



Introduction to High Performance Computing

Lecture 01 - Introduction

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg



Persons

- Lecturer: JProf. Dr. Holger Fröning
 - Juniorprofessur, Institut für Technische Informatik, Universität Heidelberg
 - Email: froening@uni-hd.de, Web: <http://ce.uni-hd.de>
- Discussions/Questions
 - During/after the lecture
 - Or via email
- Consulting hours after appointment
- Exercises: Benjamin Baumann



Computer Engineering Group

- “Currently sold on BSP styles of computing for data intensive problems”

▪ Research

- Parallel computing, computer architecture, interconnection networks and hardware design with a recent focus on application-specific heterogeneous computing, data movement optimizations and associated power and energy aspects
- “Most advanced group investigating data movements for accelerated systems“

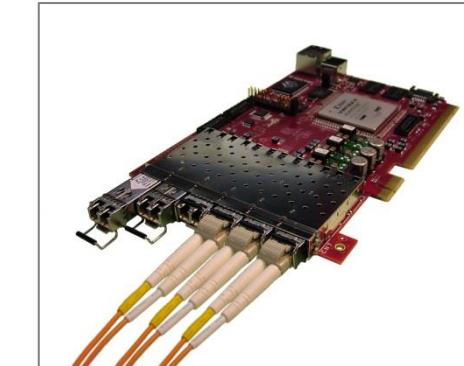
▪ Teaching

- Major „Application-Specific Computing (ASC)“

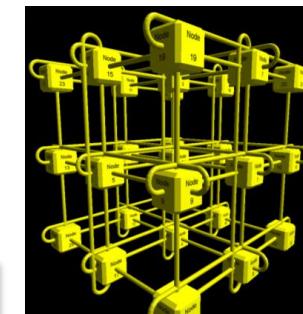
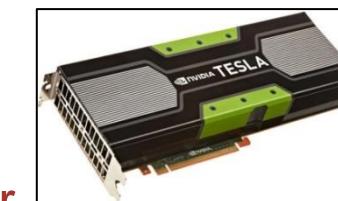
▪ Nvidia CUDA Research and Teaching Center

▪ Interactions

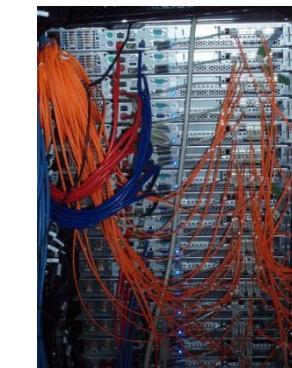
- Technical University of Valencia, Spain
- University of Castilla-La Mancha, Albacete, Spain
- Georgia Institute of Technology, U.S.
- Computer Architecture Group, UOH, Germany
- CERN, Switzerland
- AMD, NVidia, Xilinx, Intel, SAP, EXTOLL, ...



FPGA-based specialized
interconnection network



Visualization of a
network topology



Prototype of a shared-
memory cluster



Get involved!

- In the case of interest plenty of opportunities exist
 - Hardware/Software design up to benchmark level
 - GPUs, CPUs, memory architecture, parallel programming, software environments, performance evaluation, etc
- MSc- and BSc-theses
- Project work
- Seminar work
 - Advanced Seminar „Computer Engineering“
 - 24.10., 14:00 ct, INF501, R102



Organisatorisches



Studiengänge

- MSc „Technische Informatik“
- Studierende mit Haupt- oder Nebenfach Informatik
 - MSc, BSc, LA
- Am Thema Interessierte
 - Physik, DKFZ, Mathematik, ...
- Studien- bzw. Prüfungsleistung:
 - Voraussetzung Leistungsnachweis
 - Punktzahl Übungen $\geq 50\%$
 - Leistungsnachweis: **10.02.2015**
 - Mündliche Prüfung mit Schein



Aufbau

■ VL „Introduction to HPC“

- Vorlesung/Übung als 2+2
- 13 Termine im WS 2014/2015
- Vorlesung Di 14:00-16:00 (ct), Seminarraum A3.04
- 1 Übung Di 16:00-18:00, Terminalraum B2.15
 - Arbeit in 2er Gruppen

■ Anmeldung per Email

- Zu Vorlesungsbeginn
- Mit Name, Matrikelnummer, Studiengang

■ Unterlagen (vorlesungsbegleitende Folien, Übungsblätter, Papers)

- Auf Moodle, Schlüssel ist „mpi_send“



Outline

| | Datum | Vorlesung | Übung |
|----|-------------------|--|---|
| 1 | 21.10.2014 | Introduction | Reading |
| 2 | 28.10.2014 | Introduction to Message Passing | MPI Basics |
| 3 | 04.11.2014 | Basics | MPI latency & bandwidth, MMULT considerations |
| 4 | 11.11.2014 | Parallel Computing | MMULT MPI |
| 5 | 18.11.2014 | Practical Parallel Programming Example | Relaxation single-threaded |
| 6 | 25.11.2014 | Messaging | Relaxation MPI 1 (1D partitioning) |
| 7 | 02.12.2014 | Characteristics | N-Body Preparation |
| 8 | 09.12.2014 | Benchmarks | N-Body Implementation |
| 9 | 16.12.2014 | GPU Computing | CUDA Basics |
| 10 | 13.01.2015 | GPU Computing II | CUDA MMULT + MPI |
| 11 | 20.01.2015 | Interconnection Networks 1 | CUDA MMULT with communication |
| 12 | 27.01.2015 | Interconnection Networks 2/3 | - |
| 13 | 03.02.2015 | Systems / Future | Final discussion |

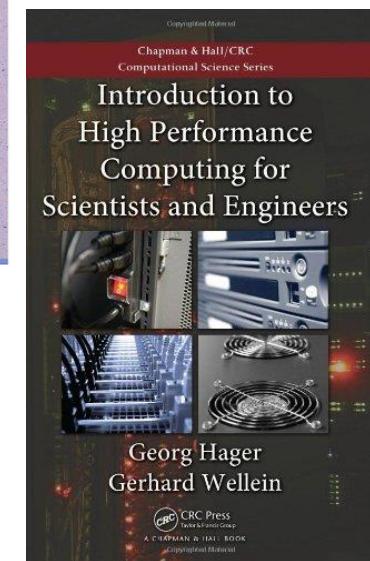
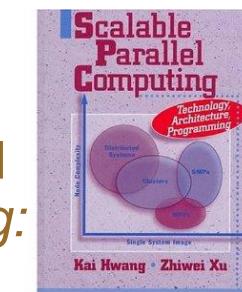
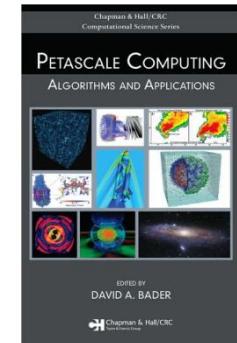
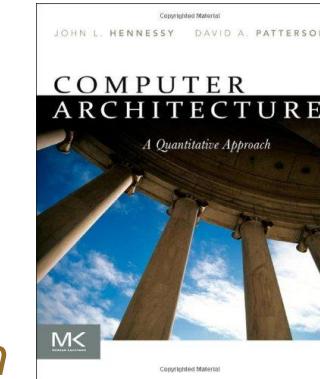


Literatur

- Vorlesungsbegleitende Folien werden laufend zur Verfügung gestellt.

- Literatur

- Georg **Hager**, Gerhard **Wellein**: *Introduction to High Performance Computing for Scientists and Engineers*, Chapman & Hall, 2010
- John L. **Hennessy** & David A. **Patterson**: *Computer Architecture – A quantitative approach*, 5th Ed., Morgan Kaufmann, 2011
- David A. **Bader** (Ed.): *Petascale Computing: Algorithms and Applications*, Chapman & Hall, 2007
- Kai **Hwang** und Zhiwei **Xu**: *Scalable Parallel Computing: Technology, Architecture, Programming*, Mcgraw Hill Book Co, 1997





Feedback

■ Rückmeldungen und Fragen

- Gebt mir Rückmeldungen bzgl. Änderungen oder Ergänzungen zum Stoff
- Stellt Fragen, auch per Email!
- Macht mich auf Fehler aufmerksam!
- Nutzt (außerhalb der Vorlesung) die Möglichkeit, mit Mit-Studierenden zu diskutieren

■ Selbstlernmöglichkeiten

- Literatur
- Bekanntgegebene Weblinks/Papers
- Infrastruktur am ZITI



Vorkenntnisse

■ Voraussetzungen:

- C Kenntnisse, Linux

■ Empfohlen:

- Grundlagen von Rechnerarchitektur, Betriebssysteme

■ Empfohlene Vorlesungen

- VL “Parallele Rechnerarchitekturen von Prof. Brüning”
 - Parallel in diesem Semester
- VL “GPU-Computing”
 - Parallel in diesem Semester
- VL „Advanced Parallel Computing“
 - Folgendes Semester
 - Vertiefung „Application-specific Computing“



Exercises: Scientific Review

- **Reading & Feedback based on paper review**
 - Ideal review here is 3 sentences for each of the following:
 1. Primary contribution
 2. Key insight of the contribution
 3. Your opinion/reaction to the content
 - Review: rating relative to all other papers (of this venue)
 - strong reject, weak reject, weak accept, accept
 - Old papers: optionally include some comments on how right this paper was
- **Reading:**
 - Michael J. Flynn and Patrick Hung. 2005. Microprocessor Design Issues: Thoughts on the Road Ahead. *IEEE Micro* 25, 3 (May 2005), 16-31.
 - Walker, 2008, benchmarking Amazon EC2 for high-performance scientific computing, ;login: *The USENIX Magazine*, 33(5).
- **Provide review (summary) using Moodle until next Tuesday**
- **Discussion round as part of the exercise**



Introduction to High Performance Computing



Questionnaire

| Buzzword | |
|------------------|--|
| Multi-Core | |
| Infiniband | |
| TOP500 | |
| Virtual Memory | |
| ISA | |
| Pthread/OpenMP | |
| Demand Paging | |
| Double Precision | |
| Virtual Machine | |
| Tianhe-2 | |
| MapReduce | |



Vorlesungsinhalte

- Einführung in HPC
 - Gebiet: „Rechnerarchitektur“
 - Großgebiet: Technische Informatik
 - „Innenleben von Rechnern“



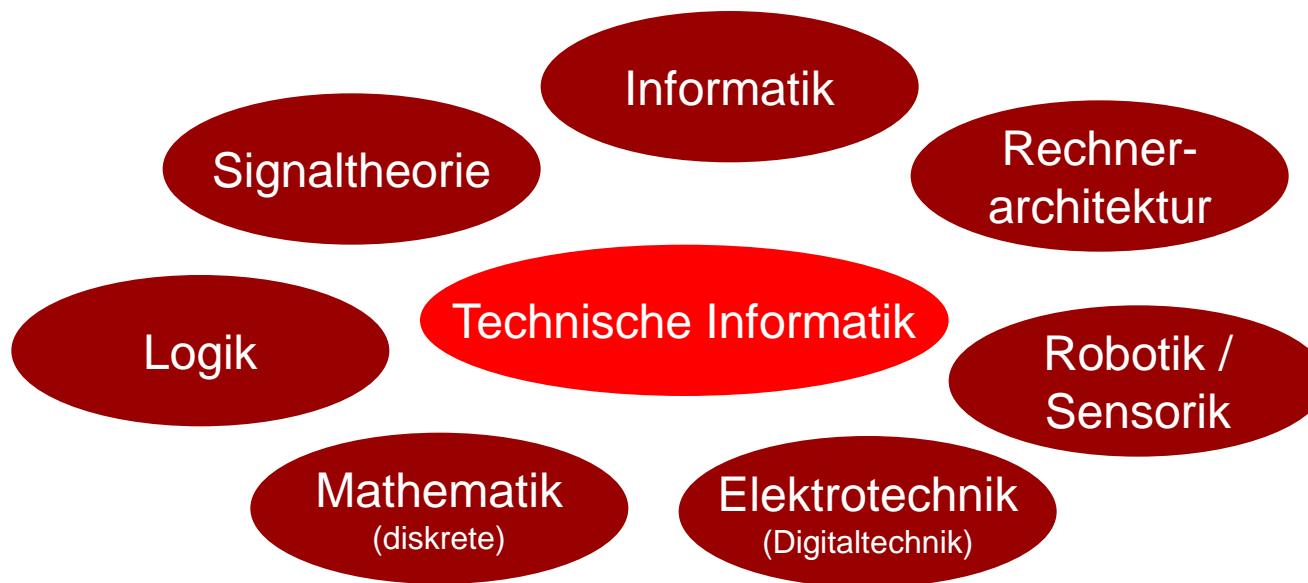
Courtesy: wikipedia.org





Vorlesungsinhalte

■ Großgebiet: Technische Informatik





High Performance Computing - Definition

■ Versuche:

1. *“By HPC we mean all clusters of servers and associated infrastructure used to solve technical/scientific or analytic problems that are computationally intensive or data intensive, and use such techniques as simulation, modeling, etc. This includes technical servers but EXCLUDES desktop computers or workstations used for technical computing.”*
2. *“Verwendung von spezialisierten (evtl. hochparallelen) Systemen zur beschleunigten Berechnung von rechen- bzw. datenintensiven Problemen, d.h. zur Reduzierung der Laufzeit im Vergleich zu Standardrechensystemen durch Erhöhung der Rechenleistung.”*

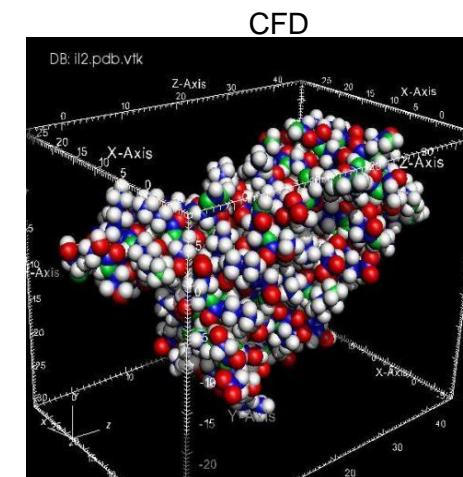
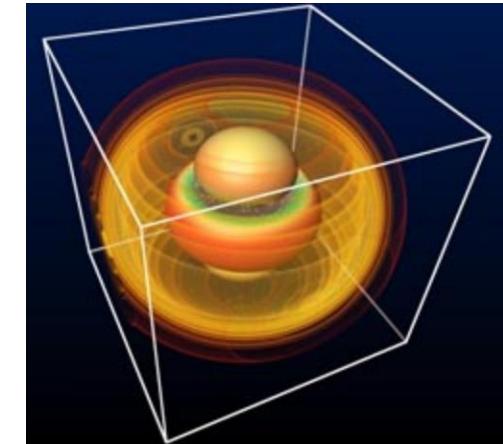
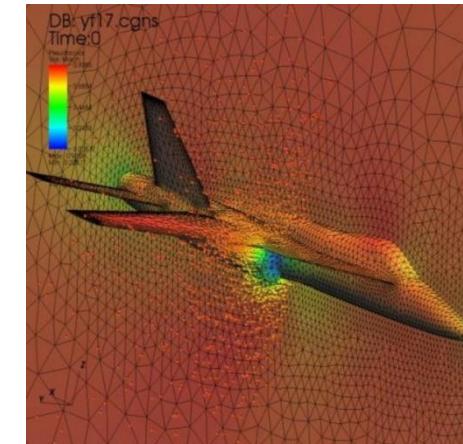
■ Verwandte Themen

- Supercomputing - Exascale

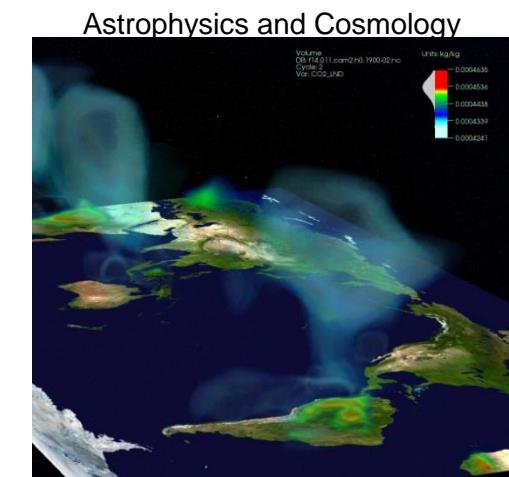


Vorlesungsinhalte

1. Basics
2. Parallel computing
3. MPI & messaging
4. Benchmarks &
application
characteristics
5. Data-parallel
processors
6. Interconnection
networks
7. Future



Molecular Dynamics

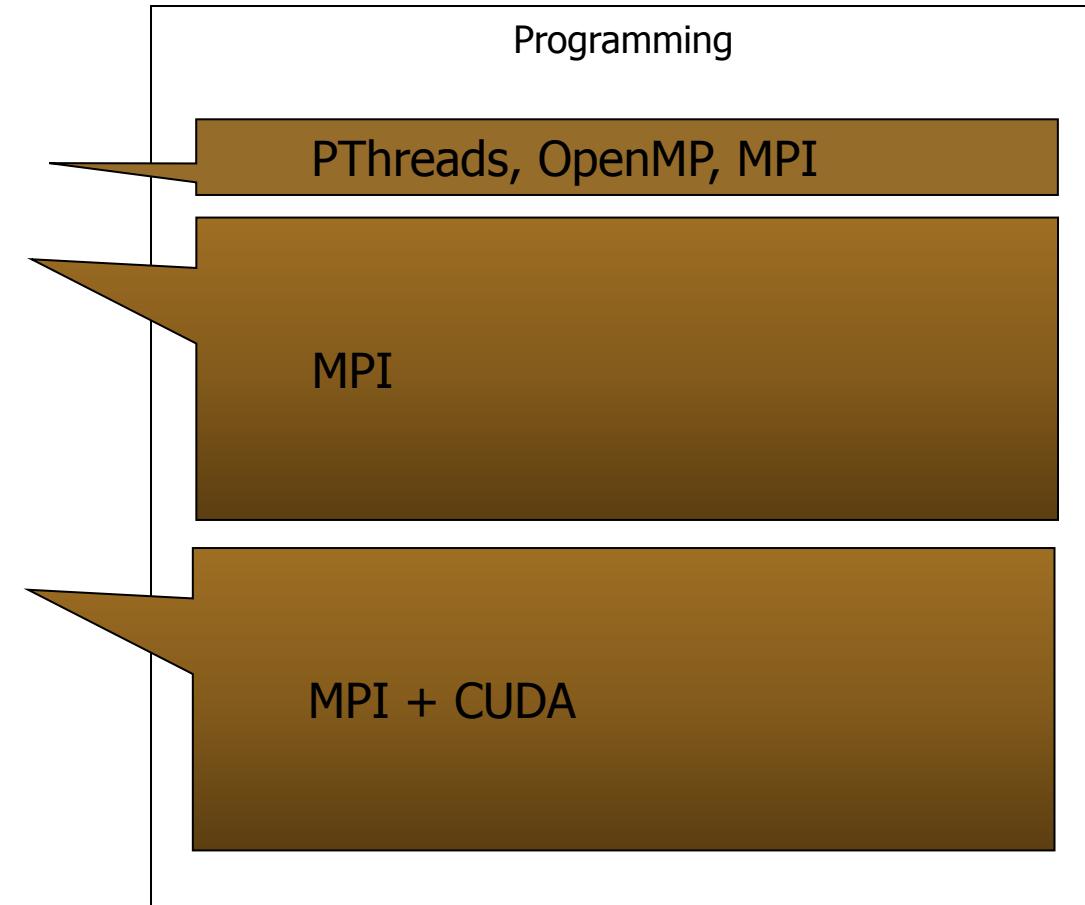


Weather and Climate Research



Vorlesungsinhalte

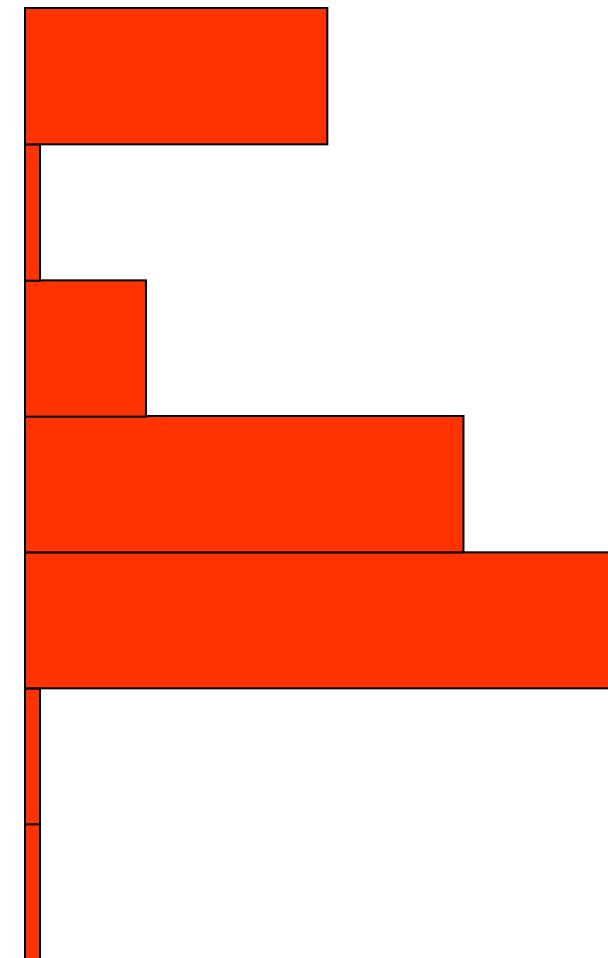
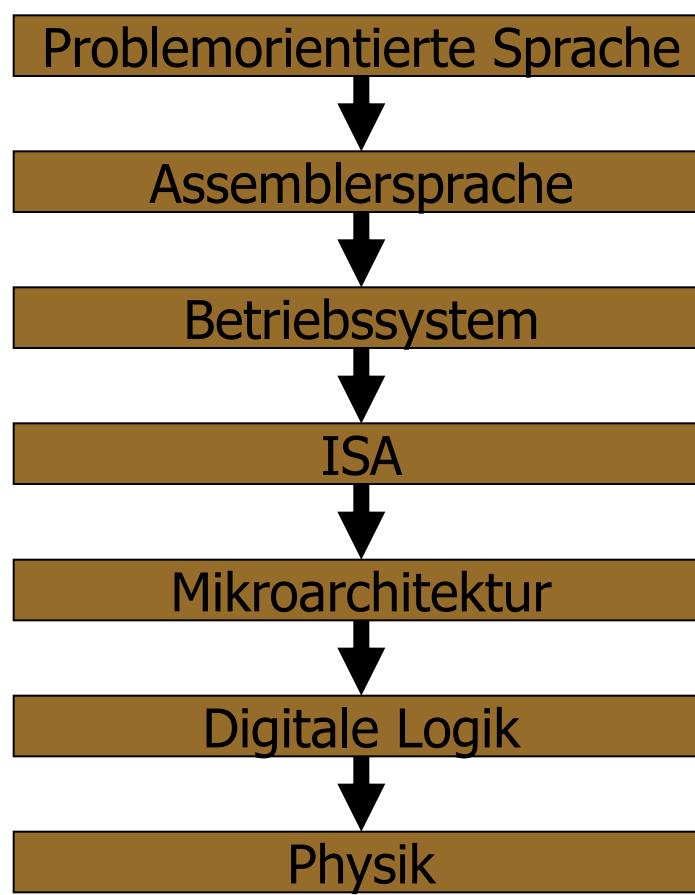
1. Basics
2. Parallel computing
3. MPI & messaging
4. Benchmarks &
application
characteristics
5. Data-parallel
processors
6. Interconnection
networks
7. Future





