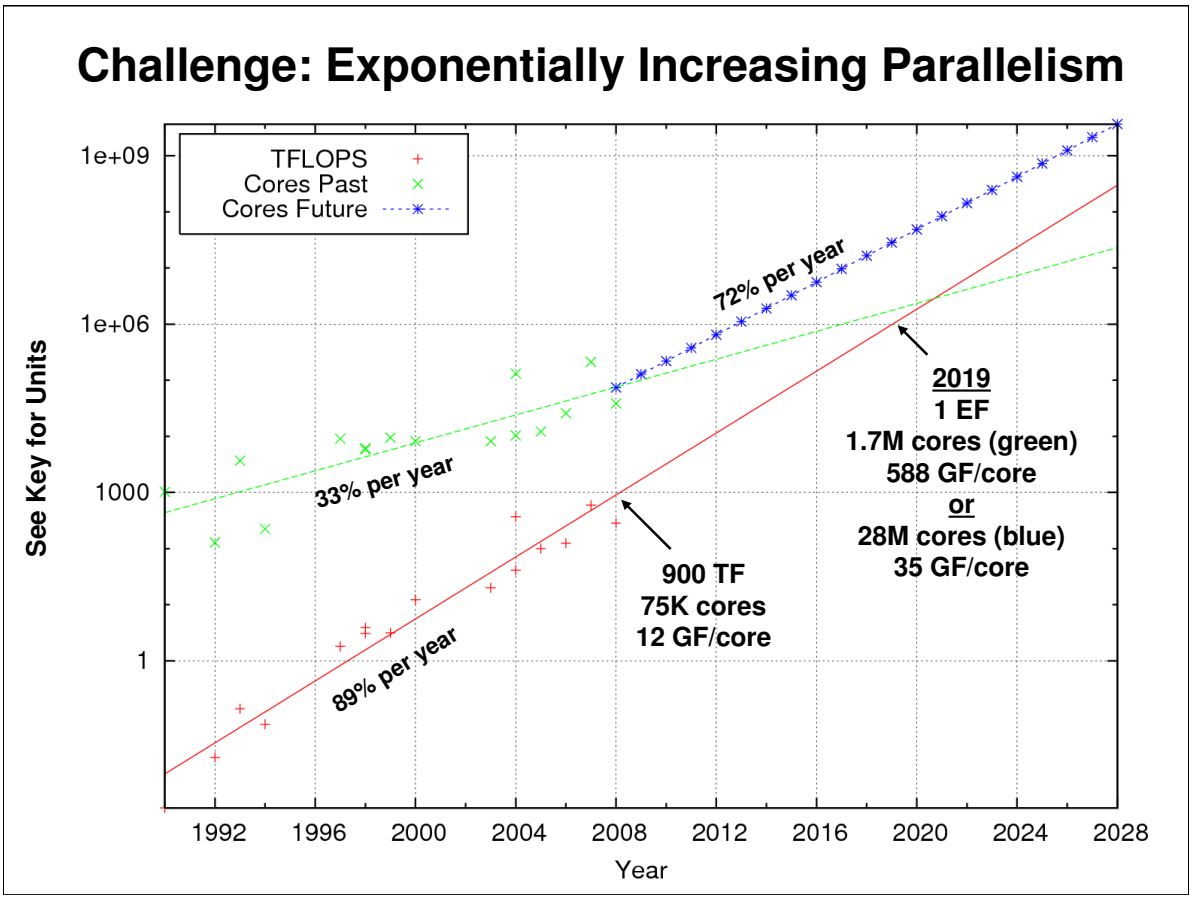

Figure 1.1. ASC Capability System Scaling Trends

Chapter 2

Multi-core Hardware

This chapter provides an overview of several current multi-core processors. General-purpose processors are described first, followed by a description of several special-purpose multi-core processors aimed at specific markets such as embedded computing.

2.1 General-purpose Processors

General-purpose processors are designed to be used in mainstream servers and desktops. They are also commonly used in distributed memory high performance computing systems ranging from small-scale Beowulf clusters to capability supercomputers such as 38,400 core Red Storm Cray XT4 system at Sandia National Laboratories. Most existing scientific computing applications simply need to be recompiled to utilize any of these processors with good performance. In some cases tuning or restructuring an application can result in significant performance gains (e.g., restructuring to expose SIMD parallelism). The Cell processor described in Section 2.1.3.3 is an extreme example—significant code restructuring *must* be performed in order to take advantage of the on-chip SPE accelerators, which deliver the bulk of the processor's performance.

The remainder of this section briefly describes several multi-core processor product families. The types of parallelism supported by each processor is listed at the end of each processor's description (see the Nomenclature section for a description of acronyms). This information can be helpful to application developers seeking to tune their application for a new processor. It is also interesting to compare the forms of parallelism provided in relation to the processors intended usage. The Sun processors, for example, do not support ILP or SIMD because they would not benefit the high-throughput applications that the chip is intended for. Finally, the parameters of a specific model of each multi-core processor family are listed in a table at the end of each processor's description.

2.1.1 AMD

2.1.1.1 Dual-core Opteron

The AMD dual-core Opteron [19] processor contains two 64-bit x86 CPUs and an integrated memory controller. AMD was first to the market with 64-bit x86 CPUs and also dual-core processors, although Intel now has similar offerings. Each AMD CPU has its own dedicated L1 and L2 caches and supports one context of execution. Each Opteron CPU is capable of performing two floating point operations per clock. The processor's on-chip memory controller supports two ganged DDR2 PC5300 channels, providing a maximum of 10.6 GB/s of memory bandwidth.

Forms of parallelism supported: ILP, SIMD, TLP, MLP

2.1.1.2 Quad-core Opteron

AMD has developed a quad-core Opteron [10] follow-on to its dual-core Opteron. This processor integrates four 64-bit x86 CPUs on one processor die and includes an integrated memory controller. Each CPU has its own dedicated L1 and L2 caches, in addition to a common L3 victim cache shared by all CPUs. The victim cache contains cache lines evicted from the private caches, buffering them before they are moved to main memory. Additionally, the cache controller attempts to detect lines that are being shared by multiple CPUs and, when detected, treats them differently than non-shared lines (e.g., by using an optimized coherency protocol).

Thanks to an improved SSE (Streaming SIMD Extensions) unit, each Opteron CPU can perform four floating point operations per clock cycle, up from two in the dual-core Opteron. SSE registers remain 128-bits wide, but both 64-bit fields can be operated on simultaneously, rather than sequentially in the dual-core Opteron. One multiply-accumulate (two FLOPs) can be performed on each 64-bit field each clock cycle.

The on-chip memory controller supports two DDR2 PC5300 channels, providing a maximum of 10.6 GB/s of memory bandwidth (DDR3 is also supported by the memory controller, but will not be available until a future processor socket). The two channels may either be operated in ganged or unganged mode, configurable on a per-boot basis. Ganged mode results in a 128-bit wide path to memory and operates identically to the dual-core Opteron. In unganged mode, each channel operates independently, effectively behaving as two 64-bit memory controllers. This allows for greater memory level parallelism (MLP), but has the disadvantage that chip-kill ECC mode is not supported due to the 128-bit error correcting code used.

Forms of parallelism supported: ILP, SIMD, TLP, MLP

2.1.1.3 Examples

Table 2.1 provides a comparison of specific instances of the processor families described in this section. The models were chosen to be roughly comparable to one another for a given generation (e.g., the AMD dual-core and Intel dual-core processors are comparable models from different vendors).

2.1.2 Intel

2.1.2.1 Dual-core Core2 Xeon

The Intel Core2 Xeon contains two 64-bit x86 CPUs. Each CPU has its own dedicated L1 cache and the two CPUs in the same chip share access to a common L2 cache. This facilitates fine-grained data sharing between CPUs at the expense of higher cache arbitration overhead. Each CPU supports one context of execution. Each Core2 CPU can perform four floating point operations per clock.

Unlike the AMD Opteron platform, the Intel platform utilizes an external chipset to supply the memory controller and Northbridge functionality. The front-side bus connecting the processor to the chipset operates at a maximum of 1333 MHz, providing 10.6 GB/s of bandwidth to the memory controller in the chipset.

