Tutorial Outline

- Parallel Computing Architecture and Resources (1 hr Alice)
- Performance and Implementation (30 min Alice)
- Coffee break
- Parallel Programming Models (1 hr 10 min Rusty)
- Optimization and Current Issues (20 min Alice)

- LUNCH BREAK

- Topics from MPI (1 hr 30 min Bill)
- Coffee break
- How to Design Real Applications (30 min David)
- Hybrid Programming (1 hr Rolf)
- Q&A

Note: all timings are approximate
Thanks! - Contributions Include:

- Ken Koch, LANL
- Anshu Dubey, U. Chicago
- Gabriele Jost, TACC/NPS
- Kirk Jordan, IBM
- John Levesque, CRAY
- Bronis de Supinski, LLNL
- Mark Seager, LLNL
- Charles Grasal, IBM/Instrumental
- Lenny Oliker, LBNL, NERSC
- Yun (Helen) He, LBNL, NERSC
- Satoru Shingu, Earth Simulator
- Hitoshi Sakagami, Himeji Inst.
- Jeff Keasler, LLNL
- Brent Gorda, LLNL
- Allison Baker, LLNL
- Jim Tomkins, Sandia
- Andrew Johnson, Army HPC
- Holger Berger, NEC
- Tom Epperly, LLNL
- Gary Kumfert, LLNL
- Erik Draeger, LLNL
- Cornell Wright, IBM, LANL
- Blaise Barney, LLNL
- David Skinner, LBNL, NERSC
- Rolf Rabenseifner, HLRS
- Subhash Saini, NASA
- Bill Camp, SNL/Intel
- Martin Galle, NEC
- Doug Joseph, IBM
- Hitoshi Sakagami, Himeji Inst.
- Dan Poznancov, SRC
- Jack Dongarra, UTK
- Wolfgang Nagel, TU Dresden
- Holger Brunst, TU Dresden
- Vasily Bulatov, LLNL
- Gregg Hommes, LLNL
- Andrew Cook, LLNL
- Fred Streitz, LLNL
- James Glosli, LLNL
- Chris Gottbrath, Etnus
- Francois Gygi, UCDavis
- Mike Campbell, CSAR
- Manuel Vigil, LANL
- Tim Pierce, LLNL
- Bert Still, LLNL
- Georg Hager, RRZE

AND many others as noted on individual slides!

Parallel Computing Architectures - Defining Terms

- **MPP**: Massively Parallel Processor with distributed memory
  - Historically each RISC-based processor had own separate memory: CRAY T3D/E, original IBM SP series
  - Ambiguous - could use shared address space model "virtual shared memory"
  - Original building blocks of parallel computing based on COTS (commodity of the shelf)
  - For a while MPPs were distinguished from clusters by their tightly-coupled single system image
  - Now it is often one or more of:
    - Proprietary interconnect
    - OS with a light weight kernel (LWK)
  - BlueGene/L and P, XT4’s, Power Architectures all classed as MPP on Top500
- **The distinction between “MPP” and “cluster” is diminishing**
Parallel Computing Architectures
Defining Terms (cont)

- **SMP**: Symmetric MultiProcessor, occasionally “Shared-Memory-Parallel”
  - Processors share a single global area of RAM
  - symmetry → all processors have equal access to memory and other parts of the system (e.g., I/O) in strictest definition
- **PVP**: Parallel Vector Processor
  - Cray J90/C90/T90 Series, mostly 80’s - 90’s
  - NEC SX series, Fujitsu VX series uses them as building blocks
  - Vector instructions included on many current CPU’s (see SSE, AltVec), also Cell
- **ccNUMA**: “cache-coherent Non-Uniform Memory Access”--allowed SMP nodes to share memory
  - Shared physical address space with automatic hardware replication
  - Sometimes called DSM or Distributed Shared Memory
  - SGI Altix with ccNUMA shared memory interconnect
- **Clusters**
  - Started with Beowulfs--build your own
  - Graduated to most common architecture on Top500
  - Generally have commodity processor with commodity interconnect

---

Simplest Comparison of SMP, MPP or shared memory vs. distributed memory

<table>
<thead>
<tr>
<th>SMP Symmetric Multi-Processor</th>
<th>MPP Massively Parallel Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="SMP_MPP.png" alt="Diagram" /></td>
<td><img src="MPP.png" alt="Diagram" /></td>
</tr>
<tr>
<td>• SMP means equal access including I/O</td>
<td>• Memory physically distributed</td>
</tr>
<tr>
<td>• Sometimes term is generalized to mean Shared Memory Parallel</td>
<td></td>
</tr>
</tbody>
</table>
Cache Coherence and NUMA introduced to enhance scalability

- SMP’s suffered from bottlenecks based on network contention and differences between processors speed and memory access rate.
- Cache-Coherency led to the introduction of ccNUMA in the early 1990’s

Shared Memory can be NUMA or UMA

Scales more like MPPs than bus-based SMP’s
Hardware tells you that the separate memories are one main memory
One address space over entire machine
Amount of time to access a memory value depends on whether it is local to a node or remote in another node
SGI Origin Series

UMA or SMP
- SMP’s still exist stand alone, e.g., Sun Servers
- Network types, e.g.
  - Crossbar: independent access from each CPU
  - BUS: one CPU can block the memory access of the other CPUs
Now the building block of most parallel architectures is the SMP

- Several CPUs on each chip—multi-processor chip
- Multiple chips on each node
- Multiple nodes clustered with a high-performance network
- Distributed Memory Systems
  - Clusters
  - MPPs
  - Constellation

Interconnection Network

Some of the “top 10” systems

www.top500.org: June 2008
(Stats given for June—it may change again at this meeting)

**CrAY XT4 Jaguar #5**
ORNL/NCCS
23016 Proc
Rmax: 205 Tflops

**IBM Blue Gene(L,P) Systems #2, 3**
LLNL (L): Rmax: 478 Tflops
Argonne National Lab (P): 450 Tflops

**Texas “Ranger” #4**
U. Texas
25544 Proc
Rmax: 326 Tflops

Interesting Architectures now being Upgraded that were featured in previous lists

- Columbia SGI Altix
  6/2006 Rmax: 61 TFlops
- NEC Earth Simulator
  6/2006 Rmax: 36 TFlops
Original MPPs were single CPUs on a chip with local memory

Simplistic View: MPP Node

- **Processors**: Commodity, Proprietary
- **Memory**: DRAM + variants for faster access
- **Communication Mechanism**: Shared Memory or Message Passing
- **Networks**: Switch, 3D Torus, Bus, Crossbar - more than one (BlueGene/L) - usually proprietary for MPP

The first introduction of multi-mode programming came with the Shared Memory Node Architecture

- Distributed memory across nodes
- Shared memory within a single node
- Optimal programming model? Didn’t get much speed-up over using MPI everywhere...
- Now, multicore / many-core is here and increasing
Chip Architecture Terminology

- How many levels of cache? Sizes?
- Stream Buffers improve data flow into cache
- Ways to bypass cache?
- Vector units?
- Pseudo vector units?

Current Multi-core or Multi-processor Cache-based Chip

Typical Layout:
- Each processor:
  - L1 caches
  - Registers
  - Functional units
- Each chip (shared)
  - L2 cache
  - L3 cache
  - Path to memory

On a multi-core chip, get more computational power with (often) same bandwidth to memory, so need to be effective with cache reuse

Note: Different access for L3 cache
Quad Cores and Beyond

Example: Intel Dunnington
6 cores, just released
16 MB shared L3

Example: 4 core AMD Barcelona
L3 Cache common to all 4
Memory

New many-core chips
64 and up are likely to be
appearing in HPC

Quad Core systems (283)
now dominate the top500

Sample Architectures
in More Detail

- Power Series, IBM ASC Purple, Upcoming Blue Waters
- Blue Genes
- Cray XT3/4 (includes Jaguar and Red Storm), XT5 is announced
- Earth Simulator (basically SX-6), SX-9 is announced
- Cells and Roadrunner
- Typical Top Cluster (Texas Ranger)
- SGI Altix 3000 (NASA’s Columbia), SGI ICE-Pleiades is installed
- Other HPC designs
IBM Power Series Evolution: Multi Chip Module (MCM) Architecture

- **POWER4:**
  - 4 processor chips
  - 2 processors per chip
  - 8 off-module L3 chips
    - L3 cache is controlled by MCM and logically shared across node
  - 4 Memory control chips
  - 16 chips

- **POWER5:**
  - 4 processor chips
  - 2 processors per chip
  - 4 L3 cache chips
  - L3 cache is used by processor pair
  - "Extension" of L2
  - 8 chips

---

**MultiChip Module Diagram**

- Power4+ processor 1.5GHz is 6.0 GF/s
- Power5 moves L3 cache (shared) on chip
- Power6 ~2X the frequency of Power5
- Power7 still under NDA

---

Various SMP Configurations Of P6's are available
A typical IBM Power Series
LLNL’s “Purple”

- **System Specs**
  - 93.4 TF/s peak from 12,288 Power5 @ 1.9 GHz
  - 50 TB memory
  - 2.88 PB global disk
    - 135 GB/s IO bandwidth to GPFS
    - Defensive IO bandwidth depends on MTBF
      - MTBF 62.7 hrs (2.6 days) \( \approx 22.15 \) GB/s
  - 1 SULU adapter for 2 copper Federation planes = 2x2x2
    = 8.0GB/s GB/s peak bandwidth
  - 8 \( \mu \)s latency, <4.8 GB/s delivered MPI bidirectional bandwidth

- **1,536 8-way SMP Squadron IH (P5-570) 2U nodes**
  - 32 GB memory from 533MHz DDR2 512MB DIMM
  - 8x1.9 GHz Power5 single core ASIC in DCM with L3 cache ASIC
  - 36 MB L3 (12 SA), 1.92 MB L2 caches (10 SA)
  - 64 KB L1I (2 SA), 32 KB L1D (4 SA)
  - 2x293 GB local disk for dual boot

LINPACK 75.8 Tflops

**Power 7 is coming**

- Blue Waters will be based on IBM's POWER7
- Timescale ~2011
- > 200,000 cores
- Private L1 and L2 caches for each core, shared L3
- NSF machine sited at U. Illinois
Blue Gene: record performance, low power, high processor count

- Blue Gene L gave surprising performance
  - Record Linpack performance of 280.6 TFlop/s
  - Sweep of HPC Challenge Competition Class 1 Benchmarks
  - 2005 Gordon Bell Award winner and 3 other finalists
  - 2006 Gordon Bell for peak performance
  - 2007 Gordon Bell
- Low power, low cost, high performance: 180-360 Tpflops
  - New generation of embedded processors
  - Relatively modest clock rate but low power consumption and low cost
  - Less cooling means pack in tighter
  - 64 cabinets (32x32x64) ~1 MW (1000 Hairdryers) <2500 sq.ft
  - Attacks the distance to memory problem
  - Low latency memory: 3 to L1, 36 to L3, 86 to memory
  - High bandwidth memory 22 GB/s to L3, 5.5 GB/s to memory
- 216 Nodes, 2 P/node, MANY Processors! (> 100,000)
- Different networks--3D torus, broadcast tree (and others)
- Blue Gene P’s are up and running

Blue Gene / L

Multiple complementary interconnects support diverse application scaling requirements
- 3D torus with bi-directional nearest-neighbor links
- 2.1 GB/s combining tree for fast global reductions
- Low-latency global barrier network
- High reliability expected from high level of integration using system-on-a-chip technology

Was Top500 #1 Before Roadrunner

SC2008 Tutorial © Koniges, Eder, Gropp, Lusk, Rabenseifner & Others
Blue Gene P upgrades Blue Gene L
First complete BG/P node card

One IO card
Physical Layer chip
XFP cages
Connector for 2nd IO card

Deep Computing
Offering Schematic – Blue Gene/P

Blue GeneP continues Blue Gene’s leadership performance in a space-saving, power-efficient package for the most demanding and scalable high-performance computing applications.

System
72 Racks

Rack
Cabled 8x8x16
32 Node Cards
1024 chips, 4096 procs
14 TF/s
2 TB
1 PF/s
144 TB

Node Card
(32 chips 4x4x2)
32 compute, 0-1 IO cards

Compute Card
1 chip, 20 DRAMs

Chip
4 processors
13.6 GF/s
8 MB EDRAM

13.6 GF/s
2.0 (or 4.0) GB DDR
Supports 4-way SMP

HPC SW:
Compilers
GPFS
ESSL
Loadleveler

Front End Node / Service Node
JS21 / Power5
Linux SLES10
Two ways for apps to use hardware on BG/L

Three different ways on BG/P

BG/L Mode 1 (Co-processor mode - CPM):
- CPU0 does all the computations
- CPU1 does the communications
- Communication overlap with computation
- Peak comp perf is $\frac{5.6}{2} = 2.8$ GFlops

BG/L Mode 2 (Virtual node mode - VNM):
- CPU0, CPU1 independent "virtual tasks"
- Each does own computation and communication
- The two CPU’s talk via memory buffers
- Computation and communication cannot overlap
- Peak compute performance is 5.6 GFlops

BG/P Virtual Node Mode, SMP Mode, Dual Mode

Blue Gene/P Highlights

- Lightweight kernel on Compute Nodes
- Linux on I/O Nodes
- Quad Core means ~2x performance enhancement
- ~1.2X frequency increase
- New/improved network architectures
- Bandwidth enhancements
Red Storm - Cray XT
Initial MPP Architecture, upgrades

- Functional hardware partitioning:
  - Service and I/O nodes,
  - Compute nodes, and
  - RAS nodes
- Partitioned Operating System
  - LINUX on service and I/O nodes,
  - LWK (Catamount) on compute nodes,
  - Stripped down LINUX on RAS nodes
- Initial Processors (circa 2005-2006)
  - AMD Opteron 2.0 GHz
  - 64 Bit extension to IA32 instruction set
  - 64 KB L1 instruction and data caches on chip
  - 1 MB L2 shared (Data and Instruction) cache on chip
- Less than 2 MW total power/cooling
- Less than 3,000 ft$^2$ of floor space
- 1st UPGRADE, more coming
  - dual core@2.4GHz
  - SeaStar 2.1
  - 12,960/25,920 Procs
  - 124,42TF Theoretical Peak

Resurgence of classic MPP

Cray XT3s were upgraded to Hood Processing Element

Node architecture
- 4 GB/sec MPI Bandwidth
- AMD Opteron
- 8.5 GB/sec Local Memory Bandwidth

Cray SeaStar2 Interconnect
- 7.6 GB/sec Local Memory Bandwidth
- 60 ns latency

Compute module
- Six Network Links
  - Each >3 GB/s x 2
  - (7.6 GB/sec Peak for each link)
- Currently Support
  - 1, 2, 3, 4, 5, 6, or 8 GB memory per socket

Hood diagrams courtesy John Levesque
Cray Introduces XT5 and XT5\textsubscript{h}

XT5\textsubscript{h} (hybrid) Integrated Scalar and Vector Processing

- Common Infrastructure
  - Seastar2+ Interconnect
  - Linux Environment
  - Common I/O and File System

Cray XT5 Blade

S1O Blade

XT5\textsubscript{h} supports two processor technologies: Opteron and Vector

XT5\textsubscript{h} diagrams courtesy John Levesque, CRAY

Mix-and-match to meet workload requirements

<table>
<thead>
<tr>
<th>Cray XT4</th>
<th>Cray XT5</th>
<th>Cray X2 Blade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized for compute/interconnect balance</td>
<td>Optimized for memory-intensive and/or compute-biased workloads</td>
<td>high bandwidth memory intensive &gt;25GFLOPs/single core vector based CPU and global address space scaling to 32K processors</td>
</tr>
</tbody>
</table>

XT5\textsubscript{h} diagrams courtesy John Levesque, CRAY
X2 Vector Blade

- X2 vector CPUs
  - 8 per blade
  - Configured as two 4-way SMP nodes

- Memory
  - 32 or 64GB per node

- XT5, diagrams courtesy John Levesque, CRAY

Earth simulator Vector Processor

- Vector Unit: 8 sets
  - 6 different types of vector pipelines
  - 72 vector registers (512 vector elements)
  - 17 mask registers (256 total)

- Main Memory Access CPU Unit

- Scalar Unit
  - 4-way super scalar
  - 64KB instruction cache
  - 64KB data cache
  - 128 general purpose register

http://www.es.jamstec.go.jp/

Based on NEC SX-6,
Held top spot in top500 for 7 Lists
Slight RAM differences, bi-dir btw differences

http://www.es.jamstec.go.jp/
Earth Simulator
http://www.es.jamstec.go.jp

System: 640 nodes, 40 TFLOP/s
10 TB memory
640x640 crossbar

Node: 8 CPUs, 64 GFLOP/s
16 GB, SMP
ext. b/w: 2x16 GB/s

CPU: Vector
8 GFLOP/s, 500 MHz
Single-Chip, 0.15 µs
32 GB/s memory b/w

Earth Simulator Upgrade
~131 Teraflops
~March 2009
Likely based on SX-9

NEC is introducing the SX-9 at several installations this year and next

Basic Unit
1.6 TF/node

SX-8R
282 GF
8 CPUs
563 GB/s
256 GB

SX-9
1.6 TF
102.4 GF/CPU
16 CPUs
“Flat” Memory
7.3x 4 TB/s
4x

Memory
1 TB

Focus on increase in bandwidth and performance

Multi Node System
0.82 PFLOPs, 512 Nodes

NEC slides courtesy NEC
Images copyright NEC

SC2008 Tutorial © Koniges, Eder, Gropp, Lusk, Rabenseifner & Others
Sun Constellation Linux Cluster “Ranger”
Texas Advanced Computing Center

- First of the new NSF Track2 HPC
- Number 3 on the Top 500 list for June 2008
- 3936 Nodes, 62,976 Cores
- Peak Performance 579.4 Tflops
- 15,744 Quad-Core AMD Opteron at 2.3 GHz

Cluster Systems are the top500 majority
Altix 3000 Overview: shared memory Linux

- Itanium2@ 1.5GHz (peak 6 GF/s)
- 128 FP reg, 32K L1, 256K L2, 6MB L3
  - Cannot store FP in values in L1
- interconnect (fat-tree)

ccNUMA in hardware
64-bit Linux w/ single system image – looks like a single Linux machine but with many processors

Columbia Configuration

Front End
- 128p Altix 3700 (RTF)

Networking
- 10GigE Switch 32-port
- 10GigE Cards (1 Per 512p)
- InfiniBand Switch (288port)
- InfiniBand Cards (6 per 512p)
- Altix 3700 2BX 2048 Numalink Kits

Compute Node (Single Sys Image)
- Altix 3700 (A) 12x512p
- Altix 3700 BX2 (T) 8x512p

Storage Area Network
-Brocade Switch 2x128port

Storage (440 TB)
- FC RAID 8x20 TB (8 Racks)
- SATARaid 8x35TB (8 Racks)
New Pleiades System NASA/Intel/SGI being installed

- 43,008 cores
- InfiniBand
- 2 quad-core processors per node
  - Quad-core dual socket 3Hz Intel Xeon (Harpertown)

Cell Processor --was enhanced for HPC

Think of the typical cell processor designed for the PlayStation PS3:
- CPU Calculation power is ~220 - 230 Gflops (Cell single precision)*
- GPU Calculation power is ~1.8 TFlops (Nvidia graphics chip)
- Total System Calculation power is 2 TFlops

*Cell has nominally 8 SPE's, this is 4GHz estimate, PS3 designed to use 7 of these. Each SPE is capable of sustaining 4 FMADD per cycle
IBM’s Cell Broadband Engine™ Blade
Cell systems and Roadrunner

- Cell double precision floating point enhancement
  - Enables each SPE to perform two DP FPMULADD ops per cycle
  - Gives a peak of 102.4 DP FP Gflops/s@3.2 Ghz (half peak SP FP rate)
- Cell Blade memory capacity increase to 16GB
- 25.6 BG/S memory bandwidth

Applications performing very near peak

Roadrunner is First Petaflop Machine!
A New top500 #1 Record

Connected Unit cluster
180 compute nodes w/ Cells
12 x3655 I/O nodes

6,120 dual-core Opterons ⇒ 44 TF
12,240 Cell eDP chips ⇒ 1.3 PF* I/O nodes not counted

17 CUs
3,264 nodes

Eight 2nd-stage 288-port IB 4X DDR switches

Roadrunner slides courtesy Ken Koch, LANL
and IBM, and RR Team

Note: dual XDR replaced by dual DDR
in HPC enhanced Cell (the PowerXCell 8i)
A Roadrunner TriBlade node integrates Cell and Opteron blades

- **QS22** is a new IBM Cell blade containing two new enhanced double-precision (eDP/PowerXCell™) Cell chips
- Expansion blade connects two QS22 via four PCI-e x8 links to LS21 & provides the node’s ConnectX IB 4X DDR cluster attachment
- **LS21** is an IBM dual-socket Opteron blade
- 4-wide IBM BladeCenter packaging
- Roadrunnertriblades are completely diskless and run from RAM disks with NFS & Panasas only to the LS21
- Node design points:
  - One Cell chip per Opteron core
  - ~400 GF/s double-precision & ~800 GF/s single-precision
  - 16 GB Cell memory & 16 GB Opteron memory

Roadrunner slides courtesy LANL and IBM RR Team

Roadrunner at a glance

- Cluster of 17 Connected Units (CU)
  - 6120 (+408) AMD dual-core Opterons
  - 12,240 IBM Cell eDP accelerators
  - 44.1 (+4.2) Teraflops peak (Opteron)
  - 1.26 (+0.02) Petaflops peak (Cell eDP)
  - 1.026 PF sustained Linpack
- InfiniBand 4x DDR fabric
  - 2-stage fat-tree; all-optical cables
  - Full bi-section BW within each CU
  - 384 GB/s (bi-directional)
  - Half bi-section BW among CUs
  - 3.3 TB/s (bi-directional)
  - Non-disruptive expansion to 24 CUs
- 98 TB aggregate memory
  - 49 TB Opteron
  - 49 TB Cell
- 204 GB/s sustained File System I/O:
  - 204x2 10G Ethernets to Panasas

Roadrunner slides courtesy LANL and IBM RR Team

- Fedora Linux (RHEL possible)
- SDK for Multicore Acceleration
  - Cell compilers, libraries, tools
- xCAT Cluster Management
  - System-wide GigEnet network
- 2.35 MW Power:
  - 437 MegaFLOPs/Watt
- Size:
  - 55 miles of IB cables
  - 500,000 lbs
  - 278 racks
  - 5200 ft²
Roadrunner Phase 3 is Cell-accelerated, not a cluster of Cells

This is what makes Roadrunner different!