Vorlesungsinhalte

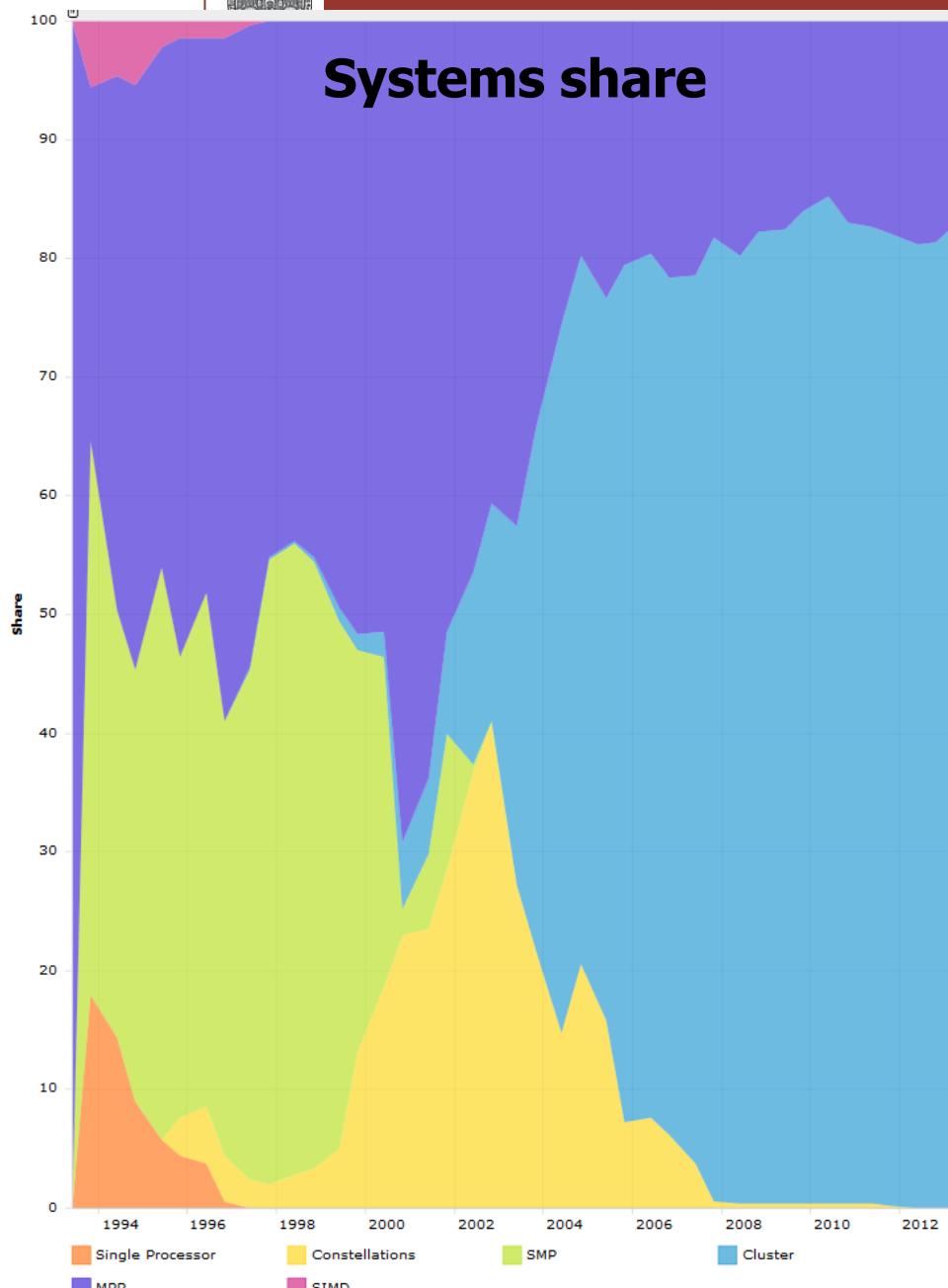
Schichten eines Rechensystems



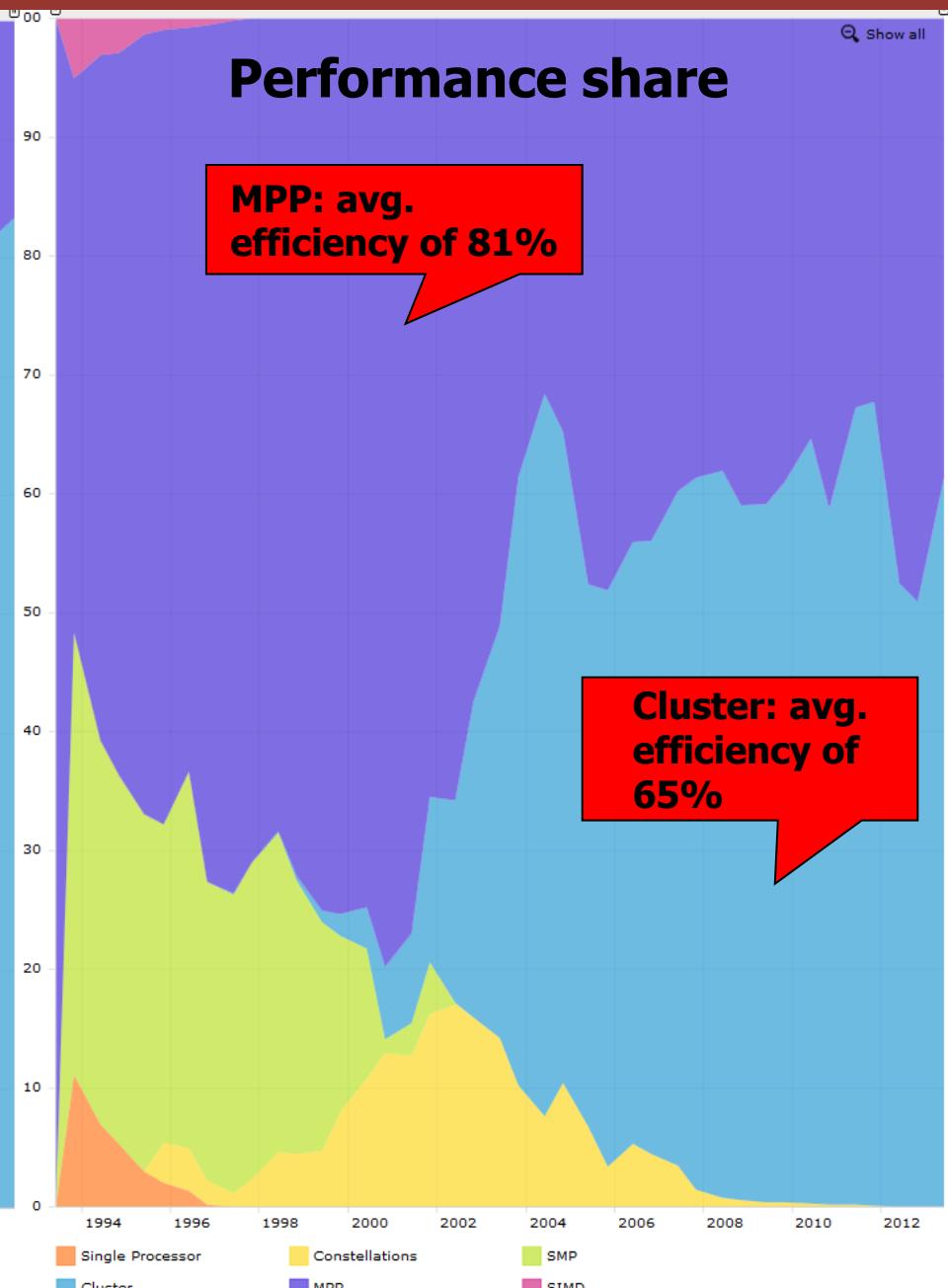


TOP500

Systems share

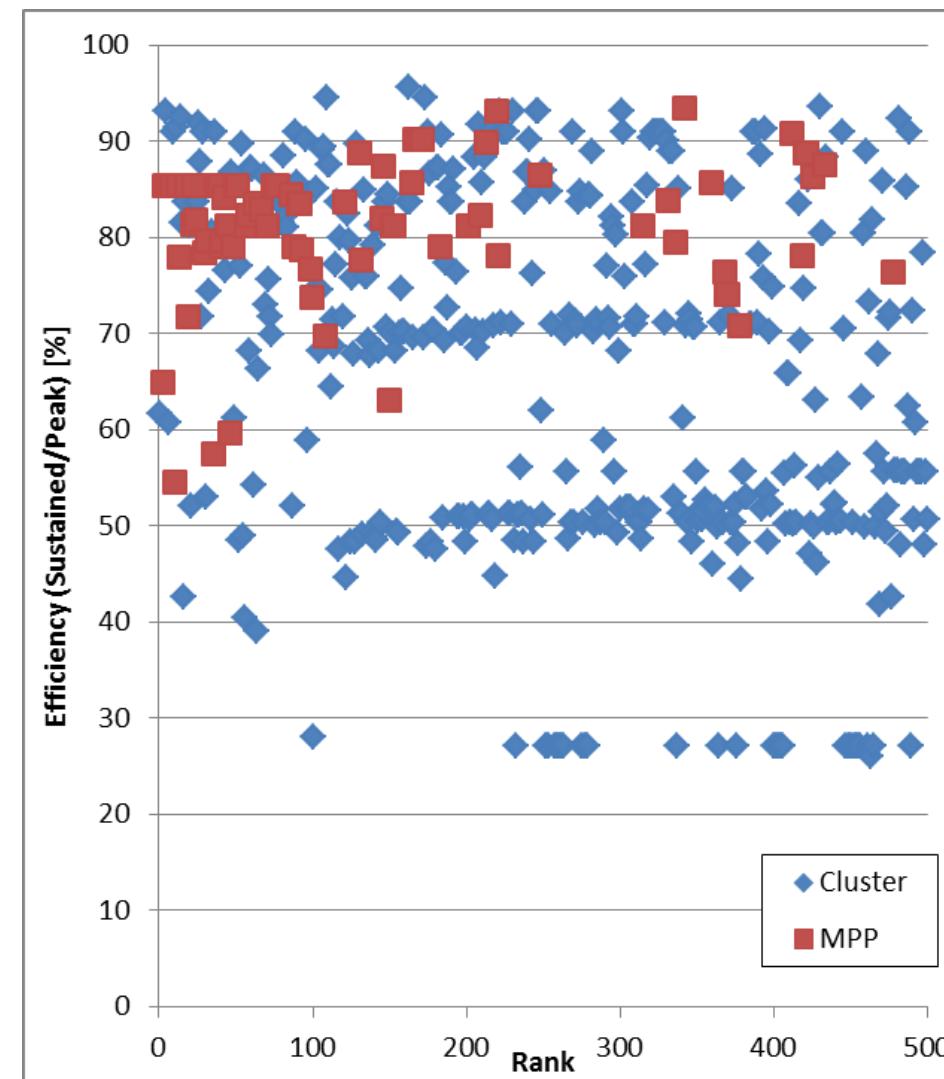


Performance share





TOP500: Efficiency (Sustained/Peak)



- **TOP500 (06/2013)**
 - Efficiency: 26%/69%/96%
(min/avg/max)
 - Rmax/core: 2.8/12/95
GFLOPs (min/avg/max)
- **~60 accelerated systems**
 - GPUs/MICs
 - Efficiency: 27%/57%/77%
(min/avg/max)
 - 7 accelerated systems back
in 2008



Power Consumption

- Constraints of Energy/Power Consumption
 - Economical: Costs
 - Technological: supply, distribution, cooling
 - Ecological

→ „Heat Wall“

„You can put more cores on a die than you can afford to turn on“

Cray 3 from 1993: 90kW

- Supercomputer today: MWatts
- Google's Power Budget
- NSA Number Cruncher





Leistungsaufnahme - 2008

TOP500 List - November 2008 (1-100)

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the [TOP500 description](#).

Power data in KW for entire system

[next](#)

| Rank | Site | Computer/Year Vendor | Cores | R _{max} | R _{peak} | Power |
|------|---|--|--------|------------------|-------------------|---------|
| 1 | DOE/NNSA/LANL United States | Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2008 IBM | 129600 | 1105.00 | 1456.70 | 2483.47 |
| 2 | Oak Ridge National Laboratory United States | Jaguar - Cray XT5 QC 2.3 GHz / 2008 Cray Inc. | 150152 | 1059.00 | 1381.40 | 6950.60 |
| 3 | NASA/Ames Research Center/NAS United States | Pleiades - SGI Altix ICE 8200EX, Xeon QC 3.0/2.66 GHz / 2008 SGI | 51200 | 487.01 | 608.83 | 2090.00 |
| 4 | DOE/NNSA/LLNL United States | BlueGene/L - eServer Blue Gene Solution / 2007 IBM | 212992 | 478.20 | 596.38 | 2329.60 |
| 5 | Argonne National Laboratory United States | Blue Gene/P Solution / 2007 IBM | 163840 | 450.30 | 557.06 | 1260.00 |
| 6 | Texas Advanced Computing Center/Univ. of Texas United States | Ranger - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband / 2008 Sun Microsystems | 62976 | 433.20 | 579.38 | 2000.00 |
| 7 | NERSC/LBNL United States | Franklin - Cray XT4 QuadCore 2.3 GHz / 2008 Cray Inc. | 38642 | 266.30 | 355.51 | 1150.00 |
| 8 | Oak Ridge National Laboratory United States | Jaguar - Cray XT4 QuadCore 2.1 GHz / 2008 Cray Inc. | 30976 | 205.00 | 260.20 | 1580.71 |
| 9 | NNSA/Sandia National Laboratories United States | Red Storm - Sandia/ Cray Red Storm, XT3/4, 2.4/2.2 GHz dual/quad core / 2008 Cray Inc. | 38208 | 204.20 | 284.00 | 2506.00 |
| 10 | Shanghai Supercomputer Center China | Dawning 5000A - Dawning 5000A, QC Opteron 1.9 Ghz, Infiniband, Windows HPC 2008 / 2008 Dawning | 30720 | 180.60 | 233.47 | |

Source: [top500.org](#)



Leistungsaufnahme - 2013

| Rank | Site | System | Cores | Rmax (TFlop/s) | Rpeak (TFlop/s) | Power (kW) |
|------|--|---|-----------|-------------------|--------------------|---------------|
| 1 | National University of Defense Technology China | Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT | 3,120,000 | 33,862.7 | 54,902.4 | 17,808 |
| 2 | DOE/SC/Oak Ridge National Laboratory United States | Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc. | 560,640 | 17,590.0 | 27,112.5 | 8,209 |
| 3 | DOE/NNSA/LLNL United States | Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM | 1,572,864 | 17,173.2 | 20,132.7 | 7,890 |
| 4 | RIKEN Advanced Institute for Computational Science (AICS) Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu | 705,024 | 10,510.0 | 11,280.4 | 12,660 |
| 5 | DOE/SC/Argonne National Laboratory United States | Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM | 786,432 | 8,586.6 | 10,066.3 | 3,945 |
| 6 | Texas Advanced Computing Center/Univ. of Texas United States | Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell | 462,462 | 5,168.1 | 8,520.1 | 4,510 |
| 7 | Forschungszentrum Juelich (FZJ) Germany | JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM | 458,752 | 5,008.9 | 5,872.0 | 2,301 |
| 8 | DOE/NNSA/LLNL United States | Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM | 393,216 | 4,293.3 | 5,033.2 | 1,972 |
| 9 | Leibniz Rechenzentrum Germany | SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM | 147,456 | 2,897.0 | 3,185.1 | 3,423 |
| 10 | National Supercomputing Center in Tianjin China | Tianhe-1A - NUDT YH MPP, Xeon X5670 6C 2.93 GHz, NVIDIA 2050 NUDT | 186,368 | 2,566.0 | 4,701.0 | 4,040 |

Source: top500.org



Datacenter vs. Cloud vs. HPC

| Characteristic | Cloud/Datacenter | HPC |
|-------------------|---|---|
| Parallelism | Massively parallel | Massively parallel |
| Nodes | Commodity x86 | Commodity x86 |
| Local storage | Huge (~1-2TB) | Medium (<1TB) |
| Network | Ethernet | Typically not Ethernet |
| <i>Coupling</i> | Loose | <i>Tight</i> |
| Technical details | 2x Xeon-SMP 4 cores per socket at 2.33GHz 7GB memory 1690GB storage Network: „High I/O performance“ | 2x Xeon-SMP 4 cores per socket at 2.33GHz 8GB memory 73GB storage Network: Infiniband |

Walker, 2008, benchmarking Amazon EC2 for high-performance scientific computing

;login: The USENIX Magazine, 33(5)

<http://www.usenix.org/publications/login/2008-10/openpdfs/walker.pdf>



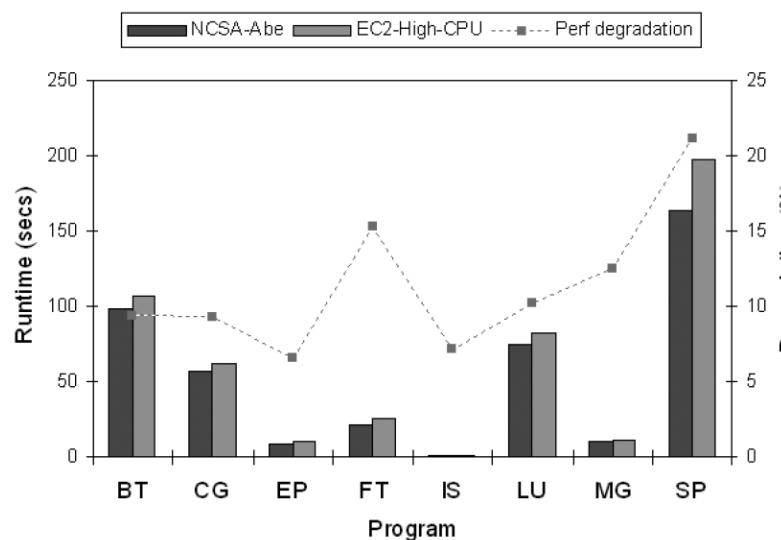
Cloud vs. HPC

■ Cloud (Amazon EC2) vs. Cluster (Infiniband, NCSA-Abe)

- NAS Parallel Benchmarks (NPB)

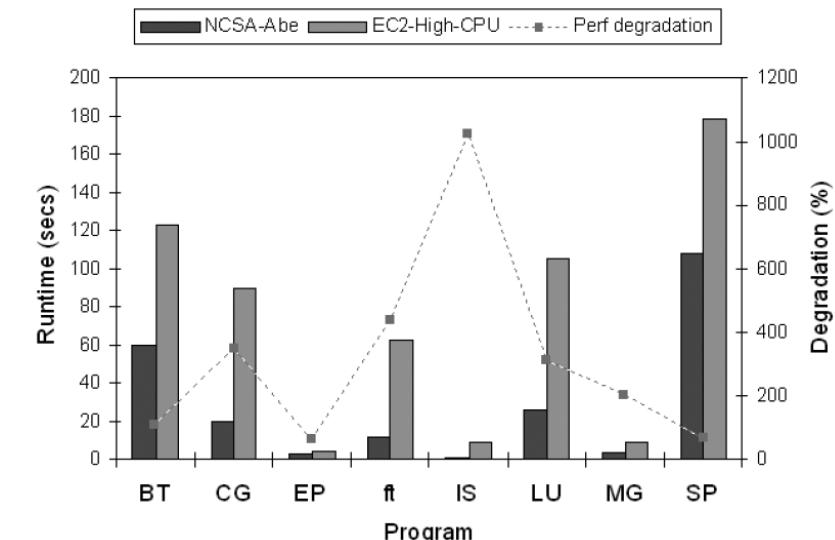
■ Node-level execution

- OpenMP
- 8 cores, 1 node



■ Cluster-level execution

- MPI
- 32 cores (resp. 16)





Summary

- Demand for computing power still not satisfied and won't be satisfied in the near future
 - During the design phase almost every product gets in touch with HPC
 - Scientific questions
 - Technical problems
- Primary tool: high parallelization
 - Interconnection network moves into the spotlight
- Power consumption is a pervasive problem, not only for huge installations
- Browse <http://top500.org> for plenty of information about facts and trends of HPC



Questions?

Next lecture:

Introduction to Message Passing



Introduction to High Performance Computing

Lecture 02 – Introduction to Message Passing

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg



Concepts of Parallel Architectures



Concepts of Parallel Architectures

■ Parallel Architectures

- Are pervasive: smartphones, netbooks, notebooks, ...
- Most important resources for HPC: computing units, memory resources, interconnection network
- Most common architecture found today: multi-core systems

■ Multi-core system consists of:

- Multiple computing/processing units, either:
 - Within one processor die (multi-core)
 - Within one computing system (SMP, multiple sockets)
- One single or multiple memory controllers

■ There are any more architecture types:

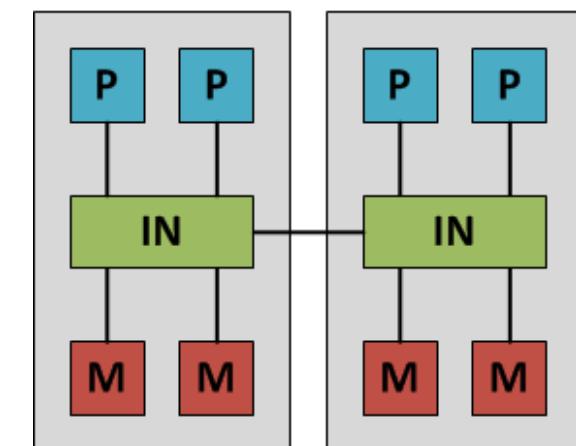
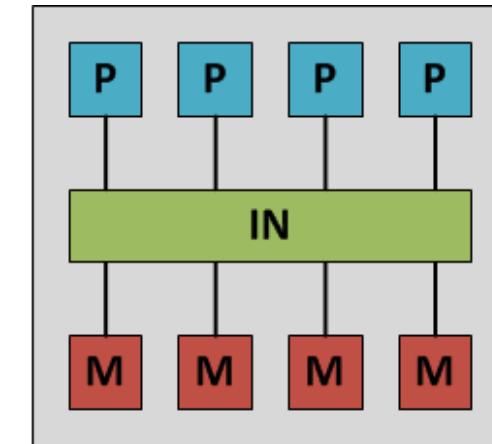
- Most important: **Shared-Memory** and **Distributed-Memory**



Shared Memory

■ Shared address space

- Each computing unit can access any memory address
- Physical memory may be distributed, access times may vary
 - UMA (uniform memory access)
 - NUMA (non-uniform memory access)
- Any cache can hold copies from any location
 - Cache coherent NUMA (ccNUMA)
 - Usually limited scalability



An address space defines a range of discrete addresses; each address may correspond to a different resource



Shared Memory

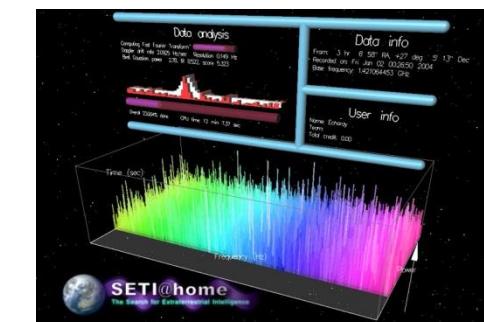
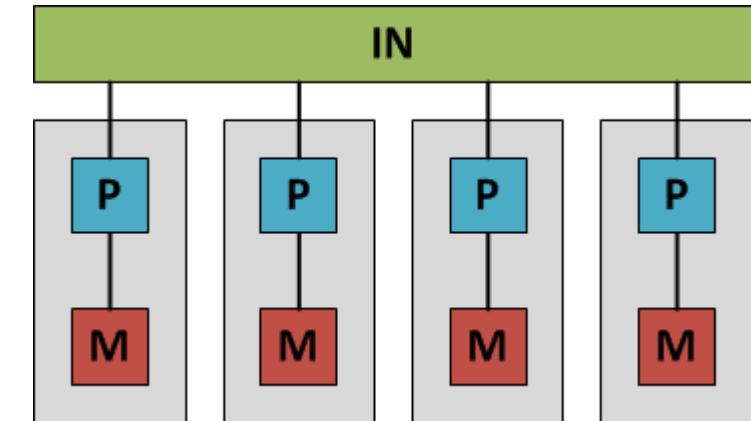
- Shared variables
 - Operations, naming, ordering
- Communication purposes
 - Each unit can read/write any address
 - Used for data
 - Hazards and race conditions require synchronization
- Synchronization purposes
 - Resolve hazards and race conditions
 - Semaphores, condition variables, mutexes, locks
- Implicit communication, explicit synchronization



Distributed Memory

■ Exclusive address spaces

- Operations, naming, ordering
- No (direct) remote access possible (NORMA)
- Physical memory is distributed
- Usually shared memory systems as building blocks
- No global cache coherency required
 - Unlimited scalability (from an architectural point of view)
- Popular example:
SETI@Home





Distributed Memory

- **Messages for communication purposes**
 - Data transfer (copies)
 - Read example: send a request, receive an answer
 - Write example: send a request, optional answer
- **Messages for synchronization purposes**
 - On the origin node, one process is sending the message
 - On the target node, one process is receiving the message
 - Both will be informed when the other has processed the message
- **Error-prone programming**
- **Programmer is responsible for:**
 - Data distribution
 - Explicit communication
 - Explicit synchronization
 - Many optimizations (huge set of messaging functions, ...)



Shared vs. Distributed Memory

- Key distinction: number of address spaces
 - Is (direct) access to the complete memory possible?

| | Shared Memory | Distributed Memory |
|--------------------------|---------------------------|--------------------|
| Number of address spaces | 1 | N |
| Communication | Implicit | Explicit |
| Synchronization | Explicit | Explicit |
| Number of data copies | Typ. 1 | Typ. [1..N] |
| Scalability | Limited (cache coherence) | Unlimited/High |

Synchronization is the most common error source!

Main reason for HPC



Example

- A human discussion as joint work example:
 - Let's assume they use a sheet of paper for the discussion
- If everybody is in the same room, one single sheet of paper can be used
 - Shared resource, but synchronization required
 - Use of pencil and eraser simultaneously, multiple instances...
- If the persons are physically distributed without access to the „local“ sheet, this sheet has to be sent around
 - Messaging
 - Only one copy – weak performance, little synchronization overhead
 - Multiple copies – improved perf., lot's of synchronization
- For both approaches, partitioning is an essential optimization



- HPC favors distributed memory
 - Scalability
- Key tool for HPC today is message passing
- Much more than send and receive
 - Plenty of send variants, plenty of receive variants
 - Collective communication
 - Even communication without saying *receive* (remote memory access, RMA)
 - More in later lectures...