Forms of parallelism supported: ILP, SIMD, TLP, MLP

2.1.2.2 Quad-core Core2 Xeon

The Intel Core2 Xeon quad-core processor is essentially two Core2 dual-core processors packaged together into a multi-chip module. An arbitrator circuit gates access to the front-side bus by the two dual-core dies, enabling the aggregate processor package to appear as a single load on the front-side bus. The cache hierarchy and front-side bus bandwidth are identical to the Core2 dual-core. The bytes/FLOP ratio is therefore decreased by a factor of two for each socket.

Forms of parallelism supported: ILP, SIMD, TLP, MLP

2.1.2.3 Upcoming Nahalem Architecture

At the time of writing, Intel has announced their next generation processor, code-named Nahalem [17]. Nahalem will initially contain four 64-bit x86 CPUs integrated on a single die, although future iterations may contain more CPUs. The Nahalem processor is the first x86 processor from Intel with more than two cores on the same chip. Each Nahalem CPU will

support two hardware threads, which will share and compete for the CPU's resources. Each quad-core processor will look like eight CPUs to the operating system, although mechanisms are provided to determine which virtual CPUs share the same physical CPU. It is likely that Intel will eventually package two quad-core chips into a single package, resulting in 8 CPUs (16 threads) per processor socket.

Like AMD's quad-core processor, Intel has moved to a three-level cache hierarchy, up from two levels in the Intel Core2. Each physical CPU will have dedicated L1 and L2 caches and share access to a common L3 cache.

This will be the first mainstream Intel x86 processor to integrate a memory controller on the processor chip, bringing it in line with the AMD Opteron processors. Initial indications are that each processor will contain three DDR3 memory controllers, each 8-bytes wide and operating at 1.33 GT/s. This results in a peak memory bandwidth of 31.92 GB/s. The integrated memory controller is expected to provide a significantly lower latency to memory than the previous Core2-based quad-core processors—initial estimates are around 60 ns for accesses to local memory, compared to 100 ns previously. This is in-line with the performance provided by AMD quad-core processors.

Nahalem will support glueless multi-processing via a new processor-to-processor communication channel called QuickPath Interconnect (QPI). QPI serves a similar purpose as the HyperTransport links on the AMD Opteron. Each Nahalem processor will include up to four point-to-point QPI channels, each channel providing 12.8 GB/s of bandwidth in each direction.

Forms of parallelism supported: ILP, SIMD, TLP, MLP, CLP

2.1.2.4 Examples

Table 2.1 provides a comparison of specific instances of the processor families described in this section. The models were chosen to be roughly comparable to one another for a given generation (e.g., the AMD dual-core and Intel dual-core processors are comparable models from different vendors).

2.1.3 IBM

2.1.3.1 Power5

The IBM Power5 [16] processor contains two cores, each supporting two hardware threads. SMT (Simultaneous Multi-threading) is employed to allow a mixture of non-conflicting instructions from a core's two threads to be executed each clock cycle. Each core has a dedicated L1 cache and both cores share a common L2 cache. An optional shared L3 cache is provided on some platforms–the L3 cache controller and tag directory are on the Power5

chip while the L3 memory is off-chip. The Power5 includes an on-chip memory controller that is connected to memory by two uni-directional buses operating at twice the DRAM frequency. The read bus is 16 bytes wide and the write bus is 8 bytes wide. Using DDR2-533 memory, this results in 17.1 GB/s of read bandwidth and 8.5 GB/s of write bandwidth.

Each core has eight execution units, allowing up to eight instructions to be executed each clock cycle. Instructions are issued out-of-order. Up to five instructions can be completed and retired in-order each clock cycle. Since a single thread is unlikely to use all of the Power5's execution units each clock cycle due to limits in ILP, SMT is beneficial for many workloads. LLNL has observed SMT performance benefit of -20% to +60% on a mix of scientific applications on the Power5-based ASC Purple system–the sPPM and UMT2K codes showed a benefit of 20-22%.[1]

SMT threads are exposed to the OS as virtual processors, meaning a single Power5 chip appears to the OS as four CPUs. A unique capability of Power5 is SMT thread priorities. Each SMT thread can be assigned one of eight priorities with higher priorities granting greater access to the shared execution resources. It is unclear whether applications can set their own priorities at user-level or if the OS must be involved. Additionally, the Power5 supports a single-threaded mode where all hardware resources are dedicated to a single thread (e.g., rename registers, reorder buffers, issue queues). The OS can enter and exit single-threaded mode dynamically at run-time by setting one thread's priority to 0. This is beneficial for applications that do not benefit from SMT.

Forms of parallelism supported: ILP, TLP, MLP, CLP

2.1.3.2 IBM Power6

The IBM Power6 [24] processor is the successor to the Power5. Like the Power5, it contains two cores, each supporting two hardware threads with configurable priorities. Each core has its own dedicated L1 cache and L2 caches. Like the Power5, there is an optional shared L3 cache with on-chip controller and off-chip memory. Two on-chip memory controllers together provide 16-byte read and 8-byte wide write paths to memory each cycle. When operated at 3.2 GHz (800 MHz DDR2 DRAM), this provides 51.2 GB/s for reads and 25.6 GB/s for writes. Other changes include the addition of AltiVec support (i.e., short-vector SIMD instructions) and a mostly in-order processor design, although floating point instructions may execute out-of-order. Instruction completion bandwidth has been improved to seven instructions per cycle, up from five in the Power5. Finally, error recovery has been significantly improved by the addition of checkpoint and restart circuitry. When an error occurs, checkpoint data is used to transparently retry an operation. If the error persists, the checkpoint data can be moved to a spare-core in the system and restarted. It is not clear if the OS needs to be involved in the process.

Forms of parallelism supported: ILP, SIMD, TLP, MLP, CLP

[1] https://computing.llnl.gov/tutorials/purple/

2.1.3.3 STI Cell Broadband Engine

The STI (Sony, Toshiba, IBM) Cell Broadband Engine [35] processor is a radical departure from the other processors discussed in this section. Unlike the other processors, which use relatively straight-forward replication of general-purpose cores, Cell is a heterogeneous chip containing one general-purpose PowerPC core (PPU, PowerPC Processing Unit) and eight DSP-like Synergistic Processor Units (SPU). An on-chip memory controller provides 25.6 GB/s of bandwidth to main memory. A normal operating system such as Linux runs on the PPU while the SPUs are too simplistic to run an OS. Applications running on the PPU off-load work onto the SPUs.

The SPUs deliver the bulk of the chip's performance, but must be programed with a new programming model that is much more constraining than what general-purpose processor developers are accustomed to. Each SPU executes entirely out of a small (256 KB) on-chip SRAM memory. While this results in very deterministic behavior, the disadvantage is that data in main memory must be explicitly moved to and from the SPU's memory using DMA commands that are part of the SPU's instruction set. Several efforts are underway to hide or reduce this burden on application developers, but often at a cost of lower performance and reduced flexibility.

Researchers have found the Cell to provide excellent performance and power efficiency on several scientific computing kernels [39]. This is tempered by the need to essentially create new implementations of the algorithms for the Cell, often using primitive tools and assembly code. The researchers observed that once they passed the initially steep learning curve, programming for the Cell was not that much more difficult than programming for a modern cache-based superscalar processor. The Cell has the advantage that SPU programs execute nearly deterministically, making performance tuning more straight-forward.

Forms of parallelism supported: SIMD, TLP, MLP, CLP, ALP

2.1.3.4 Examples

Table 2.2 provides a comparison of specific instances of the IBM and STI processor families described in this section.

2.1.4 Sun

2.1.4.1 UltraSPARC T1

The Sun UltraSPARC T1 [20] is the first generation of Sun's Niagara platform, which is targeted at high throughput work-loads such as web servers and transaction processing rather than single thread performance. Each UltraSPARC T1 processor contains eight cores, each supporting four in-order hardware threads. Fine-grained multi-threading is used to switch

between threads on a per cycle basis with one thread per core capable of issuing one instruction each cycle. All cores on a chip share a single floating-point unit, making the processor unattractive for high performance computing applications. The follow-on UltraSPARC T2 processor provides significantly improved floating-point support.