Roadrunner uses Cells to make nodes ~30x faster

400+ GFlop/s performance per hybrid node!
FPGA (Field Programmable Gate Array) and GPU (Graphics Processing Unit)

• Several efforts to use FPGAs and GPUs in high performance computing
  – E.g., NVIDIA Tesla
• FPGAs included in CRAY Cascade design

FPGAs are advancing on the Moore’s law curve, but even better regarding delivered performance because we can implement much simpler parallel logic and not “waste” transistors or such logic as instruction decode, branch prediction and out of order execute.

We were Quoting Numbers...

• Top500 List
• Rmax in Tflops

What do these numbers really mean?
What other numbers are important?
How do we record parallel performance?
Performance Issues: Chip and machine level definitions/concepts

• Peak Floating Point Performance: the maximum performance rate possible for a given operation MFLOPS ($10^6$), GFLOPS($10^9$), TFLOPS ($10^{12}$), PFLOPS ($10^{15}$)
• PFLOPS are new this year!

• Performance in OPS (operations per second) is based on the clock rate of the processor and the number of operations per clock

• On the chip, there are various ways to increase the number of operations per clock cycle

Counting Flops

• Counting Flops (Definition due to Dongarra)
  - + or * counts as 1 flop (also min,max,abs,sign)
  - Sqrt or / counts as 4 flops
  - Log counts as 8 flops
  - Compare counts as 1 flop
Theoretical Peak Performance

\[
R^* = \frac{\text{# arithmetic function units/CPU}}{P \text{ clock cycle}}
\]

"The speed that the vendor guarantees you will never exceed"
Occasionally is exceeded!
Also called Rpeak

Why \( R^* \) is rarely obtained

- The bandwidth from the memory to the cache is insufficient
- The bandwidth from the cache to registers/functional units is insufficient
- Memory bank conflicts
- Startup time or latency is too long
- Communication time is too large wrt the computation time
- The mix of floating point ops is not suitable e.g., code cannot use the ability to do two multiplies/cycle

Dongarra
So, instead we benchmark ala the Top500 list

- Competitive analysis
- Verification of performance claims, reduction of marketing hype
- Application performance
- Performance prediction
- System optimization
- Future development

Other Goals of Performance Modeling

– Future Systems

- To predict the performance on future systems
  - Procurements of new computer systems
  - Decrease in development time for future systems
  - Need for building prototype systems is reduced
How do we get $R_{max}$ for Top500?

• **Linpack Benchmark:**
  - Dense system of linear equations
    - LU decomposition via Gaussian elimination
  - Added run-rules, guidelines for implementation, execution, optimization and reporting
  - Output is Tflop/s

Specific Tests

• **100 X 100:** Performance on the unchanged Fortran code. Only compiler optimization allowed.
• **1000 X 1000:** Performance on an optimized code. For operation count, count standard Gaussian Elimination operations
• **N X N:** Optimized code for arbitrary problem size
  - This is the one for the Top500 list
Actual Application Performance

- Algorithm and problem characteristics
  - problem size, regularity of data, geometry
- Programming model (appropriate for architecture?)
- Compiler efficiency
- Available operations (vector units?, floating point support)
- R* chip performance (clock, instructions per clock)
- Processor/memory performance (cache hits and misses, specialized hardware, e.g., stream buffers)
- Optimization of problem for single CPU, and MCM (multichip module)
- Operational environment (effect of other processes, scheduling)
- I/O quantity and performance

Useful performance of a parallel computer

- Chip performance and thus R* is only the first factor in determining the speed of the application
- Bandwidth: The rate of data transferred in MB/sec
- Latency: How long it takes to get that first bit of information to a processor (message of size zero)
  - Effective Latency takes into consideration cache hits/misses
  - Newer architectures improve and hide latency
- Part of obtaining good performance is finding ways to reduce the so-called Processor - Memory Performance Gap
Some benchmark suites to compare useful performance

- Linpack Parallel Benchmark suite [performance.netlib.org](http://icl.cs.utk.edu/hpcc)
  - Provides a forum for comparison of massively parallel machines by solving a system of linear equations and allowing the performance numbers to reflect the largest problem run on the machine
  - lists $R_{\text{max}}$ (Gflop/s for largest problem) and $R^*$ (a theoretical #)
  - providing input for Top500 list
- Stream Benchmark [www.cs.virginia.edu/stream](http://icl.cs.utk.edu/hpcc)
  - sustainable memory bandwidth
    (cpu’s getting faster quicker than memory)
- NAS Parallel Benchmarks [science.nas.nasa.gov/Software/NPB](http://icl.cs.utk.edu/hpcc)
  - set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, derived from CFD applications, consist of five kernels and three pseudo-applications
  - standardized set of relevant benchmarks and metrics for performance evaluation of modern computer systems
  - has started a set of benchmarks based on large industrial applications

---

**— HPCC —**  
The HPC Challenge Benchmark Suite

- Combining
  - HPL (Linpack),
  - DGEMM (double prec. matrix-matrix multiplication),
  - STREAM (singleCPU & parallel),
  - PTRANS (parallel transpose),
  - GUP (random update),
  - FFT (Discrete Fourier Transformation),
  - Ping-Pong (latency and bandwidth),
  - b_eff (random and natural ring parallel latency and bandwidth)
Reporting Performance: Fundamental Measurements

- **Parallel speedup** \((n, P)\) where \(n\) is problem size, \(P\) is processors
  \[ S = \frac{T(N,1)}{T(N,P)} \]
  - Ideally, speed up should be measured w.r.t. the best serial algorithm, not just the parallel algorithm on a serial computer
- **Parallel efficiency**
  - Measure how well the implementation scales on the given architecture
- **Wall clock speedup**
- **Scaled speedup**
  - Important since one real utility of MPP’s is solving the big problem
  - Also, no matter what, the absolute speedup should always decrease (exceptional cases of superlinear excluded)
- **Amdahl’s Law is still often underestimated**
- **Gustafson’s Law**

---

**Example:** *parallel speedup compared to 1 processor*

<table>
<thead>
<tr>
<th>Number of CPUs</th>
<th>Peak</th>
<th>Sustained</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.86 Tflops</td>
<td>74.3%</td>
</tr>
<tr>
<td>4</td>
<td>7.61 Tflops</td>
<td>70.8%</td>
</tr>
<tr>
<td>16</td>
<td>14.50 Tflops</td>
<td>64.9%</td>
</tr>
<tr>
<td>64</td>
<td>26.58 Tflops</td>
<td>57.8%</td>
</tr>
</tbody>
</table>

SC2008 Tutorial © Koniges, Eder, Gropp, Lusk, Rabenseifner & Others

---

**Example:** *scalable performance* (AFES T1275L96, 10 steps)
Measuring \textit{wall clock speedup during the development of a scalable application}

Brian Gunney, et al. LLNL

Example of \textit{parallel efficiency and scaled efficiency}

LLNL, Hypre Team

L. Yang
Can report: fixed problem size and problem size scaled with number PE’s – *scaled speed-up*

**GRASP performance on BG/L**
and comparison to HP cluster (3 GHz Pentium+Myrinet)

![](image)

Scaling beyond 10,000 procs is now the goal for BlueGene scale architectures

**Strong scaling == fixed problem size**

**Scaled speed-up == weak scaling**
The best algorithm design requires testing both performance and scalability

MultiGrid preconditioned CG (MGCG) performance

Less iterations?  Scalable?

Example from ISPC: Bosl and Ashby, LLNL

Some perspective on Amdahl’s Law

\[ Speedup = \frac{1}{(1 - f) + f/P} \]

where \( f \) is the fraction of the code that is parallelizable and \( P \) is the PE number

Gustafson’s law: Parallel fraction increases with problem size — scaled speed-up
Tools have evolved to meet the demands of the HPC platforms

Consider the challenge:
- 16K+ Processors
- SMP nodes, Vector Processors, Levels of Cache
- OpenMP, Threads
- MPI, Parallel I/O
- F90,C,C++,scripting languages

Debugging Concepts

- **Trace Debuggers:**
  - program is instrumented to produce a trace
  - information is obtained following the execution to completion (or malfunction)

- **Process Control Debuggers**
  - program is instrumented to respond to control of user -- stop a process, resume a process, conditional breakpoints
  - can query state of a process
    - is it running, stopped, or at a breakpoint
  - can alter values of variables, memory locations

- **Popular debugger: TotalView**
  - "de facto standard" tool
  - includes threaded, MPI, OpenMP and Hybrid models
  - advanced support for C/C++ and F90
  - Blaise Barney tutorials highly recommended [www.llnl.gov/computing/tutorials](http://www.llnl.gov/computing/tutorials)
  - TotalView [www.etnus.com](http://www.etnus.com), SC tutorial by Chris Gottbrath

- **DDT** [www.streamline-computing.com](http://www.streamline-computing.com) (Linux)

- **KAP OpenMP Debugging Tools - Assure**
  - validate correctness of OpenMP program (also hybrids)
Debugging is much more difficult on highly parallel machines

- State of a sequential program depends only on its program counter and its memory map
- State of a parallel program needs that info for each PE
- Parallel codes are not deterministic in general
- A large number of processes must be presented in a way that can be displayed on a typical workstation
- Must be easy to insert breakpoints at subgroups of processes
- User needs to know the current status of the entire program in a concise manner
  - use this to determine what program is doing
  - navigate to a specific processor to obtain detailed information
- Debugger should not introduce any changes to the behavior of the code

What is TotalView?

- Source Code Debugger
  - C, C++, Fortran 77, Fortran90, UPC
    - Complex language features
  - Wide compiler and platform support
  - Multi-threaded Debugging
  - Parallel Debugging
  - Proven Scalability
  - Remote Debugging
  - Memory Debugging Capabilities
  - Powerful/Easy GUI
  - Visualization
  - CLI for Scripting

TotalView Slides courtesy
Chris Gottbrath, Etnus
TotalView Parallel Debugging

- **Setup**
  - Launch/Attach to parallel job as a unit

- **Context**
  - Look at 1 process at a time
  - Easily switch focus
  - See the whole program state

- **View data**
  - For a Single Thread or Process
  - Across Many Processes

- **Control Processes**
  - Synchronize with Breakpoints or Barriers
  - Run Individual Threads or Processes
  - Work with Groups of Threads or Processes

TotalView Subset Attach

- **You can attach to part of a large job**
  - different subsets at different times
    - You can attach to a subset, run till you see trouble and then ‘fan out’ to look at more processes if necessary.
  - greatly reduces overhead
    - with a danger of missing things
TotalView MPI Message Queue Graph

- Look for Hangs & Deadlocks
- Pending
  - Receives
  - Sends
  - Unexpected
- Inspect individual entries
- Filter Messages
  - by Tag, Communicator, or Rank
- Automatic Cycle Detection

TotalView Slides courtesy Chris Gottbrath, Etnus

TotalView Memory Debugger

- Built into the source debugger
  - 1 process or many
- Interactively Investigate Memory State
  - Look at memory issues in context of your running application
  - Solve memory problems during debug phase
- Detect and solve
  - Memory Leaks
  - Dangling Pointers
  - Malloc API usage errors (i.e. double free)
  - Heap Corruption (dynamic arrays)

TotalView Slides courtesy Chris Gottbrath, Etnus
Performance Analysis Tools

- **Profiling**
  - Gather statistics about program’s execution
  - Sample at regular intervals
  - Provide statistical information, graphical displays
  - Generally depends on input data
- **Event tracing of instrumented application**
  - Special code inserted into user’s program
  - Important events recorded
    - e.g., function entry, message sent/received
- **Research Areas:**
  - Performance prediction tools/simulators

Some Available Performance Tools

- **All/Most Major HPC platforms**
  - Performance Visualization for Parallel Programs
    - http://www-unix.mcs.anl.gov/perfvis (jumpshot, etc)
  - TAU http://www.cs.uoregon.edu/research/paracomp/tau/
    - C++, Java, C, Fortran 90, HPF, and HPC++, and OpenMP
  - Vampir: http://www.vampir-ng.com for parallel programs (Tracing supported through TAU, KOJAK, and Adaptor)
  - Pablo: http://www-pablo.cs.uiuc.edu/ I/O and visualization
  - PAPI: Performance analysis using hardware counters
  - Aims: Automated Instrumentation and Monitoring System.
    - http://www.nasa.gov/Groups/Tools/Projects/AIMS/
  - Paradyn - can operate on executable
    - http://www.cs.wisc.edu/~paradyn/
  - PARAVER http://www.ceph.upc.es/paraver
    - MPI, OpenMP, Performance counters, system activity
  - GNU gprof
  - KOJAK http://www.fz-juelich.de/zam/kojak/
- **Also most vendors support their own tools (e.g., Cray Apprentice2)**
Visualization and Analysis of Parallel Programs

- Trace data based
- Global displays show all selected processes
  - Timeline: application execution over time
  - Counter Timeline: counter dev. over time
  - Summary Timeline: shows parallelism
  - Summary Chart: aggregated profiling
  - Activity Chart: presents per-process profiling
  - Call Tree: draws dynamic call tree
  - Message Statistics: for each process pair
  - I/O Statistics: MPI I/O operation statistics
- Process displays show a single process per window
  - Activity Chart
  - Timeline
  - Call Tree

Vampir Slides courtesy of Wolfgang E. Nagel, Holger Brunst TU Dresden.
www.vampir-ng.com

Vampir – Routine Profiles and Statistics

- Setup Summary Chart
  - Global Displays ⇒ Summary Chart
  - Context Menu ⇒ Select ⇒ All Symbols
  - Context Menu ⇒ Options ⇒ Per Process
- Selected group: All Master Threads
Vampir – Timeline Display

• Can display MPI collective and I/O operations
• To zoom, draw rectangle with the mouse
• Also used to select sub-intervals for statistics

Vampir – Timeline Display – Info Dialogs
Cray Apprentice

Call Graph Profile

Function Overview

Communication & I/O Activity View

Load balance views

Source code mapping

Time Line Views

Pair-wise Communication View

TAU Performance System Components

http://www.cs.uoregon.edu/research/tau/home.php

From Sameer Shende U. Oregon
TAU Performance System Architecture

http://www.cs.uoregon.edu/research/tau/home.php

From Sameer Shende, U. Oregon

IPM Performance Tool

Integrated Performance Monitoring

- portable, lightweight
- scalable profiling
- fast hash method
- profiles MPI topology
- profiles code regions
- open source
- easy to use

Developed by David Skinner, LBNL

But, your code’s “hot” spots may vary depending on architecture

SC2008 Tutorial © Koniges, Eder, Gropp, Lusk, Rabenseifner & Others
Extremely difficult to use tools on very large numbers of processors: research area

- TotalView scalability
  - BlueGene/L port included significant scaling requirements
  - Collaborative scalability project targeting traditional OSs
  - Scalable subset debugging on BG/L
- Scalable, lightweight tools
  - “Focused tools”: fewer features, but extreme scalability
  - Stack Trace Analysis Tool (STAT)
  - Scalable infrastructure: LaunchMON, MRNet
- Automatic error detection
  - Memory correctness tools
  - Umpire
  - Threading tools

Adapted from Scott Futral, Dong Ahn, and Greg Lee, LLNL

Issues affecting code performance

- Programming Model, suitability for application/architecture
- Memory Characteristics
  - read/write to PE’s own memory (local memory)
  - read/write to shared memory or another PE’s memory
- Processing Element
  - caches: levels, size, performance, coherence
  - stream buffers, means to bypass cache
  - Vector processors
  - Advanced processor functions (multi-threading, SSE vectorization)
- Connection Network
- I/O and Peripherals
- Run-time environment
  - schedulers, contention for I/O, etc.
Advice for Debugging and Tuning Hybrid Codes

- Debug and Tune MPI code and OpenMP code separately.
- Decide which loop to parallelize. Better to parallelize outer loop. Decide whether Loop permutation or loop exchange is needed.
- Choose between loop-based or SPMD.
- Use different OpenMP task scheduling options.
- Experiment with different combinations of MPI tasks and number of threads per MPI task.
- Adjust environment variables.
- Aggressively investigate different thread initialization options and the possibility of overlapping communication with computation.

From Helen He LBL-NERSC

Performance Strategies

After isolating portions of code to optimize, what next?

- **Global Performance Strategies**
  - Degree of parallelism over entire code
  - Number of parallel tasks as compared with available numbers of processors
  - Uniformity of tasks / load balancing
  - Synchronization requirements

- **Processor-specific strategies**
  - If cache-based architecture: data locality
  - Parallelism via a “super scalar” processor
  - Pipelining
  - Vector processors
Tutorial Outline

- Parallel Computing Architecture and Resources (1 hr Alice)
- Performance and Implementation (30 min Alice)
- Coffee break
- Parallel Programming Models (1 hr 10 min Rusty)
- Optimization and Current Issues (20 min Alice)

- LUNCH BREAK

- Topics from MPI (1 hr 30 min Bill)
- Coffee break
- How to Design Real Applications (30 min David)
- Hybrid Programming (1 hr Rolf)
- Q&A

Note: all timings are approximate
Outline

- Parallel computers
- Parallel programming models
  - Shared memory
  - Distributed memory
  - Hybrid
- Some parallel programming languages and libraries
  - MPI
  - OpenMP, HPF
  - Co-Array Fortran, UPC, Titanium
  - HPCS languages
  - A high-level load-balancing library
- An experiment with hybrid programming

What Makes Parallel Computing Necessary

- Scientific problems require more computation than single processor machines can produce
  - 1 arithmetic operation = 1 Floating-point Operation = 1 FLOP
  - 1 million such operations per Second = 1 megaflops
  - Current fast PC’s: 2 gigaflops (3 billion operations per second)
  - Current fastest parallel supercomputers: 1 Petaflop
  - Achieved with ~100,000 processing elements
  - The challenge: how to program them
- Such power is needed for accuracy of simulations
  - Where did New Zealand go?
A Very Short, Very Introductory Introduction

- We start with a short introduction to parallel computing “from scratch” in order to get our terminology straight.
- You may already know this, but it will help get us synchronized (a term from parallel computing).