Basics of Message Passing



Parallel Programming

- Nobody wants to write N programs for N processes
- Single Program Multiple Data (SPMD) paradigm
 - Write one program, which is internally partitioned to act differently
 - Master/Slave(s), Consumer(s)/Producer(s), Peer model
 - Before execution, each instance is assigned a unique number
 - Let's call this „*rank*“
 - Let's call the total number of ranks „*size*“
 - During execution, the rank determines program behavior
 - Functionality
 - Associated input – better output – data
- SPMD widely used for shared-memory and distributed-memory programming



SPMD Examples

■ Master/Slave

```
if ( rank == 0 ) {  
    act_as_master();  
} else {  
    act_as_slave();  
}
```

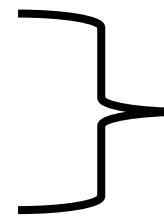
■ Peer model

```
int work[n];  
for ( i = rank/size * n;  
      i < (rank+1)/size * n;  
      i ++ ) {  
    process ( work[i] );  
}
```

Note: $(n \% \text{size} == 0)$ is not mandatory



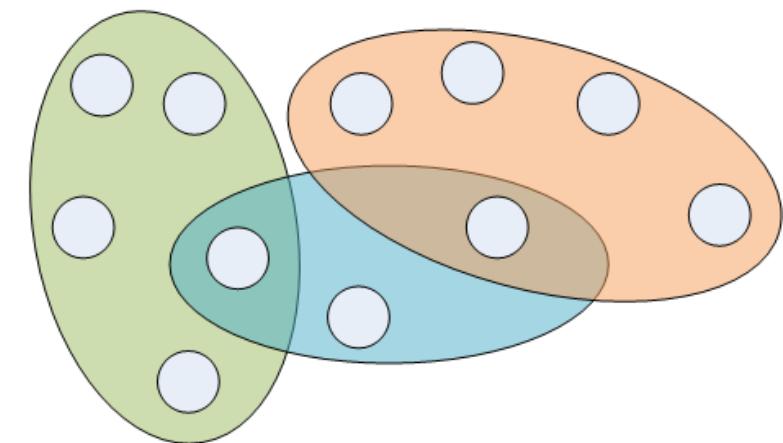
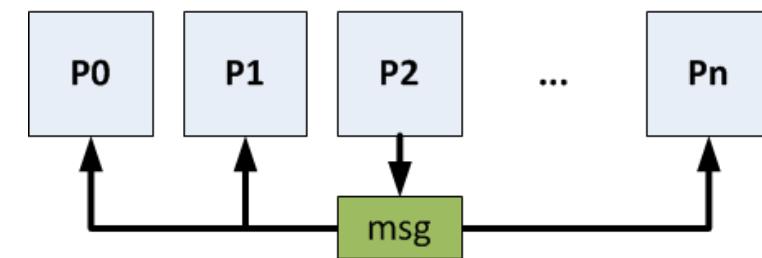
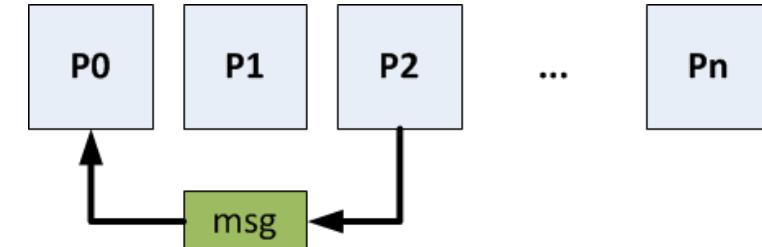
SPMD and Distributed Memory

- Distributed Memory programs run locally with local data
 - Everything explicit
 - Communication
 - Synchronization
 - Data distribution
 - Complicated programming
 - Not compatible with shared-memory programming
 - Code porting
 - Development of new algorithms
 - Some help
 - Messaging libraries
- 
- A large black brace is positioned to the right of the 'Everything explicit' section, spanning vertically from the bullet points to the text 'done using messages'.
- done using messages**



Messaging Overview

- Point-to-point operations
 - Data movement between two peers
- Collective operations
 - Collective data movement (among peers or master/slaves)
 - Barriers (among peers)
- Communicators
 - Define process groups
 - Ranks / size of a communicator
 - Every message is assigned to one communicator
 - Collective operations refer to all ranks within one communicator





Message Details

- A message consists of the following
 - Source
 - Destination
 - Tag
 - Size
 - Data (optional)
 - Type
- Send operation
 - Provide all these information
- Receive operation
 - 1. Receive any message
 - 2. Receive any message from a **given source**
 - 3. Receive any message with a **given tag**
 - 4. Receive any message from a **given source** with a **given tag**



Introduction to MPI



Message Passing Interface (MPI)

- De-facto standard for message passing in HPC
 - MPI 1.0 in 1994, to MPI 2.2 in 2009
- Library for C, Fortran, C++
- <http://mpi-forum.org>

- Plenty of functions – here only a couple
 - Mandatory are about 10-20
- One predefined communicator
 - MPI_COMM_WORLD (containing all processes)



Basic functions

- Always include MPI header file

```
#include <mpi.h>
```

- Initialize library

- Call prior to any other MPI function

```
MPI_Init (&argc, &argv) ;
```

- Finalize library

- Call after all MPI calls are done

```
MPI_Finalize () ;
```

- Get rank (unique ID) and size (number of processes) of current process for a given communicator (here default)

```
MPI_Comm_rank (MPI_COMM_WORLD, &rank) ;
```

```
MPI_Comm_size (MPI_COMM_WORLD, &size) ;
```



„Hello parallel world!“

```
// A very short MPI test routine to show the involved hosts
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int size,rank;
    char hostname[50];
    MPI_Init ( &argc, &argv );

    MPI_Comm_rank ( MPI_COMM_WORLD, &rank); // Who am I?
    MPI_Comm_size ( MPI_COMM_WORLD, &size ); // How many processes?

    gethostname (hostname, 50 );
    printf ("Hello Parallel World! I'm process %2d out of %2d (%s)\n",
           rank, size, hostname );

    MPI_Finalize ();
    return 0;
}
```



„Hello parallel world!“

■ Compile:

```
mpicc -Wall mpi_hello.c -o mpi_hello
```

- Any additional compilation parameters (-O3, -L, -I, -h, ...)
- See also mpic++, mpicxx

■ Prepare execution (only once per user):

- MPI is usually based on ssh-connections. Ensure that no password is required for remote login:

```
ssh-keygen -t dsa -b 1024
```

```
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```



„Hello parallel world!“

- Execute, simplest flavor, 2 processes on one node:

```
mpirun -host creek01[,creek01] -np 2 ./mpi_hello
```

- Execute, 16 processes on two nodes:

```
mpirun -host creek01,creek02 -np 16 ./mpi_hello
```

- Parameters:

- **-np <n>**: start n processes
- **-host**: list of hostnames, used cyclic if more processes than entries in this list
- **-mca ...**: parameters to pass to the MPI library, here use TCP for communication (-mca btl tcp,self)



„Hello parallel world!“

Output:

```
# mpirun -host creek01,creek02 -np 4 ./mpi_hello
Hello Parallel World! I'm process 0 out of 4 (creek01)
Hello Parallel World! I'm process 2 out of 4 (creek01)
Hello Parallel World! I'm process 1 out of 4 (creek02)
Hello Parallel World! I'm process 3 out of 4 (creek02)
```

```
# mpirun -host creek01,creek01,creek02,creek02 -np 4
./mpi_hello | sort
Hello Parallel World! I'm process 0 out of 4 (creek01)
Hello Parallel World! I'm process 1 out of 4 (creek01)
Hello Parallel World! I'm process 2 out of 4 (creek02)
Hello Parallel World! I'm process 3 out of 4 (creek02)
```



A first message...

- (Blocking) send of a message

```
int MPI_Send(void *buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm)
```

- Example:

```
signal = 42;  
MPI_Send (&signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
```

- A blocking send returns if the provided buffer can be used again.
- Plenty of MPI_Datatypes



A first message...

- (Blocking) receive of a message

```
int MPI_Recv(void *buf, int count, MPI_Datatype  
datatype, int source, int tag, MPI_Comm comm,  
MPI_Status *status)
```

- Example:

```
MPI_Recv ( &signal, 1, MPI_INT, i, 0,  
          MPI_COMM_WORLD, &status );
```

- A blocking receive returns if a message with the required properties has been received (source, tag)

- MPI_ANY_SOURCE
- MPI_ANY_TAG

- Status contains more information about the receive
 - MPI_STATUS_IGNORE



Communication example

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    int i, size, rank, signal;
    MPI_Status status;
    double starttime, endtime;

    MPI_Init (&argc, &argv);      //Initialisation of MPI
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    if (rank == 0) { //Master: send to every proc and wait for answer
        <see 2nd slide>
    } else { //Clients: wait for message from master and send back
        <see 3rd slide>
    }
    MPI_Finalize(); //Deinitialisation of MPI
    return 0;
}
```



Communication example

```
if (rank == 0) {  
  
    signal = 666;  
    starttime=MPI_Wtime();  
    for (i=1; i<size; i++) {  
        MPI_Send(&signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);  
        printf ("%d: Sent to %d\n", rank, i);  
        MPI_Recv (&signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD,  
                  &status);  
        printf ("%d: Received from %d\n", rank, i);  
    }  
    endtime=MPI_Wtime();  
    printf ("%d: Time passed: %f\n", rank, endtime-starttime);  
}
```

double MPI_Wtime()

Returns time in seconds since an arbitrary time in the past



Communication example

```
//Clients: wait for message from master and send back
else {
    MPI_Recv (&signal, 1, MPI_INT, MPI_ANY_SOURCE, 0,
              MPI_COMM_WORLD, &status);
    printf ("%d: Received from %d\n", rank, status.MPI_SOURCE);
    MPI_Send(&signal, 1, MPI_INT, status.MPI_SOURCE, 0,
              MPI_COMM_WORLD);
    printf ("%d: Sent to %d\n", rank, status.MPI_SOURCE);
}
```



Blocking vs. Non-blocking

■ Blocking

- Returns after send or receive complete
- What means complete?

■ Non-blocking variants

- Guarantees nothing, may return immediately
- Used for improved overlap between computation and communication

```
int MPI_Isend(void *buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm,  
MPI_Request *request)
```

```
int MPI_Irecv(void *buf, int count, MPI_Datatype  
datatype, int source, int tag, MPI_Comm comm,  
MPI_Request *request)
```



Non-blocking: test/wait for completion

■ Blocking: Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- Guarantees that request has been processed

■ Non-Blocking: Test

```
int MPI_Test(MPI_Request *request, int *flag,  
MPI_Status *status)
```

- Updates the request, but does not ensure completion



Questions?

Next lecture:

Basics



Introduction to High Performance Computing

*Lecture 03 – Applications, Performance Increase,
Top500*

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg

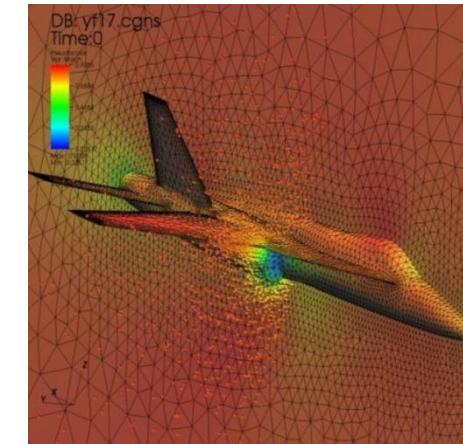


Example Applications Fields

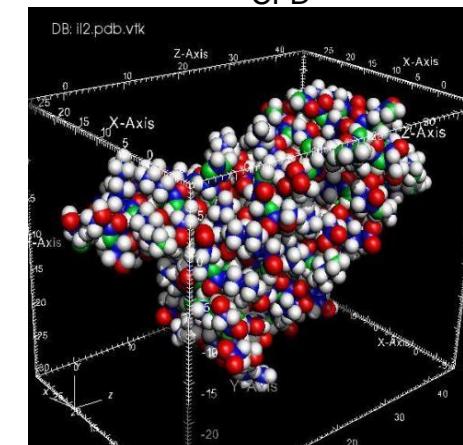


Example application fields

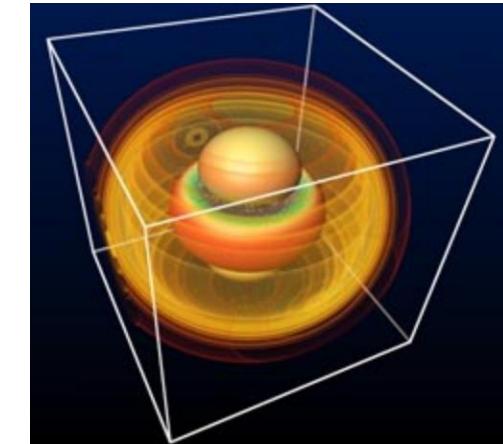
- **Oil and gas**
 - Seismic processing
- **Financial services**
 - Automated option pricing and trading
- **Bioscience**
 - Genetic sequencing and chemistry
- **Government**
 - Searching and encryption engines
- **Digital content creation**
 - Movie animation
- **Scientific research**
 - Astrophysics, particle physics
 - Biology: molecular dynamics
- **Industry**
 - Fluid dynamics
- **Meterology & Climatology**
- **Electronic design automation**
 - Chip design
- **Finite element methods**
 - Crash simulations
- ...



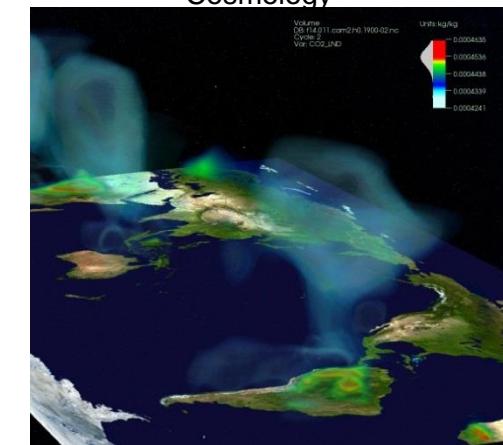
CFD



Molecular Dynamics (MD)



Cosmology



Weather/Climate Research



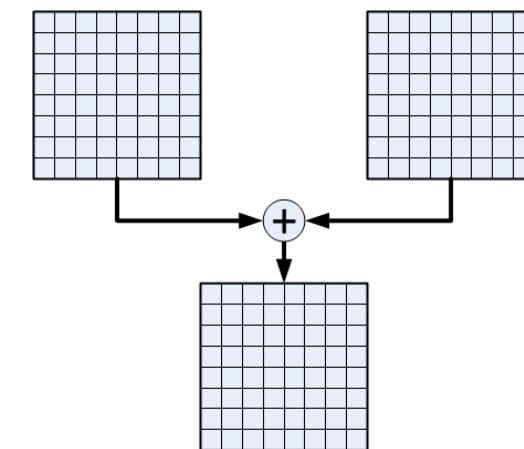
A more abstract view...

■ Basic operation types in HPC

- Complex processing of regular data structures
 - Vectors
 - Arrays
 - Elements
- High degree of parallelism

■ Trivial example: Matrix addition

- “Domain decomposition”

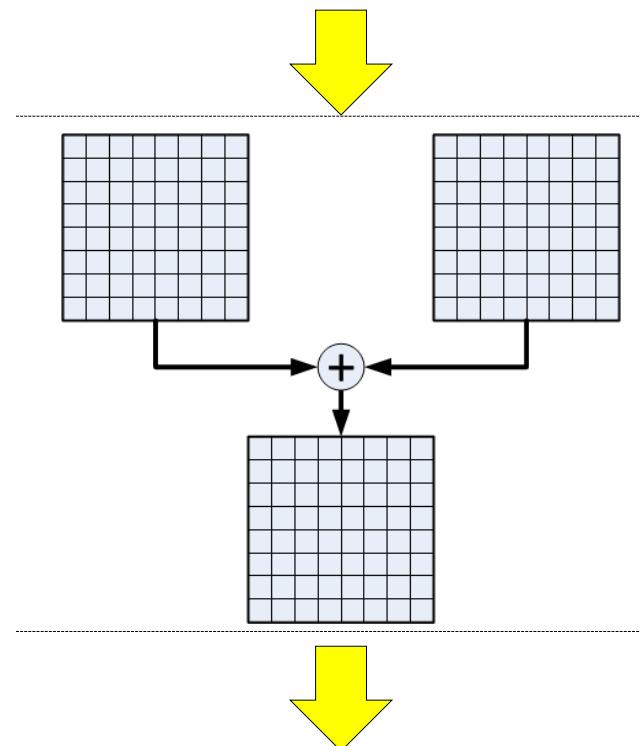




Simple parallelization example

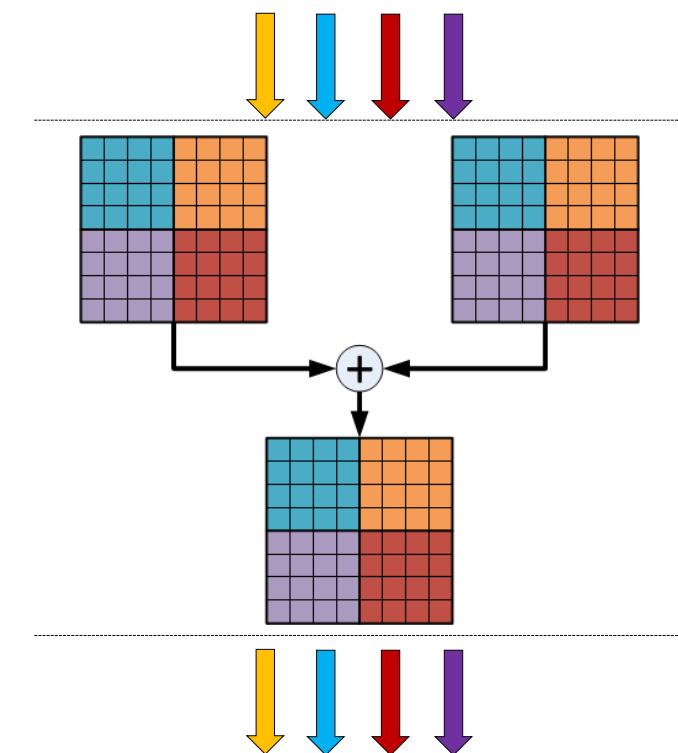
■ Serial execution

- No communication
- Sequential processing of elements



■ Parallel execution

- Communication?
- Synchronization?
- Parallel processing of (blocks of) elements





Molecular dynamics

■ Motivation

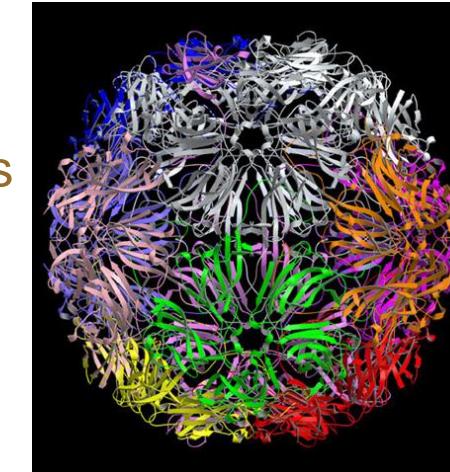
- Protein Folding
- Digital simulation instead of biochemics
- Computationally intensive
- Runtime of several months
- STMV: 160 genes, 100ns/day for Petascale-class

■ Goal: Calculation of the molecule's shape

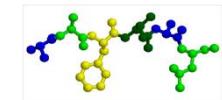
- Double precision floating point
- Calculation of forces in between the atoms of the molecule and the surrounding
 - Forces: Electrostatic (Coulomb) & Van der Waal

• Time step = 1 femto (10^{-15}) second

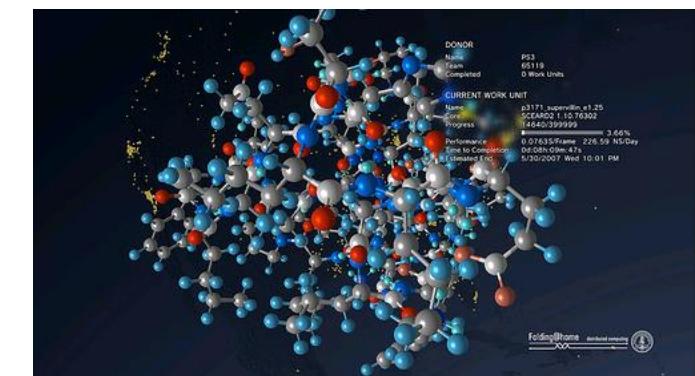
■ “N-body problem”



nasa.gov



wikipedia.org



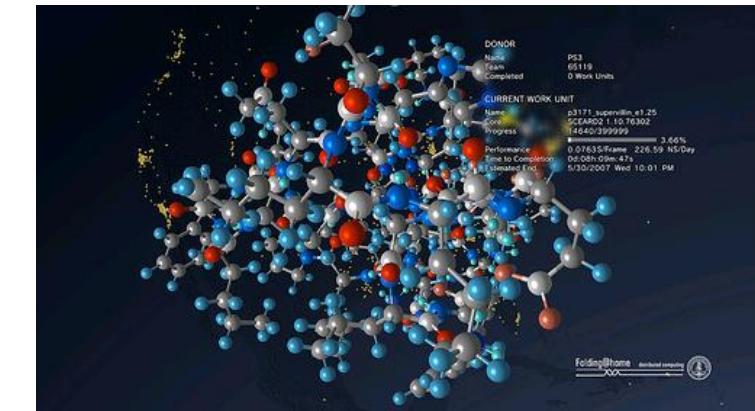


Molecular dynamics: N-body

■ N-body problem

- For each atom in a 3D system

```
Do repeat
    Increase time step t
    Foreach atom i
        Foreach atom j (j != i)
            Compute force(s) from j to i
            Sum all forces on i
        Next j
        Compute next position of i
    Next i
Repeat until stable
```



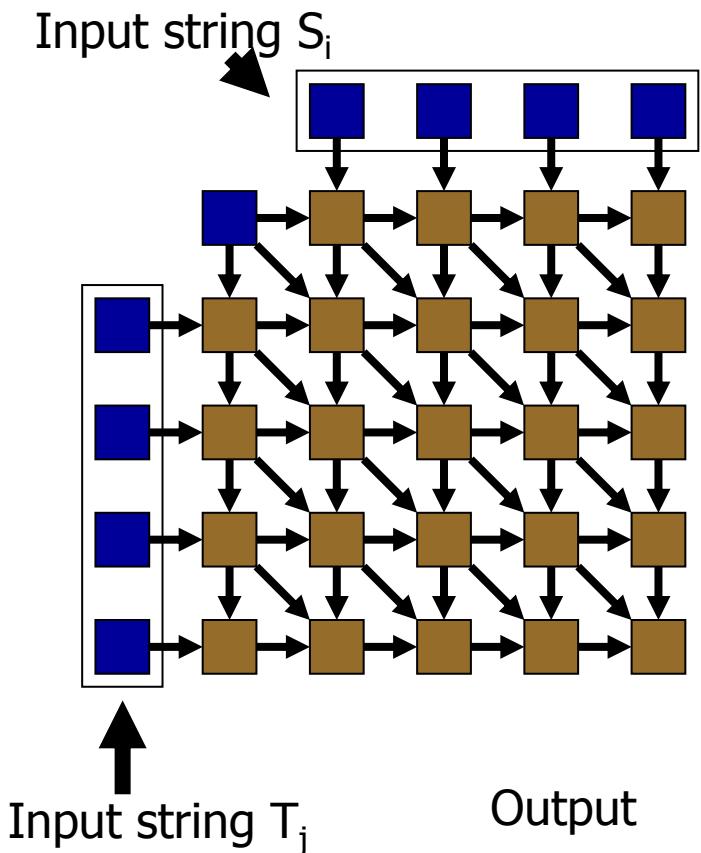
forces $\sim 1/d^2$
 $\sim 82\%$ time
 ~ 52 FP ops

- No special treatment of borders here...
 - Approx. 60 variants



- Genetics related research
- String distance computation
- Using Smith-Waterman
 - Exact string matching algorithm
 - Finds optimal local alignment
 - Computes a matching score $H(i,j)$ of two input strings S and T using a 2D matrix
- Other applications:
 - Motif discovery, data mining

$$H(i,j) = \max \left\{ \begin{array}{ll} 0 & \\ H(i-1, j-1) + w(a_i, b_j) & \text{Match/Mismatch} \\ H(i-1, j) + w(a_i, -) & \text{Deletion} \\ H(i, j-1) + w(-, b_j) & \text{Insertion} \end{array} \right\}, \quad 1 \leq i \leq m, 1 \leq j \leq n.$$





Computational Fluid Dynamics (CFD)