Forms of parallelism supported: TLP, MLP, CLP

2.1.4.2 Sun UltraSPARC T2

The UltraSPARC T2 [29] is the second generation of Sun's Niagara platform. Each UltraSPARC T2 processor contains eight simple in-order cores, each capable of supporting eight hardware threads divided into two static groups of four threads each. Each core may issue up to two integer instructions each clock cycle, one instruction from each of the core's two thread group. In contrast to the UltraSPARC T1, which contained one floating-point unit that was shared by all cores, each core in the UltraSPARC T2 has its own dedicated floating-point unit. Only one of a core's threads may utilize the core's floating-point unit each clock cycle. Each core has a dedicated L1 cache and all cores share a common L2 cache. The UltraSPARC T2 processor includes four on-chip FB-DIMM memory controllers, providing an aggregate of 42.6 GB/s of read bandwidth and 21.3 GB/s of write bandwidth. Memory-level parallelism and latency tolerance are provided by having many simultaneous in-order threads accessing memory, rather than the out-of-order instruction scheduling and prefetching mechanisms that traditional server processors employ. Hardware threads are exposed to the OS as virtual CPUs, resulting in 64-way parallelism. The architecture provides topology information to the OS so that intelligent scheduling can be performed (e.g., distribute tasks across cores first, then across thread groups).

Forms of parallelism supported: TLP, MLP, CLP

2.1.4.3 Examples

Table 2.3 provides a comparison of specific instances of the SUN processor families described in this section.

2.2 Special-purpose Processors

The special-purpose processors described in this section are multi-core processors targeted at specific application domains, such as embedded systems and high performance computing. In contrast to the general-purpose processors described in Section 2.1, these processors have a more narrow focus, are more customized for the intended usage, and in some cases require a non-traditional programming model. Many are system-on-a-chip designs that integrate more functionality than is typical for general-purpose processors, such as integrated network

Table 2.1. Comparison of x86 Multi-core Processors

	AMD Opteron 1218HE	AMD Opteron 2347HE	Intel Xeon 5160	Intel Xeon E5472
# Cores	2	4	2	4
# Threads/Core	1	1	1	1
Total OS Threads	2	4	2	4
ILP	Y	Y	Y	Y
SIMD	Y	Y	Y	Y
TLP	Y	Y	Y	Y
MLP	Y	Y	Y	Y
Heterogeneous Cores	N	N	N	N
Heterogeneous ISAs	N	N	N	N
Advanced Sync. Methods	N	N	N	N
Local Stores	N	N	N	N
Thread Priorities	N	N	N	N
On-chip Memory Ctrl(s).	Y	Y	N	N
MP NUMA	Y	Y	N	N
Shared Caches	N	Y	Y	Y
L1 Instruction	64KB x2	64KB x4	32KB x2	32KB x2
L1 Data	64KB x2	64KB x4	32KB x2	32KB x2
L2 (Unified)	1MB x2	512KB x4	4MB	6MB x2
L3 (Unified)	N/A	2MB	N/A	N/A
Total On-chip Cache	2.25MB	4.5MB	4.125MB	6.125MB x2
Addr. Bits (virt/phys)	48/40	48/48	48/36	48/40
Page Sizes	4KB, 2MB	4KB, 2MB, 1GB	4KB, 2MB	4KB, 2MB
Frequency (GHz)	2.6	1.9	3	3
GFLOPS/Core	5.2	7.6	12	12
GFLOPS Total	10.4	30.4	24	48
Mem BW (GB/s)	12.8	10.6	10.6	12.8
Bytes/FLOP	1.23	.35	.44	.27
CMOS Process (nm)	90	65	65	45
Die Area (mm^2)	199	283	144	2 x 107
Transistors (millions)	233	463	291	2 x 410
Package Pins	940	1207	771	771
Power (W TDP)	65	68	80	80
Released	2007 Q1	2007 Q4	2006 Q2	2007 Q4

Table 2.2. Comparison of IBM/STI Multi-core Processors

	IBM Power5+	IBM Power6	IBM Blue Gene/P	STI Cell
# Cores	2	2	4	1(PPC) + 8(SPE)
# Threads/Core	2	2	1	2(PPC) + 2(SPE)
Total OS Threads	4	4	4	2
ILP	Y	Y	Y (dual-issue)	Y (PPC dual-issue)
SIMD	N	Y	Y	Y
TLP	Y	Y	Y	Y
MLP	Y	Y	Y	Y
Heterogeneous Cores	N	N	N	Y
Heterogeneous ISAs	N	N	N	Y
Advanced Sync. Methods	Y	Y	N	Y
Local Stores	N	N	N	Y
Thread Priorities	Y	Y	N	N
On-chip Memory Ctrl(s).	Y	Y	Y	Y
MP NUMA	Y	Y	N/A	Y
Shared Caches	Y	Y	Y	Y (PPC-only)
L1 Instruction	64K x2	64K x2	32K x4	32K (PPC-only)
L1 Data	32K x2	64K x2	32K x4	32K (PPC-only)
L2 (Unified)	1.875M	4M x2	2K x4	512K (PPC-only)
L3 (Unified)	36M (off-chip)	32M (off-chip)	8M	N/A
Total On-chip Cache	2.06M	8.25M	8.26M	.56M
Local Store	N/A	N/A	N/A	256K x8 (SPE-only)
Total On-chip Memory	2.06M	8.25M	8.26M	2.56M
Addr. Bits (virt/phys)	64/40	64/40	32/32	64/42
Page Sizes	4K,64K, 16M,16G	4K,64K, 16M,16G	1K,4K,16K, 64K,256K,1M 16M,256M	4K + Two of (64K, 4M, 256M)
Frequency (GHz)	2.3	4.7	.85	3.2
GFLOPS/Core	9.2	18.8	3.4	6.4/PPC + 1.8/SPE
GFLOPS Total	18.4	37.6	13.6	20.8
Mem BW (GB/s)	17.05	50	13.6	25.6
Bytes/FLOP	.93	1.33	1	1.23
CMOS Process (nm)	90	65 (+some 90)	90	90
Die Area (mm^2)	243	341	173	235
Transistors (millions)	276	790	208	241
Package Pins	5370	7352	?	3349
Power (W TDP)	170	100+	16	40
Released	2005 Q4	2007 Q2	2007 Q4	2006 Q4

Table 2.3. Comparison of Sun Niagara Multi-core Processors

	Sun UltraSPARC T1	Sun UltraSPARC T2
# Cores	8	8
# Threads/Core	4	8
Total OS Threads	32	64
ILP	N	N
SIMD	N	N
TLP	Y	Y
MLP	Y	Y
Heterogeneous Cores	N	N
Heterogeneous ISAs	N	N
Advanced Sync. Methods	N	N
Local Stores	N	N
Thread Priorities	N	N
On-chip Memory Ctrl(s).	Y	Y
MP NUMA	N/A	N/A
Shared Caches	Y	Y
L1 Instruction	16K x8	16K x8
L1 Data	8K x8	8K x8
L2 (Unified)	3M	4M
L3 (Unified)	N/A	N/A
Total On-chip Cache	3.2M	4.2M
Addr. Bits (virt/phys)	48/40	48/48
Page Sizes	8K,64K, 4M,256M	8K,64K, 4M,256M
Frequency (GHz)	1.2	1.4
GFLOPS/Core	1 FPU per chip	1.4
GFLOPS Total	1.2	11.2
Mem BW (GB/s)	25.6	42.6
Bytes/FLOP	21.33	3.8
CMOS Process (nm)	90	65
Die Area (mm^2)	379	342
Transistors (millions)	279	503
Package Pins	1933	1831
Power (W TDP)	79	123
Released	2005 Q4	2007 Q4

interfaces and PCI controllers. This typically results in much lower power utilization than state-of-the-art general purpose processors.

The remainder of this section briefly describes several special-purpose multi-core processors that are available from a number of vendors. In general these processors exhibit more architectural diversity than the general-purpose processors due to the higher level of design freedom afforded by targeting specific applications. Successful architectural ideas from these processors are likely to eventually migrate to general-purpose processors.