What Makes Parallel Computing Possible: How Parallel Computers Work

- A serial (“normal”) computer

  ![Diagram](image)

  - Program: single sequence of instructions
    - “Usual” programming languages compiled into instructions understood by CPU
  - Process: program + address space + program counter + stack
A Bunch of Serial Computers

- Program: Many instances of same sequential program operating on different data in different address spaces
- Multiple independent processes with same program
- Examples
  - seti@home
  - Data mining
  - Sequence matching in biology

Multiple CPU’s per Memory System

- May use multiple processes with own memory plus some way of sharing memory and coordinating access (operating-system-specific)
- One process may have multiple threads (pc + stack)’s operating on same address space, guided by same program
  - Sequential programming languages with smart compilers
  - OpenMP versions of C and Fortran
Hardware Variations on Multiple CPU's per Memory System

- SMPs (Symmetric Multiprocessors)
- Multi-core Chips
  - Similar programming model, but enhances performance of multithreaded programs

Multiple Memory Systems

- Program: multiple processes, each of which may be multi-threaded, with separate address space
- The program needs some way of expressing communication among processes, so that data can move between address spaces
- Many large-scale machines look like this, leading to more interest in programming models that combine the shared and distributed memory approaches to programming.
Programming Models

- A *programming model* is an abstraction that we program by writing instructions for
- Programming models are implemented in languages and libraries
- Implementations of the “standard” serial model of a CPU
  - Assembly language
  - Language models
    - C
    - C++
    - Fortran
- Implementations of various parallel models
  - For shared memory: OpenMP (C and Fortran versions), pthreads library
  - For multiple-memory systems: Message Passing (MPI)
  - Hybrid models for hybrid systems

Higher-Level Models

- Parallel Languages
  - UPC
  - Co-Array Fortran
  - Titanium
- Abstract, declarative models
  - Logic-based (Prolog)
  - Spreadsheet-based (Excel)
- The programming model research problem: Define a model (and language) that
  - Can express complex computations
  - Can be implemented efficiently on parallel machines
  - Is easy to use
- It is hard to get all three
  - Specialized libraries can implement very high-level, even application-specific models
Parallel Programming Models

- Multiple classes of models differ in how we think about communication and synchronization among processes or threads.
  - Shared memory
  - Distributed memory
  - Some of each
  - Less explicit
- Shared Memory (really globally addressable)
  - Processes (or threads) communicate through memory addresses accessible to each
- Distributed memory
  - Processes move data from one address space to another via sending and receiving messages
- Multiple cores per node make the shared-memory model efficient and inexpensive; this trend encourages all shared-memory and hybrid models.

Writing Parallel Programs

- Parallel programming models are expressed:
  - In libraries callable from conventional languages
  - In languages compiled by their own special compilers
  - In structured comments that modify the behavior of a conventional compiler
- We will survey some of each of these and consider a single example written in each
  - Not an adequate tutorial on any of these approaches
  - Many detailed sources are available
  - Here we are only trying to convey the “flavor” of each approach
The Poisson Problem

- Simple elliptic partial differential equation
- Occurs in many physical problems
  - Fluid flow, electrostatics, equilibrium heat flow
- Many algorithms for solution
- We illustrate a sub-optimal one, since it is easy to understand and is typical of a data-parallel algorithm

Jacobi Iteration (Fortran Ordering)

- Simple parallel data structure

- Processes exchange columns with neighbors
- Local part declared as xlocal(m,0:n+1)
Serial Fortran Version

real u(0:n,0:n), unew(0:n,0:n), f(1:n, 1:n), h

! Code to initialize f, u(0,*), u(n:*), u(*,0), and 
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter
  do j=1, n-1
    do i=1, n-1
      unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                          u(i,j+1) + u(i,j-1) - &
                          h * h * f(i,j) )
    enddo
  enddo
  ! code to check for convergence of unew to u.
  ! Make the new value the old value for the next iteration
  u = unew
enddo

MPI

- The Message-Passing Interface (MPI) is a standard library interface specified by the MPI Forum
- It implements the message passing model, in which the sending and receiving of messages combines both data movement and synchronization. Processes have separate address spaces.
- Send(data, destination, tag, comm) in one process matches Receive(data, source, tag, comm) in another process, at which time data is copied from one address space to another
- Data can be described in many flexible ways
- SendReceive can be used for exchange
- Callable from Fortran-77, Fortran-90, C, C++ as specified by the standard
  - Other bindings (Python, java) available, non-standard
**MPI Version**

```fortran
use mpi
real u(0:n,js-1:je+1), unew(0:n,js-1:je+1)
real 0 f(1:n-1, js), h
integer nbr_down, nbr_up, status(MPI_STATUS_SIZE), ierr

! Code to initialize f, u(0,*), u(n,*), u(*,0), and
! u(*,n) with g

h = 1.0 / n
do k=1, maxiter
  ! Send down
  call MPI_Sendrecv( u(1,js), n-1, MPI_REAL, nbr_down, k &
                   u(1,je+1), n-1, MPI_REAL, nbr_up, k, &
                   MPI_COMM_WORLD, status, ierr )
  ! Send up
  call MPI_Sendrecv( u(1,je), n-1, MPI_REAL, nbr_up, k+1, &
                   u(1,js-1), n-1, MPI_REAL, nbr_down, k+1, &
                   MPI_COMM_WORLD, status, ierr )
  do j=js, je
    do i=1, n-1
      unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                          u(i,j+1) + u(i,j-1) - &
                          h * h * f(i,j) )
    enddo
  endo
  ! code to check for convergence of unew to u.
  ! Make the new value the old value for the next iteration
  u = unew
endo
```

**HPF**

- HPF is a specification for an extension to Fortran 90 that focuses on describing the distribution of data among processes in structured comments.
- Thus an HPF program is also a valid Fortran-90 program and can be run on a sequential computer
- All communication and synchronization if provided by the compiled code, and hidden from the programmer
**HPF Version**

```fortran
real u(0:n,0:n), unew(0:n,0:n), f(0:n, 0:n), h
!HPF$ DISTRIBUTE u(:,BLOCK)
!HPF$ ALIGN unew WITH u
!HPF$ ALIGN f WITH u

! Code to initialize f, u(0,*), u(n:*,) , u(*,0),
! and u(*,n) with g

h = 1.0 / n
do k=1, maxiter
  unew(1:n-1,1:n-1) = 0.25 * &
    ( u(2:n,1:n-1) + u(0:n-2,1:n-1) + &
      u(1:n-1,2:n) + u(1:n-1,0:n-2) - &
      h * h * f(1:n-1,1:n-1) )
  ! code to check for convergence of unew to u.
  ! Make the new value the old value for the next iteration
  u = unew
enddo
```

**OpenMP**

- OpenMP is a set of compiler directives (in comments, like HPF) and library calls
- The comments direct the execution of loops in parallel in a convenient way.
- Data placement is not controlled, so performance is hard to get except on machines with real shared memory
OpenMP Version

```fortran
real u(0:n,0:n), unew(0:n,0:n), f(1:n-1, 1:n-1), h
!
! Code to initialize f, u(0,*), u(n,*), u(*,0),
! and u(*,n) with g
!
h = 1.0 / n
!
!$omp parallel
$omp do k=1, maxiter
$omp do
    do j=1, n-1
        do i=1, n-1
            unew(i,j) = 0.25 * ( u(i+1,j) + u(i-1,j) + &
                              u(i,j+1) + u(i,j-1) - &
                              h * h * f(i,j) )
        enddo
    enddo
$omp enddo
! code to check for convergence of unew to u.
! Make the new value the old value for the next iteration
u = unew
$omp end parallel
enddo
```

The PGAS Languages

- PGAS (Partitioned Global Address Space) languages attempt to combine the convenience of the global view of data with awareness of data locality, for performance
  - Co-Array Fortran, an extension to Fortran-90
  - UPC (Unified Parallel C), an extension to C
  - Titanium, a parallel version of Java
Co-Array Fortran

- SPMD – Single program, multiple data
- Replicated to a number of images
- Images have indices 1,2, ...
- Number of images fixed during execution
- Each image has its own set of local variables
- Images execute asynchronously except when explicitly synchronized
- Variables declared as co-arrays are accessible of another image through set of array subscripts, delimited by [ ] and mapped to image indices by the usual rule
- Intrinsics: this_image, num_images, sync_all, sync_team, flush_memory, collectives such as co_sum
- Critical construct

CAF Version

```fortran
real (0:n,js-1:je+1,0:1)[*], f (0:n,js:je), h
integer np, myid, old, new
np = NUM.Images()
myid = THIS_IMAGE()
new = 1
old = 1-new
! Code to initialize f, and the first and last columns of u on the extreme processors and the first and last row of u on all processors
h = 1.0 / n
do i=1, n-1
do j=1, maxiter
if (myid < np) u(:,js-1,old)[myid+1] = u(:,je,old)
if (myid > 0) u(:,je+1,old)[myid-1] = u(:,js,old)
call sync_all
do j=1, js, je
do i=1, n-1
u(i,j,new) = 0.25 * ( u(i+1,j,old) + u(i-1,j,old) +
                      u(i,j+1,old) + u(i,j-1,old) -
                      h^2 * f(i,j) )
enddo
dodo
do i=1, n-1
u(i,j,old) = 0.25 * ( u(i+1,j,old) + u(i-1,j,old) +
                      u(i,j+1,old) + u(i,j-1,old) -
                      h^2 * f(i,j) )
enddo
enddo
! code to check for convergence of u(:,:,new) to u(:,:,old).
! Make the new value the old value for the next iteration
new = old
old = 1-new
enddo```

**UPC**

- UPC is an extension of C (not C++) with shared and local addresses

---

**UPC Version**

```c
#include <upc.h>
#define n 1024
shared [*] double u[n+1][n+1];
shared [*] double unew[n+1][n+1];
shared [*] double f[n][n];
int main() {
    int maxiter = 100;
    // Code to initialize f, u(0, *), u(n, *), u(*, 0), and u(*, n) with g
    double h = 1.0 / n;
    for (int k = 0; k < maxiter; k++) {
        for (int i = 1; i < n; i++) {
            upc_forall (int j = 1; j < n; j++) {
                unew[i][j] = 0.25 * ( u[i+1][j] + u[i-1][j] +
                                             u[i][j+1] + u[i][j-1] -
                                             h * h * f[i][j] );
            }
        }
        upc_barrier;
        // code to check for convergence of unew to u.
        // Make the new value the old value for the next iteration
        for (int i = 1; i < n; i++) {
            upc_forall(int j = 1; j < n; j++) {
                u[i][j] = unew[i][j];
            }
        }
    }
}
```
Titanium

- Titanium is a PGAS language based on Java
  - Implementations do not use the JVM
- We show both a serial and parallel version

Titanium Serial Version

```java
public class Poisson_seq {
    public static void main(String[] argv) {
        int n = 10; // grid side length of f grid
        int maxiter = 100; // number of iterations

        double [2d] u = new double [[0,0],[n+1,n+1]]; // initialize u and f
        double [2d] unew = new double [u.domain()];
        double [2d] f = new double [u.domain().shrink(1)];
        double [2d] temp; // used for switching arrays

        double h = 1.0/n;
        for (int i = 0; i < maxiter; i++) {
            foreach (p in unew.domain().shrink(1)) {
                // perform computation
                unew[p] = 0.25 * (u[p + [1, 0]] + u[p + [-1, 0]] + u[p + [0, 1]] + u[p + [0, -1]] - h * h * f[p]);
            }

            // swap u and unew
            temp = unew;
            unew = u;
            u = temp;
        }
    }
}
```
Titanium Version – Part 1

public class Poisson_par {
  public static void main (String[] argv) {
    int n = 10; // grid side length of f (RHS) grid
    int maxiter = 100; // number of iterations

    RectDomain<2> myDomain = [[0, Ti.thisProc()] * n / Ti.numProcs()] :
      [n+1, (Ti.thisProc()+1)* n / Ti.numProcs()+ 1];
    RectDomain<2> myInterior = myDomain.shrink(1);

    // create distributed array (auto-initialized to zero)
    double[1d][] allu = new double[0:1][0:Ti.numProcs()-1] single[2d];
    allu[0].exchange(new double[myDomain]);
    allu[1].exchange(new double[myDomain]);

    // create & initialize f
double[2d] f = new double[myInterior];
f.set(1.0);
    double h = 1.0/n;
    for (int i = 0; i < maxiter; i++) {
      // fetch reference to local arrays
      double[2d] local u = (double[2d])allu[0][Ti.thisProc()];
      double[2d] local unew = (double[2d])allu[1][Ti.thisProc()];

      // update ghost cells
      if (Ti.thisProc() > 0)
        allu[0][Ti.thisProc()-1].copy(u.restrict(myInterior));
      if (Ti.thisProc()+1 < Ti.numProcs())
        allu[0][Ti.thisProc()+1].copy(u.restrict(myInterior));
      Ti.barrier();

      // perform computation
      foreach (p in myInterior) {
        unew[p] = 0.25 * (u[p + [ 1, 0]] + u[p + [-1, 0]]
          + u[p + [ 0, 1]] + u[p + [0, -1]])
          - h * h * f[p];
      }
      // swap u and unew
      double[1d] single[2d] temp = allu[0];
      allu[0] = allu[1];
      allu[1] = temp;
    }
  }
}

Titanium Version – Part 2

  // update ghost cells
  if (Ti.thisProc() > 0)
    allu[0][Ti.thisProc()-1].copy(u.restrict(myInterior));
  if (Ti.thisProc()+1 < Ti.numProcs())
    allu[0][Ti.thisProc()+1].copy(u.restrict(myInterior));
  Ti.barrier();

  // perform computation
  foreach (p in myInterior) {
    unew[p] = 0.25 * (u[p + [ 1, 0]] + u[p + [-1, 0]]
      + u[p + [ 0, 1]] + u[p + [0, -1]])
      - h * h * f[p];
  }
  // swap u and unew
  double[1d] single[2d] temp = allu[0];
  allu[0] = allu[1];
  allu[1] = temp;
}
Global Operations

Example: checking for convergence

Serial Version

real u(0:n,0:n), unew(0:n,0:n), twonorm

! ...
  twonorm = 0.0
  do j=1, n-1
    do i=1, n-1
      twonorm = twonorm + (unew(i,j) - u(i,j))**2
    enddo
  enddo
  twonorm = sqrt(twonorm)
  if (twonorm .le. tol) ! ... declare convergence
**MPI Version**

```plaintext
use mpi
real u(0:n,js-1:je+1), unew(0:n,js-1:je+1), twonorm
integer ierr

! ...

twonorm_local = 0.0
do j=js, je
  do i=1, n-1
    twonorm_local = twonorm_local + &
      (unew(i,j) - u(i,j))**2
  enddo
enddo
call MPI_Allreduce( twonorm_local, twonorm, 1, &
  MPI_REAL, MPI_SUMM, MPI_COMM_WORLD, ierr )
twonorm = sqrt(twonorm)
if (twonorm .le. tol) ! ... declare convergence
```

**HPF Version**

```plaintext
real u(0:n,0:n), unew(0:n,0:n), twonorm
!HPF$ DISTRIBUTE u(:,BLOCK)
!HPF$ ALIGN unew with u
!HPF$ ALIGN f with u

! ...

twonorm = sqrt ( &
  sum ( (unew(1:n-1,1:n-1) - u(1:n-1,1:n-1))**2 )
  if (twonorm .le. tol) ! ... declare convergence
endo
OpenMP Version

real u(0:n,0:n), unew(0:n,0:n), twonorm

!..
   twonorm = 0.0
!$omp parallel
!$omp do private(ldiff) reduction(+:twonorm)
   do j=1, n-1
      do i=1, n-1
         ldiff = (unew(i,j) - u(i,j))**2
         twonorm = twonorm + ldiff
      enddo
   enddo
!$omp enddo
!$omp end parallel
   twonorm = sqrt(twonorm)
enddo

The HPCS languages

- DARPA funded three vendors to develop next-generation languages for programming next-generation petaflops computers
  - Fortress (Sun)
  - X10 (IBM)
  - Chapel (Cray)
- All are global-view languages, but also with some notion for expressing locality, for performance reasons.
  - They are more abstract than UPC and CAF in that they do not have a fixed number of processes.
- Sun’s DARPA funding was discontinued, and the Fortress project made public. See http://fortressproject.sun.com
- Work continues at Cray and IBM
Some Experiments With Hybrid Programming on Scalable Computers with Multi-Core Nodes

- These test the portability of the OpenMP + MPI programming model on three quite different architectures
- A well-known example
- A tool for visualizing performance in all three environments

Supercomputers are Hierarchical

- Which programming model is fastest?
  - MPI everywhere?
  - Fully hybrid MPI & OpenMP?
  - Something between (Mixed model)

Often hybrid programming can be slower than pure MPI
**MPI and Threads**

- MPI describes parallelism between processes (with separate address spaces)
- Thread parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism
  - Pthreads provide more complex and dynamic approaches
- We focus here on the OpenMP approach

**Standards Issues**

- Hybrid programming (two programming models) requires that the standards make commitments to each other on semantics.
- OpenMP’s commitment: if a thread is blocked by an operating system call (e.g. file or network I/O), the other threads remain runnable.
  - This is a major commitment; it involves the thread scheduler in the OpenMP compiler’s runtime system and interaction with the OS.
  - What this means in the MPI context: An MPI call like MPI_Recv or MPI_Wait only blocks the calling thread.
- MPI’s commitments are more complex.
MPI’s Four Levels of Thread Safety

- Note that these are not specific to OpenMP
- They are in the form of commitments that the multithreaded application makes to the MPI implementation
  - MPI_THREAD_SINGLE: there is only one thread in the application
  - MPI_THREAD_FUNNELED: there is only one thread that makes MPI calls
  - MPI_THREAD_SERIALIZE: Multiple threads make MPI calls, but only one at a time
  - MPI_THREAD_MULTIPLE: Any thread may make MPI calls at any time

MPI-2 defines an alternative to MPI_Init
- MPI_Init_thread(requested, provided)
  - Allows applications to say what level it needs, and the MPI implementation to say what it provides

What This Means in the OpenMP Context

- MPI_THREAD_SINGLE
  - There is no OpenMP multithreading in the program.

- MPI_THREAD_FUNNELED
  - All of the MPI calls are made by the master thread. I.e. all MPI calls are
    - Outside OpenMP parallel regions, or
    - Inside OpenMP master regions, or
    - Guarded by call to MPI_Is_thread_main MPI call.
      - (same thread that called MPI_Init_thread)

- MPI_THREAD_SERIALIZE
  #pragma omp parallel
  ...
  #pragma omp atomic
  { ...
    ...MPI calls allowed here...
  }

- MPI_THREAD_MULTIPLE
  - Anything goes; any thread may make an MPI call at any time
Threads and MPI in MPI-2

- An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is not required to be thread safe.
- A fully thread-compliant implementation will support MPI_THREAD_MULTIPLE.
- A portable program that does not call MPI_Init_thread should assume that only MPI_THREAD_SINGLE is supported.

For MPI_THREAD_MULTIPLE

- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order.
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions.
- It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls.
- User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads.
The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE (duh).
- They probably support MPI_THREAD_FUNNELED even if they don’t admit it.
  - Does require thread-safe malloc
  - Probably OK in OpenMP programs
- “Thread-safe” usually means MPI_THREAD_MULTIPLE.
- This is hard for MPI implementations that are sensitive to performance, like MPICH2.
  - Lock granularity issue
- “Easy” OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED.
  - So don’t need “thread-safe” MPI for many hybrid programs
  - But watch out for Amdahl’s Law!