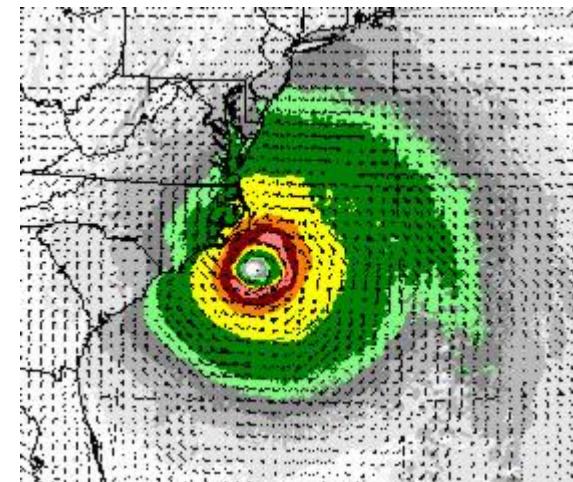
- Numerical fluid simulations
- Lattice-Boltzmann method (LBM)
 - Simulation at particle scope
 - Discretization by grid, at microscopic level solution of mutual reaction as described by the Boltzmann equation
- Boltzmann equation in the case of large average free path lengths
 - Currents in (depleted) gases
 - Neutron distribution in atomic reactors
- Otherwise: E.g. Navier-Stokes equation
 - Much simpler
 - Current in liquids
- Rather few communication, large messages between pairs of two



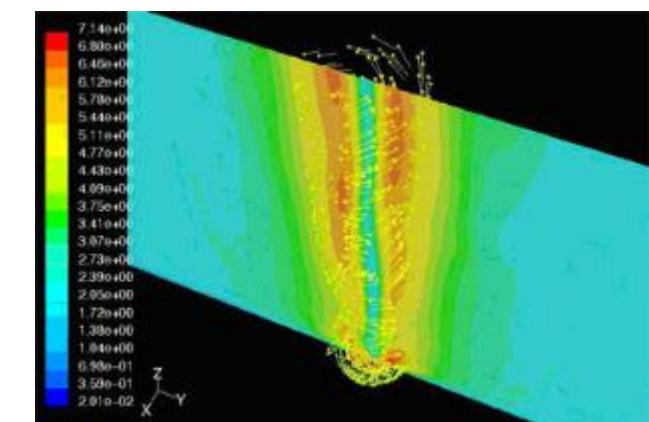
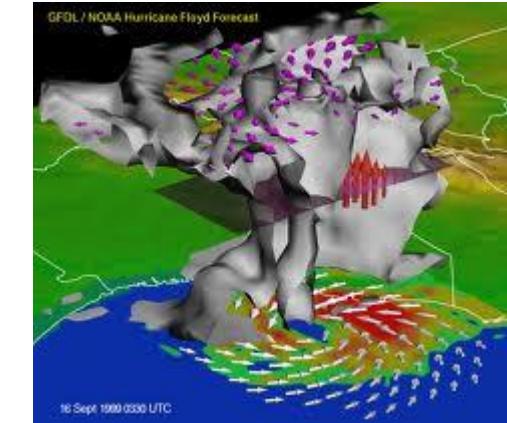
Weather and Climate Research

- See CFD
- Examples

- Hurricane prediction
- Tornado simulation
- Local weather forecast
- Global Climate Modeling



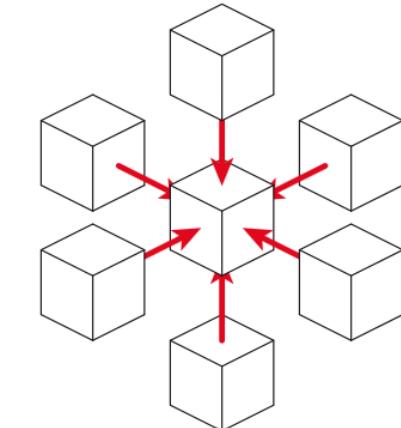
ucar.edu



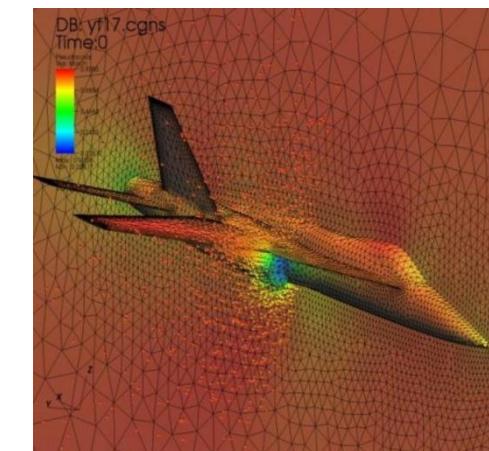


Implementations

- Solving/Approximation of Partial Differential Equations (PDE) or CFD
- Stencil codes
 - Iterative kernels
 - Regular, invariable structure
- Finite Element Methods (FEM)
 - Irregular structures
 - For complex or variant structures
 - Different accuracies
 - Adaptive Mesh Refinement (AMR)
- Technical and scientific computing is mostly modeling
 - Based on time steps
 - Iterative solution until results are stable or simulation period is over



6-point 3D stencil, courtesy: wikipedia.org





Basics of Performance Increase

*Performance Increase of
Technologies and Applications*

Moore vs. Amdahl



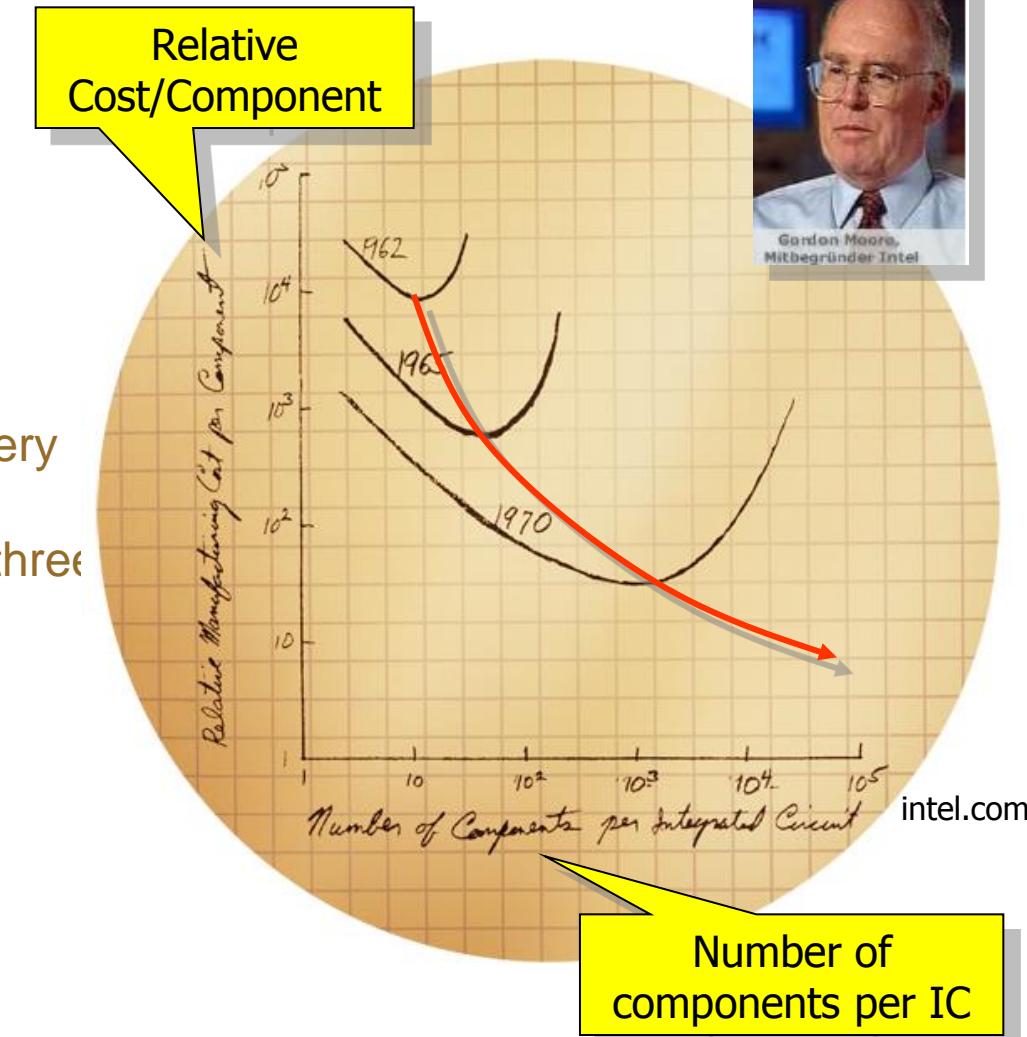
Moore's Law

■ Gordon Moore

- 1965: Doubling each year
- 1975: Transistor count of ICs doubling every two years

■ Derived “laws”

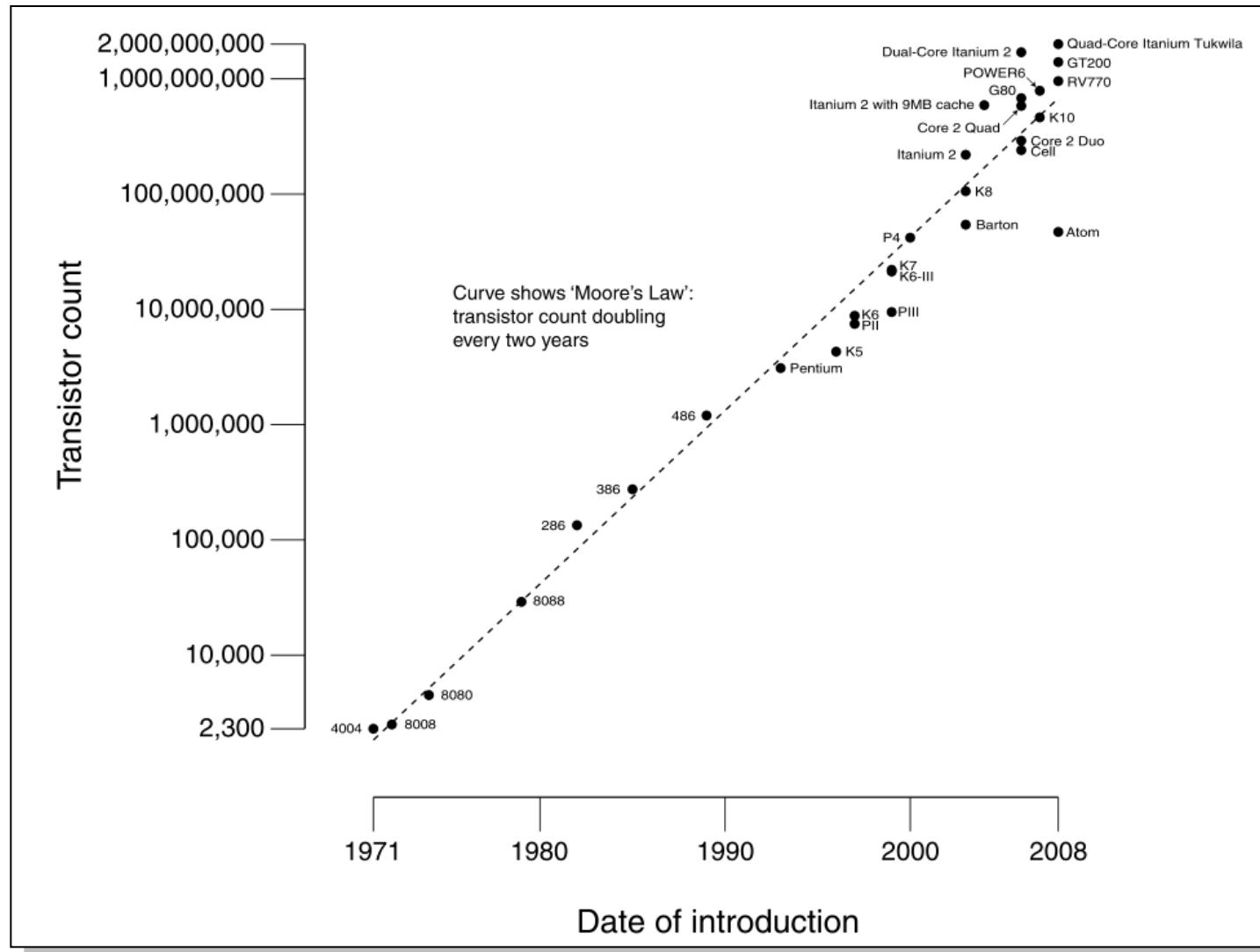
- CPU performance doubling every 18 months
- Memory size four times every three years
- Memory performance doubling every 10 years
- At the same costs double performance every two years





Moore's Law

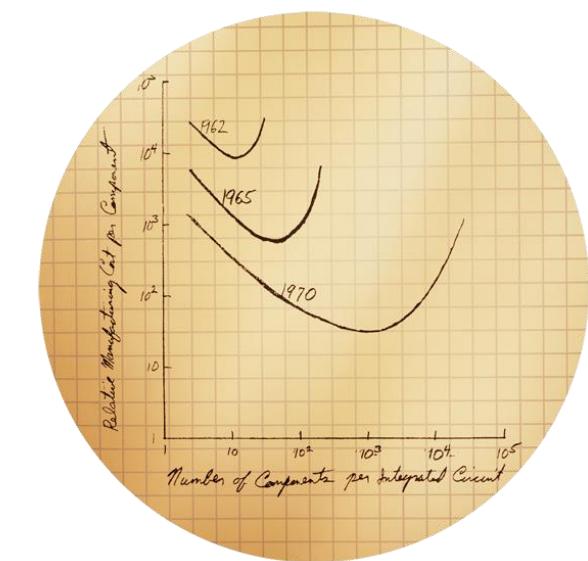
wikipedia.org





Moore's Law

- Industry is trying to keep the pace
 - Self-fulfilling prophecy
 - “positive feedback between belief and behavior”
- Atoms as fundamental lower bound
 - Even then, increase of die size can maintain the law
 - Intel’s statements about end of Moore’s law
 - 2003: 2013-2018
 - 2005: until 2015
 - 2008: until 2029
- Bernie Meyerson (IBM):
7-9nm is the limit
 - Quantum mechanics effects





Speed-up

- Speed-up: „How much faster can one program be executed“
- Assumption: instead of one resource, N identical resources are available
- Naive: More resources, faster execution
- A bit more realistic: N resources yield an execution time of $1/N$
 - No overhead assumed
- Reality: significant loss
 - Break-even point when execution time starts to increase again



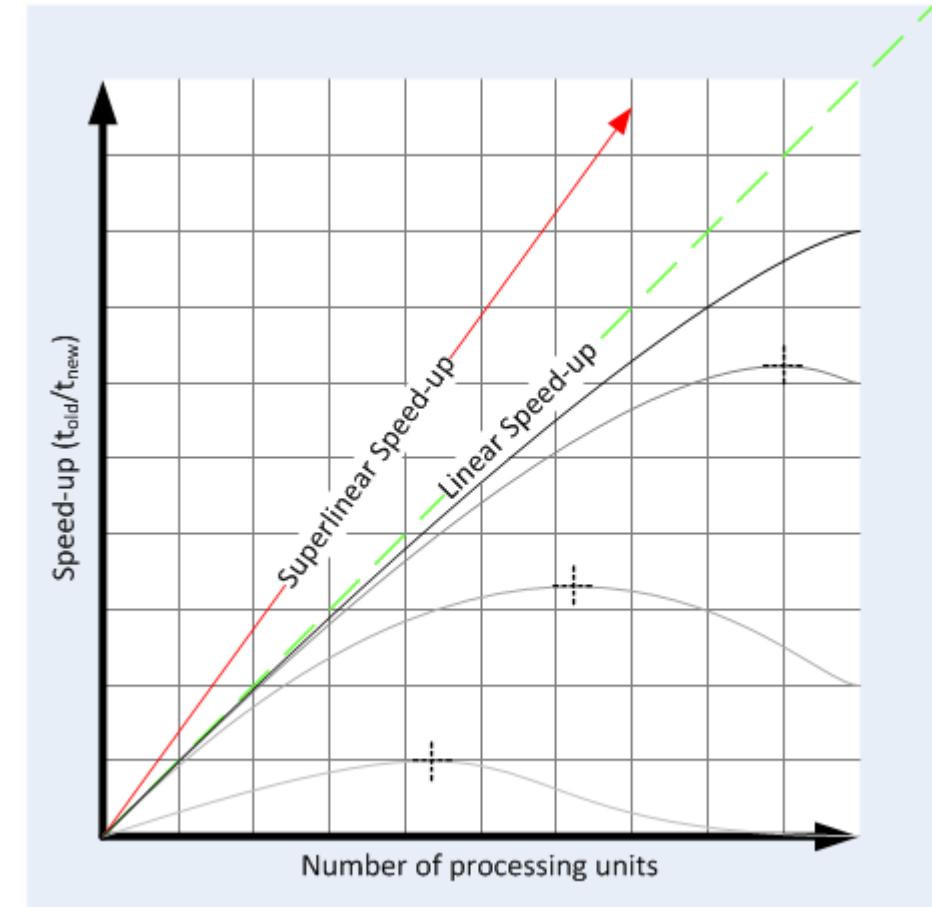
Speed-up - Definitions

- For a given algorithm:
 - $\text{SerTime}(n)$ = time of the best serial implementation for an input of size n
 - $\text{ParTime}(n,p)$ = time of the parallel implementation, using p parallel computing units
- Sanity check: $\text{SerTime}(n) \geq \text{ParTime}(n,1)$
 - The other case is not uncommon
- Speed-up:
 - $\text{Speedup}(p)$ = $\text{SerTime}(n) / \text{ParTime}(n,p)$
 - $\text{Efficiency}(p)$ = $\text{SerTime}(n) / (p * \text{ParTime}(n,p))$



Speed-up - Notes

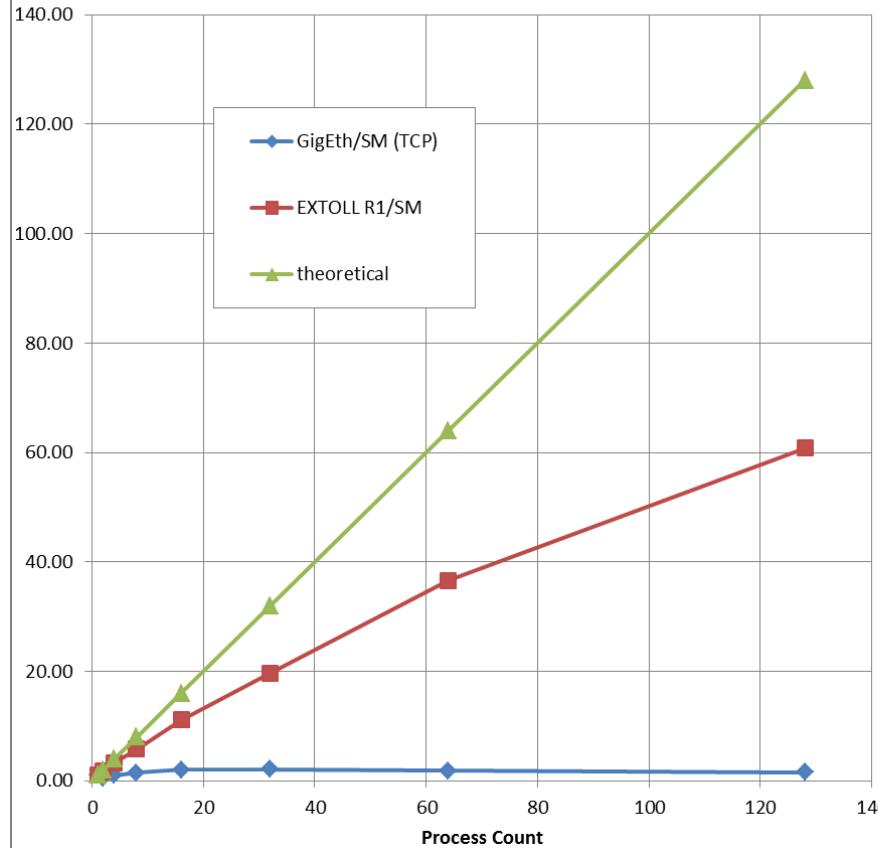
- $1 \leq \text{Speedup}(p) \leq p$
- $0 \leq \text{Efficiency}(p) \leq 1$
- Linear speed-up:
 $\text{Speedup}(p) = p$
- Superlinear speed-up:
 $\text{Speedup}(p) > p$
 - Usually not possible



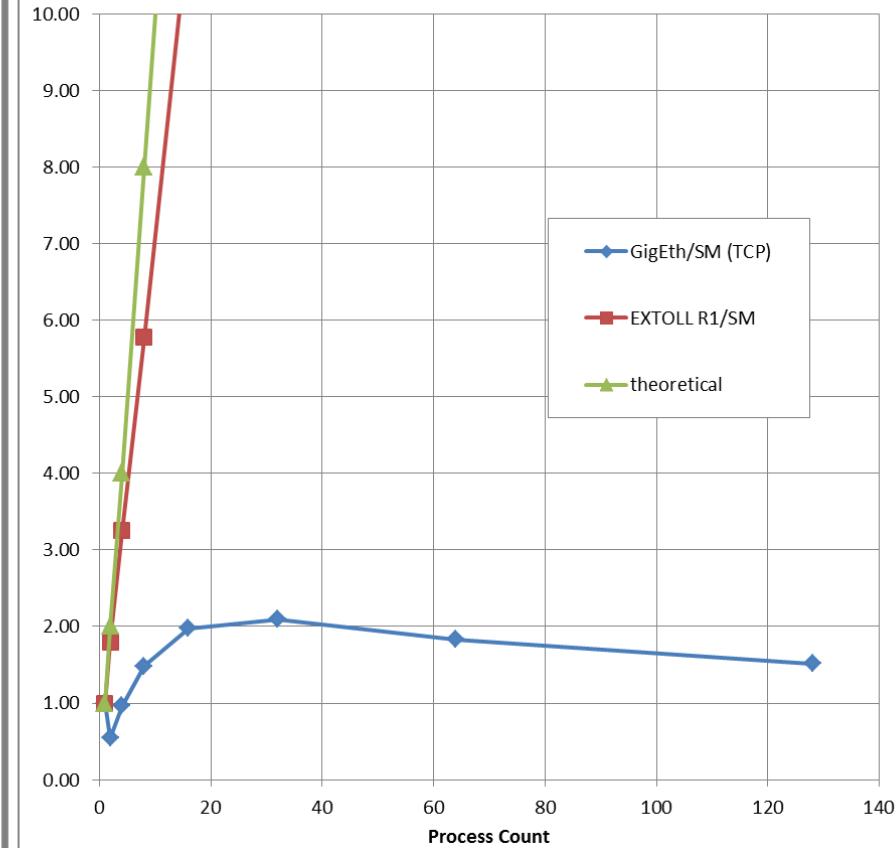


Speed-up – Real experiment

Speed-up of HPCC RandomAccess



Speed-up of HPCC RandomAccess





Amdahl's Law

- Model to find the maximum improvement in terms of performance
 - Assumption: only a fraction of the runtime can be parallelized (parallel fraction P).
 - Assumption that the other fraction is the serial one: serial fraction S
 - Then: $P + S = 1$
 - As fraction P is processed in parallel, this fraction of time is reduced (N parallel execution units)

$$\text{Speedup} = \frac{1}{(1 - P) + \frac{P}{N}}$$

■ Notes:

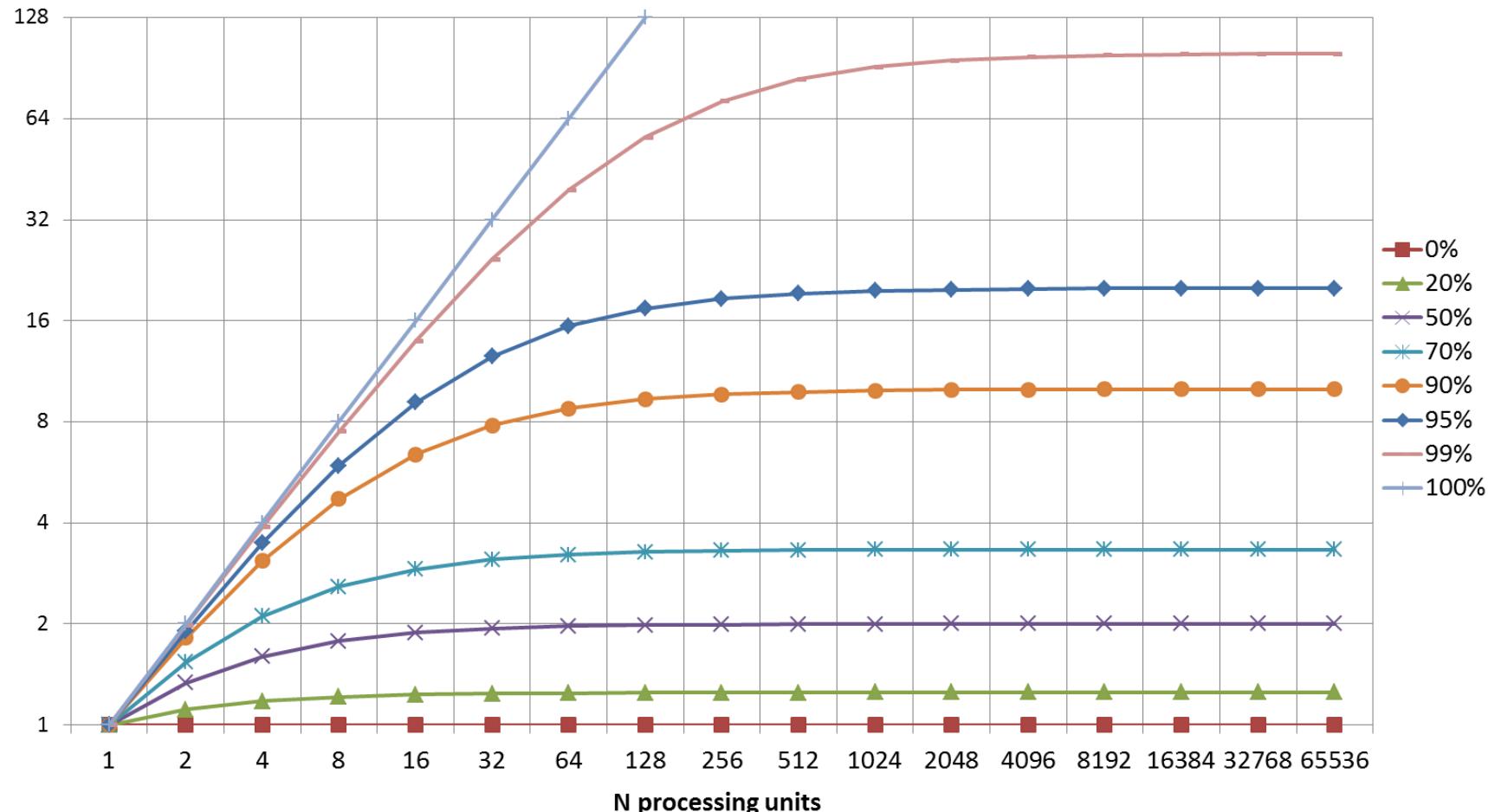
- Speed-up has an upper limit dependent on S , not on N !