2.2.1 Cray XMT/Threadstorm

The Cray XMT [12] system is a massively multi-threaded architecture designed to perform well on applications with little locality and abundant fine-grained parallelism, i.e., those that perform extremely poorly on distributed memory systems made up of commodity processors [37]. Each XMT Threadstorm processor supports 128 hardware threads (streams) each consisting of 32 general purpose registers, a program counter, and other state necessary to manage a thread. The processor is able to switch between threads on a cycle by cycle basis. When a thread blocks due to a high latency operation (e.g., a memory load or network operation), the processor switches to another thread rather than wait for the operation to complete. As long as there is at least one of the 128 threads ready to execute, the processor will not stall. This scheme affords a very high degree of latency tolerance compared to commodity processors, in effect mitigating the memory wall [40].

The XMT system scales to 8192 Threadstorm processors, each providing 1.5 GFLOPS peak at 500 MHz. The XMT system leverages the architecture and packaging developed by Cray and Sandia for the Red Storm system, which was productized by Cray as the XT series (XT3, XT4, XT5). The Threadstorm processor is socket compatible with the AMD Opteron, the processor used in Red Storm, and directly connects to the SeaStar2 network interface and router ASIC that was developed for Red Storm. An XMT processor board is virtually identical to a Red Storm board except that the AMD Opterons are replaced with Threadstorm processors.

2.2.2 IBM BlueGene/P

IBM has developed a custom system-on-a-chip processor for the BlueGene/P [23] supercomputer product. The processor integrates four PowerPC 450 cores, a memory controller, and a network interface and mesh router on a single chip. Each core is capable of executing four FLOPs per clock. At 850 MHz, each core provides 3.4 GFLOPS. This is low in comparison to state-of-the-art x86 processors such as the Intel Core2, which provides 12 GFLOPS per core at 3 GHz; however, the BlueGene/P uses significantly less power due to its low clock frequency. Each quad-core processor consumes approximately 16 W, compared to 65+ W for typical x86 general-purpose processors. A (peak) peta-flop BG/P system would contain 73,530 quad-core processors and 294,120 cores total.

The BlueGene/P processor uses 32-bit PowerPC 450 cores, which are cores targeted at and designed for the embedded processor market. All four cores on the processor are cache coherent with one another, so shared memory programming models such as POSIX Threads and OpenMP are possible. This is an improvement from the BlueGene/L processor, which did not support standard shared-memory programming model due to its non-cache-coherent cores.

2.2.3 IBM Cyclops-64

IBM and partners are currently developing a supercomputer based on a custom processor called Cyclops-64 [9] that contains 80 cores, each supporting two threads of execution and one shared floating-point unit. At a clock frequency of 500 MHz, each processor will provide an aggregate of 80 GFLOPs (2 flops per FPU per clock). The processor chip also contains an integrated memory controller, network interface, and mesh router. All on-chip components, including cores, on-chip memory, and network interface ports, are arranged in a "dance-hall" configuration connected by an on-chip 96-port 7-stage non-blocking crossbar switch. Each core contains a small amount of SRAM that can be configured to either be local to the core (a scratchpad memory), global to the processor (accessible to all cores on the chip), or some combination of the two. The processor architecture is unique in that the physical memory hierarchy is exposed directly to the application programmer. On-chip scratchpad memory, on-chip global memory, and off-chip memory are explicitly exposed. There is no support for virtual memory, although a simple segment-based protection scheme is provided to protect privileged system software such as the OS kernel.

2.2.4 SiCortex

SiCortex is a startup company that is focused on creating power efficient and high performance small to mid-range Linux clusters. The company has custom engineered a system-on-a-chip processor for this system that integrates six 64-bit MIPS cores, a memory controller, a high-performance network interface, a Gigabit Ethernet network interface, and a PCI Express controller on a single chip. Each chip (socket) contains everything necessary for a compute node except for the memory, which are commodity DDR2 DIMM parts. Each node consisting of a processor and 4 GB of memory consumes approximately 12 W. While each core has relatively low performance (1 GFLOP), the bytes to FLOP ratio is quite good at approximately 1.8 (six cores share two DDR2 channels, 10.6 GB/s total with PC5300 DIMMs). Currently the largest SiCortex system contains 972 nodes (5832 cores) in a single cabinet. It may be possible to construct larger multi-cabinet systems in the future, although hardware modifications would likely be required.

2.2.5 ClearSpeed e620 Accelerator

ClearSpeed is a startup company that is creating 64-bit floating point accelerators for high performance computing applications. The ClearSpeed CSX600 processor incorporates 96 processing elements (cores) with 6 KB of local scratchpad memory per core. Additionally, there is an on-chip 128 KB scratchpad memory that is accessible by all processing elements. The processor provides 33 GFLOPS of sustained double precision floating point and consumes approximately 10 W of power. The processor is designed to operate as an add-on accelerator to a general-purpose processor and is not capable of running stand-alone. ClearSpeed's e620 product incorporates two CSX600 processors on an add-in PCIe card designed to plug into a standard server. The board provides 66 GFLOPS sustained on DGEMM and has an average power dissipation of 33 W.

The CSX600 processing elements operate in SIMD data-parallel [14] fashion and are user programmable. A suite of libraries is provided that implement many common HPC operations (e.g., BLAS), which often allows developers to accelerate their applications without having to resort to low-level custom programming of the CSX600's SIMD units.

2.2.6 Tilera Tile64

Tilera is a startup company that is commercializing the technology developed by the MIT RAW [36] [38] research project. Its first product, TILE64, is a "tiled" multi-core processor consisting of 64 cores (tiles) arranged in a mesh network topology. Each chip integrates the processor cores with four DDR2 memory controllers, two 10-gigabit Ethernet interfaces, a PCIe interface, and a software-configurable "Flexible I/O" interface. The cores in the TILE64 can be partitioned into cache-coherent groups where each group can run a full operating system such as SMP Linux. Multiple simultaneously running operating system images are supported and hardware protection mechanisms isolate the partitions from one another.

The Tilera64 cores do not include hardware support for floating point operations, making them unattractive for scientific computing. The processor is targeted at embedded applications such as video encoding and network packet processing. According to the TILE64 product brief, each 64-core processor provides up to 192 billion operations per second. When operating at 700 MHz, power consumption is 15-22 W with all cores running an application.

2.2.7 SPI Storm-1

SPI (Stream Processors Inc.) is a startup company that is commercializing the STREAM processor technology developed at Stanford University as part of the Imagine [8], Merrimac [7], and other related projects. Like Tilera, the company is initially targeting the embedded market with its first product, the STORM-1 STREAM processor. STORM-1

consists of a general-purpose MIPS core coupled with a multi-lane STREAM execution unit. The MIPS core runs an embedded version of Linux and manages the STREAM execution unit, which does not run an operating system.

In STREAM programming, the task of the programmer is to break the problem into streams of data and define a graph of small self-contained kernels that operate on the streams. The model is similar to vector programming except that instead of applying one simple operation to the vector operands at a time (e.g., add, multiply) more complex operations can be constructed in the kernels under full programmer control. The STREAM compiler generates a dependency graph and then creates an efficient schedule of kernel executions on the available STREAM execution lanes. Fundamentally the STREAM approach is aimed at making efficient use of the available bandwidth by operating on large streams of data, rather than scalar operands. This can lead to much greater power efficiency and latency tolerance for applications that are well-suited to this model.

The STORM-1 SP16HP-G220 is currently the highest performing model, operating at 700 MHz with 16 lanes providing 224 GMACS (billion multiply-accumulates per second). The STORM-1 processor family does not appear to support floating-point operations, making it unattractive for scientific computing. However, nothing about the STREAM model precludes floating-point support so it may be added in future processor generations. The Merrimac project at Stanford proposed to construct a supercomputer aimed at scientific computing using STREAM processors with floating point support.

2.2.8 Ambric Am2045

Ambric is a startup company that is developing single-chip MPPA (massively parallel processor arrays) processors and a message passing style programming model to go along with it. MPPAs are essentially an MPP on a chip, where many processors with distributed memories communicate via peer-to-peer message passing rather than through a single shared memory. The company is focused on the embedded market and is positioning itself as a general-purpose replacement for DSPs, FPGAs, and other application-specific devices.