Visualizing the Behavior of Hybrid Programs

- Jumpshot is a logfile-based parallel program visualizer of the “standard” type. Uses MPI profiling interface.
- Recently it has been augmented in two ways to improve scalability.
  - Summary states and messages are shown as well as individual states and messages.
    - Provides a high-level view of a long run.
    - SLOG2 logfile structure allows fast interactive access (jumping, scrolling, and zooming) for large logfiles.
Jumpshot and Multithreading

- Newest additions are for multithreaded and hybrid programs that use pthreads.
  - Separate timelines for each thread id
  - Support for grouping threads by communicator as well as by process

Using Jumpshot with Hybrid Programs

- SLOG2/Jumpshot needs two properties of the OpenMP implementation that are not guaranteed by the OpenMP standard
  - OpenMP threads must be pthreads
    - Otherwise, the locking in the logging library necessary to preserve exclusive access to the logging buffers would need to be modified.
  - These pthread ids must be reused (threads are “parked” when not in use)
    - Otherwise Jumpshot would need zillions of time lines.
Three Platforms for Hybrid Programming

- Linux cluster
  - 24 nodes, each consisting of two Opteron dual-core processors, 2.8 Ghz each
  - Intel 9.1 fortran compiler
  - MPICH2-1.0.6, which has MPI_THREAD_MULTIPLE
  - Multiple networks; we used GigE

- IBM Blue Gene/P
  - 40,960 nodes, each consisting of four PowerPC 850 MHz cores
  - XLF 11.1 Fortran cross-compiler
  - IBM’s MPI V1R1M2 (based on MPICH2), which has MPI_THREAD_MULTIPLE
  - 3D Torus and tree networks

- SiCortex SC5832
  - 972 nodes, each consisting of six MIPS 500 MHz cores
  - Pathscale 3.0.99 fortran cross-compiler
  - SiCortex MPI implementation based on MPICH2, has MPI_THREAD_FUNNELED
  - Funky Kautz graph network

Experiments

- Basic
  - Proved that necessary assumptions for our tools hold
    - OpenMP threads are pthreads
    - Thread id’s are reused

- NAS Parallel Benchmarks
  - NPB-MZ-MPI, version 3.1
  - Both BT and SP
  - Two different sizes (W and B)
  - Two different modes (“MPI everywhere” and OpenMP/MPI)
    - With four nodes on each machine

- Demonstrated satisfying level of portability of programs and tools across three quite different hardware/software environments

- But we didn’t get it right the first time…
**It Might Not Be Doing What You Think**

- An early run:
  - Nasty interaction between the environment variables `OMP_NUM_THREADS` and `NPB_MAX_THREADS`

**More Like What You Expect**

- BT class B on 4 BG/P nodes, using OpenMP on each node
## MPI Everywhere

- BT class B on 4 BG/P nodes, using 16 MPI processes

![Image of MPI Everywhere](image_url)

## Observations on Experiments

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Cluster</th>
<th>BG/P</th>
<th>SiCortex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bt-mz.W.16x1</td>
<td>1.84</td>
<td>9.46</td>
<td>20.60</td>
</tr>
<tr>
<td>Bt-mz-W.4x4</td>
<td>0.82</td>
<td>3.74</td>
<td>11.26</td>
</tr>
<tr>
<td>Sp-mz.W.16x1</td>
<td>0.42</td>
<td>1.79</td>
<td>3.72</td>
</tr>
<tr>
<td>Sp-mz.W.4x4</td>
<td>0.78</td>
<td>3.00</td>
<td>7.98</td>
</tr>
<tr>
<td>Bt-mz.B.16x1</td>
<td>24.07</td>
<td>113.51</td>
<td>257.67</td>
</tr>
<tr>
<td>Bt-mz.B.4x4</td>
<td>27.96</td>
<td>124.60</td>
<td>399.23</td>
</tr>
<tr>
<td>Sp-mz.B.16x1</td>
<td>21.19</td>
<td>70.69</td>
<td>165.82</td>
</tr>
<tr>
<td>Sp-mz.B.4x4</td>
<td>24.03</td>
<td>81.47</td>
<td>246.76</td>
</tr>
<tr>
<td>Bt-mz.B.24x1</td>
<td></td>
<td></td>
<td>241.85</td>
</tr>
<tr>
<td>Bt-mz.B.4x6</td>
<td></td>
<td></td>
<td>337.86</td>
</tr>
<tr>
<td>Sp-mz.B.24x1</td>
<td></td>
<td></td>
<td>127.28</td>
</tr>
<tr>
<td>Sp-mz.B.4x6</td>
<td></td>
<td></td>
<td>211.78</td>
</tr>
</tbody>
</table>

On the small version of BT (W), hybrid was better
For SP and size B problems, MPI everywhere is better
On SiCortex, more processes or threads are better than fewer
Conclusions

- This particular benchmark has been studied much more deeply elsewhere
  - Rolf Rabenseifner, “Hybrid parallel programming on HPC platforms,” *Proceedings of EWOMP’03*.

- Adding Hybridness (Hybriditude?) to a well-tuned MPI application is not going to speed it up. So this NPB study doesn’t tell us much.
- More work is needed to understand the behavior of hybrid programs and what is needed for future application development.

More Analysis Tools Needed

- Kojak is particularly designed to look at hybrid programs.
KOJAK / Expert: Example 1 – “Idle threads”

- Idle time
- Probably due to serial parts in the OpenMP parallelization of “phyec”
- I.e., other threads are sleeping

Screenshots, courtesy of SCALASCA / KOJAK group, JSC, FZ Jülich

KOJAK / Expert: Example 2 – “Idle threads”

MPI_Recv is called outside of OpenMP parallel regions

Screenshots, courtesy of SCALASCA / KOJAK group, JSC, FZ Jülich
Conclusion

- Some sources:
  - http://www.openmp.org
  - http://www mpi-forum.org
  - http://upc.lbl.gov
  - http://www.co-array.org
  - http://www.cs.berkeley.edu/projects/titanium

Tutorial Outline

- Parallel Computing Architecture and Resources (1 hr Alice)
- Performance and Implementation (30 min Alice)
- Coffee break
- Parallel Programming Models (1 hr 10 min Rusty)
- Optimization and Current Issues (20 min Alice)

- LUNCH BREAK

- Topics from MPI (1 hr 30 min Bill)
- Coffee break
- How to Design Real Applications (30 min David)
- Hybrid Programming (1 hr Rolf)
- Q&A

Note: all timings are approximate
Single PE Optimization Techniques

• know the architecture details:
  – cache(s), size, access time
  – direct methods to bypass cache?
  – effects of moving data around, e.g., to force cache coherency, a read may trigger a write
  – vector or pseudo-vector processors
• kinds of data locality
  – spatial locality $a(i) = b(i) + c(i)$ (vector-like)
  – temporal locality $a(i) = x(i) + x(i-1)$ (was in cache already)
  – no locality: gather/scatter
    ▪ cdir$cache_bypass
• Each vendor should have a document with architecture specific details

Some examples of optimizing for the processor and system

• Generic Cache-based Concepts
• Power 3,4,5 under IBM AIX
• Opteron
• Some Vector processors/differences
• Layout for vector machines and auto parallelization

Special Thanks to Charles Grassl, Jeff Brooks, Ping Wang
How Caches and Stream Buffers Work

1. The microprocessor requests the value of B(1) from data cache. Data cache does not have B(1), so it requests B(1) from secondary cache.

2. Secondary cache does not have B(1). This is called a secondary cache miss. It retrieves a line (8 words for secondary cache) from local memory. This includes elements B(1-8).

3. Data cache receives a line (4 words for data cache) from secondary cache. This is elements B(1-4).

4. The microprocessor receives B(1) from data cache. When the microprocessor needs B(2) through B(4), it need only go to data cache.

5. When the microprocessor needs B(5), data cache does not have it. Data cache requests B(5) through B(8) from secondary cache, which has them and passes them on.

6. Data cache passes B(5) through B(8) on to the microprocessor as it gets requests. When the microprocessor finishes with them, it requests B(9).

7. Data cache requests a new line of data elements from secondary cache, which does not have them. This is the second secondary cache miss, and it is the signal to the system to begin streaming data.

8. Secondary cache requests another 8-word line from local memory and puts it into another of its three-line compartments.

9. A 4-word line is passed from secondary cache to data cache, and a single value is moved to the microprocessor, B(9).

10. Because streaming has begun, data is now prefetched. Secondary cache anticipates continuing need for consecutive data.

Note 1: Power4/5 has 8 stream buffers per processor
Note 2: in some processors data goes directly to L1, and the only way to get into L2 is when it leaves L1
Cache Issues

**Prefetch**

Without Prefetch:

With Prefetch:

- Hiding latency with the cache can be very effective if done with care

When multiple arrays are needed in cache, there is a benefit in having the leading dimension of a cache-aligned array to be a multiple of the cache-line size

Bandwidth Exploitation

- **Computational intensity**
  - Increase number of flop/s per memory reference
  - E.g., find variable that is constant w.r.t. outer loop
    - Unroll s.t. variable is loaded once, used multiple times
- **Cache Blocking**
  - Work with smaller chunks of data (usually in matrix operations in loops) that are "cache-sized" if possible
- **Load streams**
  - Coding for hardware prefetch may include splitting up loops
  - Load-miss streams are pre-fetched

```plaintext
sum = 0.0
do i=1,N
   sum = sum + a(i)
end do
sum1 = 0.0
sum2 = 0.0
do i=1,N/2
   sum1 = sum1 + a(i)
   sum2 = sum2 + a(i+N/2)
end do
sum = sum1 + sum2
```

Adapted from Grassl, IBM
Example: RISC Power

- **IBM's Power2**
  - Functional Parallelism: up to 6 instructions simultaneously if they don't conflict
  - Pipelining: long sequences of the same operation e.g., multiply and add
  - Cache optimization

- **IBM's Power3**
  - RISC Superscalar (all units can run in parallel) with out-of-order execution, branch predicti data pre-fetch
  - SMP enabled

- **IBM's Power4**
  - Multiprocessor chip with high clock rate
    - Long pipelines, Three cache levels, can use latency hiding
  - Shared Memory
    - Large memory size

- **IBM’s Power5**
  - L3 cache moved to processor side of interconnect fabric

- **BG/L differences**
  - BG/L has minimal L2 cache so attempt L1 cache reuse
  - BG/L does not have a hardware square root
  - Mathematical intrinsic routines (exp, log, …) expensive

---

POWER5 Simultaneous Multi-Threaded

- Presents SMP programming model to software
- Natural fit with superscalar out-of-order execution core

Legend
- Thread0 active
- No Thread active
- Thread1 active

Charles Grassl, IBM
Conventional Multi-Threading

- Threads alternate
  - Nothing shared

Simultaneous Multi-Threading

- Simultaneous execution
  - Shared registers
  - Shared functional units
Manual Pipelining
Old:
For (i=0; i<n; i++)
S=S+a1[i]*a2[i]+a3[i]*a4[i]+
....ak[i];

New:
For (i=0; i<n; i++)
S1=S1+a1[i]*a2[i];
S2=S2+a3[i]*a4[i];
S=S1+S2+....;

Elimination of redundant computations/branches
Old code:
For (i=0; i<n; i++)
getA ( A, ....);
D[i]=A[i]+e[i];

New code:
For (i=0; i<n; i++)
getA ( A, ....);
D[i]=A[i]+e[i];
F[i]=A[i]+ k[i];

If statements
Old:
For (i=0; i<n; i++)
getA ( A, ....);
D[i]=A[i]+e[i];

New:
If (fracke==0) {
For (i=0; i<n; i++)
getA ( A, ....);
Z1=x;
F[i]=A[i]+ k[i];

Loop jamming
Old: For (i=0; i<n; i++)
Do-stuff (x);
} For (j=0; j<n; j++)
Get (y);

New: For (i=0; i<n; i++)
Do-stuff(x);
Get (y);

Sometimes, two or more loops may be combined into one reducing overhead
Depending on chip, this may or may not be an advantage

AMD Opteron Processor

Some PowerSeries Code Examples
Courtesy Ping Wang, LLNL

L1 Instruction Cache 64KB
L1 Data Cache 64KB
System Request Queue
Crossbar
Memory Controller
HyperTransport
L2 Cache 1 MB
16-way assoc

16 instruction bytes fetched per cycle
Fetch
Branch Prediction
Fastpath
Microcode Engine
Instruction Control Unit (72 entries)
FP Decode & Rename
36-entry FP scheduler
ALU
AGU
AGU
FADD
FMISC
FMUL

9-way Out-Of-Order execution

• 36 entry FPU instruction scheduler
• 64-bit/80-bit FP Realized throughput (1 Mul + 1 Add)/cycle: 1.9 FLOPs/cycle
• 32-bit FP Realized throughput (2 Mul + 2 Add)/cycle: 3.4+ FLOPs/cycle

Courtesy John Levesque
Simplified memory hierarchy on the AMD Opteron

16 SSE2 128-bit registers
16 64-bit registers

2 x 8 Bytes per clock, i.e. Either 2 loads, 1 load 1 store, or 2 stores (38 GB/s on 2.4 GHz)

L1 data cache

8 Bytes per clock

L2 cache

16 bytes wide data bus => 6.4 GB/s for DDR400

Main memory

64 Byte cache line
- complete data cache lines are loaded from main memory, if not in L2 cache
- if L1 data cache needs to be refilled, then storing back to L2 cache

64 Byte cache line
- write back cache: data offloaded from L1 data cache are stored here first
  until they are flushed out to main memory

Other Important Techniques

- Function in-lining
- Enable SSE vectorization (when available) - streaming SIMD extensions
  - Fine-grained data parallelism
  - Check compiler output for vectorization of loops
    - C and C++ codes can inhibit vectorization

SIMD is single instruction multiple data
Vector Processor Optimization

• Raise the vectorization ratio
  – Lengthen vector loop length. A vector register on the SX-8 has 256 entries. Make sure that your application makes good use of the maximum vector length

• Maximize efficiency of memory access
  – For a continuous or a constant stride with odd stride, load and store speeds are highest (32GB/s on SX-6) or 64GB/s on SX-8 => 0.5 words per flop.
  – Make sure flop/memory access ratio enables the memory subsystem to keep the arithmetic units on the CPU busy
  – Watch vector strides. On SX-6, e.g., For stride of 2, the rate is down by half (16GB/s). For stride of 4, the rate is down by the half (8GB/s).

• Increase efficiency of floating point operations
  – loop fusion and outer loop unrolling is effective

• SX-8 CPU has a multiply, an add and a divide pipe.
  – For maximum performance, add and multiply operations should be balanced. There is no need to avoid divide or square root operations, since both are done in hardware.

• The memory subsystem of the SX-8 has 4096 banks.
  – Try to distribute the memory load evenly over all banks.

Roadrunner Multiscale: Three levels of parallelism are exposed along with remote offload of an algorithm “solver”

• MPI message passing still used between nodes and within-node (2 levels)
  – Global Arrays, IPC, UPC, Global Address Space (GAS) languages, etc. also remain possible choices
  – Additional parallelism can be introduced within the node (“divide & conquer”)
    • Roadrunner does not require this due to it’s modest scale

• Offload large-grain computationally intense algorithms for Cell acceleration within a node process
  – This is equivalent to function offload and similar to client-server & RPCs
    • One Cell per one Opteron core (1:1 process ratio)
    • Opteron would typically block, but could do concurrent work
  – Embedded MPI communications are possible via “relay” approach

• Threaded fine-grained parallelism within the Cell itself (1 level)
  – Create many-way parallel pipelined work units for SPMD on the SPEs
    • MPMD, RPCs, streaming, etc. are also possible
    • Consistent with heterogeneous chips future trends

• Considerable flexibility and opportunities exist
  From LA-UR-07-6213
  Ken Koch, LANL
Global Techniques

- Compiler Optimizations
- Load Balancing
- Cache-Based Architecture Examples
- Mapping Problem to System
- Use Language Options to fit machine
- Scaling to 10,000 Procs and More

Using directives can give benefit with no major code changes

For example, using the directive:

```c
!DIR$ UNROLL 4
```

is the same as manually unrolling...

```c
DO I = 1, 256
   A(I) = B(I) + 2.5*C(I)
ENDDO
```

...to...

```c
TEMP1 = 2.5*C(1)
TEMP2 = 2.5*C(2)
TEMP3 = 2.5*C(3)
TEMP4 = 2.5*C(4)
DO I = 1, 252, 4
   A(I) = B(I) + TEMP1
   TEMP1 = 2.5*C(I +4)
   A(I+1) = B(I +1) + TEMP2
   TEMP2 = 2.5*C(I +5)
   A(I+2) = B(I +2) + TEMP3
   TEMP3 = 2.5*C(I +6)
   A(I +3) = B(I +3) + TEMP4
   TEMP4 = 2.5*C(I +7)
END DO
```

On X1, using non-cachable & unroll(4) pragma improves strided access bandwidth test by 20X (result measured by Oliker, 2004)

Many compilers now have an optimization flag that covers several optimizations at once, e.g., on the PG compilers, try `-fastsse -Mipa=fast` for opteron
Sample of Incremental Optimization

- Compiler options for optimization are added with no code changes: aggress, noieeedivide, unroll, rdahead
- Memory optimization for specific architecture:
  - 2-d arrays have frequent stride-one addressing. Replace this with explicit one-dimensional addressing
  - \( \text{KUM(ImT, JMT)} \rightarrow \text{KMU(ImT*JMT)} \)
- Use optimized libraries
  - Matrix and vector computations done using BLAS
  - Authors found that use of libraries was not always an improvement -- it had to be tested at each increment
- Replace logical IF and WHERE statements (to distinguish ocean from land points) with masking arrays which store 1 for ocean points and 0 for land points. Arrays then used as multiplying factors

Chao, Li, Wang, Katz, Cheng, and Whitman, Caltech & Equator Tech. in ISPC

Result of Optimization Process

- Optimization techniques are added incrementally
- Addition of mask arrays increases number of floating point operations, yet decreases the total run time
- Using mask arrays introduces code which can be replaced by BLAS calls (use of libraries)
- Substituting floating point ops for conditional instr.

<table>
<thead>
<tr>
<th>Floating Point Ops (x 10^9)</th>
<th>Time (sec)</th>
<th>Rate (GFLOPS)</th>
<th>Optimization Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1904</td>
<td>6779.26</td>
<td>0.281</td>
<td>Original</td>
</tr>
<tr>
<td>2305</td>
<td>6046.90</td>
<td>0.381</td>
<td>Mask</td>
</tr>
<tr>
<td>2305</td>
<td>4517.81</td>
<td>0.510</td>
<td>Mask+BLAS</td>
</tr>
<tr>
<td>2305</td>
<td>3337.32</td>
<td>0.690</td>
<td>Mask+BLAS+Compiler Options</td>
</tr>
<tr>
<td>2305</td>
<td>2858.56</td>
<td>0.807</td>
<td>Full Optimization</td>
</tr>
</tbody>
</table>

Chao, Li, Wang, Katz, Cheng, and Whitman, Caltech & Equator Tech. in ISPC
Optimized code shows improvement in both weak and strong scaling

- Wall-clock time per time step per processor shows optimization benefit
- Scaled Results: Problem size per processor is fixed

Original Code: 182 times faster on 256 processors
Optimized Code: 250 times faster on 256 processors

Getting good performance may involve mapping application to architecture

- MPI process → node
  - for distributed memory system → max : 640 (nodes)

- Microtask thread → CPU on a node
  - for shared memory system → max : 8 (CPUs)

- Vector processing → Vector unit on a CPU
  - for vector processor → 256 (vector register length)

AFES -- Atmospheric General Circulation Model (AGCM) in F90 on Earth Simulator
Parallel processing & physical layout

Grid Points: (I,J,K)=(3840,1920,96)

Latitudinal decomposition

One MPI process

T1279L96

I=3840

J/P = 1920/640 = 3

K=96

PN01

PN02

PN03

vector processing

One microtask

I=3840

K=96

I=3840

J=1920

PN640.

PN=AP8

AP2

AP3

AP4

AP5

AP6

AP7

AP8

vector processing

Scalable performance (AFES T1279L96, 10 steps)

<table>
<thead>
<tr>
<th>Total CPUs</th>
<th>Nodes</th>
<th>CPUs</th>
<th>Elapsed time (sec)</th>
<th>Speedup</th>
<th>Tflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>/Node</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>80 x 1</td>
<td>238.04</td>
<td>1.00</td>
<td>0.52</td>
<td>81.1%</td>
</tr>
<tr>
<td>160</td>
<td>160 x 1</td>
<td>119.26</td>
<td>2.00</td>
<td>1.04</td>
<td>81.0%</td>
</tr>
<tr>
<td>320</td>
<td>320 x 1</td>
<td>60.52</td>
<td>3.93</td>
<td>2.04</td>
<td>79.8%</td>
</tr>
<tr>
<td>640</td>
<td>80 x 8</td>
<td>32.06</td>
<td>7.42</td>
<td>3.86</td>
<td>75.3%</td>
</tr>
<tr>
<td>1280</td>
<td>160 x 8</td>
<td>16.24</td>
<td>14.66</td>
<td>7.61</td>
<td>74.3%</td>
</tr>
<tr>
<td>2560</td>
<td>320 x 8</td>
<td>8.52</td>
<td>27.93</td>
<td>14.50</td>
<td>70.8%</td>
</tr>
<tr>
<td>5120</td>
<td>640 x 8</td>
<td>4.65</td>
<td>51.18</td>
<td>26.58</td>
<td>64.9%</td>
</tr>
</tbody>
</table>

This results was measured for only 10 time steps excluding pre-post, and radiation processes.

Satoru Shingu (ESC)
HPF Directives for Automatic Parallelization: Application-fusion

- The first dimension of three-dimensional array variable is reserved for vectorization to secure enough vector length.
- The third dimension of them is used for HPF parallelization with (*,*,BLOCK) DISTRIBUTE and SHADOW.
- All loops except one loop of reduction operation are automatically parallelized without INDEPENDENT directives.