Amdahl's Law

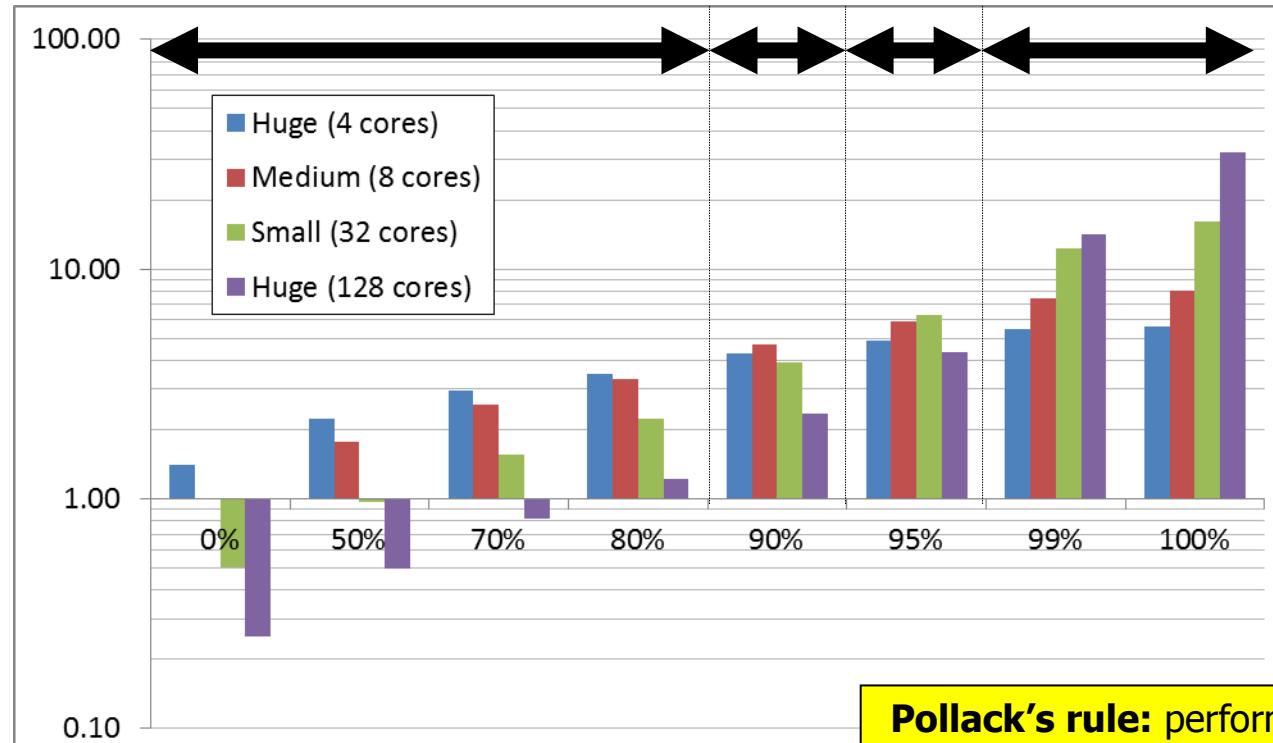
$$\boxed{Speedup = \frac{1}{(1-P) + \frac{P}{N}}}$$

Amdahl's Law
(dependant on parallel portion P)





Amdahl's Law - Implications



Assumptions
80W &
200mm² for
cores

Pollack's rule: performance increase $\sim \sqrt{(\text{complexity increase})}$

| Number of Cores | 4 | 8 | 32 | 128 |
|--------------------------------|-------|-------|------|------|
| Power (W) / Core | 20 | 10 | 2,5 | 0,6 |
| Area (mm ²) / Core | 50 | 25 | 6 | 1,5 |
| Relative Performance R | 140 % | 100 % | 50 % | 25 % |



Notes on Amdahl and his law

■ Amdahl himself ...

1. ... wanted to claim that **parallel computing is not viable**
 - "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", *AFIPS Conference Proceedings*, 1967.
2. ... was an **optimist**
 - Extra work is required for parallelization
 - Synchronization, communication, management, ...
 - In this regard his law is too optimistic
3. ... was a **pessimist**
 - We can (have to?) scale the problem size with **N**
 - Gustafson's law – superlinear speedup (1988)
 - Parallel algorithms exist that reduce fraction **S**
 - Superlinear speed-up due to caching effects



Performance increase - Summary

- Increase of performance according to Moore's Law
 - Technology ☺
- Increase of performance according to Amdahl's Law
 - Limited by serial fraction ☹

"Everyone knows Amdahl's Law, but quickly forgets"

Dr. Tom Puzak, IBM Research, 2007

- Sources for serial fraction
 - Data dependencies
 - Communication & synchronisation is costly
- ⇒ Optimize these components ☺
- ⇒ Increase problem size ☺
- Increases percental fraction of P



TOP500 List



- <http://www.top500.org>
- Biannual list: 500 fastest computer systems world-wide
 - LinPACK benchmark
 - „Dense system of linear equations“
 - $2/3 n^3 + O(n^2)$ double precision floating point operations
 - Highly scalable, problem size can be chosen arbitrary
- Computationally intensive, not memory-bound
 - Little requirements on memory bandwidth and capacity
- Old lists available on-line
 - History and trends

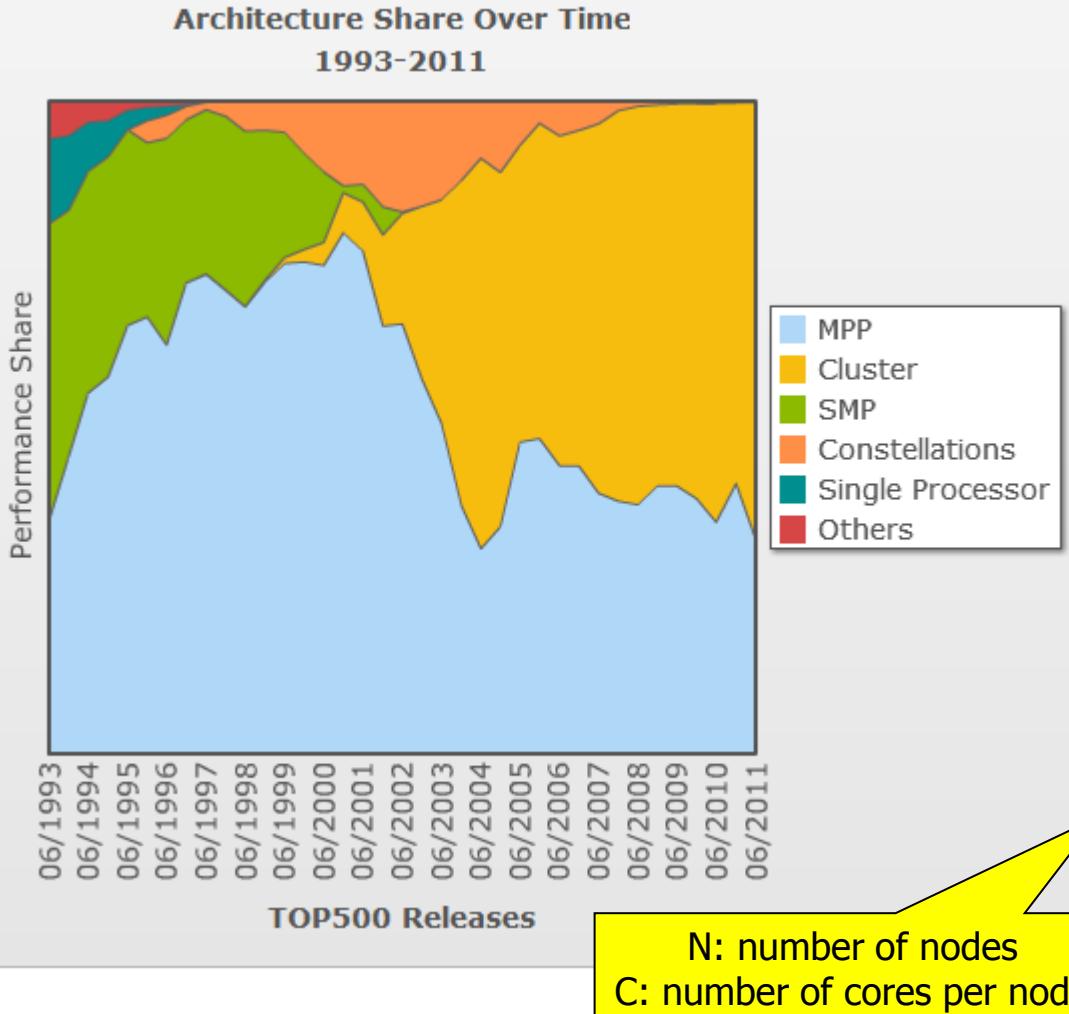


TOP500 – List of 06/2014

| Rank | Site | System | Cores | Rmax (TFlop/s) | Rpeak (TFlop/s) | Power (kW) |
|------|---|--|-----------|-------------------|--------------------|---------------|
| 1 | National Super Computer Center in Guangzhou China | Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT | 3,120,000 | 33,862.7 | 54,902.4 | 17,808 |
| 2 | DOE/SC/Oak Ridge National Laboratory United States | Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc. | 560,640 | 17,590.0 | 27,112.5 | 8,209 |
| 3 | DOE/NNSA/LLNL United States | Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM | 1,572,864 | 17,173.2 | 20,132.7 | 7,890 |
| 4 | RIKEN Advanced Institute for Computational Science (AICS) Japan | K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu | 705,024 | 10,510.0 | 11,280.4 | 12,660 |
| 5 | DOE/SC/Argonne National Laboratory United States | Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM | 786,432 | 8,586.6 | 10,066.3 | 3,945 |
| 6 | Swiss National Supercomputing Centre (CSCS) Switzerland | Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc. | 115,984 | 6,271.0 | 7,788.9 | 2,325 |
| 7 | Texas Advanced Computing Center/Univ. of Texas United States | Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell | 462,462 | 5,168.1 | 8,520.1 | 4,510 |
| 8 | Forschungszentrum Juelich (FZJ) Germany | JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM | 458,752 | 5,008.9 | 5,872.0 | 2,301 |
| 9 | DOE/NNSA/LLNL United States | Vulcan - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM | 393,216 | 4,293.3 | 5,033.2 | 1,972 |
| 10 | Government United States | Cray XC30, Intel Xeon E5-2697v2 12C 2.7GHz, Aries interconnect Cray Inc. | 225,984 | 3,143.5 | 4,881.3 | |



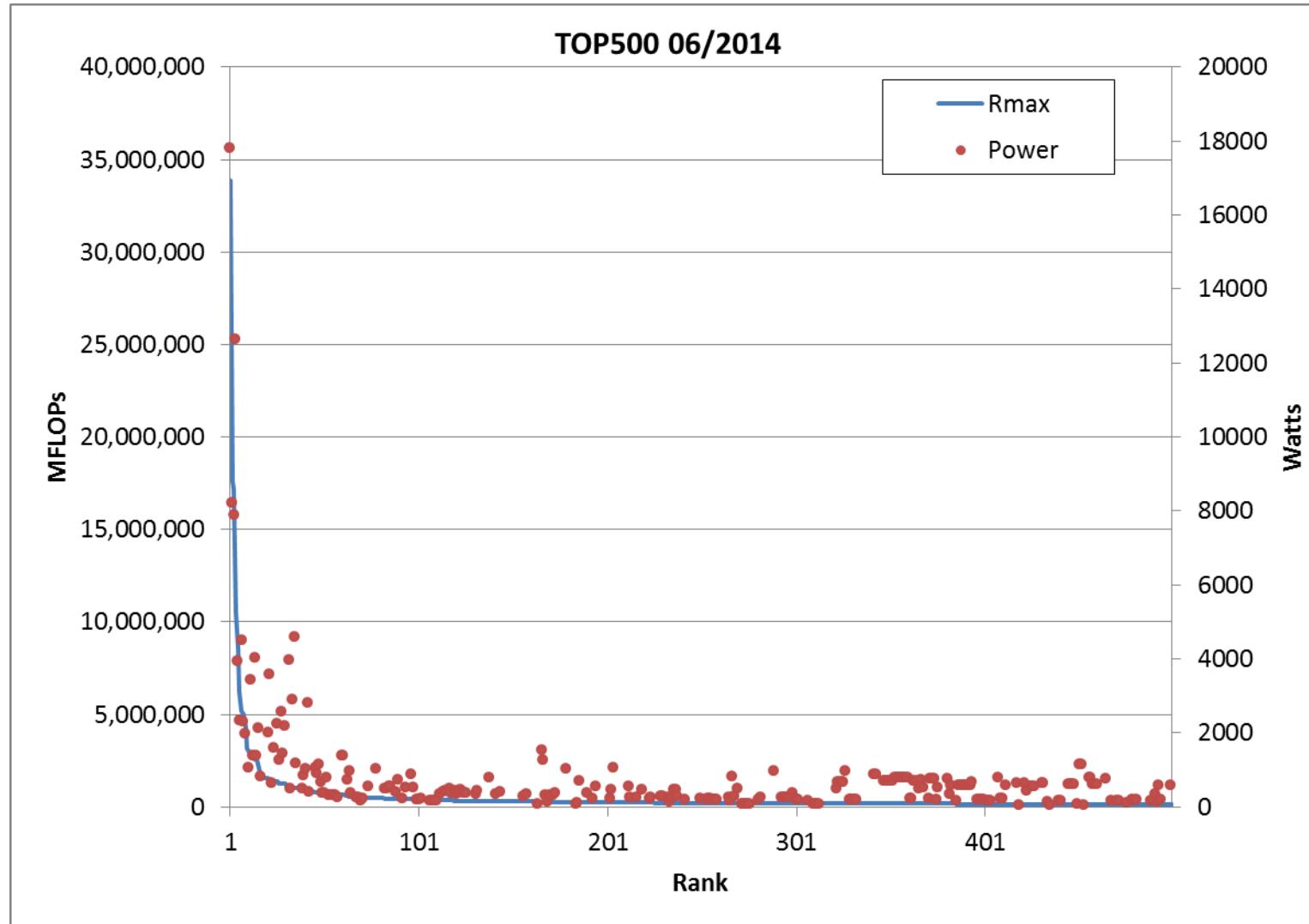
TOP500 – Architecture Share



- System with N nodes, each C cores
 - 1 node equals 1 address space
- Massively Parallel Processors (MPP)
 - $N > C, N \gg 1$
(whatever that means)
- Cluster
 - $N > C, N > 1$
- Symmetric Multi-Processors (SMP)
 - $N = 1$
- Constellations
 - $N < C, N > 1$
- Single Processor
 - $N = C = 1$
- Others
 - Never seen

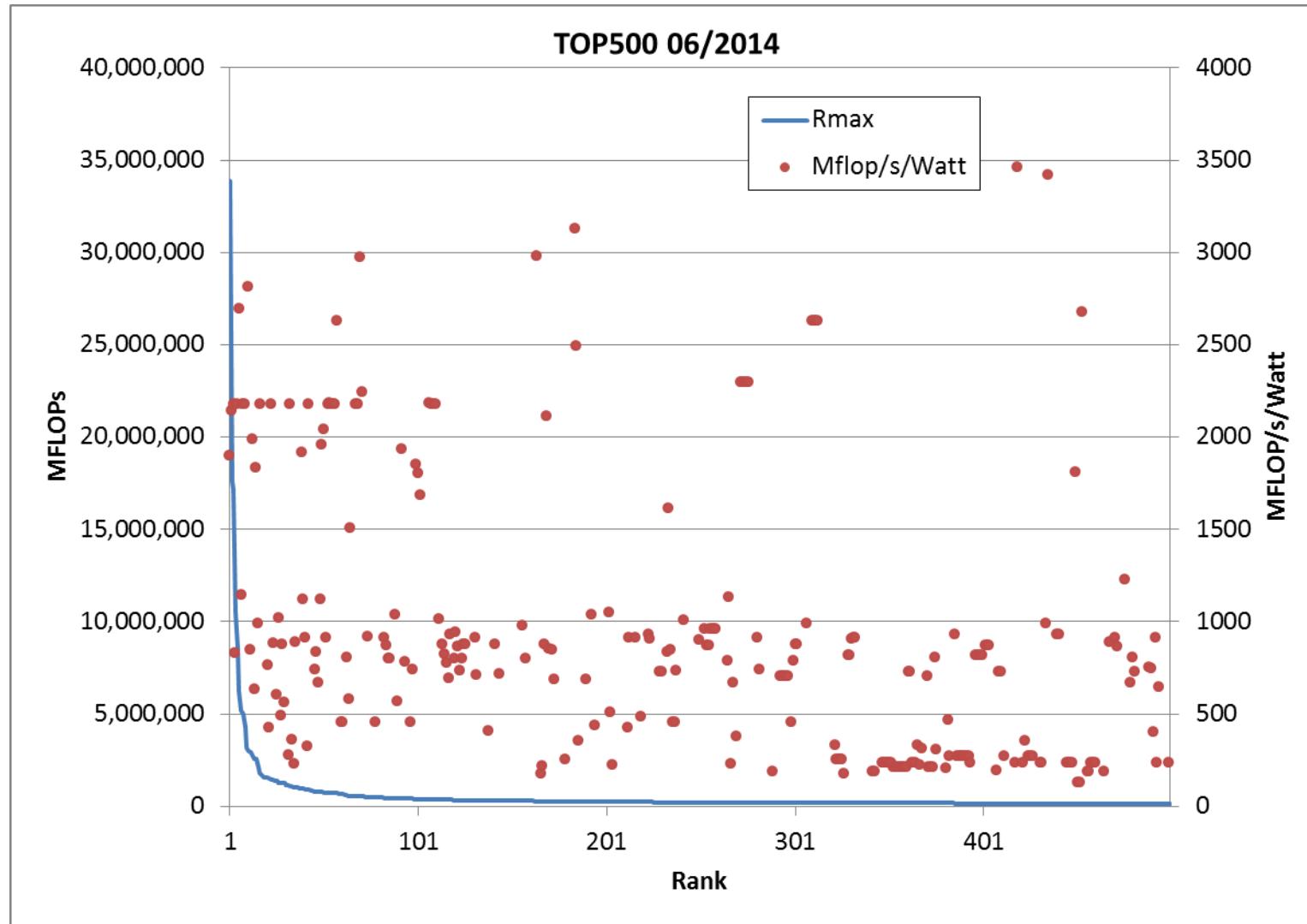


TOP500 – Performance & Power



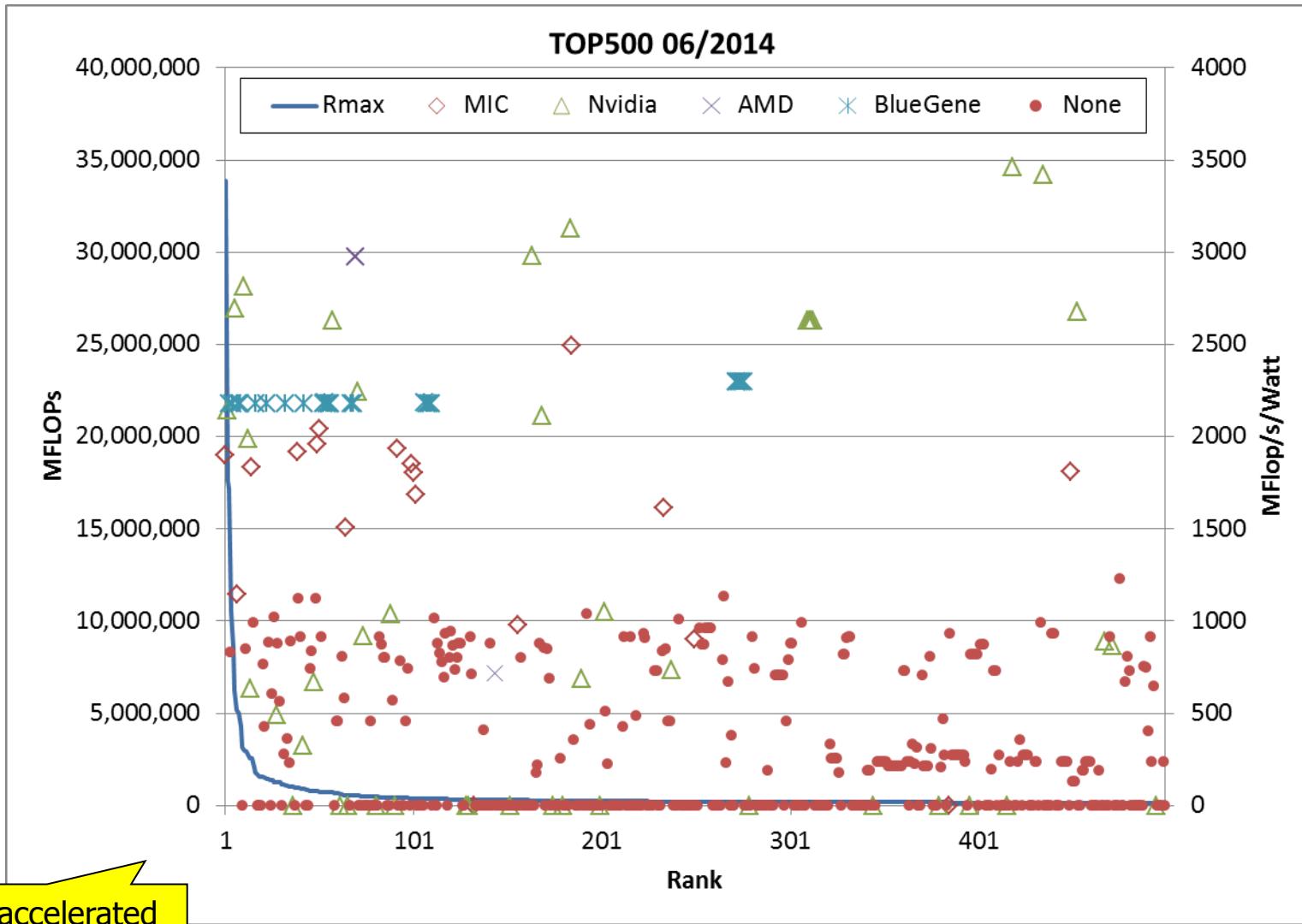


TOP500 – Power Efficiency





TOP500 – Accelerated Systems





TOP500 - Summary

- Excellent tool for trend analysis
 - Introductions, motivating data
- Maybe too limited due to the single workload
 - No scalability worries
- Alternatives
 - Graph500:
<http://www.graph500.org>
 - Green500:
<http://www.green500.org>
 - HPC-Challenge:
<http://icl.cs.utk.edu/hpcc>

▪ Exascale at 2GFLOPs/Watt (BlueGene):

- 1,000,000,000,000,000 Flops (1,000,000,000 GFlops)
- 500,000,000 Watts (500 MWatt)
- @50MWatt: 20GFLOPs/Watt required

→ Extreme specialization

- What about too specialized?