Ambric's initial product, the Am2045 processor, incorporates 336 32-bit RISC cores and 336 2 KB memories. Each core may execute its own program and communicates with other cores over a configurable on-chip multi-level interconnection network. Cores may be programmed in Java or assembly language. The Am2045 operates at 350 MHz and provides a peak of 60 GMACS (billion multiply-accumulates per second). The processor does not appear to support floating-point operations, making it unattractive for scientific computing.

2.3 Future Multi-core Hardware Trends

It is widely expected that Moore's Law [28] will continue for at least another decade. This means that the number of transistors available to chip designers will continue to double at a rate of approximately once every two years. Novel technologies such as 3-D stacking and chip-to-chip interconnects may actually increase the rate of transistor growth. However, single processor core performance is not expected to grow significantly due to power constraints and a lack of viable architectural improvements to pursue. Most of the single core performance gains have already been realized.

The extra transistors will likely be used to integrate more processor cores along with more functionality in each processor. Graphics, network interfaces, routers, system memory, and special-purpose accelerators are all candidates to be integrated into general-purpose processors. The remaining transistors will be used to increase the number of cores per processor. If the number of cores increases with Moore's Law, thousand core processors are roughly sixteen years away. It is likely that the core population will be comprised of heterogeneous cores, with some large cores aimed at single thread performance and an army of small and simple cores aimed at highly parallelizable workloads. Research has shown that a heterogeneous mix of cores can have significant power and performance advantages [3]. Determining the right mix of cores for a given application is a challenging optimization problem [33].

As more cores are integrated into each processor, connecting them becomes a problem. Shared bus-based schemes begin to break down after a few cores. As more cores are added, point-to-point on-chip interconnection networks become necessary. Efficiently mapping applications to the intra-chip topology is an important research area going forward. It is unclear what the right mix of system software and application developer involvement is for creating the mapping. Not only will the cores themselves be arranged in a topology, but on-chip memory will also be distributed and have non-uniform access (NUMA) behavior.

Finally, novel architectural capabilities such as hardware support for transactional memory [13] and advanced synchronization mechanisms [21] [34] [42] [22] are being pursued to make multi-threaded programming easier and more efficient. A significant problem with threaded programming is its inherent non-determinism and the need to be very careful when accessing shared resources [25]. Many decades of research in parallel computing have failed to make parallel programming easy. With massive levels of parallelism being introduced into mainstream computing, there is now more motivation than ever for exploring simpler and less error-prone parallel hardware and programming models.

Chapter 3

Programming Models for Parallel Scientific Applications on Multi-core

As part of the study on required Light Weight Kernel functionality, it is important to understand how scientific applications achieve parallelism on massively parallel processor (MPP) supercomputers. Current and evolving programming models are identified. We begin with a short discussion of the parallelization mechanisms that are applicable to scientific applications on MPPs. Then we discuss specific software products that are important to support.

3.1 Parallelization Mechanisms

This section is not a tutorial on programming models. It simply identifies the ones appropriate for this study. The Wikipedia web site provides an adequate starting point should the reader desire more information on parallel programming models in general.[1]

3.1.1 Vector-like Instructions

In the simplest form, a sequential application can be parallelized by the compiler without any explicit instructions from the application. The compiler can take advantage of the specific processor characteristics. For example, when targeting an AMD Opteron, the compiler can make use of the SSE (Streaming SIMD [Single Instruction, Multiple Data] Extensions) instructions when implementing application loops.

Processor support required: ILP, MLP, SIMD

OS requirement: none

[1] http://en.wikipedia.org/wiki/Parallel_programming_model

3.1.2 Vector Instructions

Some MPPs may contain vector processors, which can concurrently operate on multiple data elements. The compiler will employ the vector instructions without explicit directives from the application. Optimum use of the vectors capabilities is dependent on the structure of the application code, however.

Processor support required: MLP, SIMD

OS requirement: none

3.1.3 Implicit Threading

From an application perspective, the next level of "free" parallelism is done by the system libraries. It can launch threads to parallelize a section of code that the library developer has identified as a key kernel of computational logic. While threading can be implemented on a processor that has only one computational unit, to achieve true parallelism, the hardware must provide multiple concurrent threads of computation. The hardware support might be in the form of 1) multiple hardware threads within a core (called Hyper-Threading Technology (HTT) on Intel Xeon and Pentium 4 chips and Simultaneous Multithreading (SMT) for upcoming chips), 2) multiple cores within a single socket, 3) multiple tightly-coupled sockets, or 4) or combination of the first three.

Processor support required: CLP, MLP, SMT, TLP

OS requirement: light weight thread support or active threads; thread-safe libraries; libraries that use threading capabilities to achieve parallelism; support for HTT functionality

3.1.4 Multi-Threading with OpenMP

The OpenMP API can be used by an application (or libraries) to explicitly identify threading opportunities in the code. OpenMP "consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior."[1] When run in serial mode, the compiler directives are ignored. The threads make use of shared memory features. OpenMP addresses single-process parallelism only.

Processor support required: CLP, SMT, TLP, MLP

OS requirement: shared memory capabilities within a process; POSIX thread support and/or compliance with compiler generated or OpenMP-generated function calls

3.1.5 Explicit Message Passing with MPI

This programming model has proven highly scalable and highly successful for scientific calculations for over a decade. It is used on clusters of computers, MPP supercomputers, and even on desktops with multiple CPUs. Independent images of an application run as separate processes on each CPU. The CPUs need some form of communication, which is typically implemented over a network protocol, but can use on-node inter-process communication mechanisms. Each process performs its portion of a calculation independently until it must share data with some or all of its cooperating processes. At that point, explicit messages are passed. Both the sender and receiver plan on this communication and the recipient must provide instructions on where to put the message. This is called two-sided message passing. The MPI-2 standard also includes APIs for a relatively heavy-weight one-sided message communication mechanism.

Processor support required: CLP and/or TLP, MLP

OS requirement: efficient network stack; matching capabilities for incoming messages; inter-process, intra-node communication mechanism

3.1.6 Partitioned Global Address Spaces (PGAS)

Like MPI, applications using the PGAS programming model assume multiple independent CPUs with a communication mechanism between them. However, PGAS makes heavy use of one-sided communication mechanisms that usually have significantly lower overhead than two-sided MPI communication. One-sided communication mechanisms allow an initiator to access a remote CPU's memory without the explicit involvement of the application running on the remote CPU. Upon receipt of an incoming message, the protocol processing engine (e.g., the OS kernel or an intelligent NIC) is provided sufficient information so that the receiving application need not be involved in completing the communication. The implementation can be done in a number of ways depending on using available OS and hardware features.

OS requirement: native get/put/increment support; shared memory between processes; threads

3.2 Software Products Gaining Popularity in Parallel Programming

In this section, we look at some specific software products that have achieved a level of community adoption. Figure 3.1 presents a representative set of them in a block diagram organized by their "closeness" to the application. It also shows the OS features upon which

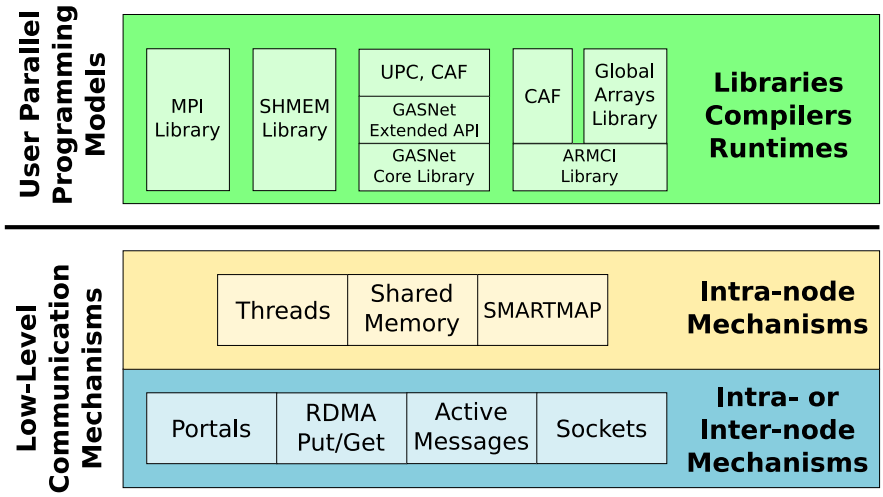

Figure 3.1. Example Software Components Used in Parallel Scientific Applications

they rely.