Example of HPF Automatic Parallelization

```hpflang
parameter(lx=1024, ly=1024, lz=1024)
!HPF$ PROCESSORS es(NUMBER_OF_PROCESSORS())
dimension sr(lx,ly,lz), sm(lx,ly,lz)
!HPF$ DISTRIBUTE (*,*,BLOCK) onto es :: sr, sm
!HPF$ SHADOW (0,0,0:1) :: sr, sm
... do iz = 1, lz-1
    do iy = 1, ly
        do ix = 1, lx
            wul = sm(ix,iy,iz+1) / sr(ix,iy,iz+1)
        ... end do
    end do
... do iz = 1, lz-1
    do iy = 1, ly
        do ix = 1, lx
            sram = max( sram, ... )
        ... end do
... end do
```

Sakagami, et al. Himeji Institute
TFLOPS and Efficiency of Automatic Communication

- TFLOPS (solid) and efficiency (hollow) to the theoretical peak performance are shown with different mesh sizes.
- Adding additional HPF directives boosts performance to 14.9 Tflop/s

Application strategies for scalability

- How to scale applications to O(100K) cpus
  - Use local operations as much as possible
  - Careful load balancing is required
    - Rebalancing may be required for dynamic applications (e.g., automatic mesh refinement or Monte Carlo methods)
    - Analysis of MPI messaging pattern may lead to increased performance by changing task layout to minimize hops
  - Careful use of global operations that map onto special network hardware
  - Assuming a pattern is a useful technique for scalability
- Single node performance
  - SIMD dual FPU operations
  - 16B alignment of data
  - Careful use of L3 cache
  - Which node mode usage?
- LWK (Light Weight Kernel) Issues
  - Many of the large processor architectures use LWK (BlueGene, Red Storm)
  - Code modifications may be necessary, programming models restricted
- multi-core issues
We recommend the use of open-source libraries and other aids when developing codes

Team Development Approach: text messaging allows modern developers to work together

- Text messaging takes over
  - Probably single most-used resource in cutting development time

Chat Room Security
- LLNL internal chat does not work from offsite unless you set up an SSH tunnel to the server.
- Not all rooms are “public”, a user must be invited before they can successfully join a room.
- There may be more rooms in existence than what you see: rooms may be defined to be invisible.
Team Development Approach - 2: TiddlyWiki provides an easy collective document

TiddlyWiki
a reusable non-linear personal web notebook

- **Wiki**
  - enables documents to be written collectively (co-authoring) in a simple markup language using a
    web browser
  - A single page in a wiki is referred to as a "wiki page"
  - entire body of pages, which are usually highly interconnected via hyperlinks, is called "the wiki"
  - An easy-to-use, easy-to-write data base

- **TiddlyWiki**
  - Microcontent WikiWikiWeb
  - Written in HTML, CSS and JavaScript
  - Runs on any modern browser without needing any ServerSide logic
  - Allows anyone to create personal self contained hypertext documents that can be posted to any
    webserver, sent by email or kept on a USB thumb drive
  - written by Jeremy Ruston "CascadingStyleSheets

Verification Basics

- **Build Tests**
  - The multitude of libraries means that daily (nightly) builds are necessary
  - Build tests automatically check code out of repository
  - Specify systems of interest
  - Usually finds errors in configuration, not our code

- **Smoke Tests**
  - Designed to see if things are working in general without necessarily verifying correctness -- a few iterations of various problems

- **Unit Tests**
  - Tests that examine smaller units of code for correctness
For verification, choice between simple developmental approach and tools

- **Low-key approach:**
  - Cron script, automatic email, html file, daily checking

- **Sophisticated Production Tools**
  - LLNL’s Shawn Larson has developed Tapestry (next slides)
    - MPI parallel application
    - Batch and interactive
    - Tests spanning multiple codes
    - Multiple test suites support
    - And many more features

- **Open Source Options like Trac integrate other objects**
  - wiki pages
  - tickets
  - changesets
  - reports
  - etc.

---

TAPESTRY
Shawn Dawson
Tutorial Outline

- Parallel Computing Architecture and Resources (1 hr Alice)
- Performance and Implementation (30 min Alice)
- Coffee break
- Parallel Programming Models (1 hr 10 min Rusty)
- Optimization and Current Issues (20 min Alice)

- LUNCH BREAK

- Topics from MPI (1 hr 30 min Bill)
- Coffee break
- How to Design Real Applications (30 min David)
- Hybrid Programming (1 hr Rolf)
- Q&A
Application Supercomputing and the Many-Core Paradigm Shift

Alice Koniges, David Eder
Lawrence Livermore National Laboratory (LLNL)

William Gropp
University of Illinois at Urbana-Champaign

Rusty Lusk
Argonne National Laboratory (ANL)

Rolf Rabenseifner
HLRS, University of Stuttgart, Germany

SC 2008
November 16
Austin, TX, USA
Tutorial S02
PART II

Tutorial Outline

- Parallel Computing Architecture and Resources (1 hr Alice)
- Performance and Implementation (30 min Alice)
- Coffee break
- Parallel Programming Models (1 hr 10 min Rusty)
- Optimization and Current Issues (20 min Alice)

- LUNCH BREAK

- Topics from MPI (1 hr 30 min Bill)
- Coffee break
- How to Design Real Applications (30 min David)
- Hybrid Programming (1 hr Rolf)
- Q&A

Note: all timings are approximate
Topics From MPI
A Quick Review of the Message Passing Interface

William Gropp

University of Illinois at Urbana-Champaign

Outline

- Topics from MPI
  - Quick Reminder
    - Message Passing, location of standards, MPI 2.1 released
    - Parts of MPI you should know
      - Mpiexec
      - Process topology
      - Profiling Interface
      - Thread levels (-> covered later as part of programming models)
  - Efficient Mesh Exchanges
    - With Point-to-point
    - With MPI RMA
  - Example Using Libraries in MPI
  - Experiments on Current Machines
Message Passing Concepts

- Parallel programs consist of separate processes, each with its own address space
  - Programmer manages memory by placing data in a particular process
- Data sent explicitly between processes
  - Programmer manages memory motion
- Collective operations
  - On arbitrary set of processes
- Data distribution
  - Also managed by programmer
    - Message passing model doesn’t get in the way
    - It doesn’t help either

Message Passing Interface

- The Message Passing Interface (MPI) is a standard library specification for implementing the message passing model for parallel computation
- MPI was developed in two steps:
  - MPI-1 provided an API for basic message passing operations, including point-to-point and collective communication and computation
  - MPI-2 extended the message passing model in several important ways, including parallel I/O, one-sided messaging, and dynamic process interactions
- The MPI Forum has continued to work on MPI
  - MPI 2.1, a single version of the MPI standard that incorporates all of MPI-1, MPI-2, and various errata, was accepted by the MPI Forum on September 3, 2008, and is the current official version of the standard
    - Freely available at [www.mpi-forum.org/docs](http://www.mpi-forum.org/docs)
    - A printed version is available purchase at cost
  - MPI 2.2, a minor revision, will be released in mid to late 2009.
  - MPI 3.0 is under development by the MPI Forum and will consider new functionality as well as changes to track the evolution of languages and parallel computer architecture
Undiscovered MPI

• This tutorial assumes that you are familiar with basic MPI programming
• This section covers some lesser known features of MPI, including
  – A portable way to start MPI programs (mpiexec)
  – A portable way to assign MPI processes to processors on a parallel computer to achieve better communication performance
  – A portable way to instrument the MPI part of any program without recompiling your (or someone else’s) code
• Covered later but also part of “Undiscovered MPI”
  – A portable way to request different levels of thread support for hybrid programming
• These are defined as part of the MPI standard

Running MPI Programs

• The MPI-1 Standard does not specify how to run an MPI program, just as the Fortran standard does not specify how to run a Fortran program.
  – Many implementations provided mpirun --np 4 a.out to run an MPI program
• In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
• mpiexec <args> is part of MPI-2, as a recommendation, but not a requirement, for implementors.
  – Many implementations do provide mpiexec
  – You should insist that your implementation does
• Many parallel systems use a batch environment to share resources among users
  – The specific commands to run a program on a parallel system are defined by the environment installed on the parallel computer
Using MPI For Process Placement

- MPI describes how processes communicate
  - Note processors - the distinction is important
  - All communication is between processes in a communicator, with “rank” in the communicator enumerating the processes
  - Different orderings of the ranks achieve different assignments of processes to processors.
- You can create a new mapping of processes to processors by using MPI_Comm_split
  - MPI_Comm_split( old_communicator, 0, key, &new_comm )
  - Set “key” to the desired rank of the MPI process in the new communicator
  - Very general approach; any mapping may be performed this way
  - Use the resulting “new_comm” instead of MPI_COMM_WORLD
- But this still requires that you determine the values of “key” on each process

Cartesian Process Topologies in MPI

- For the special cases of regular (or Cartesian) grids, MPI provides “process topology” routines to create a new communicator with a “better” layout
- For example, when using a regular grid, consider using these routines:
  ```c
  int dims[2], periodic[2];
  for (i=0; i<2; i++) { dims[i] = 0; periodic[i] = 0; }
  MPI_Dims_create( size, 2, dims );
  MPI_Cart_create( MPI_COMM_WORLD, 2, dims, periodic, 1, &newComm );
  ```
- MPI_Dims_create returns a good “shape” for the grid on the parallel system
- The “1” tells MPI_Cart_create to reorder the mapping of processes to create a “better” communicator for neighbor communication.
- Use newComm instead of MPI_COMM_WORLD in neighbor communication
- There’s also an MPI_Graph_create, but it isn’t very useful (it is too general).
- As we’ll see later, not all MPI implementations do a good job of implementing this routine
  - They could - you should insist on it
Tools Enabled by the MPI Profiling Interface

- The MPI profiling interface: how it works
- Some freely available tools
  - Those to be presented in other talks
  - A few that come with MPICH2
    - SLOG/Jumpshot: visualization of detailed timelines
    - FPMPI: summary statistics
    - Collcheck: runtime checking of consistency in use of collective operations

The MPI Profiling Interface

Call MPI_Send

Call MPI_Bcast

User Program

Profiling Library

MPI_Send

PMPI_Send

MPI_Bcast

MPI_Send

MPI Library
Building Efficient Mesh Exchanges in MPI

• Arises even in common grid exchange patterns
• Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable stalls

Building Efficient Mesh Exchanges in MPI

• Consider a mesh exchange, like the one used to illustrate the different parallel programming models
• We'll look at a 2-dimensional decomposition of a 2-d mesh
Sample Code

- Do $i=1,n_{\text{neighbors}}$
  - Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag, comm, ierr)
Enddo

- Do $i=1,n_{\text{neighbors}}$
  - Call MPI_Recv(edge,len,MPI_REAL,nbr(i),tag, comm,status,ierr)
Enddo

- May deadlock (no processor completes all sends before needing a receive, so no receive is ever issued)

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of
  - if (has down nbr)
    - Call MPI_Send( ... down ... )
  - if (has up nbr)
    - Call MPI_Recv( ... up ... )
... sequentializes (all except the bottom process blocks)
Fix 1: Use Irecv

- Do i=1,n_neighbors
  Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
  comm,requests(i),ierr)
Enddo
- Do i=1,n_neighbors
  Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
  comm, ierr)
Enddo
- Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
- Does not perform well in practice. Why?

Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)
Mesh Exchange - Step 1

- Exchange data on a mesh

Mesh Exchange - Step 2

- Exchange data on a mesh
Mesh Exchange - Step 3

- Exchange data on a mesh

Mesh Exchange - Step 4

- Exchange data on a mesh
Mesh Exchange - Step 5

- Exchange data on a mesh

Mesh Exchange - Step 6

- Exchange data on a mesh
Timeline from IBM SP

- Note that process 1 finishes last, as predicted

Distribution of Sends
Why Six Steps?

• Ordering of Sends introduces delays when there is contention at the receiver
• Takes roughly twice as long as it should
• Bandwidth is being wasted
• Same thing would happen if using memcpy and shared memory

Fix 2: Use Isend and Irecv

• Do i=1,n_neighbors
  Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
  comm, request(i), ierr)
Enddo
• Do i=1,n_neighbors
  Call MPI_Isend(edge, len, MPI_REAL, nbr(i), tag,
  comm, request(n_neighbors+i), ierr)
Enddo
• Call MPI_Waitall(2*n_neighbors, request, statuses, ierr)
Mesh Exchange - Steps 1-4

- Four interleaved steps

Timeline from IBM SP

Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors
Lesson: Defer Synchronization

• Send-receive accomplishes two things:
  – Data transfer
  – Synchronization
• In many cases, there is more synchronization than required
• Use nonblocking operations and MPI_Waitall to defer synchronization

MPI One-Sided Communication

• The following is joint work with Rajeev Thakur, Argonne National Laboratory
• Three data transfer functions
  – Put, get, accumulate
    ![Diagram](image)
    – We’ll look at Put and Get
• Three synchronization methods
  – Fence
  – Post-start-complete-wait
  – Lock-unlock
• A natural choice for implementing halo exchanges
  – Key Feature: Multiple communication per synchronization
Halo Exchange

- Decomposition of a mesh into 1 patch per process
  - Update formula typically \( a(l,j) = f(a(l-1,j), a(l+1,j), a(l,j+1), a(l,j-1), \ldots) \)
  - Requires access to "neighbors" in adjacent patches

Performance Tests

- "Halo" exchange or ghost-cell exchange operation
  - Each process exchanges data with its nearest neighbors
  - Part of the mpptest benchmark; works with any MPI implementation
    - Even handles implementations that only provide a subset of MPI-2 RMA functionality
    - Similar code to that in halocompare, but doesn’t use process topologies (yet)
  - One-sided version can use any of the 3 synchronization methods
    - Available from http://www.mcs.anl.gov/mpi/mpptest
  - Careful with clock resolution; insensitive to transient effects.
  - Reports minimum measured time

- Experiments were run on
  - Sun Fire SMP at RWTH, Aachen, Germany
  - IBM p655+ SMP at San Diego Supercomputer Center
  - SGI Altix at NASA Ames
  - *Jazz* Linux Cluster at Argonne National Laboratory
Point-to-Point Halo

- For (j=0; j<n_partners; j++) {
  MPI_Irecv( rbuffer[j], len, MPI_BYTE, partners[j], 0,
    MPI_COMM_WORLD, &req[j] );
  MPI_Isend( sbuffer[j], len, MPI_BYTE, partners[j], 0,
    MPI_COMM_WORLD, &req[n_partners+j] );
}
MPI_Waitall( 2*n_partners, req, MPI_STATUSES_IGNORE );
- Rbuffer, sbuffer are arrays of pointers to buffers of length len
- Also compared with persistent communication version of this
  - Persistent version was often the fastest

Performance Tuning RMA

- MPI Provides a variety of flags and info parameters to help tune RMA use
- Our benchmark allows selection of these:
- Fence. This is active-target synchronization with MPI Win fence.
  1. Allocate send and receive buffers with MPI Alloc mem.
  2. Specify "no locks" in MPI Win create.
  3. Provide assert option MPI MODE NOPRECEDE on MPI Win fence before RMA calls and all
     of MPI MODE NOSTORE, MPI MODE NOPUT, and MPI MODE NOSUCCEED on the MPI
     Win fence after the RMA calls.
- Post/Start/Complete/Wait. This is scalable active-target synchronization with MPI Win post,
  MPI Win start, MPI Win complete, and MPI Win wait.
  1. Allocate send and receive buffers with MPI Alloc mem.
  2. Specify "no locks" in MPI Win create.
- Passive. This is passive-target synchronization with MPI Win lock and MPI Win unlock.
  1. Use locktype MPI LOCK SHARED (instead of MPI LOCK EXCLUSIVE).
  2. Do not use a separate MPI Barrier for the target processes to know that all RMA operations
     have completed.
Implementation Use of MPI RMA Tuning

<table>
<thead>
<tr>
<th>Option</th>
<th>SGI Altix</th>
<th>Sun Fire</th>
<th>IBM p655+</th>
<th>MPICH2</th>
<th>OpenMPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocmem w Fence</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nolocks w Fence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asserts w Fence</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All w Fence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocmem w PSCW</td>
<td>NA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nolocks w PSCW</td>
<td>NA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All w PSCW</td>
<td>NA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Locks</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NA - Not implemented; X - Option improves performance

Performance Measurements

- The following graphs compare the performance of the best performing point-to-point operation against MPI RMA with the best choice of tuning options
- Short messages used (emphasizes benefit of RMA model)
- “psendrecv” is persistent sends and receives (point-to-point)
- Putlocksharenb or getlocksharenb is a special case:
  - When passive target RMA is used, a separate synchronization step is needed:
    - If get is used, a barrier or equivalent is needed before the get to ensure that
      the get accesses data from the current iteration
    - If put is used, a barrier or equivalent is needed after the put to ensure that the
      target processes accesses the new data from the put
  - Thus, in the cases using passive target (MPI_Win_lock/MPI_Win_unlock), an
    MPI_Barrier is used as well, and is included in the timings. These are indicated
    with “putlockshared”
  - In many cases in applications, there is some other operation that serves to
    provide the “barrier equivalent” (e.g., an MPI_Allreduce). These results use “nb”
    (no barrier) at the end, as in “putlocksharenb”
SGI Altix Halo Performance

Halo Performance (8 nhrs) Columbia 21
One-Sided Communication on Sun SMP with Sun MPI

Sun Fire E2900 with 8 dual core UltraASPARC IV 1.2 GHz CPUs
Note on this system, with ClusterTools version 5, MPI_Win_create and MPI_Alloc_mem could take seconds. Fixed in version 7, we are told.

MPI RMA on IBM 655p: Effect of Threads

Single SMP (no use of Federation Switch)
8 process version 40 x slower than point-to-point
7 process version RMA is much faster (but still relatively slow)
4 MPI process version is 2x faster than 7 process version
MPI_Barrier takes about 9us; that’s not the problem here.
Observations on MPI RMA

- MPI RMA can provide a performance benefit on systems that have hardware support for remote memory operations
- Systems from SGI and SUN provide very good performance with passive-target MPI RMA
- Some systems appear not to have optimized their MPI RMA implementation
- Few implementations take advantage of the flags provided by MPI to enable optimizations

Poisson Solver in PETSc

- The following 7 slides show a complete 2-d Poisson solver in PETSc. Features of this solver:
  - Fully parallel
  - 2-d decomposition of the 2-d mesh
  - Linear system described as a sparse matrix; user can select many different sparse data structures
  - Linear system solved with any user-selected Krylov iterative method and preconditioner provided by PETSc, including GMRES with ILU, BiCGstab with Additive Schwarz, etc.
  - Complete performance analysis built-in
- Only 7 slides of code!
/ * -*- Mode: C; c-basic-offset:4 ; -*- *
#include <math.h>
#include "petscsles.h"
#include "petscda.h"
extern Mat FormLaplacianDA2d( DA, int );
extern Vec FormVecFromFunctionDA2d( DA, int, double (*)(double,double) );
/* This function is used to define the right-hand side of the
Poisson equation to be solved */
double func( double x, double y ) {
  return sin(x*M_PI)*sin(y*M_PI);}
int main( int argc, char *argv[] )
{
  SLES sles;
  Mat A;
  Vec b, x;
  DA grid;
  int its, n, px, py, worldSize;
  PetscInitialize( &argc, &argv, 0, 0 );

  /* Get the mesh size. Use 10 by default */
  n = 10;
  PetscOptionsGetInt( PETSC_NULL, "-n", &n, 0 );
  /* Get the process decomposition. Default it the same as without
DAs */
  px = 1;
  PetscOptionsGetInt( PETSC_NULL, "-px", &px, 0 );
  MPI_Comm_size( PETSC_COMM_WORLD, &worldSize );
  py = worldSize / px;

  /* Create a distributed array */
  DACreate2d( PETSC_COMM_WORLD, DA_NONPERIODIC, DA_STENCIL_STAR,
              n, n, px, py, 1, 1, 0, 0, &grid );

  /* Form the matrix and the vector corresponding to the DA */
  A = FormLaplacianDA2d( grid, n );
  b = FormVecFromFunctionDA2d( grid, n, func );
  VecDuplicate( b, &x );

  PETSc provides routines to create, allocate, and manage distributed data structures

  PETSC "objects" hide details of distributed data structures and function parameters

  PETSc provides routines to access parameters and defaults

  Solve a Poisson Problem with Preconditioned GMRES
SLESCreate( PETSC_COMM_WORLD, &sles );
SLESSetOperators( sles, A, A, DIFFERENT_NONZERO_PATTERN );
SLESSetFromOptions( sles );
SLESSolve( sles, b, x, &its );

PetscPrintf( PETSC_COMM_WORLD, "Solution is:\n" );
VecView( x, PETSC_VIEWER_STDOUT_WORLD );
PetscPrintf( PETSC_COMM_WORLD, "Required %d iterations\n", its );

MatDestroy( A ); VecDestroy( b ); VecDestroy( x );
SLESDestroy( sles ); DADestroy( grid );
PetscFinalize( );
return 0;

PETSc provides routines that solve systems of sparse linear (and nonlinear) equations.

PETSc provides coordinated I/O (behavior is as-if a single process), including the output of the distributed "vec" object.