Introduction to High Performance Computing

Lecture 04 – Parallel Computing

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg



Parallelism



Motivation

- Sequential vs. parallel processing completely different
- Multi-/Many-core era
 - Applications designed for single-core
 - **Concurrency** is fundamental for algorithms and applications
- Number of cores/CPU increasing
 - **Scalability** also fundamental
- Further motivations:
 - Performance increase, distributed systems, tolerating I/O Blocking

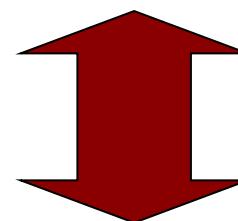
Parallel programming: Concurrency & Scalability (I & II)



Concurrency

Sequential Program

- Single thread of control
- Instructions executed sequentially



Concurrent Program

- Several autonomous sequential threads
- Parallel execution
- Execution determined by implementation

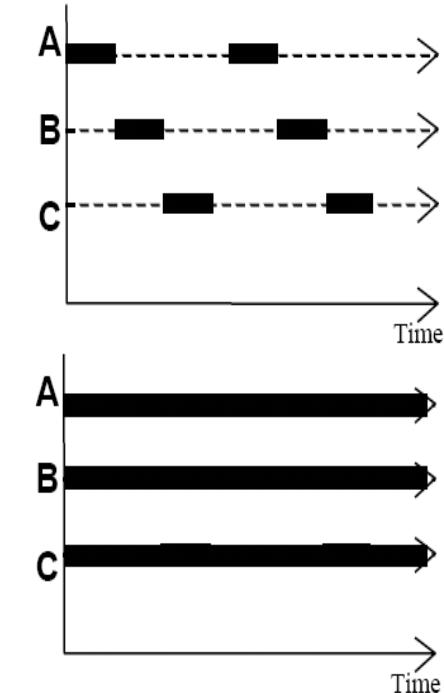
Implementations

- Multi-programming
 - Executing multiple threads on one single resource
 - Time-multiplexing
- Multi-processing
 - Executing multiple threads on one multiple resources
 - Multi-processor, Multi-core
- Distributed processing
 - Executing multiple threads on one multiple independent resources
 - Cluster, Grid, Cloud



Concurrency vs. Parallelism

- Concurrency is not (only) parallelism!
- Concurrency by interleaving
 - Only logical “parallel” execution on one single resource
 - Appearance of “simultaneous” execution
- Parallelism
 - True parallel, simultaneous execution
 - Requires several, parallel resources
- Example for concurrency:
 - Multiple ATMs (“EC-Automaten”) and account balance



Error-free execution on sequential hardware not necessarily implies error-free execution on parallel hardware



Levels of Parallelism – Traditional Approach

▪ Program level

- Coarse grained
- Concurrent execution of multiple programs, or of a single program with different input data sets

▪ Procedure level

- Medium grained
- Different parts of a program are executed concurrently on different parts of a computing system, or one single part with different input data sets

▪ Instruction level

- Fine grained
- Concurrent computation of multiple variables in one procedure

▪ Microcode level

- More fine grained
- Instruction → OperationsPhases
- Execution of different phases of different instruction in different pipeline stages or superscalar execution units simultaneously

▪ Bit level

- Extreme fine grained
- Processing of words only, consisting of multiple bits



Levels of Parallelism – Modern Approach

■ Instruction Level Parallelism (**ILP**)

- Parallelism of one instruction stream
- Huge amount of dependencies and branches
- Limited parallelism (~4-6)

■ Thread Level Parallelism (**TLP**)

- Parallelism of multiple independent instruction streams
- Less amount of dependencies, no limitations due to branches
- Limited by maximal concurrently executable I-streams

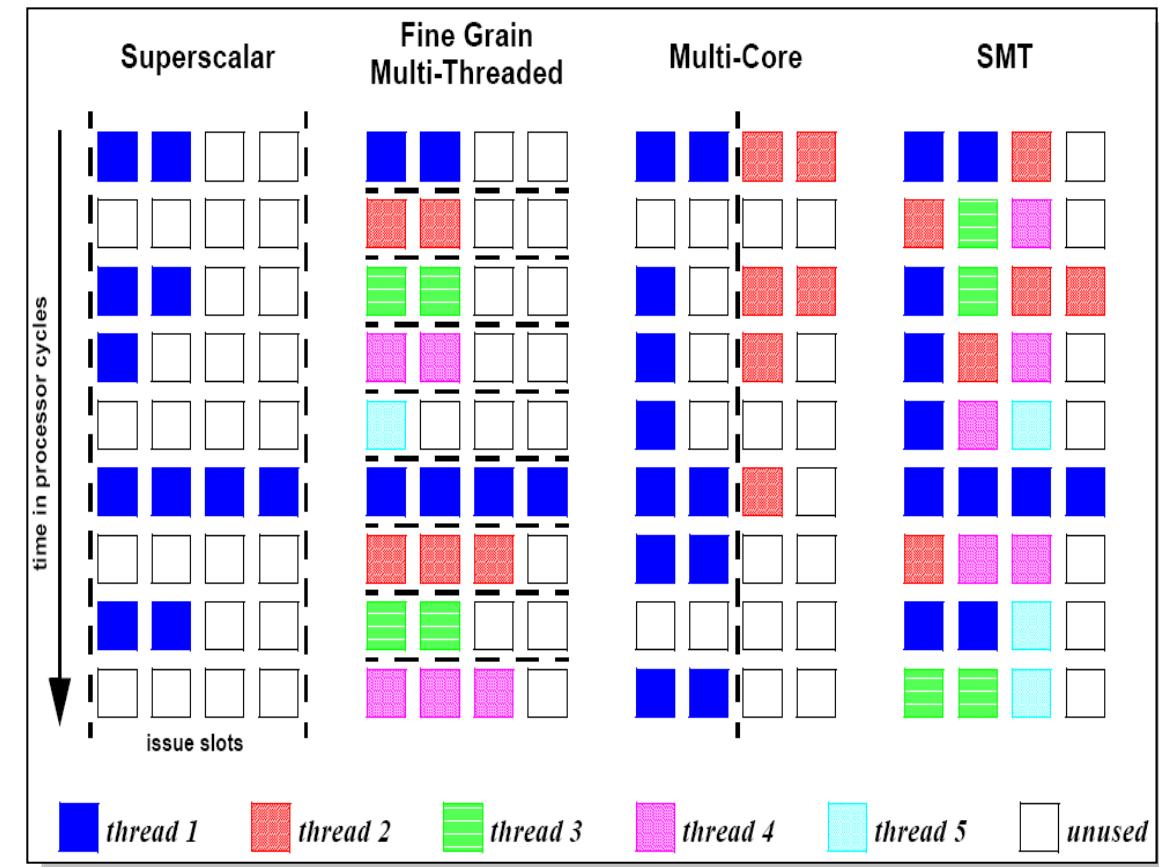
■ Data Level Parallelism (**DLP**)

- Vectorization techniques
- Applying one operation on multiple elements of a data structure
- Parallelism dependent on data structure



Levels of Parallelism (3)

- Exploiting parallelism in different CPU architectures
 - ILP
 - TLP
 - Why no DLP?
- What about GPUs?





Computing Model



Computing model (1)



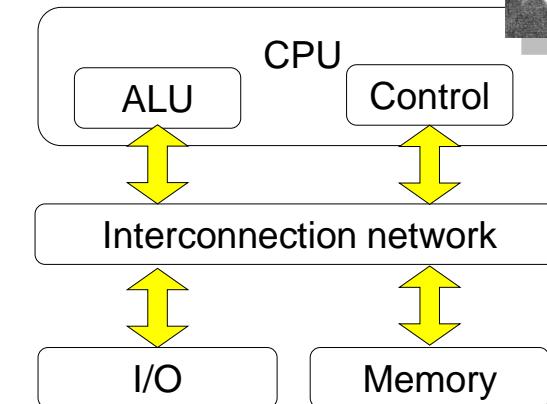
■ von-Neumann architecture

- Main units
 - CPU (Control & Compute)
 - I/O
 - Memory
- “Node” for HPC systems

■ von-Neumann bottleneck

- ALU faster than memory
- Costs for control and communication higher than computing costs

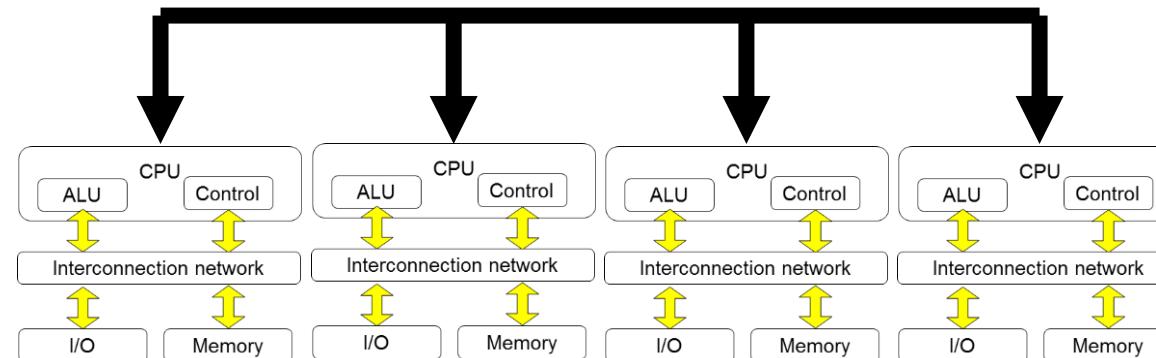
■ Harvard Architecture: Separation of data and instruction memory





Computing model (2)

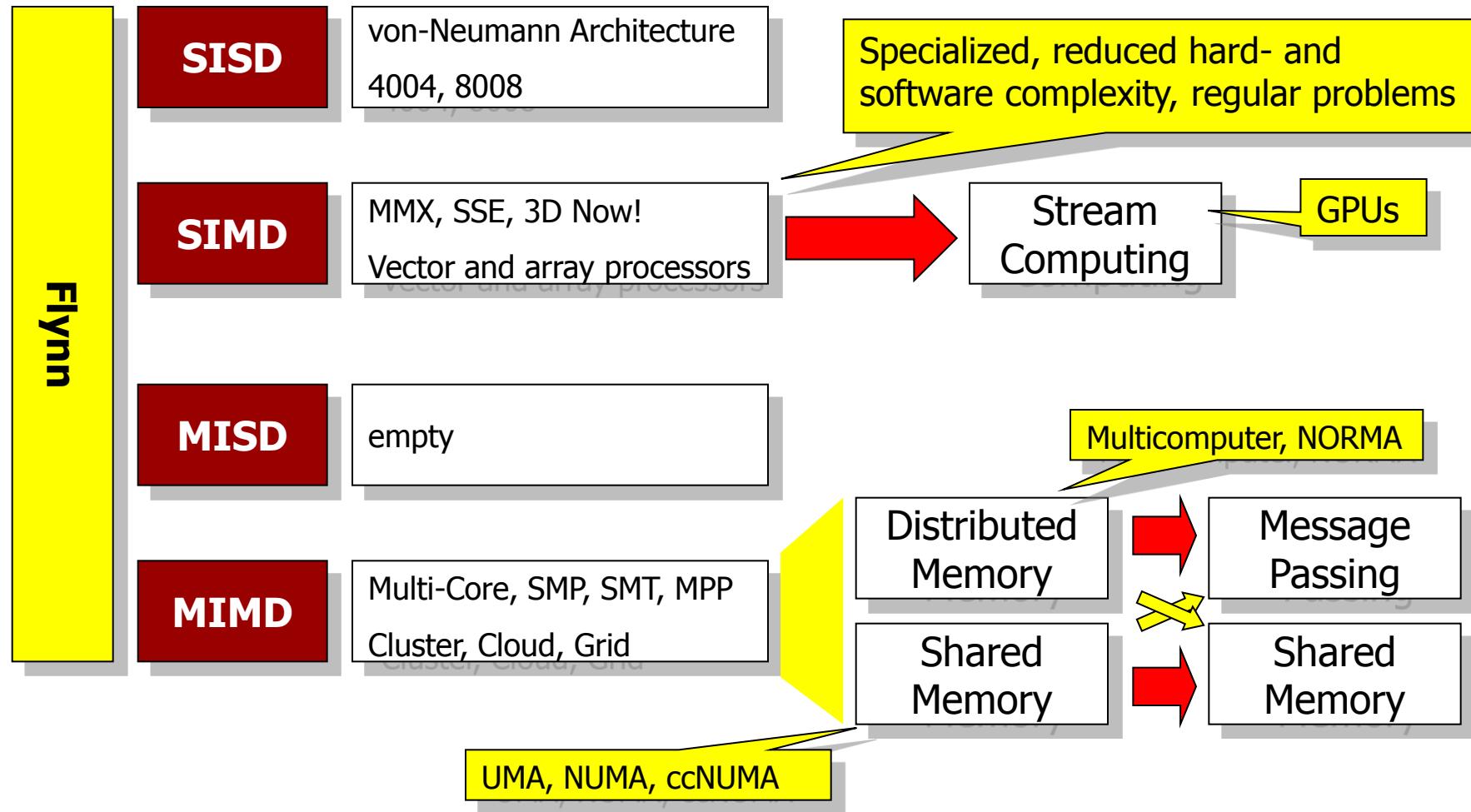
- Multicomputer
 - Multiple nodes
 - Interconnection network
- Many parallel instruction streams
- Local (and remote) memory accesses
- According to Flynn?



Parallel programming: Locality (III)



Classification by Flynn

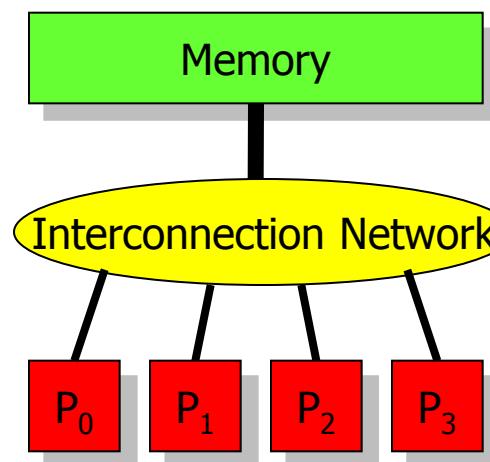




Distributed and Shared Memory

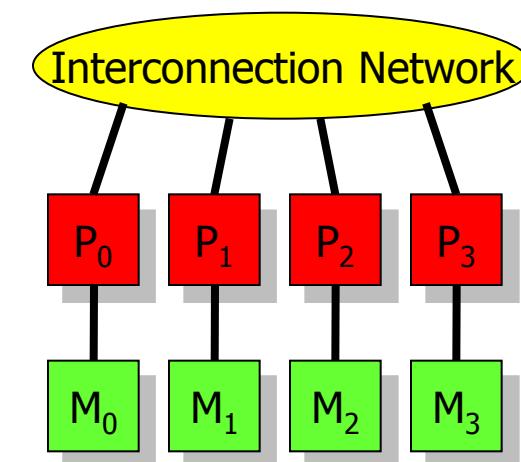
■ Shared Memory

- Shared use of one copy
- Scalability issues
- Atomicity, locking, synchronization



■ Distributed Memory

- Explicit data exchange
- Only access to local memory
- Data distribution and communication scheme





1. Concurrency

- Functional Decomposition, Domain Decomposition, Pipeline Decomposition
- Re-engineering for parallelism
 - Control dependencies, data dependencies

2. Parallel programming paradigms

- Shared memory: PThreads, OpenMP
- Distributed memory: Message-passing
- Data parallel operations (SIMT): CUDA, OpenCL

3. Supporting structures

- SPMD, loops, master/worker, fork/join, data structures



- von-Neumann:
 - 1 node → 1 instruction stream
- Multicomputer:
 - n nodes → n instruction streams
 - Too complex
- Modular approach
 - Simple components made of abstract elements
 - Data structures, loops, procedures
- Single Program Multiple Data (SPMD)

Parallel Programming: Modularity (IV)

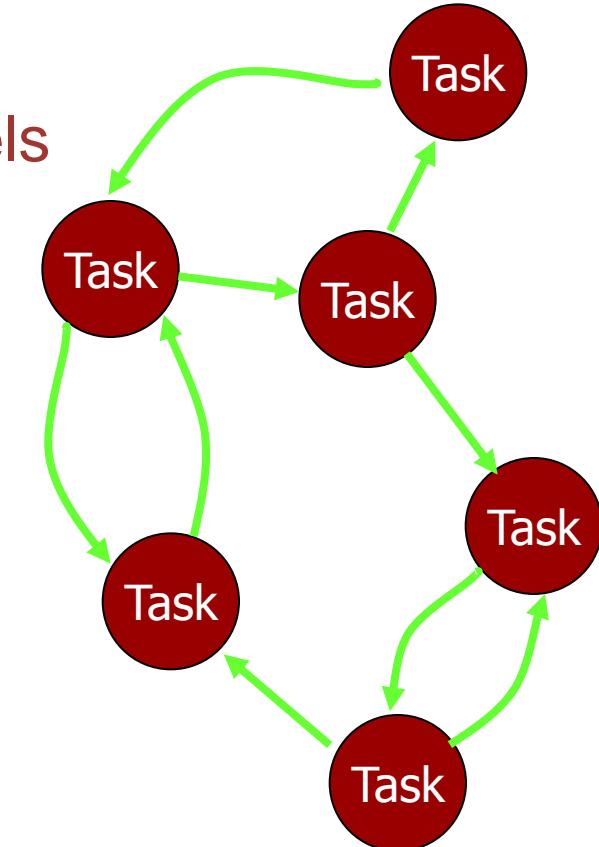


Programming Model

- Used for this lecture: tasks & Channels

- Task
 - Computations
 - Instructions & Memory
- Channel
 - Communication among tasks
 - Message-based
 - Blocking receives

- Computation & Communication
- Data dependencies





■ Message Passing

- Difference:
 - Message Passing: send to x
 - Task/Channel: send over channel y

- SPMD

■ Data Parallelism

- Applying one operation to multiple elements of a data structure
- SPMD

■ Shared Memory

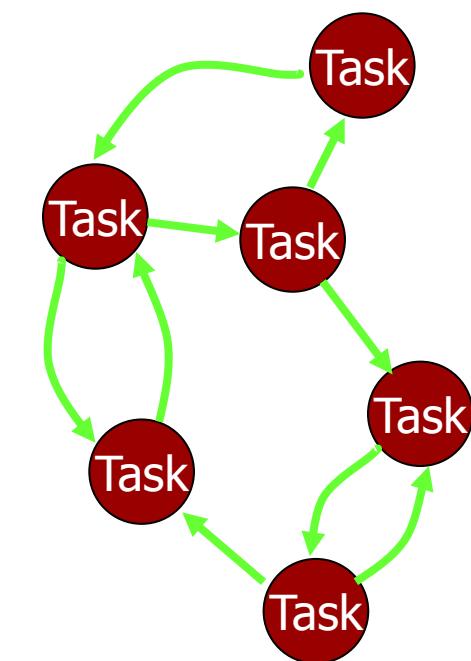
- Uniform memory access from user's point of view
 - No explicit communication
 - Locks & Semaphores
- SPMD



Synchronization

Synchronization is the enforcement of a defined logical order between events. This establishes a defined time-relation between distinct places, thus defining their behavior in time.

- Communication & synchronization
 - Explicit / implicit
- SIMD: one instruction stream, no synchronization necessary
- MIMD: synchronization necessary
 - Shared variables
 - Process synchronization
 - Blocking message exchange





Algorithm Design

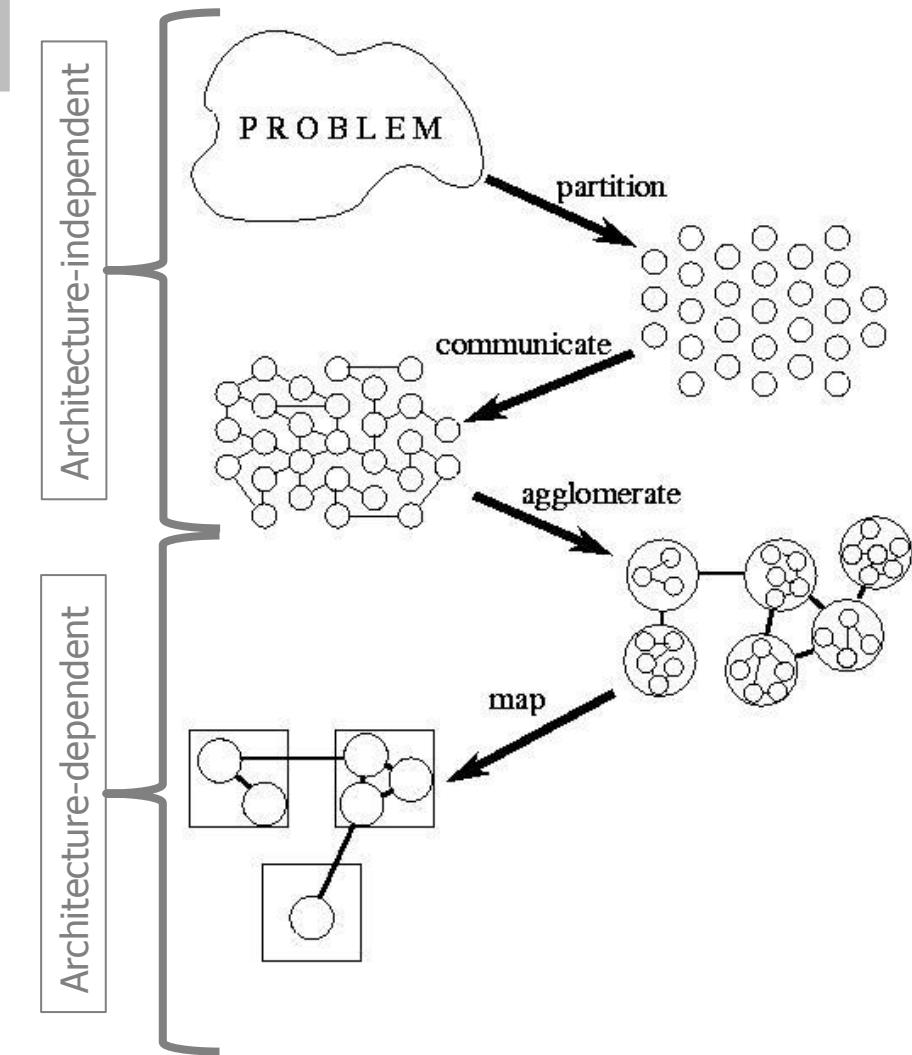
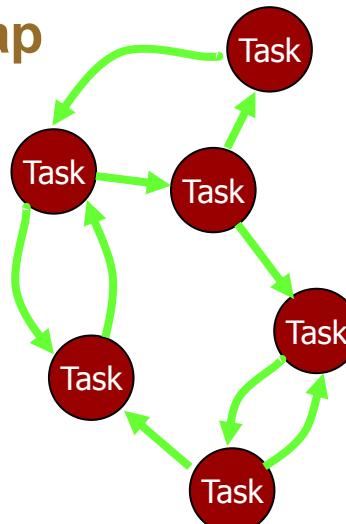


Algorithm Design

Book is online at:
<http://www.mcs.anl.gov/~itf/dbpp>

■ Foster's PCAM

- Partition
- Communicate
- Agglomerate
- Map





▪ PCAM: Partitioning

- Ignore technical aspects like number of processing units
- Maximal granularity
- Partition computation and data
 - Domain Decomposition
 - Functional Decomposition
 - Pipeline Decomposition
- Avoid replication, disjoint partitioning
 - See also minimization of communication

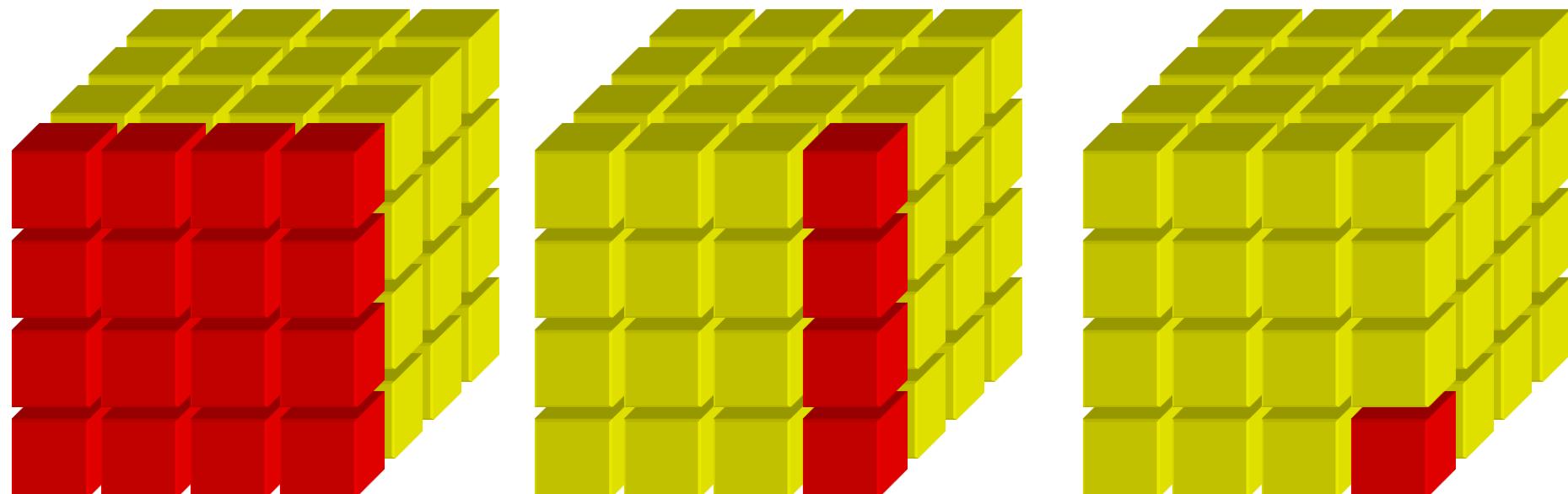
Number of Tasks >> Number of P

→ **Partitioning**



▪ PCAM: Partitioning

- Example 1: Domain Decomposition
- Typical uses: data parallelism, e.g. arrays & trees

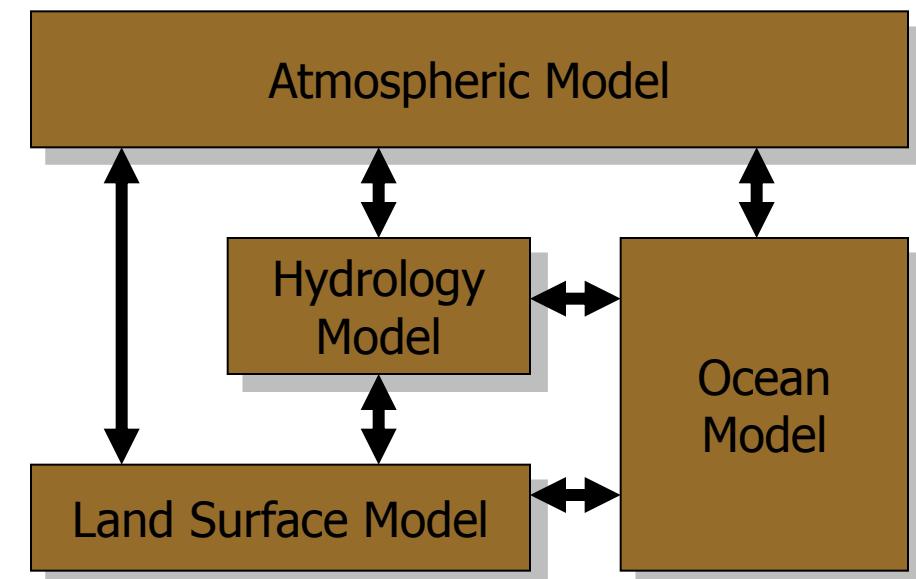




▪ PCAM: Partitioning

- Example 2: Functional Decomposition
- Typical uses:
 - Function calls
 - Different loop iterations
- Rather too many tasks than too few!