3.2.1 PGAS Languages

While Figure 3.1 does not depict them all, the PGAS languages gaining the most attention are: UPC [5], CAF, Chapel, Titanium, X10, Fortress, and BEC. They are expressive languages that lend themselves well to capturing a computational algorithm [27] [41]. While this is often beneficial from a programmer productivity standpoint, the trade-off is that performance and scalability are often lower than expected. An important performance-enhancing feature common to most PGAS languages is the ability for the programmer to identify local (private) data and global (shared, possibly remote) data [11]. Careful management of local and global data access is typically required to obtain good performance.

Sandia National Laboratories has recently released a virtual shared memory (a.k.a. PGAS) programming model (or support tool) called BEC[2], which is implemented as a lightweight runtime library plus a simple extension to the C language to allow shared variables. Compared with other PGAS languages/models, BEC has built-in support for unstructured applications that require high-volume, random, and fine-grained communication. For these types of applications, BEC-based implementations are very straight-forward. By contrast, other PGAS models/languages are roughly comparable to MPI in terms of code complexity. Data from initial application testing has demonstrated the effectiveness of the BEC approach. Compared to MPI implementations of the applications tested, the BEC codes had comparable (or slightly better) performance and scalability, and many times (5-10x) fewer source lines of code.

3.2.2 User Friendly Languages

The current PGAS languages tend to be a superset of a traditional language (e.g., C, C++, Fortran, and Java.). There is another set of languages that grew out of the tools area: Python, Perl, and Matlab. Originally these languages were used primarily for scripting and one-time use (quick and dirty) jobs. With time, they have shown themselves to be useful for calculations. As CPUs increased in speed, their efficiency was also sufficient. It is not clear that they will parallelize sufficiently for MPPs, but that concern is based on thought experiments, rather than actual tests.

3.2.3 Libraries

In addition to careful language selection to facilitate physical model implementation, the application developer can take advantage of third party libraries (TPLs). This section explores TPLs that enable distributed memory and/or distributed communication. It does not address the entire spectrum of scientific libraries that can provide useful functionality such as numeric algorithms, mesh manipulation, or load balancing. The broader spectrum of libraries do not have any specific OS requirements and are not addressed.

3.2.3.1 SHMEM

SHMEM refers to a collection of APIs that provide one-sided put/get style access to remote memory in a distributed memory system. The API provides point-to-point, atomic, and collective operations. There is no official standard for the API. The first SHMEM library appeared on the Cray T3D computer in the 1990's and continues to help solve important classes of problems requiring high performance computation. SHMEM libraries have been available on a few platforms [26]. The HPVM (High Performance Virtual Machine) project[3]

[2]http://www.cs.sandia.gov/BEC

[3]http://www-csag.ucsd.edu/projects/hpvm.html

provided solutions for clusters running Windows NT or Linux. The most widely used implementations of the SHMEM API are those provided by Cray, Inc. [15]

3.2.3.2 Global Arrays

The Global Arrays library allows every process to independently access a distributed array structure. Explicit library calls are used to access portions of the array. However, these library calls are only available for the explicitly created global array. The calls cannot be used on arbitrary memory locations [31]. Global Arrays offers a compromise between explicit message passing programming models and shared memory models.

3.2.3.3 ARMCI

ARMCI (Aggregate Remote Memory Copy Interface) [30] is a PGAS style library intended to be used by higher-level system libraries, such as the Global Arrays library. While ARMCI may be used directly by application developers, that was not the intent of its designers. ARMCI provides platform independent remote memory access functionality. Each implementation takes advantage of the system hardware and software features available on the platform. Operations to the same remote location are guaranteed to complete in order.

3.2.3.4 GASNet Extended API

The "upper" level of the GASNet library builds upon the lower level, which is described in Section 3.2.4.1. GASNet stands for Global Address Space Networking. The upper level provides more functionality and is solely dependent upon the lower layer. Therefore, the upper level is highly portable once the lower level is ported. The extended API provides remote memory access operations and collective operations.[4]

3.2.4 Networking Software

Networking software for parallel applications must balance the need for scalable high performance, low overhead, and yet provide reliable and deterministic results. The need exists for a thin, reasonably portable API that is independent of the underlying operating system and hardware. Three such specifications are described in this section.

[4] http://gasnet.cs.berkeley.edu/

3.2.4.1 GASNet

GASNet is based heavily upon the Active Messages 1.1 specification [6] and is implemented directly on top of each individual network architecture. Like the extended API, its API is visible to higher level software. This allows for more sophisticated uses than what may be feasible with the machine-independent extended API.

3.2.4.2 Active Messages

"Active Messages represent a RISC approach to communication, providing simple primitives, rather than solutions, which expose the full hardware performance to higher layers. Active Messages are intended to serve as a substrate for building libraries that provide higher-level communication abstractions and for generating communication code from a parallel-language compiler, rather than for direct use by programmers." [2]

3.2.4.3 Portals

The Portals message passing interface can be implemented on top of a native networking layer (e.g. sockets, Infiniband verbs, RDMA) or implemented directly in the network. For the Cray XT3/XT4/XT5h line, Portals is the native networking layer [4] and is implemented via a combination of hardware and firmware. Version 3.3 of the specification is best suited for higher-level explicit message passing protocols such as MPI. It is lightweight, connectionless, and highly scalable.

3.3 Shared Libraries

A discussion on an optional-use OS feature such as shared libraries may seem inappropriate for a section dedicated to programming models. However, their role in high performance computing remains an outgoing debate. Because of the polarization on the issue, we have explored the feature in some depth. While not a programming model, some developers used this feature as a foundation for achieving flexibility and run-time decision making.

On a standard desktop system, a dynamically linked application offers several advantages. The latest copy of a shared library can be accessed when the application is run. By contrast, statically linked applications use the version of the library that was available when the application was built. Explicit action, in the form of a relink, is required to update the libraries used by the application. Secondly, a shared library image can be shared across all applications running on the system. This can relieve memory pressure, since the same text is accessed by all. Lastly, although more of an implementation detail than a pro or con, shared libraries are often memory mapped from disk to minimize the memory usage and swap space. Only the portions of the library that are being accessed need to reside in memory.

For the MPP community, shared libraries are a blessing and a curse. The first advantage mentioned above can be very important. Software library developers are often distinct from the application developers. Their release cycles may not be synchronized. Library developers may not be able to notify application developers or users of critical updates to the library. Additionally, application developers may rely on the dynamic invocation of a library only if the input test specification requires it. If an application is statically linked, the sum of all possible libraries may make for a very large application binary. However, there is negative aspect to this real-time update feature of shared libraries. An application may have passed regression and acceptance testing with one version of shared libraries. Problems may appear without warning due to an update in a shared library that uses a different release schedule. An application that ran fine one day, may not the next.

The second advantage of shared libraries is rarely helpful in the MPP and cluster environments with distributed memory. Each node has its own memory from which a program executes. In this case, shared libraries cannot be shared by different applications or even the same application running on a different node in the cluster. Not only is it no longer an advantage, but it can also become a disadvantage. Either each node must have a copy of the library on its local disk or the memory mapping will inefficiently swap to a remote disk. Typically, cluster systems will maintain copies of system libraries on each node's local disk. Application-specific libraries must somehow be copied over to each node on which the application is running. If they are not copied locally, remote accesses will severely degrade the performance of the application. Swapping of library text sections will occur over the cluster's or MPP's network. Currently, the degradation due to remote swapping typically becomes visible when using 500 or more nodes.

The remainder of this section explores the research to provide a scalable shared library access mechanism when the library image is not available on a local disk.

3.3.1 Performance Impact of Dynamic Access to a Library

We begin with metrics. In order to understand why dynamic linking is an issue, we studied the performance of each node in an application accessing a shared file (e.g., a library). Scientific parallel programs typically follow the same code paths and then synchronize at specific points during the computation. Given this assumption, we further assume that each node will request a portion of the shared file at nearly the same time. If these accesses are independent of one another (i.e., not collective), significant serialization will occur as each node attempts to read the file and bring all or some of its contents into the node's memory. The serialization of requests will lead to poor performance that worsens with increasing node count.