/* Mode: C; c-basic-offset:4 ; */
#include "petsc.h"
#include "petscvec.h"
#include "petscdsa.h"

/* Form a vector based on a function for a 2-d regular mesh on the unit square */
Vec FormVecFromFunctionDA2d( DA grid, int n,
    double (*f)( double, double ) )
{
    Vec V;
    int is, ie, js, je, in, jn, i, j;
    double h;
    double **vval;
    
    h = 1.0 / (n + 1);
    DACreateGlobalVector( grid, &V );
    
    DAVecGetArray( grid, V, (void **)vval );
    
    for (j = 0; j < jn; j++) {
        int js = 0;
        for (i = 0; i < in; i++) {
            double x = i * h;
            double y = j * h;
            double v;
            double r = sqrt(x * x + y * y);
            v = f(x, y);
            vval[js][i] = v;
        }
    }
    
    return V;
}
Almost the uniprocess code

Creating a **Sparse** Matrix, Distributed Across All Processes

Creates a parallel distributed matrix using compressed sparse row format
for (i=is; i<ie; i++) {
    for (j=js; j<je; j++) {
        row.j = j; row.i = i; nelm = 0;
        if (j - 1 > 0) {
            vals[nelm] = oneByh2;
            cols[nelm].j = j - 1; cols[nelm++].i = i;
        }
        if (i - 1 > 0) {
            vals[nelm] = oneByh2;
            cols[nelm].j = j; cols[nelm++].i = i - 1;
            vals[nelm] = -4 * oneByh2;
            cols[nelm].j = j; cols[nelm++].i = i;
            if (i + 1 < n - 1) {
                vals[nelm] = oneByh2;
                cols[nelm].j = j + 1; cols[nelm++].i = i;
                MatSetValuesStencil(A, 1, &row, nelm, cols, vals, INSERT_VALUES);
            }
        }
        if (j + 1 < n - 1) {
            vals[nelm] = oneByh2;
            cols[nelm].j = j + 1; cols[nelm++].i = i;
        }
    }
}
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
return A;

Full-Featured PDE Solver

- Command-line control of Krylov iterative method (choice of algorithms and parameters)
- Integrated performance analysis
- Optimized parallel sparse-matrix operations

- Question: How many MPI calls used in example?
Other Libraries

• Many libraries exist
  – Freely available libraries
    ▪ PETSc, ScaLAPACK, FFTW, HDF5, DOUG, GeoFEM, MP_SOLVE, MpCCI, PARASOL, ParMETIS, Prometheus, PSPASES, PLAPACK, S+, TCGMSG-MPI, Trilinos, SPRNG, TAO, …
  – Commercially supported libraries
    ▪ E.g., NAG, IBM PESSL, IMSL, …
  – More at www.mcs.anl.gov/mpi/libraries.html
• These provide the building blocks for building large, complex, and efficient programs

Experiments with Topology and Halo Communication on LC Machines

• The following slides show some results for a simple halo exchange program (halocompare) that tries several MPI-1 approaches and several different communicators:
  – MPI_COMM_WORLD
  – Dup of MPI_COMM_WORLD
    ▪ Is MPI_COMM_WORLD special in terms of performance?
    ▪ Reordered communicator - all even ranks in MPI_COMM_WORLD first, then the odd ranks
      ▪ Is ordering of processes important?
    – Communicator from MPI_Dims_create/MPI_Cart_create
      ▪ Does MPI Implementation support these, and do they help
  • Communication choices are
    – Send/receive
    – Isend/irecv
    – “Phased”
Phased Communication

• It may be easier for the MPI implementation to either send or receive
• Color the nodes so that all senders are of one color and all receivers of the other. Then use two phases
  – Just a “Red-Black” partitioning of nodes
  – For more complex patterns, more colors may be necessary

Experiments with Topology and Halo Communication on LC Machines

• The following slides show some results for a simple halo exchange program (halocompare) that tries several MPI-1 approaches and several different communicators:
  – MPI_COMM_WORLD
  – Dup of MPI_COMM_WORLD
    ▪ Is MPI_COMM_WORLD special in terms of performance?
  – Reordered communicator - all even ranks in MPI_COMM_WORLD first, then the odd ranks
    ▪ Is ordering of processes important?
  – Communicator from MPI_Dims_create/MPI_Cart_create
    ▪ Does MPI Implementation support these, and do they help
• Communication choices are
  – Send/Irecv
  – Isend/Irecv
  – “Phased”
Halo Exchange on BG/L

- 64 processes, co-processor mode, 2048 doubles to each neighbor
- Rate is MB/Sec (for all tables)

<table>
<thead>
<tr>
<th></th>
<th>4 Neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ircv/Send</td>
<td>ircv/Isend</td>
</tr>
<tr>
<td>World</td>
<td>112</td>
<td>199</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>81</td>
<td>114</td>
</tr>
<tr>
<td>Cart_create</td>
<td>107</td>
<td>218</td>
</tr>
</tbody>
</table>

Halo Exchange on BG/L

- 128 processes, virtual-node mode, 2048 doubles to each neighbor
- Same number of nodes as previous table

<table>
<thead>
<tr>
<th></th>
<th>4 Neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ircv/Send</td>
<td>ircv/Isend</td>
</tr>
<tr>
<td>World</td>
<td>64</td>
<td>120</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>48</td>
<td>64</td>
</tr>
<tr>
<td>Cart_create</td>
<td>103</td>
<td>201</td>
</tr>
</tbody>
</table>
Halo Exchange on BG/P

- 64 processes, co-processor mode, 2048 doubles to each neighbor
- Rate is MB/Sec (for all tables)

<table>
<thead>
<tr>
<th></th>
<th>4 Neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>irecv/Send</td>
<td>irecv/isend</td>
</tr>
<tr>
<td>World</td>
<td>208</td>
<td>328</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>219</td>
<td>327</td>
</tr>
<tr>
<td>Cart_create</td>
<td>301</td>
<td>581</td>
</tr>
</tbody>
</table>

Halo Exchange on BG/P

- 256 processes, virtual-node mode, 2048 doubles to each neighbor
- Same number of nodes as previous table

<table>
<thead>
<tr>
<th></th>
<th>4 Neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>irecv/Send</td>
<td>irecv/isend</td>
</tr>
<tr>
<td>World</td>
<td>112</td>
<td>132</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>74</td>
<td>84</td>
</tr>
<tr>
<td>Cart_create</td>
<td>109</td>
<td>132</td>
</tr>
<tr>
<td>World(txyz)</td>
<td>132</td>
<td>177</td>
</tr>
<tr>
<td>Even/Odd(txyz)</td>
<td>78</td>
<td>84</td>
</tr>
<tr>
<td>Cart_create(txyz)</td>
<td>132</td>
<td>177</td>
</tr>
</tbody>
</table>
Halo Exchange on Cray XT3

- 1024 processes, 2000 doubles to each neighbor

<table>
<thead>
<tr>
<th>4 Neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irecv/Send</td>
</tr>
<tr>
<td>World</td>
<td>134</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>118</td>
</tr>
<tr>
<td>Cart_create</td>
<td>114</td>
</tr>
</tbody>
</table>

(P) 4 Neighbors | 8 Neighbors |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irecv/Send</td>
</tr>
<tr>
<td>World</td>
<td>109</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>100</td>
</tr>
<tr>
<td>Cart_create</td>
<td>125</td>
</tr>
</tbody>
</table>

Halo Exchange on Cray XT4

- 1024 processes, 2000 doubles to each neighbor

<table>
<thead>
<tr>
<th>4 Neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irecv/Send</td>
</tr>
<tr>
<td>World</td>
<td>153</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>128</td>
</tr>
<tr>
<td>Cart_create</td>
<td>133</td>
</tr>
</tbody>
</table>

(P) 4 Neighbors | 8 Neighbors |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Irecv/Send</td>
</tr>
<tr>
<td>World</td>
<td>131</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>113</td>
</tr>
<tr>
<td>Cart_create</td>
<td>151</td>
</tr>
</tbody>
</table>
Halo Exchange on Cray XT4

- 1024 processes, SN mode, 2000 doubles to each neighbor

<table>
<thead>
<tr>
<th></th>
<th>4 Neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>irecv/recv</td>
<td>irecv/recv</td>
</tr>
<tr>
<td>World</td>
<td>311</td>
<td>306</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>257</td>
<td>247</td>
</tr>
<tr>
<td>Cart_create</td>
<td>265</td>
<td>275</td>
</tr>
</tbody>
</table>

(P) 4 Neighbors 8 Neighbors

<table>
<thead>
<tr>
<th></th>
<th>irecv/recv</th>
<th>irecv/recv</th>
<th>Phased</th>
<th>irecv/recv</th>
<th>irecv/recv</th>
</tr>
</thead>
<tbody>
<tr>
<td>World</td>
<td>264</td>
<td>268</td>
<td>262</td>
<td>230</td>
<td>233</td>
</tr>
<tr>
<td>Even/Odd</td>
<td>217</td>
<td>217</td>
<td>220</td>
<td>192</td>
<td>197</td>
</tr>
<tr>
<td>Cart_create</td>
<td>300</td>
<td>306</td>
<td>319</td>
<td>256</td>
<td>254</td>
</tr>
</tbody>
</table>

Observations on Halo Exchange

- Topology is important (again)
- For these tests, MPI_Cart_create always a good idea for BG/L; often a good idea for periodic meshes on Cray XT3/4
- Cray performance is significantly under what the “ping-pong” performance test would predict
  - The success of the “phased” approach on the Cray suggests that some communication contention may be contributing to the slow-down
  - To see this, consider the performance of a single process sending to four neighbors
Discovering Performance Opportunities

- Let's look at a single process sending to its neighbors. We expect the rate to be roughly twice that for the halo (since this test is only sending, not sending and receiving).

<table>
<thead>
<tr>
<th>System</th>
<th>4 neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Periodic</td>
<td>Periodic</td>
</tr>
<tr>
<td>BG/L</td>
<td>488</td>
<td>389</td>
</tr>
<tr>
<td>BG/L, VN</td>
<td>294</td>
<td>239</td>
</tr>
<tr>
<td>BG/P</td>
<td>1139</td>
<td>892</td>
</tr>
<tr>
<td>BG/P, VN</td>
<td>468</td>
<td>601</td>
</tr>
<tr>
<td>XT3</td>
<td>1005</td>
<td>1053</td>
</tr>
<tr>
<td>XT4</td>
<td>1634</td>
<td>1773</td>
</tr>
<tr>
<td>XT4 SN</td>
<td>1701</td>
<td>1811</td>
</tr>
</tbody>
</table>

- BG gives roughly double the halo rate. XTn is much higher.
  - It should be possible to improve the halo exchange on the XT by scheduling the communication.
  - Or improving the MPI implementation.

Discovering Performance Opportunities

- Ratios of a single sender to all processes sending (in rate).
- Expect a factor of roughly 2 (since processes must also receive).

<table>
<thead>
<tr>
<th>System</th>
<th>4 neighbors</th>
<th>8 Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Periodic</td>
<td>Periodic</td>
</tr>
<tr>
<td>BG/L</td>
<td>2.24</td>
<td>2.01</td>
</tr>
<tr>
<td>BG/L, VN</td>
<td>1.46</td>
<td>1.81</td>
</tr>
<tr>
<td>BG/P</td>
<td>3.8</td>
<td>2.2</td>
</tr>
<tr>
<td>BG/P, VN</td>
<td>2.6</td>
<td>5.5</td>
</tr>
<tr>
<td>XT3</td>
<td>7.5</td>
<td>9.08</td>
</tr>
<tr>
<td>XT4</td>
<td>10.7</td>
<td>13.7</td>
</tr>
<tr>
<td>XT4 SN</td>
<td>5.47</td>
<td>6.73</td>
</tr>
</tbody>
</table>

- BG gives roughly double the halo rate. XTn is much higher.
  - It should be possible to improve the halo exchange on the XT by scheduling the communication.
  - Or improving the MPI implementation.
Observations on Halo Exchange

- **Topology is important (again)**
- For these tests, MPI_Cart_create always a good idea for BG/L; often a good idea for periodic meshes on Cray XT3/4
  - Not clear if MPI_Cart_create creates optimal map on Cray
  - On BG/P, there is an environment variable controlling mapping. MPI_Cart_create has no effect in vn mode :(.
- **Cray performance is significantly under what the “ping-pong” performance test would predict**
  - The success of the “phased” approach on the Cray suggests that some communication contention may be contributing to the slow-down
    - Either contention along links (which should not happen when MPI_Cart_create is used) or contention at the destination node.
  - To see this, consider the performance of a single process sending to four neighbors

Acknowledgments

- **For computing time:**
  - RWTH Aachen University
  - NASA Ames
  - San Diego Supercomputing Center
  - Oak Ridge National Laboratory
  - Argonne National Laboratory
- **For help with SGI runs**
  - Subhash Saini and Dale Talcott
- **For help with Linux Cluster runs at Argonne**
  - Anthony Chan
Conclusions

• **Avoiding unnecessary synchronization**
  – Enabling overlap of communication and computation
    ▪ Supported by an increasing number of MPI implementations
  – Enable efficient scheduling of network use
    ▪ Particularly on highly scalable systems
  – MPI RMA (where implemented well) offers further improvements

• **Good software design makes it relatively easy to exploit these options**
  – Even better, libraries can provide powerful components and frameworks that can eliminate the need for low-level MPI programming

Tutorial Outline

- Parallel Computing Architecture and Resources (1 hr Alice)
- Performance and Implementation (30 min Alice)
- Coffee break
- Parallel Programming Models (1 hr 10 min Rusty)
- Optimization and Current Issues (20 min Alice)
- LUNCH BREAK
- Topics from MPI (1 hr 30 min Bill)
- Coffee break
- **How to Design Real Applications** (30 min David)
- Hybrid Programming (1 hr Rolf)
- Q&A

Note: all timings are approximate
How to design real applications

• Hydrodynamic simulations and general multiscale issues
• Simulation of Rockets at University of Illinois at Urbana-Champaign
• Astrophysical Thermonuclear Flashes at University of Chicago
• Range of applications running on BG/L at LLNL
  – Propagation and backscatter of light
  – Dislocation dynamics to model strength of materials
  – Fluid instabilities and turbulent mix
  – Investigation of rapid solidification
  – First-principles molecular dynamics

Partitioning overview

Problem

Domains

Processor Mapping

Node 1

Keasler, et al. LLNL
Unstructured mesh requires a connectivity list and larger number of ghost zones

- Domains created using recursive spectral bisection that minimizes surface area and give approximately equal domain volumes

<table>
<thead>
<tr>
<th>Zone</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 4 3</td>
</tr>
<tr>
<td>1</td>
<td>1 2 5 4</td>
</tr>
<tr>
<td>2</td>
<td>3 4 6 7</td>
</tr>
<tr>
<td>3</td>
<td>4 5 7 8</td>
</tr>
</tbody>
</table>

- Unstructured mesh can lead to “stair-step” domain edges & more ghost zones
- Domains small enough for cache have large surface to volume ratios
- Ghost zones are typically over 1/3 of total

Slide (contact) surfaces

Slide (contact) surfaces occur when independent meshes are allowed to interact with each other at mesh boundaries.

This leads to sliding interfaces which result in dynamic communication patterns between domains.

Keasler, et al. LLNL
New algorithm for slide surfaces gives scalability

- Old algorithm did not scale well past 128 processors
- In new algorithm there is no need for global comm
- Nodes stay “aware” of location of neighbors
- Small number of elements per proc (~2500) kept constant

Sphere problem with advection and two slide surfaces

<table>
<thead>
<tr>
<th>Num Processors</th>
<th>Old Cycle Time (s)</th>
<th>New Cycle Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pierce, et al. LLNL

General multiscale issues

- A portion of a simulation, e.g., laser-ray trace, can require time and spatial scales different from the rest
  - Original domain partitioning can be inappropriate
  - Sub-calculation may not need the full set of variables in a zone

- An other example is adaptive mesh refinement (AMR) codes where one may want to solve different physics equations on different levels

- Currently, most codes do not change the number of CPU’s assigned to the simulation during runtime
  - Can be a problem when “fine scale” calculations are only needed after the “rough scale” calculation has been running for some time
Some multiscale simulations will want to use multiple codes to solve complex problems

- Mixed MPMD and SPMD parallelism is required
- One approach is to treat a given code as a “symponent∗,” which can be dynamically launched on dynamically allocated nodes
- SARS∗ - Symponents Architecture Runtime System
  - inter-symponent communication and synchronization
  - exception handling
  - organized into parent-child hierarchy for purposes of error handling
  - checkpoint / restart / failure management
- There are simulations where adaptive sampling is appropriate
  - Fine scale dynamics needed at only small fraction of space and time
  - Not a priori known when or where
  - Fine scale results can be used to set coefficient in coarse modeling

Babel solves some of the key integration challenges for multiscale simulations

- Babel enables bi-directional communication among programs
- No impact on the parallel scalability of application
- Provides a consistent set of object-oriented features across all supported languages
- Remote Method Invocation (RMI) provides interprocess communication
  - Not an MPI replacement, can bind MPI jobs together

Babel Consists of Two Parts:
Code Generator + Runtime Library
Center for Simulation of Advanced Rockets at University of Illinois at Urbana-Champaign

- Simulation of solid propellant rockets, e.g., space shuttle
- Problem requires expertise in diverse disciplines
- Currently, most codes do not change the number of CPU's assigned to the simulation during runtime
- Desire high-resolution, 3-D, integrated model
- Studying potential instabilities and failures modes
- Effects of temperature, strain rate, and structure on stress
- [http://www.csar.uiuc.edu](http://www.csar.uiuc.edu)

Issues associated with running on large (<10,000) processors

- Dynamic determination of neighboring processors is non-viable strategy
- Never access the same file from all processors, never search file system
- Remove any code construct that depends on # of partitions
- Preprocessing tools, and input data structures and layout had to be changed extensively
- Vast majority of our performance bottlenecks have been in the computation, and rarely have anything to do with communication
- Firm believers in tuning the "serial" performance of your application
- We've gotten a great deal of mileage from inlining (either by code construct or compiler option)
Center for Astrophysical Thermonuclear Flashes at University of Chicago

- Modeling explosions due to mass build-up on white dwarfs and neutron stars
  - Observed as an x-ray burst

- Studying element formation in supernovae
  - Iron and all heavy elements formed this way

- Turbulent mixing of multi-component fluids
  - Model nuclear flames on unresolved scales

- Flash is an Eulerian AMR (Adaptive Mesh Refinement) code

- http://www.flash.uchicago.edu

Challenges of multi-core and a reason for using hybrid computing model

- The memory footprint during refinement can be temporarily bigger
  - Uses Paramesh, an oct-tree based block structured AMR package
  - Each block has physical data, and non-trivial amount of meta-data
  - Each refined block generates 8 child blocks in 3D
  - Meta-data is created before redistributing the blocks
  - Application fails during refinement process, even though there is enough memory per core to run the calculation

- In general, when running multiple MPI processes on a core the memory fraction is fixed

- When running multiple threads on a core, the memory can be shared between the tasks

That is why FLASH is looking at hybrid MPI/OpenMP model of communication
Algorithmic necessities of removing global communication

Example: Lagrangian Tracer Particles in Paramesh based Grid

- **Quick Solution:**
  - Copy the entire oct-tree metadata on all processors
  - Computationally extremely efficient, but memory wise extremely inefficient
  - Cannot scale with large number of blocks
  - Dismal failure on machines with very limited memory per processor
    - The tree metadata can use up half or more of the available memory on the 64K processor BG/P if all processors have reasonable number of blocks

  **Solution:** A suite of new parallel algorithms that move particles data without worrying about the topology of the grid as long as good spatial locality is maintained in the distribution of blocks

Improving I/O Performance

- **Split I/O**
  - Instead of a single file, write to a few files
  - Easy in parallel libraries with grouping of processors

- **Utility to stitch them together, most often concatenation enough**
What are the challenges in using BGL?

- The CPU clock is only 700 MHz, so it takes more processors to achieve a given level of performance than on other systems.
- A node only has 512 MB of memory (256 MB per CPU) so it takes more nodes to fit a given problem than on other systems.
- The large number of processors used in a typical BGL run requires some activities to be parallel that could be run on a single processor on other systems.
- BGL uses a microkernel that lacks many of the features of a traditional operating system. There is no support for threads.