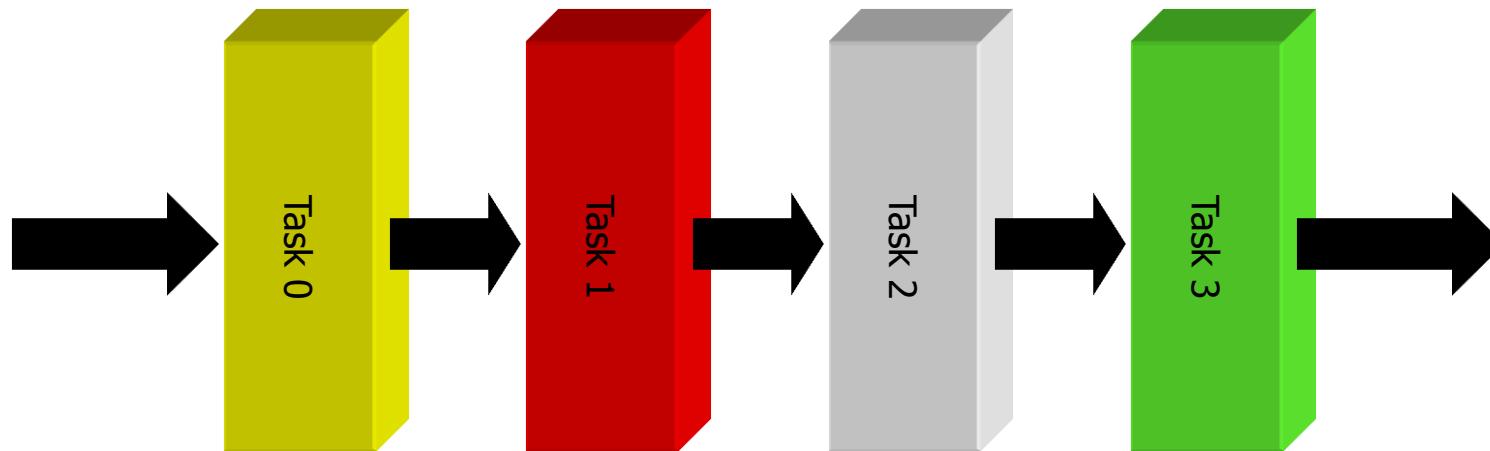
Climate Computing Model





■ PCAM: Partitioning

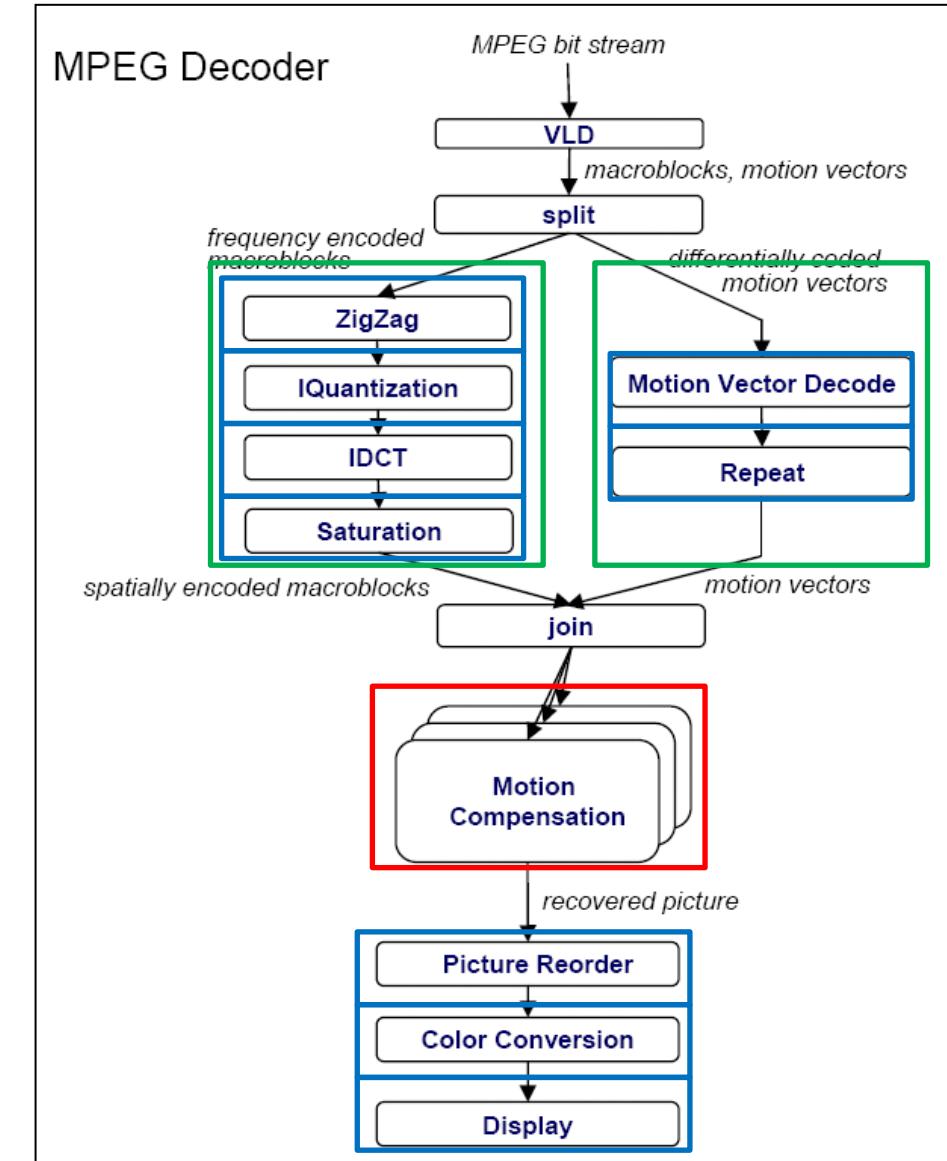
- Example 3: Pipeline Decomposition
- Data flow through several pipeline stages
- Typical uses:
 - Instruction pipelining in modern CPUs





Algorithm Design

- Identify possible decomposition techniques!
- Domain Decomposition
 - Red
- Functional Decomposition
 - Green
- Pipeline Decomposition
 - Blue





▪ PCAM: Communicate

- Execution of partitions **concurrently**, but not **independently**
 - Data dependencies → communication & synchronization
- Complex for DD, rather simple for FD
 - Local/global, structured/unstructured, static/dynamic, synchronous/asynchronous

→ Communication scheme

▪ Data-parallel language

- Requires data-parallel operations and data distribution. Channels actually not necessary, but help for locality and communication costs



■ PCAM: Communicate

- Example for local communication: stencil operation
 - Simple numerical computation: finite difference method (iterative method used to solve a linear system of equations)
 - **Gauss-Seidel (GS)**

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t+1)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t+1)} + X_{i,j+1}^{(t)}}{8}$$

vs. **Jacobi**

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8}$$

- GS optimal for sequential execution (fewer iterations), but too many dependencies for parallel execution
- GS execution: diagonal wave front or Red/Black method

■ **PCAM: Communicate**

- Global communication
 - E.g. global addition (parallel reduction)

$$S = \sum_{i=0}^{N-1} X_i$$

- Cons: **O(N)**, centralized & sequential
- More equal distribution of computation and communication, **O(N-1)**

$$S_i = X_i + S_{i-1}$$

■ **Divide & Conquer to exploit parallelism**

- Tree structures, as long as partitions can be computed independently
- Associativity of addition, **O(log N)**



■ **PCAM: Agglomeration**

- From the abstract to the concrete
- Fixing the parallel computer model

■ **Goal**

- Increase granularity (coarse-grain)
- Maintaining flexibility, therefore reducing development costs

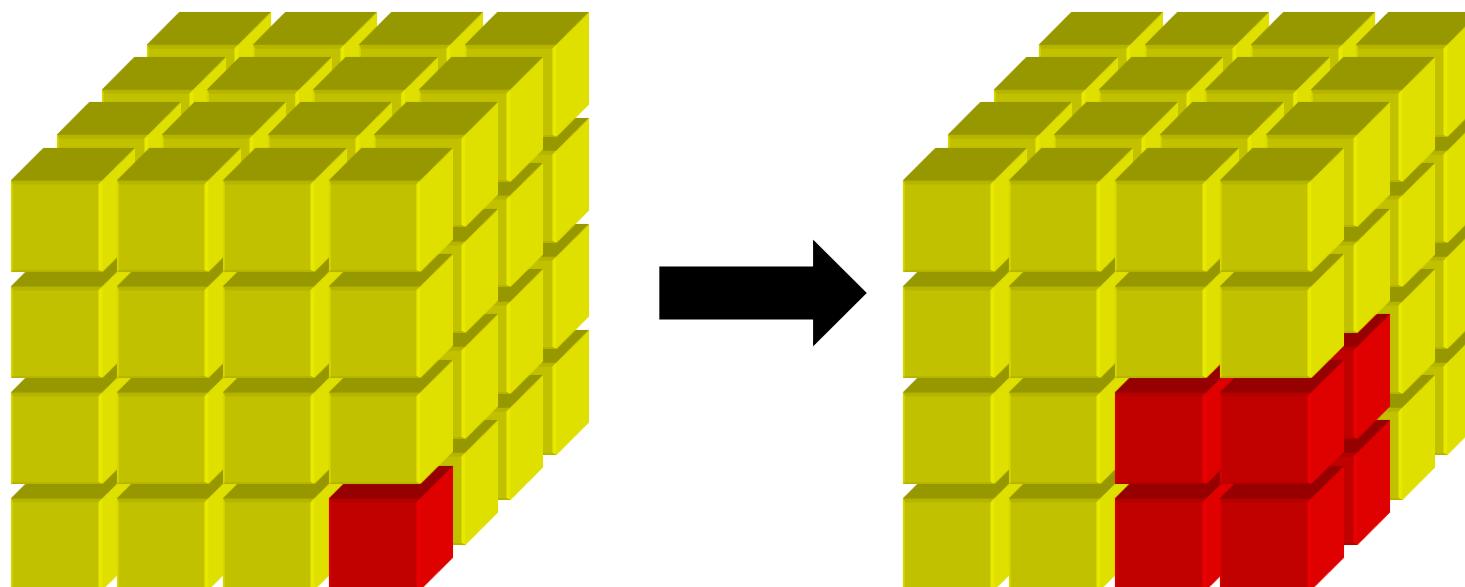
■ **Number of tasks T \geq number of processors P**

- Depending on use case:
 - One order of magnitude more Ts than Ps (parallel slackness)
 - $T = P$ (HPC)
- SIMD: $T = 1$
- If $T = P$, then mapping (almost) done



■ PCAM: Agglomeration

- Combining of tasks
 - Increase of granularity
- Motivation: Reducing communication costs
 - Fixed & variable fraction (surface-to-volume effects)





■ PCAM: Agglomeration

- Replication of data and computation

■ Example: global sum

- Chained: $2(N-1)$ steps (sum & broadcast)
→ Redundant computation in a ring, no broadcast ($(N-1)$)
- Tree-based: $2 \log N$ steps (sum & broadcast)
→ Redundant computation in a butterfly, no broadcast ($\log N$)

→ Reducing Communication



■ PCAM: Mapping

- Assignment: task \leftrightarrow processor & memory
 - Place tasks that can execute concurrently on different processors
 - Place tasks that communicate frequently on the same processor
 - Note that this implies conflicts
- Mapping not necessary for:
 - Uni-processors or shared memory systems with automatic mapping
 - Hardware mechanism or the OS responsible for scheduling

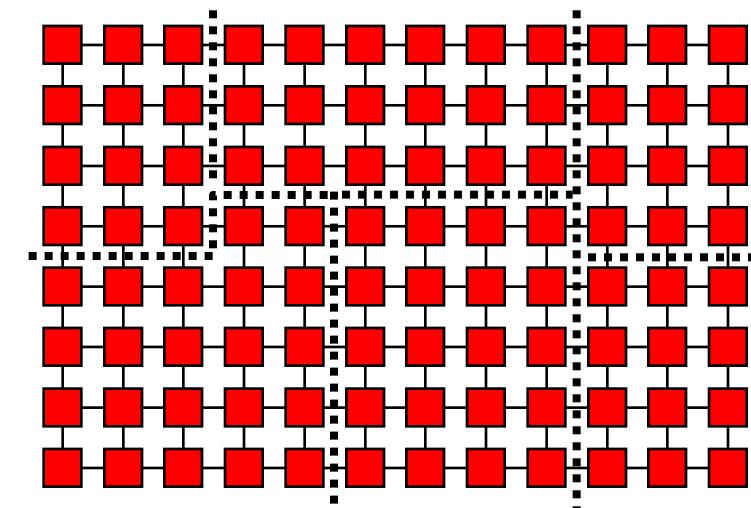
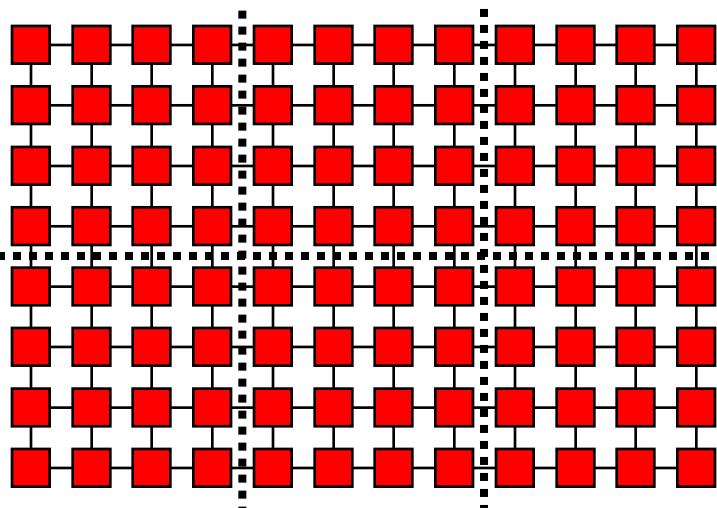
■ Mapping problem is NP-complete

■ Dynamic Load Balancing



■ PCAM: Mapping

1. Concurrent tasks on different Ps
2. Frequently communicating tasks on same P



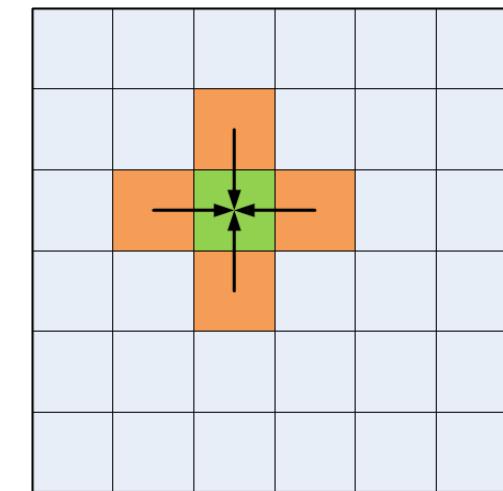
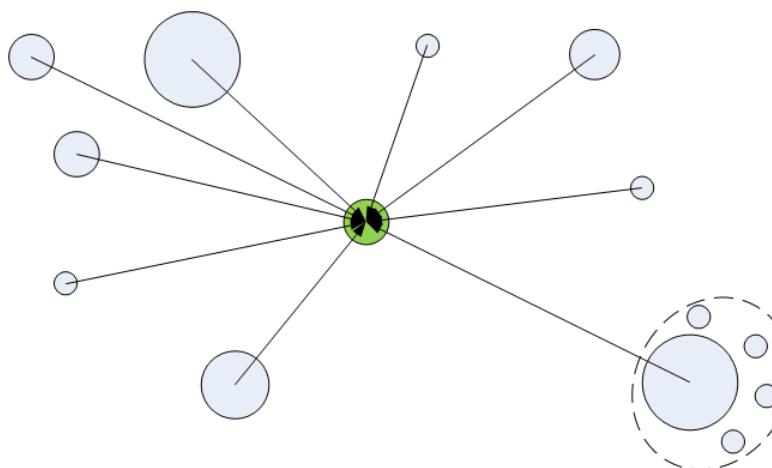
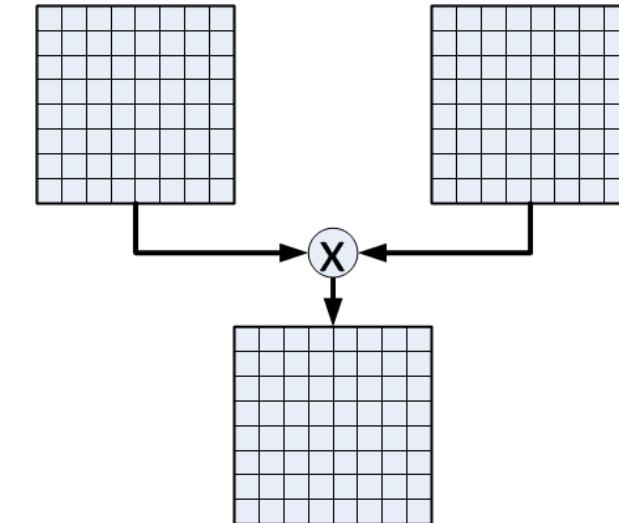
Ready to run!



Parallel Computing

Algorithm Design - Examples

- Matrix multiply
- Stencil operation
- N-Body problem



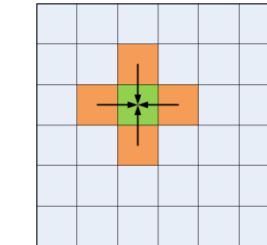
Parallel Computing
Algorithm Design - Examples

■ Stencil codes (e.g. Jacobi method)

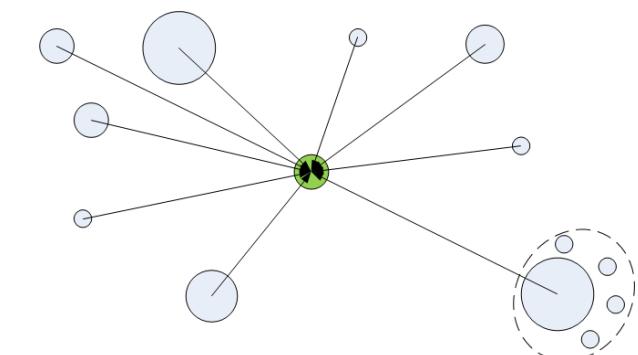
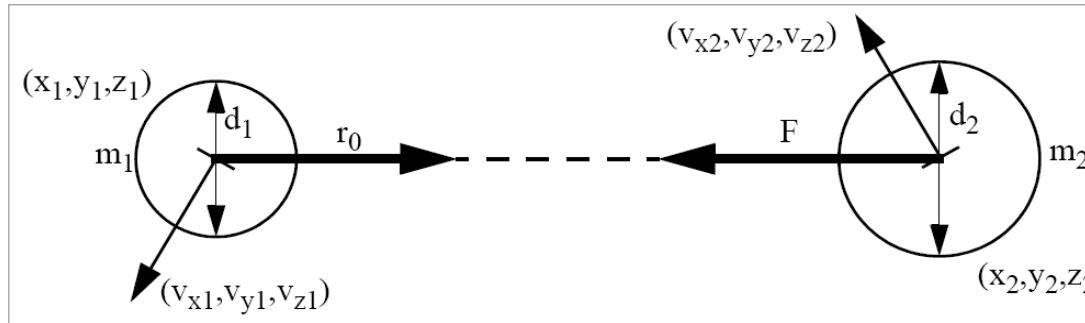
- Approximation by time steps

■ N-Body codes

- Gravitational forces, electrostatical forces
- Smoothed particle hydrodynamics (simulating fluid flows)
- Superposition
- Approximation by time steps



$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8}$$





- Concurrency and parallelism of fundamental importance
 - Granularity
 - ILP, TLP, DLP
- Characteristics of “good” parallel programs
 - Concurrency, Scalability, Locality and Modularity
- Algorithm design
 - Partition, Communicate, Agglomerate, Map
- Parallel computing highly dependent on architecture!
 - (Flynn’s classification,) shared & distributed memory
- Literature
 - Foster Online
 - <http://www.mcs.anl.gov/~itf/dbpp>
 - Introduction to Parallel Computing
 - <http://www-users.cs.umn.edu/~karypis/parbook>



Introduction to High Performance Computing

Lecture 05 – Practical Parallel Programming Example

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg



Matrix Multiply

❖ Why always Matrix Multiply?

- One of the most heavily optimized codes in HPC
- Interesting access patterns
- Good mixture of sufficient complexity but still simple enough for a comprehensive understanding
- Finally, it's an important operation!