Continuing with the premise that a significant number of the nodes need to access the shared library, we studied the viability of using a collective operation for the access. In this scheme, one node reads the file and then distributes its contents to the other nodes using a broadcast over the high-speed network. The experimental results shown in Figure 3.2

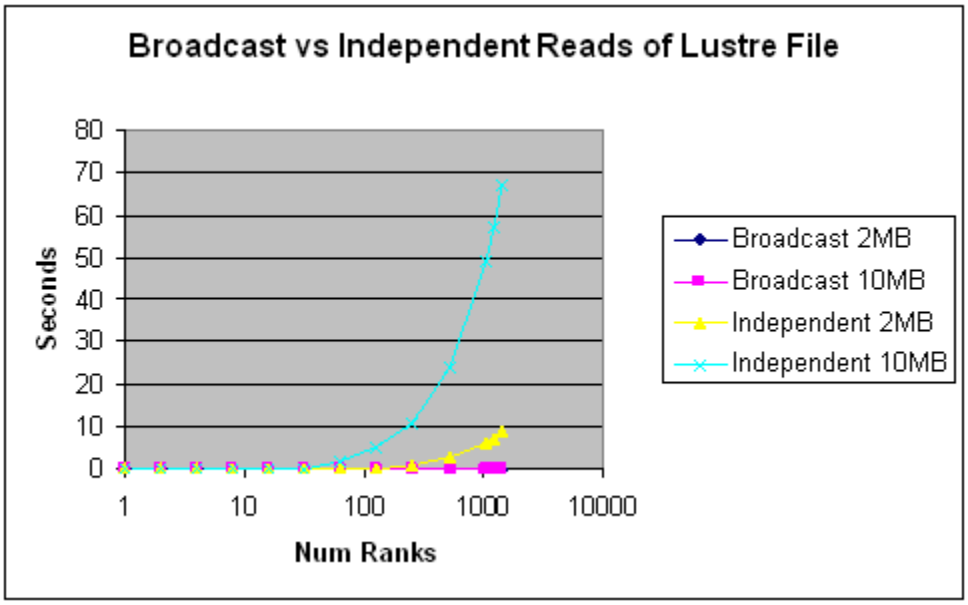

Figure 3.2. Performance Comparison of Independent Reads versus Single Read followed by Broadcast

indicate that this improves performance and scalability significantly.

The independent reads take much longer than the single read followed by a broadcast. This data was collected on Red Storm using a Lustre file system. Even Lustre cannot parallelize the independent accesses since the file resides on exactly one disk. Every node must wait its turn for the data. A disadvantage of the broadcast approach is the constraint that all (or a pre-determined subset of) nodes must cooperate in a collective operation. This is not the default behavior for current dlopen() implementations. Most applications prefer portability and do not wish to include special code for a particular system, unless absolutely necessary.

3.3.2 Loadtime Linking of Dynamic Libraries

Light weight kernels, such as the Catamount operating system make design choices that can make dynamic libraries difficult to implement, even if a scalable solution is found. For example, the Catamount OS has no knowledge of file systems, so it cannot store shared libraries for possible future use by applications. Given this constraint, we looked at a solution that addresses only the first advantage described above. (Dynamic linking accesses the latest version of the shared library.) At job launch time, all shared libraries are collected and loaded onto each compute node in a scalable fashion.

3.3.3 Overview of UNIX Dynamic Load Process

We begin by briefly describing what happens when a dynamically linked application is executed on full-featured UNIX or Linux system. When an application binary is invoked, the application's ELF image is memory mapped (via `mmap()`). The shared object called ld.so is launched as the application. The "real" application information (e.g., the load address) is passed in as arguments to ld.so. The file ld.so checks the load address of the application and relocates itself if necessary. The ld.so parses the application and loads all libraries. It then processes the relocation entries and calls the init function of each library. At this point, ld.so starts the application.

The ld.so image is primarily self-contained code and does not call other libraries. It uses open(), close(), write(), mmap(), and munmap() system calls to communicate with the operating system kernel.

3.3.4 Overview of Catamount Dynamic Load Process

Armed with this understanding of the functional flow of a dynamic load process, we prototyped a scalable implementation using the Catamount LWK. This section assumes a general understanding of the Catamount architecture. See [18] for more details.[5]

There are no changes to the command line options for the application launcher, yod. Yod automatically detects that the application specified on the command line is dynamic by checking for the ELF program segment of type PT_INTERP. This segment points to the dynamic library loader ld.so to use. Yod loads ld.so and the application program into memory before ld.so is launched. This was done by extending the load protocol to the compute nodes. With Catamount, a static application is loaded in two phases: text followed by the data. A dynamic application is loaded in three phases: application ELF image, followed by ld.so text, followed by ld.so data. The application ELF image uses the same "I have section" followed by "Send me section" protocol that is used for the text and data sections of ld.so. The application ELF image is loaded into low heap space and heap_base is incremented.

For statically linked application launches, yod sends a "start program" message to the first PCT. This message is fanned out to the rest of the PCTs. When the leaf nodes' PCTs receive the message, they start the application. The application then sends an "app started" message to its parent node PCT, which then starts the application on that node. This results in a fan-in synchronization back to yod.

This approach does not work when a dynamically linked application is launched because ld.so will need to take part in collective operations (for gathering the shared libraries) before `cstart()` is called. To facilitate this, the PCT starts the application (i.e. ld.so) when it receives the "start program" message during the fan-out. This allows ld.so to join in collective loading of libraries.

[5] available at http://www.cs.sandia.gov/ smkelly/papers.html

ld.so parses the application ELF image and loads the necessary shared libraries by performing the following actions on each shared library:

1. dl_open()
2. dl_read(), read the first page in the ELF image so it can validate the ELF file and read the program section headers
3. dl_mmap(), a region of the memory large enough to contain the library's text, data, and bss
4. dl_mmap(), the text portion of the file into the memory region
5. dl_mmap(), the data portion of the file into the memory region
6. dl_mmap(), the bss into the memory region
7. dl_close()

Note that any given time, there is only one file open.

The Catamount version of dl_open() loads the ELF image into a temporary buffer positioned at the end of the application heap. The Catamount version of _dl_mmap() allocates memory from the beginning of application heap and increments the application's heap_base as needed. If _dl_mmap() is called to map a file into memory, it copies the requested portion from the temporary file buffer.

A critical aspect of the scenario described above is how a library image gets into the temporary buffer. The library load protocol is basically the same as the one used to load the text/data sections. It is an "I have data", "Send me data" protocol which two exceptions: 1. During the application load phase, the node is running completely in the context of the PCT. During the library load phase, the node is running in the context of the ld.so which does not yet have portals support. Instead, it traps to the PCT which then communicates with yod. 2. During the application load phase, the PCT knows the size of the text and data segments. During the library load phase, the size of each library is only known by the service node. This means that the base (node 0) PCT needs to send a file stat request to yod. Yod replies with "I have N bytes of data". This message kicks off the "I have data", "Send me data" fan-out.

When the application traps to the PCT for a library read, it blocks waiting for the read to complete. Once the PCT has received the library and has fanned it out to all children, it unblocks ld.so.

Once the library loads are complete, the application still needs to participate in the "app started" fan-in protocol. This is done by blocking the application until the non-leaf node PCTs get the "app started" message. The application blocks in crt0.o by making a trap to the PCT signaling that it is ready to call cstart().

The role of crt0.o in Catamount is to call cstart() with pointers to the PCB and NIDPID map. For a static application, these pointers are on the stack which is set up by the PCT. For dynamic applications, these pointers are obtained by traps to the PCT. This change was needed because the PCT does not directly launch the dynamic application as it does a static application.

The Catamount version of ld.so was obtained from the uClibc distribution. ld.so interfaces with the system via a limited set of system calls: _dl_open(), _dl_mmap, etc. Other than this set of system calls, the ld.so source code was unmodified. ld.so was compiled in the uClibc distribution tree with minimal changes to the Makefiles. The Catamount versions of _dl_open(), _dl_read() and _dl_mmap() are custom and limited to just the functionality needed for ld.so.

One final implementation detail is to describe the application image. A dynamic application differs from a static application in two ways: first, it is linked differently, and second, the pre-main() initialization code in crt0.o is different. Conspicuously absent is a difference in cstart(). It is the same for static and dynamic binaries. Given the complexity of Catamount's cstart(), this is an important feature. The only real difference in application linking is that the dynamic library loader (ld.so) must be specified as a link option.