Dislocation Dynamics - ParaDiS
(Parallel Dislocation Simulator)

- New LLNL code for direct computation of plastic strength of materials
- Tracks simultaneous motion of millions of dislocation lines

Plasticity occurs by slip along well defined crystal planes and directions

Bulatov, Arsenlis, Bartelt, Cai, Hiratani, Hommes, Rhee, Tang, LLNL
Dislocation lines tend to cluster in space and increase in number

- ParaDiS uses hierarchical recursive bisection to partition problem
  - Originally on single processor but not possible for large problems

- Domain boundaries are shifted to maintain good load balance

- Overlay a regular grid of cubic cells to compute short- and long range forces
  - Short range done explicitly but only require local communication
  - Long range lumped but require global communication

- # of dislocations increases 1-2 orders of magnitude affecting machine usage

Bulatov, Arsenlis, Bartelt, Cai, Hiratani, Hommes, Rhee, Tang, LLNL
Instability and Turbulence - Miranda

- High order hydrodynamics code for computing fluid instabilities and turbulent mix
- Employs FFTs and band-diagonal matrix solvers to compute spectrally-accurate derivatives, combined with high-order integration methods for time advancement
- Contains solvers for both compressible and incompressible flows
- Has been used primarily for studying Rayleigh-Taylor (R-T) and Richtmyer-Meshkov (R-M) instabilities, which occur in supernovae and Inertial Confinement Fusion (ICF)

Cook, Cabot, Welcome, Williams, Miller, et al., LLNL

Transpose methods can be used to solve band-diagonal linear systems

Data are rearranged to give each processor all the data it needs to compute solution in serial

Sample 2D domain decomposition

Block transfer in MPI_Alltoall operation for data transpose

Cook, Cabot, Welcome, Williams, Miller, et al., LLNL
Strong scaling for a fixed problem size of 2048 X 2048 X 512

Weak scaling with 8 X 16 X 2048 grid points per processor
Investigation of rapid solidification

- Use molecular dynamics to uncover details of the solidification process
  - Determine structure and stability of metals under dynamic loading conditions
  - Identify relevant time scales (from ps to µs) for solidification
  - Locate non-equilibrium phase boundaries
  - Describe rate and path dependence of approach to final structure

- Condense results into phenomenological models which can be incorporated into hydrocodes

Molecular dynamics simulations

- Model solidification of tantalum (d-electron, bcc metal with no structural phase transitions)
- Liquid condensed from random (hot gas) positions
- Isothermally compressed on exponential ramp
- Interatomic interaction modeled using MGPT potentials

MGPT Interaction Potentials:

$$E_{MGPT} = E_0(\Omega) + \frac{1}{2} \sum_v v_i(i,j,\Omega) + \frac{1}{6} \sum_o v_o(i,j,k,\Omega) + \frac{1}{24} \sum_{ijkl} v_{ijkl}(i,j,k,l,\Omega) + \ldots$$

where the $v_v(\Omega)$ represent many-body terms which are volume dependent but structure independent and thus rigorously transferable

- About 20x slower than EAM
ddcMD = domain decomposition + Molecular Dynamics

- Independent of MD code - with a simple interface
- Flexible domains
  - Not a regular grid - domains defined by particles not geometry
  - Re-domain to improve performance or to achieve load balance.
    - Large void would not unbalance the load distribution
  - For a given domain structure finds minimum communication.
- No minimum number of particles per domain.
  - Can reduce particles per domain to increase time simulated.

<table>
<thead>
<tr>
<th></th>
<th>MD</th>
<th>ddc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force</td>
<td>86%</td>
<td>1%</td>
</tr>
<tr>
<td>I/O</td>
<td>4%</td>
<td>5%</td>
</tr>
<tr>
<td>Barrier</td>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>90%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Currently running ~ 10 TFlop with 8M atoms on 65,536 CPUs

Small system size has large effect on modeling of solidification

64,000 atoms at 250 ps
256,000 atoms at 250 ps
Is 2,048,000 atoms large enough?

Evolution of largest clusters displays substantial size effects - even at 2M atoms!

- Size of largest clusters at coalescence increases systematically with simulation size
- Growth rates show size dependence for simulation of less than 2M atoms
- Evolution of cluster size is sample size independent for simulations larger than 8M atoms

Streitz, Glosli, Patel, et al., LLNL
Qbox: massively parallel first principles molecular dynamics

- Qbox was written from the ground up to take advantage of massively-parallel supercomputers
- Rewritten to include features needed to accurately simulate high-Z metals
  - multiple simultaneous solutions at different points in the Brillouin zone ("k-points")
  - non-local f-electron pseudopotential support

- Optimized for BlueGene/L
  - good parallel efficiency up to full machine (131,072 cpus)
  - 2006 Gordon Bell award for record peak performance of 207 TFlop/s

First-Principles Molecular Dynamics

For dynamical properties, both ions and electrons have to move simultaneously and consistently:

1. Solve Schrödinger’s Equation to find ground state electron density.
2. Compute inter-atomic forces.
3. Move atoms incrementally forward in time.
4. Repeat.

Because step 1 represents the vast majority (>99%) of the computational effort, we want to choose time steps that are as large as possible, but small enough so that previous solution is close to current one:
Collaboration yields exceptional numerical performance

- To fully exploit BlueGene/L hardware, we need well-optimized single-node kernels for key routines
  - Use double FPU instructions ("double hummer")
  - Use both CPUs on the node
- DGEMM/ZGEMM kernel (John Gunnels, IBM)
  - Hand optimized, uses double FPU very efficiently
  - Algorithm tailored to make best use of L1/L2/L3
  - Dual-core version available: uses all 4 FPUs on the node
- FFTW kernel (Technical University of Vienna)
  - Uses hand-coded intrinsics for DFPU instructions

Physical task layout significantly affects parallel performance

- 65,536 nodes, in a 64x32x32 torus
- 512 tasks per MPI subcommunicator
- 64% speedup!
Qbox allows for platform-specific optimization

sustained performance

1000 Mo atoms:
112 Ry cutoff
12 electrons/atom
1 k-point

Optimization path:
- communication
- four fpu zgemm
- node mapping
- four fpu dgemm

Metals require explicit k-point sampling

- Finite crystalline systems require explicit sampling of the Brillouin zone to accurately represent electrons:
  - each wave vector ("k-point") requires a separate $2\pi/L$ solution of Schrödinger's Equation, increasing the computational cost
- Qbox was originally written for non-crystalline calculations and assumed $k=0$. Rewriting the code for k-points had several challenges:
  - more complicated parallel distribution
  - complex linear algebra
Record performance on BlueGene/L

Summary of real applications

- Reducing global communication is critical for scalability
- Dynamic load balancing often limited by having fixed number of CPU’s
- Single processor/node optimization is still important
- Mapping computational mesh to CPU/node configuration is important
- There are challenges in using BGL type machines
- However, BGL class machines are allowing “first ever” calculations
- Large teams are required to achieve top performance
Tutorial Outline

- Parallel Computing Architecture and Resources  (1 hr Alice)
- Performance and Implementation  (30 min Alice)
- Coffee break
- Parallel Programming Models  (1 hr 10 min Rusty)
- Optimization and Current Issues (20 min Alice)

- LUNCH BREAK

- Topics from MPI (1 hr 30 min Bill)
- Coffee break
- How to Design Real Applications  (30 min David)
- Hybrid Programming (1 hr Rolf)
- Q&A

Note: all timings are approximate

Hybrid Programming – Outline

- Introduction / Motivation
- Programming models on clusters of SMP nodes
- Case Studies / Practical “How-To” on hybrid programming
- Mismatch Problems & Pitfalls
- Opportunities: Application categories that can benefit from hybrid parallelization
- Summary on hybrid parallelization
**Motivation**

Hybrid MPI/OpenMP programming seems natural

- Which programming model is fastest?
- MPI everywhere?
- Fully hybrid MPI & OpenMP?
- Something between? (Mixed model)
- Often hybrid programming slower than pure MPI
  - Examples, Reasons, ...

**Example from SC**

- Pure MPI versus Hybrid MPI+OpenMP (Masteronly)
- What's better? It depends on?

Goals of this part of the tutorial

• Sensitize to problems on clusters of SMP nodes
  see sections → Case studies
  → Mismatch problems & Pitfalls
• Technical aspects of hybrid programming
  see sections → Programming models on clusters
  → "How-To"
• Opportunities with hybrid programming
  see section → Application categories that can benefit from hybrid parallelization

Hybrid Programming – Outline

• Introduction / Motivation

• Programming models on clusters of SMP nodes
  – Repetition from overview given by Rusty Lusk in Section Parallel Programming Models

• Case Studies / Practical “How-To” on hybrid programming
• Mismatch Problems & Pitfalls
• Opportunities: Application categories that can benefit from hybrid parallelization
• Summary on hybrid parallelization
Major Programming models on hybrid systems

- Pure MPI (one MPI process on each CPU)
- Hybrid MPI+OpenMP
  - shared memory OpenMP
  - distributed memory MPI
- Other: Virtual shared memory systems, PGAS, HPF, ...
- Often hybrid programming (MPI+OpenMP) slower than pure MPI
  - why?

Parallel Programming Models on Hybrid Platforms

- pure MPI
  - one MPI process on each CPU
- hybrid MPI+OpenMP
  - MPI: outside of parallel regions of the numerical application code
  - OpenMP: inside of each SMP node
  -MPI only outside of parallel regions
  - Overlapping Comm. + Comp. MPI communication by one or a few threads while other threads are computing
  - Master thread, other threads
  - Master only
Pure MPI

Advantages
- No modifications on existing MPI codes
- MPI library need not to support multiple threads

Major problems
- Does MPI library uses internally different protocols?
  - Shared memory inside of the SMP nodes
  - Network communication between the nodes
- Does application topology fit on hardware topology?
- Unnecessary MPI-communication inside of SMP nodes!

Discussed in detail later on in the section Mismatch Problems

Hybrid Masteronly

Advantages
- No message passing inside of the SMP nodes
- No topology problem

for (iteration …) 
{
  #pragma omp parallel
  numerical code
  /* end omp parallel */

  /* on master thread only */
  MPI_Send (original data to halo areas in other SMP nodes)
  MPI_Recv (halo data from the neighbors)
} /* end for loop

Masteronly
MPI only outside of parallel regions

Major Problems
- All other threads are sleeping while master thread communicates!
- Which inter-node bandwidth?
- MPI-lib must support at least MPI_THREAD_FUNNELED

→ Section on Parallel Programming Models
Overlapping Communication and Computation
MPI communication by one or a few threads while other threads are computing

if (my_thread_rank < ...) {
    MPI_Send/Recv, ...
    i.e., communicate all halo data
} else {
    Execute those parts of the application
    that do not need halo data
    (on non-communicating threads)
}

Execute those parts of the application
that need halo data
(on all threads)

Pure OpenMP (on the cluster)

- Distributed shared virtual memory system needed
- Must support clusters of SMP nodes
- e.g., Intel® Cluster OpenMP
  - Shared memory parallel inside of SMP nodes
  - Communication of modified parts of pages
    at OpenMP flush (part of each OpenMP barrier)

i.e., the OpenMP memory and parallelization model
is prepared for clusters!

Experience: → Mismatch section
Hybrid Programming – Outline

- Introduction / Motivation
- Programming models on clusters of SMP nodes

**Case Studies & Practical “How-To” on hybrid programming**

Courtesy of
Gabriele Jost (University of Texas, TACC / Naval Postgraduate School, Monterey CA)
Georg Hager (Regionales Rechenzentrum Erlangen, RRZE)

- Mismatch Problems & Pitfalls
- Application categories that can benefit from hybrid parallelization
- Summary on hybrid parallelization

Benchmark Characteristics

- **Aggregate sizes and zones:**
  - Class B: 304 x 208 x 17 grid points, 64 zones
  - Class C: 480 x 320 x 28 grid points, 256 zones
  - Class D: 1632 x 1216 x 34 grid points, 1024 zones
  - Class E: 4224 x 3456 x 92 grid points, 4096 zones

  Expectations:
  - Pure MPI: Load-balancing problems!
  - Good candidate for MPI+OpenMP

- **BT-MZ:**
  - Block tridiagonal simulated CFD application
  - Size of the zones varies widely:
    - large/small about 20
    - requires multi-level parallelism to achieve a good load-balance

- **SP-MZ:**
  - Scalar Pentadiagonal simulated CFD application
  - Size of zones identical
    - no load-balancing required

  Load-balanced on MPI level: Pure MPI should perform best

Courtesy of Gabriele Jost (TACC/NPS)
Sun Constellation Cluster Ranger (1)

- Located at the Texas Advanced Computing Center (TACC), University of Texas at Austin (http://www.tacc.utexas.edu)
- 3936 Sun Blades, 4 AMD Quad-core 64bit 2.3GHz processors per node (blade), 62976 cores total
- 123TB aggregate memory
- Peak Performance 579 Tflops
- InfiniBand Switch interconnect
- Sun Blade x6420 Compute Node:
  - 4 Sockets per node
  - 4 cores per socket
  - HyperTransport System Bus
  - 32GB memory

Sun Constellation Cluster Ranger (2)

- Compilation:
  - PGI pgf90 7.1
  - mpi90 -tp barcelona-64 -r8
- Cache optimized benchmarks Execution:
  - MPI MVAPICH
  - setenv
    - OMP_NUM_THREAD NTHREAD
  - ibrun numactl bt-mz.exe
- numactl controls
  - Socket affinity: select sockets to run
  - Core affinity: select cores within socket
  - Memory policy: where to allocate memory
NPB-MZ Class E Scalability on Ranger

- Scalability in MFlops
- MPI/OpenMP outperforms pure MPI
- Use of numactl essential to achieve scalability

Hybrid Programming How-To: Overview

- A practical introduction to hybrid programming
  - How to compile and link
  - Getting a hybrid program to run on a cluster

- Running hybrid programs efficiently on multi-core clusters
  - Affinity issues
    - ccNUMA
    - Bandwidth bottlenecks
  - Intra-node MPI/OpenMP anisotropy
    - MPI communication characteristics
    - OpenMP loop startup overhead
  - Thread/process binding
How to compile, link and run

- Use appropriate OpenMP compiler switch (-openmp, -xopenmp, -mp, -qsmp=openmp, …) and MPI compiler script (if available)
- Link with MPI library
  - Usually wrapped in MPI compiler script
  - If required, specify to link against thread-safe MPI library
    - Often automatic when OpenMP or auto-parallelization is switched on
- Running the code
  - Highly non-portable! Consult system docs! (if available…)
  - If you are on your own, consider the following points
  - Make sure OMP_NUM_THREADS etc. is available on all MPI processes
    - Start "env VAR=VALUE … <YOUR BINARY>" instead of your binary alone
    - Use Pete Wyckoff’s mpiexec MPI launcher (see below):
      http://www.osc.edu/~pw/mpiexec
    - Figure out how to start less MPI processes than cores on your nodes

Some examples for compilation and execution (1)

- Standard Intel Xeon cluster:
  - Intel Compiler
  - mpif90 -openmp ...
  - Execution (handling of OMP_NUM_THREADS, see next slide):

    $ mpirun_ssh -np <num MPI procs> -hostfile machines a.out
Some examples for compilation and execution (2)

Handling of OMP_NUM_THREADS

• without any support by mpirun:
  - E.g. with mpich-1
  - Problem: mpirun has no features to export environment variables to the via ssh automatically started MPI processes
  - Solution: Set
    `export OMP_NUM_THREADS=<# threads per MPI process>`
    in `~/.bashrc`
  - If you want to set OMP_NUM_THREADS individually when starting the MPI processes:
    • Add
      `test -s ~/myexports && . ~/myexports`
      in your `~/.bashrc`
    • Add
      `echo "$OMP_NUM_THREADS=<# threads per MPI process>"` > ~/myexports
      before invoking mpirun
    • Caution: Several invocations of mpirun cannot be executed at the same time with this trick!

Some examples for compilation and execution (3)

Handling of OMP_NUM_THREADS (continued)

• with support by OpenMPI -x option:
  `export OMP_NUM_THREADS=<$# threads per MPI process>`
  `mpirun -x OMP_NUM_THREADS -n <$# MPI processes> .executable`

• Sun Constellation Cluster:
  •
Some examples for compilation and execution (4)

- Cray XT4:
  - `ftn -fastsse -tp barcelona-64 -mp=nonuma ...
  - `aprun -n nprocs -N nprocs_per_node a.out`

- NEC SX8
  - NEC SX8 compiler
  - `mpif90 -C chopt -P openmp ... # -ftrace for profiling info`
  - Execution:
    - `$ export OMP_NUM_THREADS=<num_threads>`
    - `$ MPIEXPORT="OMP_NUM_THREADS"
    - `$ mpirun -nn # MPI procs per node >nnp # of nodes> a.out`

Interlude: Advantages of mpiexec

- Uses PBS/Torque Task Manager (“TM”) interface to spawn MPI processes on nodes
  - As opposed to starting remote processes with ssh/rsh:
    - Correct CPU time accounting in batch system
    - Faster startup
    - Safe process termination
    - Understands PBS per-job nodefile
    - Allowing password-less user login not required between nodes
  - Support for many different types of MPI
    - All MPICHs, MVAPICHs, Intel MPI, ...
  - Interfaces directly with batch system to determine number of procs
  - Downside: If you don’t use PBS or Torque, you’re out of luck...

- Provisions for starting less processes per node than available cores
  - Required for hybrid programming
  - “-pernode” and “-npernode #” options – does not require messing around with nodefiles
Running the code

- Example for using mpiexec on a dual-socket dual-core cluster:

```bash
$ export OMP_NUM_THREADS=4
$ mpiexec -pernode ./a.out
```

- Same but 2 MPI processes per node:

```bash
$ export OMP_NUM_THREADS=2
$ mpiexec -npernode 2 ./a.out
```

- Pure MPI:

```bash
$ export OMP_NUM_THREADS=1 # or nothing if serial code
$ mpiexec ./a.out
```

Intra-node MPI characteristics: IMB Ping-Pong benchmark

- Code (to be run on 2 processors):

```fortran
wc = MPI_WTIME()
do i=1,NREPEAT
   if(rank.eq.0) then
      MPI_SEND(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD,ierr)
      MPI_RECV(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD, &
                 status,ierr)
   else
      MPI_RECV(...)  
      MPI_SEND(...)  
   endif
endo
c
```

- Intranode (1S): `mpirun -np 2 -pin "1 3" ./a.out`
- Intranode (2S): `mpirun -np 2 -pin "2 3" ./a.out`
- Internode: `mpirun -np 2 -pernode ./a.out`
IMB Ping-Pong on DDR-IB Woodcrest cluster: Bandwidth Characteristics
Intra-Socket vs. Intra-node vs. Inter-node

Hybrid Programming – Outline

- Introduction / Motivation
- Programming models on clusters of SMP nodes
- Case Studies / Practical “How-To” on hybrid programming

- Mismatch Problems & Pitfalls
  - Opportunities: Application categories that can benefit from hybrid parallelization
  - Summary on hybrid parallelization
Mismatch Problems & Pitfalls

- None of the programming models fits to the hierarchical hardware (cluster of SMP nodes)
- Several mismatch problems → following slides
- Benefit through hybrid programming → opportunities, see next section
- Quantitative implications → depends on your application

<table>
<thead>
<tr>
<th>Examples:</th>
<th>No.1</th>
<th>No.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benefit through hybrid (see next section)</td>
<td>30%</td>
<td>10%</td>
</tr>
<tr>
<td>Loss by mismatch problems</td>
<td>-10%</td>
<td>-25%</td>
</tr>
<tr>
<td>Total</td>
<td>+20%</td>
<td>-15%</td>
</tr>
</tbody>
</table>

In most cases: Both categories!

The Topology Problem with pure MPI

Exa.: 2 SMP nodes, 8 cores/node

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Minimizing inter-node communication, i.e.,</td>
</tr>
<tr>
<td>- To fit application topology on hardware topology</td>
</tr>
</tbody>
</table>

Solutions for Cartesian grids:
- E.g. choosing ranks in MPI_COMM_WORLD ???
  - round robin (rank 0 on node 0, rank 1 on node 1, ...)
  - Sequential (ranks 0-7 on 1st node, ranks 8-15 on 2nd ...)

... in general
- load balancing in several steps (e.g. with ParMetis):
  - all cells among the SMP nodes
  - Inside of each SMP node:
    → distribute cells among CPU sockets
  - Inside of each CPU socket:
    → distribute cells among all cores

→ or using hybrid programming models
The Topology Problem with hybrid MPI+OpenMP

Problem
- Does application topology inside of SMP parallelization fit on inner hardware topology of each SMP node?

Solutions:
- Domain decomposition inside of each thread-parallel MPI process, and
- first touch strategy with OpenMP

Successful examples:
- Multi-Zone NAS Parallel Benchmarks (MZ-NPB)

The Mapping Problem with mixed model

Do we have this? ... or that?

Several multi-threaded MPI process per SMP node:

Problem
- Where are your processes and threads really located?

Solutions:
- Depends on your platform,
- e.g., lbrun numactl option on Sun

As seen in case-study on Sun Constellation Cluster Ranger with BT-MZ and SP-MZ
Unnecessary intra-node communication

Problem:
- If several MPI processes on each SMP node → unnecessary intra-node communication

Solution:
- Only one MPI process per SMP node

Remarks:
- MPI library must use appropriate fabrics / protocol for intra-node communication
- Intra-node bandwidth higher than inter-node bandwidth → problem may be small
- MPI implementation may cause unnecessary data copying → waste of memory bandwidth

Quality aspects of the MPI library

Sleeping threads and network saturation with Masteronly

for (iteration ...)
{
    #pragma omp parallel
    numerical code
    /*end omp parallel */
    /* on master thread only */
    MPI_Send (original data to halo areas in other SMP nodes)
    MPI_Recv (halo data from the neighbors)
    /* end for loop */

Problem 1:
- Can the master thread saturate the network?
Solution:
- If not, use mixed model
  - i.e., several MPI processes per SMP node

Problem 2:
- Sleeping threads are wasting CPU time
Solution:
- Overlapping of computation and communication

Problem 1&2 together:
- Producing more idle time through lousy bandwidth of master thread
OpenMP: Additional Overhead & Pitfalls

- Using OpenMP
  - may prohibit compiler optimization
  - may cause significant loss of computational performance
- Thread fork / join
- On ccNUMA SMP nodes:
  - E.g. in the masteronly scheme:
    - One thread produces data
    - Master thread sends the data with MPI
      - data may be internally communicated from one memory to the other one
- Amdahl’s law for each level of parallelism
- Using MPI-parallel application libraries?
  - Are they prepared for hybrid?

Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

Three problems:
- the application problem:
  - one must separate application into:
    - code that can run before the halo data is received
    - code that needs halo data
  - very hard to do !!!
- the thread-rank problem:
  - comm. / comp. via thread-rank
  - cannot use work-sharing directives
  - loss of major OpenMP support (see next slide)
- the load balancing problem

```c
if (my_thread_rank < 1) {
    MPI_Send/Recv....
} else {
    my_range = (high-low-1) / (num_threads-1) + 1;
    my_low = low + (my_thread_rank+1)*my_range;
    my_high = high+ (my_thread_rank+1)*my_range;
    my_high = max(high, my_high)
    for (i=my_low; i<my_high; i++) {
        ....
    }
}
```
Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

Subteams

- Important proposal for OpenMP 3.x or OpenMP 4.x

```c
#pragma omp parallel
{
    #pragma omp single onthreads(0)
    {
        #pragma omp for onthreads(1:omp_get_numthreads()-1)
        for (........)
            { /* work without halo information */
            } /* barrier at the end is only inside of the subteam */
        ...
    }
    #pragma omp for
    for (........)
        { /* work based on halo information */
        }
} /* end omp parallel */
```

Barbara Chapman et al.: Toward Enhancing OpenMP’s Work-Sharing Directives.

Experiment: Matrix-vector-multiply (MVM)

- Same experiment on IBM SP Power3 nodes with 16 CPUs per node
- funneled & reserved is always faster in this experiments
- Reason: Memory bandwidth is already saturated by 15 CPUs, see inset
- Inset: Speedup on 1 SMP node using different number of threads

OpenMP/DSM

- Distributed shared memory (DSM)
- Distributed virtual shared memory (DVSM)
- Shared virtual memory (SVM)

- Principles
  - emulates a shared memory
  - on distributed memory hardware

- Implementations
  - e.g., Intel® Cluster OpenMP

Comparison:
MPI based parallelization  ↔  DSM

- MPI based:
  - Potential of boundary exchange between two domains in one large message
  - Dominated by bandwidth of the network

- DSM based (e.g. Intel® Cluster OpenMP):
  - Additional latency based overhead in each barrier
    - May be marginal
  - Communication of updated data of pages
    - Not all of this data may be needed
    - i.e., too much data is transferred
    - Packages may be too small
    - Significant latency
  - Communication not oriented on boundaries of a domain decomposition
    - probably more data must be transferred than necessary

by rule of thumb:
Communication may be 10 times slower than with MPI
No silver bullet

• The analyzed programming models do not fit on hybrid architectures
  – whether drawbacks are minor or major
    ➢ depends on applications’ needs
  – But there are major opportunities → next section

• In the NPB-MZ case-studies
  – We tried to use optimal parallel environment
    ▪ for pure MPI
    ▪ for hybrid MPI+OpenMP
  – i.e., the developers of the MZ codes and we tried to minimize the mismatch problems
  → the opportunities in next section dominated the comparisons

Hybrid Programming – Outline

• Introduction / Motivation
• Programming models on clusters of SMP nodes
• Case Studies / Practical “How-To” on hybrid programming
• Mismatch Problems & Pitfalls

• Opportunities: Application categories that can benefit from hybrid parallelization

• Summary on hybrid parallelization
Opportunities of hybrid parallelization (MPI & OpenMP)

- **Nested Parallelism**
  - Outer loop with MPI / inner loop with OpenMP

- **Load-Balancing**
  - Using OpenMP *dynamic* and *guided* worksharing

- **Memory consumption**
  - Significantly reduction of replicated data on MPI level

- **Opportunities, if MPI speedup is limited due to "algorithmic" problem**
  - Significantly reduced number of MPI processes

- ... (→ slide on ‘Further Opportunities’)

### Nested Parallelism

- **Example NPB: BT-MZ** (Block tridiagonal simulated CFD application)
  - Outer loop:
    - limited number of zones → limited parallelism
    - zones with different workload → speedup < \(\frac{\text{Max workload of a zone}}{\text{Sum of workload of all zones}}\)
  - Inner loop:
    - OpenMP parallelized (static schedule)
    - Not suitable for distributed memory parallelization

- **Principles:**
  - Limited parallelism on outer level
  - Additional inner level of parallelism
  - Inner level not suitable for MPI
  - Inner level may be suitable for static OpenMP worksharing
Load-Balancing
(on same or different level of parallelism)

• **OpenMP enables**
  - Cheap *dynamic* and *guided* load-balancing
  - Just a parallelization option (clause on omp for / do directive)
  - Without additional software effort
  - Without explicit data movement

• **On MPI level**
  - **Dynamic load balancing** requires moving of parts of the data structure through the network
  - Significant runtime overhead
  - Complicated software / therefore not implemented

• **MPI & OpenMP**
  - Simple static load-balancing on MPI level, medium quality dynamic or guided on OpenMP level, cheap implementation

Memory consumption

• **Shared nothing**
  - Heroic theory
  - In practice: Some data is duplicated

• **MPI & OpenMP**
  - **With n threads per MPI process:**
    - Duplicated data is reduced by factor n

• **Future:**
  - **With 100+ cores per chip the memory per core is limited.**
    - Data reduction though usage of shared memory may be a key issue
    - No halos between
Memory consumption (continued)

- **Future:**
  With 100+ cores per chip the memory per core is limited.
  - Data reduction through usage of shared memory may be a key issue
  - Domain decomposition on each hardware level
    - Maximizes
      - Data locality
      - Cache reuse
    - Minimizes
      - CCnuma accesses
      - Message passing
  - No halos between domains inside of SMP node
    - Minimizes
      - Memory consumption

How many multi-threaded MPI processes per SMP node

- **SMP node = 1 Chip**
  - 1 MPI process per SMP node
- **SMP node is n-Chip ccNUMA node**
  - With x NICs (network interface cards) per node
- **How many MPI processes per SMP node are optimal?**
  - somewhere between 1 and n

In other words:

- **How many threads (i.e., cores) per MPI process?**
  - Many threads
    - overlapping of MPI and computation may be necessary,
    - some NICs unused?
  - Too few threads
    - too much memory consumption (see previous slides)
Opportunities, if MPI speedup is limited due to “algorithmic” problems

- Algorithmic opportunities due to larger physical domains inside of each MPI process
  - If multigrid algorithm only inside of MPI processes
  - If separate preconditioning inside of MPI nodes and between MPI nodes
  - If MPI domain decomposition is based on physical zones

Further Opportunities

- Reduced number of MPI messages, reduced aggregated message size compared to pure MPI
- Functional parallelism
  - e.g., I/O in an other thread
- MPI shared memory fabrics not loaded if whole SMP node is parallelized with OpenMP
Hybrid Programming – Outline

- Introduction / Motivation
- Programming models on clusters of SMP nodes
- Case Studies / Practical “How-To” on hybrid programming
- Mismatch Problems & Pitfalls
- Opportunities: Application categories that can benefit from hybrid parallelization

"Hybrid Parallelization” – Summary

“Hybrid” – Summary – the good news

MPI + OpenMP
- Significant opportunity → higher performance on fixed number of cores
- Seen with NPB-MZ examples
  - BT-MZ → strong improvement (as expected)
  - SP-MZ → small improvement (none was expected)
- Usable on higher number of cores
- Advantages
  - Load balancing
  - Memory consumption
  - Two levels of parallelism
    - Outer → distributed memory → halo data transfer → MPI
    - Inner → shared memory → ease of SMP parallelization → OpenMP
- You can do it → “How To”
“Hybrid” – Summary – the bad news

MPI+OpenMP – There is a huge amount of pitfalls:

- Pitfalls of MPI
- Pitfalls of OpenMP
  - On ccNUMA → e.g., first touch
  - Pinning of threads on cores
- Pitfalls through combination of MPI & OpenMP
  - E.g., topology and mapping problems
  - Many mismatch problems
- Tools are available
  - It is not easier than analyzing pure MPI programs
- Most hybrid programs → Masteronly style 😊
- Overlapping communication and computation with several threads
  - Requires thread-safety quality of MPI library
  - Loss of OpenMP support → future OpenMP subteam concept

“Hybrid” – Summary – good and bad

- Problems may be small
  - x% loss efficiency  mismatch problem f x% loss
  - If loss is small x=1%
    and factor f=3 is medium
    → don’t worry ?!
- Optimization
  - 1 MPI process 1 MPI process
    per core ............................................... per SMP node
    ∧ somewhere between
    may be the optimum
- 😍 Efficiency of MPI+OpenMP is not for free:
  The efficiency strongly depends on
  😊 the amount of work in the source code development
“Hybrid” – Summary – Alternatives

Pure MPI
+ Ease of use
– Topology and mapping problems may need to be solved
  (depends on loss of efficiency with these problems)
– Number of cores may be more limited than with MPI+OpenMP
+ Good candidate for perfectly load-balanced applications

Pure OpenMP
+ Ease of use
– Limited to problems with tiny communication footprint
– Source code modifications are necessary
  (Variables that are used with "shared" data scope
  must be allocated as "sharable")
± (Only) for the appropriate application a suitable tool

“Hybrid” – Summary

• This hybrid-section tried to
  – help to negotiate obstacles with hybrid parallelization,
  – give hints for the design of a hybrid parallelization,
  – and technical hints for the implementation → "How To",
  – show tools if the application does not work as designed.

• This hybrid-section was not an introduction into other parallelization models:
  – Partitioned Global Address Space (PGAS) languages
    (Unified Parallel C (UPC), Co-array Fortran (CAF), Chapel, Fortress, Titanium, and X10).
  – High Performance Fortran (HPF)
→ Many rocks in the cluster-of-SMP-sea do not vanish into thin air by using new parallelization models
→ Area of interesting research in next years
“Hybrid” – Conclusions

- Future hardware will be more complicated
  - Heterogeneous
  - ccNUMA quality may be lost on cluster nodes
  - ....
- High-end programming → more complex
- Medium number of cores → more simple
  (if cores / SMP-node will not shrink)
- MPI+OpenMP → work horse on large systems
- Pure MPI → still on smaller cluster
- OpenMP → on large ccNUMA nodes
  (not ClusterOpenMP)

- Further information:
  “Hybrid MPI and OpenMP Parallel Programming”
  SC08 half-day tutorial M09, Monday, 8:30am – 12:00pm

Tutorial–Conclusions

- Item 1
- Item 2

Thank you for your interest

Q & A
Please fill in the feedback sheet – Thank you
Appendix

• Bibliography
• Authors' Biographies
• Index

Bibliography

• Parallel Computing Resources and Performance
  ▪ Hennessy and Patterson, Computer Architecture, Morgan Kaufmann Publishers, 1996
  ▪ Culler and Singh, Parallel Computer Architecture, Morgan Kaufmann Publishers, 1999
  ▪ TOP 500: http://www.top500.org/
  ▪ TOP 500 Clusters: http://clusters.top500.org/
  ▪ NAS Parallel Benchmarks: http://science.nas.nasa.gov/Software/NPB
  ▪ Linpack Parallel Benchmark suite: http://performance.netlib.org
  ▪ Stream Benchmark: http://www.cs.virginia.edu/stream
  ▪ B_eff – parallel communication: http://www.hlrs.de/mpi/b_eff
  ▪ B_eff_io – parallel file I/O: http://www.hlrs.de/mpi/b_eff_io
  ▪ HPC Challenge Benchmark: http://icl.cs.utk.edu/hpcc/
Bibliography

• Tools and Implementation
  – Debugging:
    • Totalview: http://www.etnus.com/
    • DDT: http://www.streamline-computing.com/
    • OpenMP Debugging Tool Assure → Intel® Thread Checker
    • Intel® Thread Checker: http://www.intel.com/software/products/threading/index.htm
  – Performance Analysis
    • VAMPIR:
    • Intel® VTune™: http://www.intel.com/software/products/vtune/index.htm

• Parallel Programming Models and Languages
  – MPI:
    • William Gropp, Ewing Lusk and Rajeev Thakur:
      And
    • MPI: A Message-Passing Interface Standard (1.1, June 12, 1995) http://www.mpi-forum.org/docs
    • MPI-2: Extensions to the Message-Passing Interface (July 18, 1997) http://www.mpi-forum.org/docs
    • MPI-2.1: (Contains all of MPI 1 and MPI 2, with corrections) http://www.mpi-forum.org/docs
    • MPI Forum: http://www.mpi-forum.org/
    • MPICH Homepage: http://www.mcs.anl.gov/mpi/mpich/
    • OpenMPI Homepage: http://www.openmpi.org/
    • HLRS MPI page: http://www.hlrs.de/mpit with links to many other MPI pages
  – General:
    • ONLINE Parallel Programming Workshop http://www.hlrs.de/organization/par/par_prog ws/
    • G. Fox, M. Johnson, G. Lyzenga, S. Otto, S. Salmon, D. Walker:
    • Barry F. Smith, Petter E. Bjerstad, William D. Gropp:
Bibliography

- Parallel Programming Models and Languages (continued)
  - MPI benchmarking:
    - HPC Challenge Benchmark: http://icl.cs.utk.edu/hpcc/
    - B_eff and b_eff_io: http://www.hlrs.de/mpi/b_eff/ and http://www.hlrs.de/mpi/b_eff_io/
- OpenMP:
  - OpenMP Homepage: http://www.openmp.org/
  - OpenMP user group: http://www.community.org/
  - HPF:
Bibliography

• How to Design Real Applications

Bibliography

• Hybrid MPI & OpenMP
  • NAS Parallel Benchmarks: http://www.nas.nasa.gov/Resources/Software/npb.html
  • Rolf Rabenseifner: Comparison of Parallel Programming Models on Clusters of SMP Nodes. In proceedings of the 45nd Cray User Group Conference, CUG SUMMIT 2003, May 12-16, Columbus, Ohio, USA.
Bibliography

- Hybrid MPI & OpenMP (continued)

Further references, see
- Rolf Rabenseifner, Georg Hager, Gabriele Jost, and Rainer Keller: Hybrid MPI and OpenMP Parallel Programming. SC08 tutorial M09.
Dr. Alice E. Koniges

Alice Koniges is an internationally known authority on parallel application development. As leader of the Parallel Applications Technology Program at Lawrence Livermore National Laboratory, she directed researchers in the largest set of agreements between industries and national laboratories ever funded by the US Department of Energy. She has served as a consultant to the Max Planck Institutes of Germany on parallelization and high performance computing issues. She has a Ph.D. in Applied and Computational Mathematics from Princeton University, an MA and an MSME from Princeton and a BA from the University of California, San Diego. As a member of the Center for Applied Scientific Computing at LLNL, her research interests include benchmarking and parallel performance, arbitrary Lagrange Eulerian methods and adaptive mesh refinement algorithms, and plasma physics.

Dr. Ewing Lusk

Rusty Lusk received his B.A. in mathematics from the University of Notre Dame in 1965 and his Ph.D. in mathematics from the University of Maryland in 1970. He is currently a senior computer scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. His current projects include implementation of the MPI Message-Passing Standard, parallel performance analysis tools, and system software for clusters. He is a leading member of the team responsible for MPICH implementation of the MPI message-passing interface standard. He is the author of five books and more than a hundred research articles in mathematics, automated deduction, and parallel computing.
Dr. David Eder

David Eder is a computational physicist at the Lawrence Livermore National Laboratory in California. He has extensive experience with application codes for the study of multiphysics problems. He was awarded a research prize in 2000 for use of advanced codes to design the National Ignition Facility (NIF) 192 beam laser currently under construction. He leads the NIF facility modeling efforts that includes 3D hydrodynamics/radiation transport calculations for debris/shrapnel assessment, 3D electromagnetic simulations for EMP effects, and 3D neutron/gamma Monte Carlo transport calculations for radiation dose evaluations. He has a PhD in Astrophysics from Princeton University and a BS in Mathematics and Physics from the Univ. of Colorado.

Dr. William Gropp

William Gropp is the Paul and Cynthia Saylor Professor in the Department of Computer Science and Deputy Directory for Research for the Institute of Advanced Computing Applications and Technologies at the University of Illinois in Urbana-Champaign. After receiving his Ph.D. in Computer Science from Stanford University in 1982, he held the positions of assistant and associate professor in the Computer Science Department of Yale University. In 1990, he joined the Numerical Analysis group at Argonne, where he held the positions of Senior Scientist and Associate Division Director. His research interests are in parallel computing, software for scientific computing, and numerical methods for partial differential equations. He is a co-author of several books on parallel programming, including Using MPI and Using MPI-2, and is leading the MPI 2.2 effort for the MPI Forum. His current projects include leading the software effort for Blue Waters, the NSF-funded sustained PeteFLOP system to be installed at Illinois, the design and implementation of MPICH, a portable implementation of the MPI Message-Passing Standard, and research into programming models for parallel architectures.

http://www.cs.uiuc.edu/homes/wgropp
Dr. Rolf Rabenseifner

Rolf Rabenseifner studied mathematics and physics at the University of Stuttgart. Since 1984, he has worked at the High-Performance Computing-Center Stuttgart (HLRS). He led the projects DFN-RPC, a remote procedure call tool, and MPI-GLUE, the first metacomputing MPI combining different vendor's MPIs without losing the full MPI interface. In his dissertation, he developed a controlled logical clock as global time for trace-based profiling of parallel and distributed applications. Since 1996, he has been a member of the MPI-2 Forum and since Dec. 2007, he is in the steering committee of the MPI-3 Forum. From January to April 1999, he was an invited researcher at the Center for High-Performance Computing at Dresden University of Technology. Currently, he is head of Parallel Computing - Training and Application Services at HLRS. He is involved in MPI profiling and benchmarking, e.g., in the HPC Challenge Benchmark Suite. In recent projects, he studied parallel I/O, parallel programming models for clusters of SMP nodes, and optimization of MPI collective routines. In workshops and summer schools, he teaches parallel programming models in many universities and labs in Germany.

Index - 1

· Parallel Computing Resources (slide 4)
  - Parallel Architecture Overview (slide 4)
  - Sample architectures in more detail (slide 16)
· Performance and Implementation (slide 48)
  - Measuring and Reporting Performance (slide 49)
  - Tools and Techniques (slide 69)
· Parallel Programming Models (slide 92)
  - Some parallel programming languages and libraries (slide 103)
  - Experiments with hybrid programming (slide 129)
· Optimization and current issues (slide 152)
  - Single CPU Optimizing Issues (slide 153)
  - Global Optimization (slide 169)
  - Other issues and aids (slide 181)
Index - 2

- Topics from MPI (slide 191)
  - Quick reminder (slide 193)
  - Efficient Mesh Exchanges (slide 201)
  - Example using Libraries in MPI (slide 232)
  - Experiments on Current Machines (slide 242)
- How to Design Real Applications (slide 259)
  - Hydrodynamic simulations and general multiscale issues (slide 260)
  - Simulation of rockets and thermonuclear flashes (slide 269)
  - More applications on BG/L and elsewhere (slide 273)

Index - 3

- MPI + OpenMP on clusters of SMP nodes (slide 295)
  - Programming models on clusters of SMP nodes (slide 301)
  - Case Studies / Practical "How-To" on hybrid programming (slide 308)
  - Mismatch Problems & Pitfalls (slide 323)
  - Opportunities: Application categories that can benefit from hybrid parallelization (slide 344)
  - Summary on hybrid parallelization (slide 353)
- Appendix: Bibliography (slide 362), Biographies (slide 371), Index (slide 376)