3.3.5 Results of Executing a Dynamically Linked Binary

We ran HPL with a shared ACML library and with a static link. There were no performance differences when using a 28-node test system. We ran both in single-core (default used by yod) and dual-core (specified VN argument to yod) modes. Any performance difference would have come from the extra level of indirection in resolving entry points.

```
mjleven@boot_cage2:~> yod   -sz all xhpl-dyn
================================================================================
T/V                N    NB     P     Q               Time             Gflops
--------------------------------------------------------------------------------
WR05C2C8       40000    64     7     4             349.73          1.220e+02
--------------------------------------------------------------------------------
||Ax-b||_oo / ( eps * ||A||_1  * N        ) =        0.0180851 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =        0.0165362 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =        0.0032472 ...... PASSED
================================================================================


mjleven@boot_cage2:~> yod   -sz all xhpl
================================================================================
T/V                N    NB     P     Q               Time             Gflops
--------------------------------------------------------------------------------
WR05C2C8       40000    64     7     4             350.34          1.218e+02
```

```
--------------------------------------------------------------------------------
||Ax-b||_oo / ( eps * ||A||_1  * N        ) =        0.0180851 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =        0.0165362 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =        0.0032472 ...... PASSED
================================================================================


mjleven@boot_cage2:~> yod -VN   -sz all xhpl-dyn
================================================================================
T/V                N    NB     P     Q               Time             Gflops
--------------------------------------------------------------------------------
WR05C2C8       50000    64     8     7             350.26          2.379e+02
--------------------------------------------------------------------------------
||Ax-b||_oo / ( eps * ||A||_1  * N        ) =        0.0059334 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =        0.0139478 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =        0.0027270 ...... PASSED
================================================================================

mjleven@boot_cage2:~> yod -VN   -sz all xhpl
================================================================================
T/V                N    NB     P     Q               Time             Gflops
--------------------------------------------------------------------------------
WR05C2C8       50000    64     8     7             349.79          2.382e+02
--------------------------------------------------------------------------------
||Ax-b||_oo / ( eps * ||A||_1  * N        ) =        0.0059334 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =        0.0139478 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =        0.0027270 ...... PASSED
================================================================================
```

Chapter 4

Conclusion

The diverse set of multi-core hardware and diverse set of programming models presented in this report are indicative of the general uncertainty in both mainstream and scientific computing. In the area of processors, it is unclear how to best mix heavy-weight, light-weight, and special-purpose cores for particular application domains. System software must support this heterogeneity and provide mechanisms for efficiently managing it. Similarly, system software must be agile and able to adapt to special-purpose processors. Power constraints make it likely that future capability systems will increasingly rely on embedded processor technology that is often not well-suited for use with mainstream commodity operating systems.

In the area of programming models, the "MPI-everywhere" approach that has been employed for more than a decade is expected to be strained on future platforms. It is unclear what the best approach is for managing many million-way parallelism and complex, hierarchical compute nodes. Multi-level programming models (e.g., MPI+OpenMP) have not been very successful in the past on multi-chip SMP systems due to productivity and performance issues. They are being re-examined in the context of single-chip multi-core processors with the hope of better exploiting intra-chip locality. System software must provide the functionality, such as support for threading and shared memory, needed to perform this experimentation. The complex memory management policies provided by general-purpose operating systems often hinder this experimentation, lead to sub-optimal performance, or both.

We are using the knowledge gained from the investigations summarized in this report to design an open-source lightweight kernel operating system targeting future capability platforms made up of multi-core processors. A goal of this effort is to create an agile system that is able to adapt to and efficiently support whatever multi-core hardware and programming models gain acceptance by the community.

References

[1] http://en.wikipedia.org/wiki/openmp.

[2] http://now.cs.berkeley.edu/am/activemessages.html.

[3] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 29–40, 2006.

[4] Ron Brightwell, Trammell Hudson, Kevin T. Pedretti, Rolf Riesen, and Keith Underwood. Implementation and performance of Portals 3.3 on the Cray XT3. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, September 2005.

[5] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[6] D. Culler, K. Keeton, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. Generic active message interface specification v1.1. Technical report, U.C. Berkeley Computer Science Technical Report, November 1994.

[7] William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, Franois Labont, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, and Ian Buck. Merrimac: Supercomputing with Streams. In *Supercomputing Conference (SC2003)*, November 2003.

[8] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors: Programmability with efficiency. *ACM Queue*, 2:52–62, 2004.

[9] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the cyclops-64 cellular architecture. In *Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment (HPCS'06)*, volume 0, page 9, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[10] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An Integrated Quad-Core Opteron Processor. In *Solid-State Circuits Conference (ISSCC)*, 2007.

[11] T. El-Ghazawi, F. Cantonnet, Y. Yao, and R. Rajamony. Developing an optimized upc compiler for future architectures. Technical report, IDA Center for Computing Sciences, 2005.

[12] John Feo, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd Conference on Computing Frontiers*, pages 28–34, 2005.

[13] M.P. Herlihy and J.E.B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 1993 International Symposium on Computer Architecture (ISCA)*, 1993.

[14] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29:1170–1183, 1986.

[15] Cray Inc. *Cray XT(TM) Series Programming Environment User's Guide*, October 2007.

[16] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: a Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, Mar-Apr 2004.

[17] David Kanter. Inside Nahalem: Intel's Future Processor and System (http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719).

[18] Suzanne M. Kelly and Ron B. Brightwell. Software architecture of the light weight kernel catamount. In *Cray User Group Annual Technical Conference*, May 2005.

[19] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmet, and Pat Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23:66–76, March-April 2003.

[20] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[21] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient Synchronization of Multiprocessors with Shared Memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(4):579–601, 1988.

[22] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[23] Gary Lakner and Carlos Sosa. *Evolution of the IBM System Blue Gene Solution*. IBM, 2007.

[24] H.G. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 Microarchitecture. *IBM Journal of Research and Development*, 2007.

[25] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences, University of California at Berkeley, January 2006.

[26] Quadrics Ltd. Shmem programming manual. Technical report, Quadrics Supercomputers World Ltd., June 2001.

[27] Rusty Lusk. The hpcs languages issues and challenges. In *PMUA 2005 Workshop in programming models for HPCS ultra-scale applications*, June 2005.

[28] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38:114–117, 1965.

[29] U.G. Nawathe, M. Hassan, K.C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-Core, 64-Thread, Power-Efficient SPARC Server on a Chip. *IEEE Journal of Solid-State Circuits*, 43:6–20, 2008.

[30] J. Nieplocha and J. Ju. Armci: A portable aggregate remote memory copy interface. Technical report, Pacific Northwest National Laboratory, 2000.

[31] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, Novmeber 1994.

[32] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[33] JoAnn M. Paul and Brett H. Meyer. Amdahl's Law Revisited for Single Chip Systems. *International Journal of Parallel Programming*, 35(2):101–123, 2007.

[34] Jack Sampson, Rubén González, Jean-Francois Collard, Norman P. Jouppi, and Mike Schlansker. Fast Synchronization for Chip Multiprocessors. *ASM SIGARCH Computer Architecture News*, 33(4):64–69, 2005.

[35] STI. *Cell Broadband Engine Programming Handbook, Version 1.0*. STI, April 2006.

[36] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of International Symposium on Computer Architecture*, June 2004.

[37] Keith D. Underwood, Megan Vance, Jonathan Berry, and Bruce Hendrickson. Analyzing the scalability of graph algorithms on eldorado. In *IPDPS '07: Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.

[38] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.

[39] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine A. Yelick. The potential of the cell processor for scientific computing. In *Proceedings of the Third Conference on Computing Frontiers*, pages 9–20, Ischia, Italy, May 2006.

[40] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–24, 1995.

[41] K. Yelick, D. Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Ton Wen. Productivity and performance using partitioned global address space languages. In *Parallel Symbolic Computation (PASCO 07) London Canada*, July 2007.

[42] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 35–45, 2007.

DISTRIBUTION:

1 MS 0899 Technical Library, 9536 (1 electronic)

Sandia National Laboratories