❖ Used in many applications as computational kernel

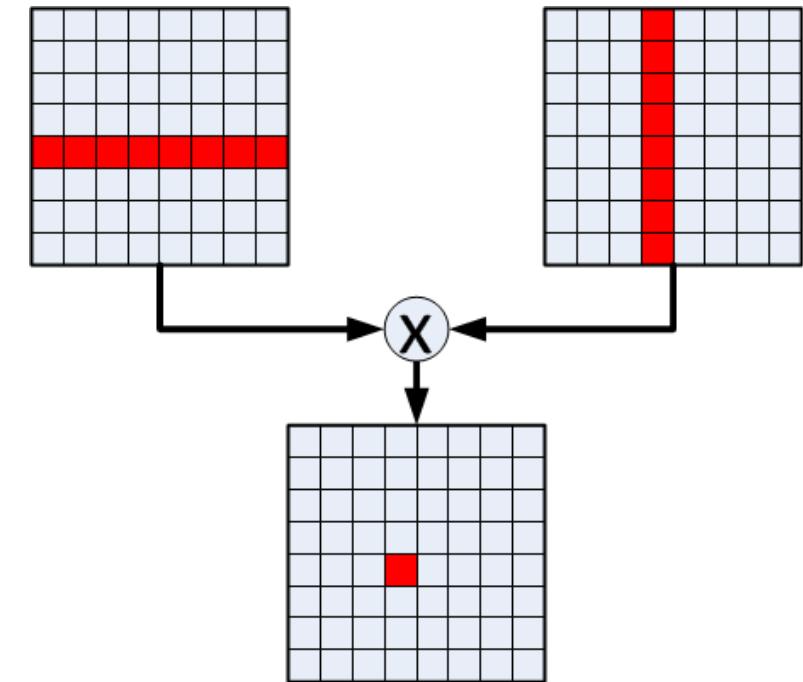
- In particular for sparse matrix operations

❖ Here: for dense matrices

- Experiments and learning
- High sustained/peak ratio
- Test system/compiler/OS

❖ Note on notation

- $M[\text{row},\text{column}] = M[\text{row}][\text{column}]$
- Analogous to C





Matrix Multiply – Sequential version

• $\text{C} = \text{A}[][] * \text{B}[][]$

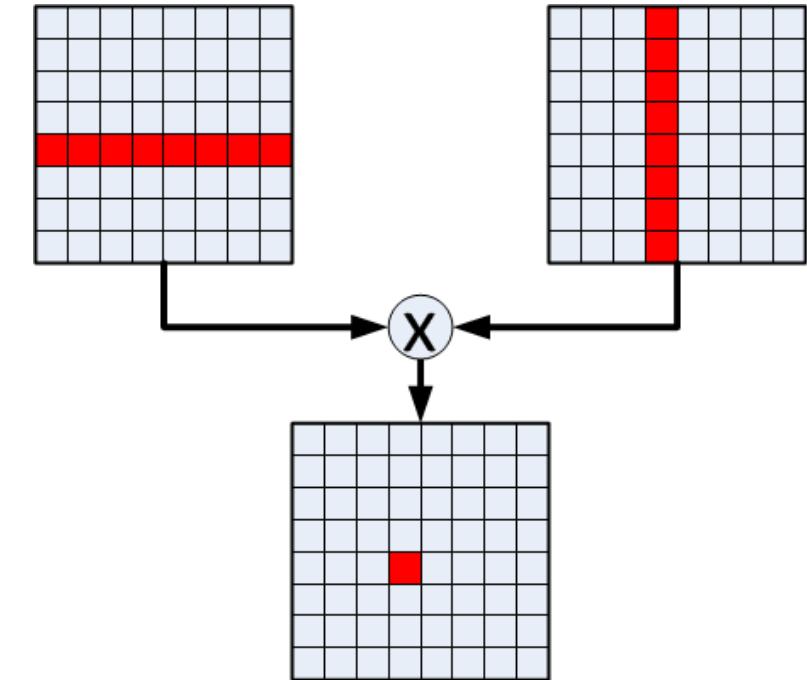
- $\text{A}[i,j] == \text{A}[i][j]$

• Locality

- Row-major order storage for C
programming language

• Caching effects

- Nice for A, bad for B
- True/False sharing
- Potentially evicts other useful blocks (3C)
 - Compulsory
 - Conflict
 - Capacity



```
for i in n
    for j in n
        for k in n
            C[i,j] = C[i,j] + A[i,k]*B[k,j]
```



Matrix Multiply - Analysis

❖ Analysis

- Assume square matrices
- Assume perfect write-through cache (no conflict, no capacity)

❖ Number of flops: $f = 2 * N^3$

- N^2 elements in C, each N steps
- Each step: multiply & add

❖ Number of cache misses:

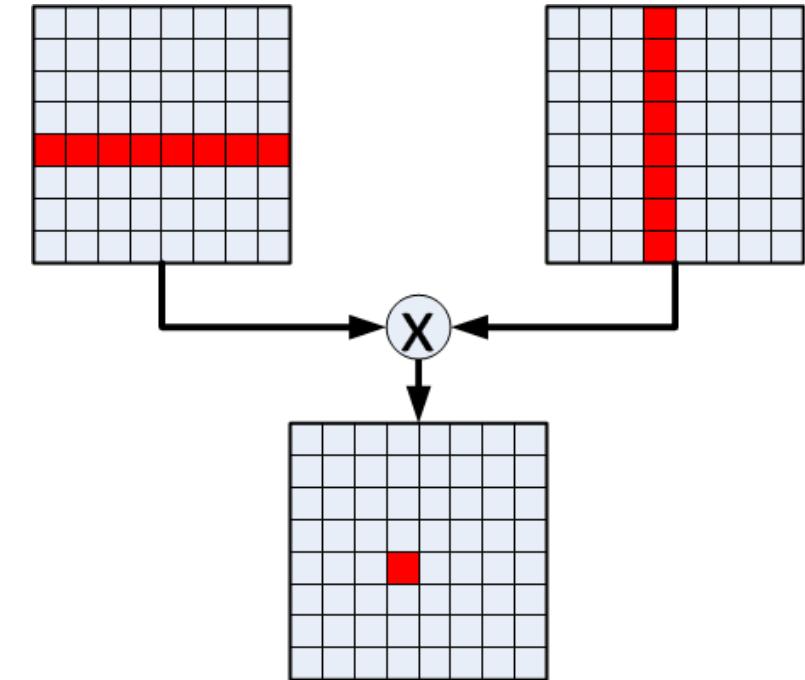
$$m = 3 * N^2$$

- Load from A,B,C, store to C
- Counting all accesses: $4 * N^3$

❖ Ratio mem/flop:

$$r = m/f = 2/N = O(1/N)$$

- Computationally intensive
- Peak performance expected for cache-based processor

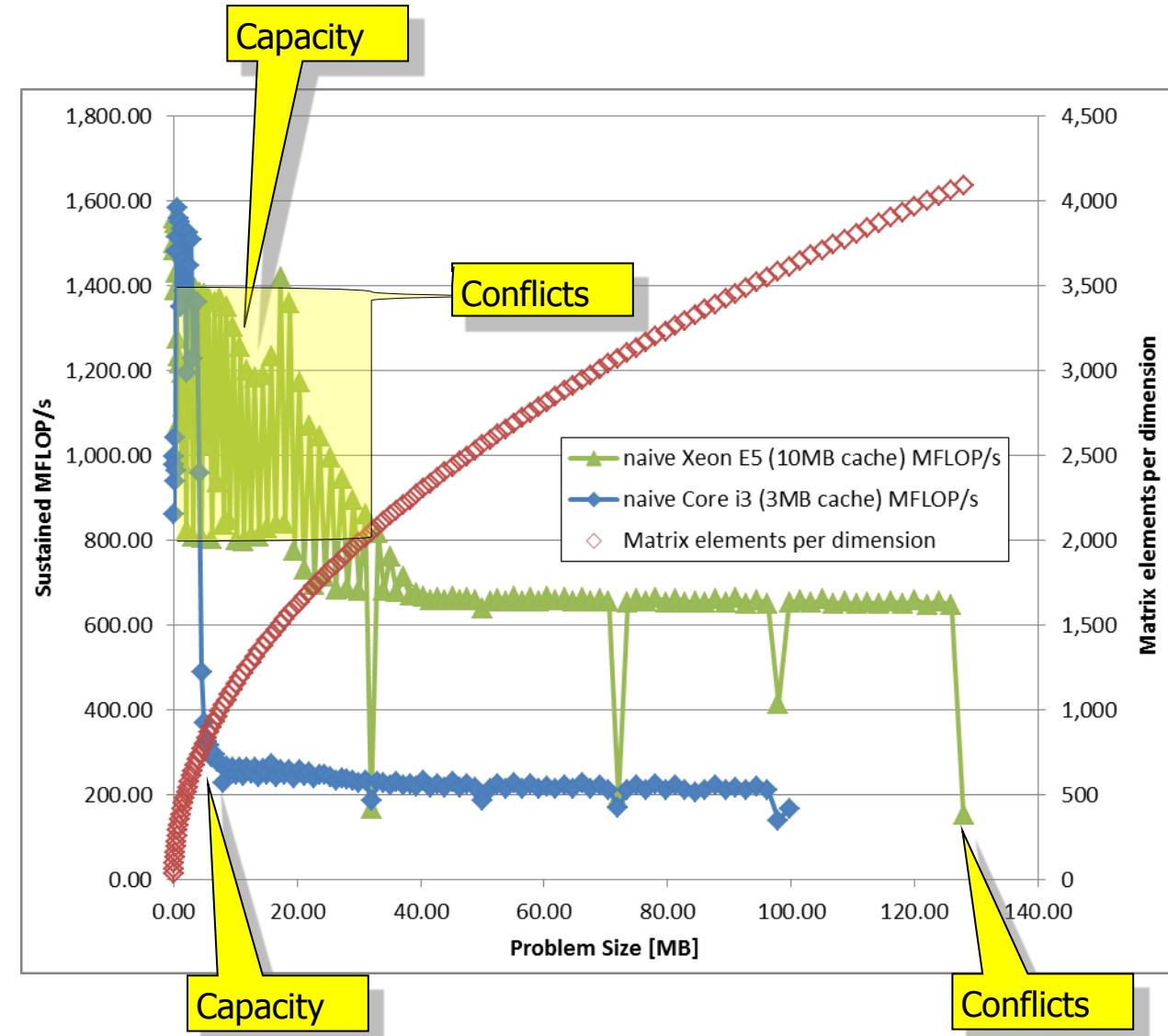


```
for i in n
    for j in n
        for k in n
            C[i,j] = C[i,j] + A[i,k]*B[k,j]
```



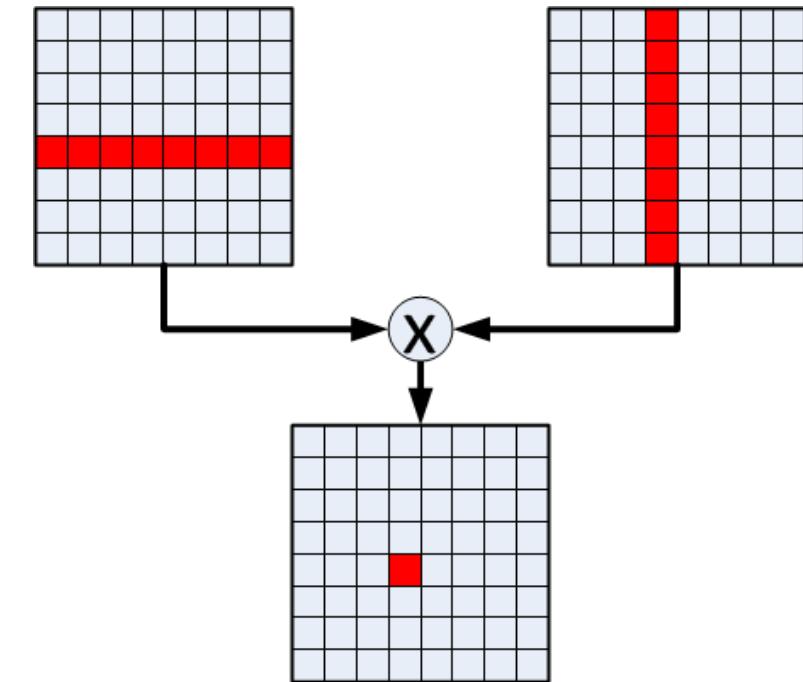
Matrix Multiply - Baseline Performance

- ❖ Core i3, 3.1GHz
dual core, AVX
 - 49.6 GFLOP/s peak
- ❖ Xeon E5, 2.4GHz
quad core, AVX
 - 76.8 GFLOP/s peak
- ❖ Size = 3 matrices
each N^2
- ❖ Flops = $2N^3$
- ❖ Nice caching
effects visible
 - Core i3
 - Xeon E5





- ❖ Suffers from non-linear access to B
 - „Pseudo-Random“ from the cache point of view
 - Neither spatial nor temporal locality exploitable
- ❖ $r = O(1/N)$ only holds for optimal caching
- ❖ Otherwise:
 - $m_{uncached} = 3 * N^3$ for all accesses (worst caching)
 - $F = 2 * N^3$
 - $r = m/f = 2$ or $O(1)$



```
for i in n
    for j in n
        for k in n
            C[i,j] = C[i,j] + A[i,k]*B[k,j]
```



❖ Solution: transpose B

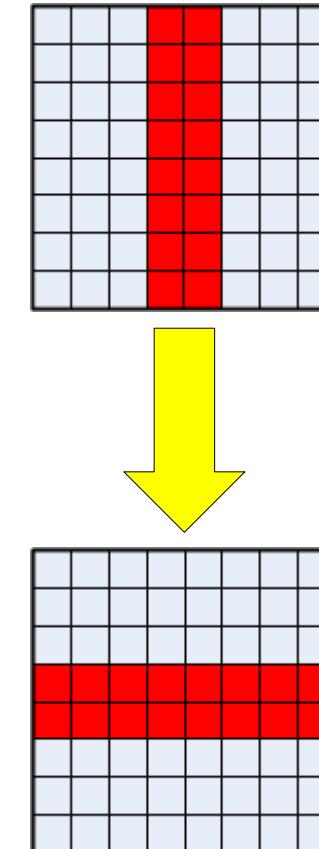
- In a linear load fashion, written back with stride
- Optimized for caches

❖ Costs:

- $f_t = 0$
- $m_t = 2N^2$

❖ New overall r:

$$\begin{aligned} r &= (m_{\text{cached}} + m_t) / (f + f_t) \\ &= (3N^2 + 2N^2) / (2^*N^3 + 0) \\ &= 5/(2N) \\ &= O(1/N) \end{aligned}$$

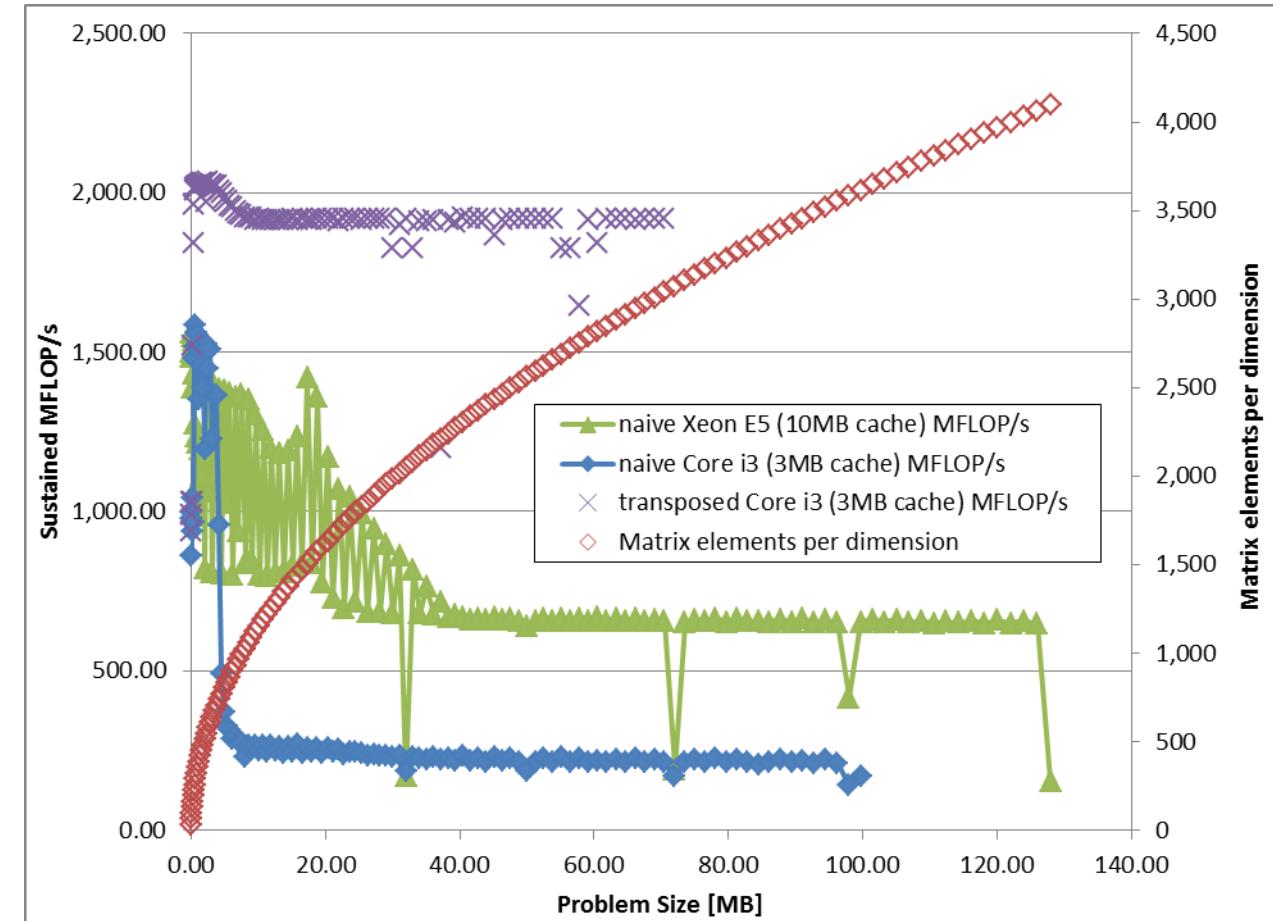




Matrix Multiply – Seq. Version First Optimization

❖ Theoretical peak in this case (single threaded, no AVX or SSE)

- Core i3: 3.1×2 (MADD) = 6.2 GFLOP/s





Tiling/Blocking

❖ Increase locality by reordering memory accesses

- Associativity
- Tiling or blocking
- Each TxT tile uses each element T times

❖ Calculate only parts of the elements of C, so that access pattern has high locality

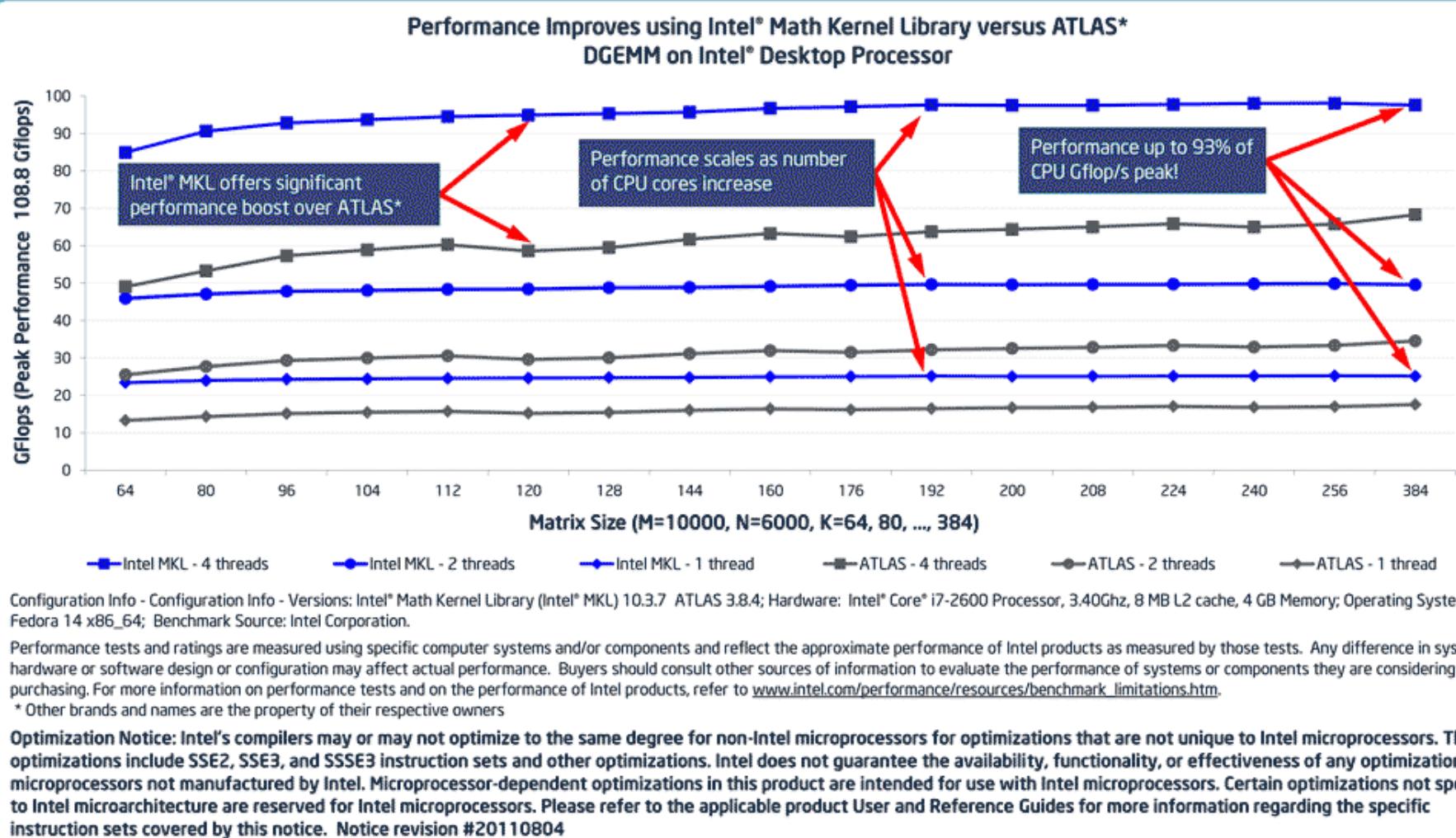
- Beneficial for both sequential and parallel algorithms

Time ↓

| $C_{0,0}$ | $C_{1,0}$ | $C_{0,1}$ | $C_{1,1}$ |
|---------------------|---------------------|---------------------|---------------------|
| $B_{0,0} * A_{0,0}$ | $B_{0,0} * A_{1,0}$ | $B_{0,1} * A_{0,0}$ | $B_{0,1} * A_{1,0}$ |
| $B_{1,0} * A_{0,1}$ | $B_{1,0} * A_{1,1}$ | $B_{1,1} * A_{0,1}$ | $B_{1,1} * A_{1,1}$ |
| $B_{2,0} * A_{0,2}$ | $B_{2,0} * A_{1,2}$ | $B_{2,1} * A_{0,2}$ | $B_{2,1} * A_{1,2}$ |
| $B_{3,0} * A_{0,3}$ | $B_{3,0} * A_{1,3}$ | $B_{3,1} * A_{0,3}$ | $B_{3,1} * A_{1,3}$ |



Commercial BLAS library



Source: <http://software.intel.com/en-us/articles/intel-mkl>

Per core: achieved about 25 GFLOP/s out of 27.2 GFLOP/s peak



Parallelization



★Now: local and remote accesses

- Distinguish between local accesses m_l and remote accesses m_r
- Remote access much worse than local access

★Serial version:

- Due to memory gap, f/m crucial for overall performance

★Parallel version:

- Hierarchical approach, first optimize m_r , then m_l
- Remote communication more important for overall performance



Parallel MMULT - Parallelization

❖ PCAM: Partition, Communicate, Agglomerate, Map

❖ Partition

- Domain decomposition of C (output matrix)
- Each $C[i,j]$ assigned to one task

❖ Resulting **Communication** pattern depends on distribution of input matrices A,B

1. Copies: no communication except for initialization/completion
 - For large problem sizes not possible (capacity constraints)
 - Huge initialization overhead, $2N^2$ elements to copy
2. Partitioning of A,B according to C: same as unique memory references ($m = 4N^2$), but here m = number of (potential) messages
 - Depends on tiling/blocking

❖ Agglomeration

- Increase computational load per thread/process
- Reduce communication overhead

❖ Mapping

- Easy as communication pattern is rather simple (global communication dominates)



Parallel MMULT - Overlap

Overlap between computation and communication

- For complete matrix copies, dependencies will prevent overlap
 - 3 clearly separated steps: distribute, compute, collect
- For partitioning, fine grain communication allows for overlap:
 - While the first blocks are processed, further distribution of input matrices can already take place

Goal is provide as much overlap as possible

- Latency hiding resp. hiding of communication costs
- Blocking/Tiling



Parallel MMULT – Data distribution

❖ Data distribution primary concern for agglomeration

- 1D partitioning by blocks of rows or columns
- 2D partitioning by both blocks of rows and columns

❖ Optimal choice depends on algorithm

- Locality, communication pattern and dependencies

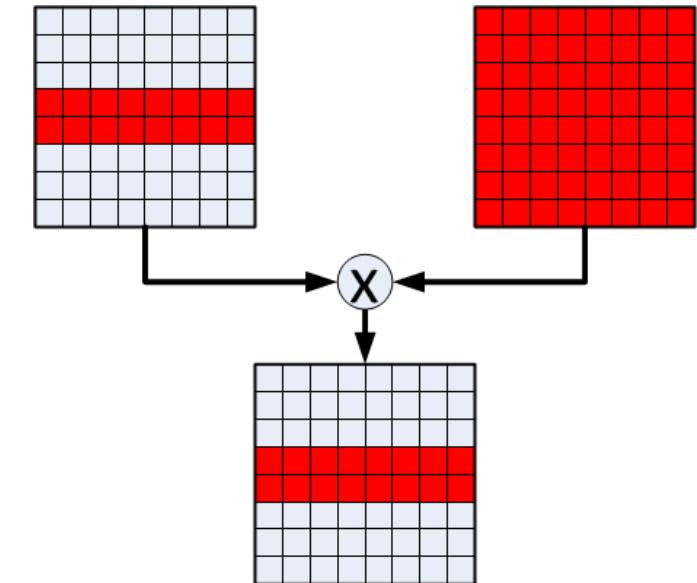
❖ Here: 1D (row and column) and 2D analyzed

❖ In general for MPI: owner computes and handles associated communication



Parallel MMULT – 1D row partitioning

- Data layout: process i owns row(s) $C[i, *]$ (short: $C[i, :]$)
- Owner computes: process i calculates $C[i, :]$
- $C[i, :] = A[i, :] * B[:, :]$
 - B has to be owned completely, but only parts of A and C

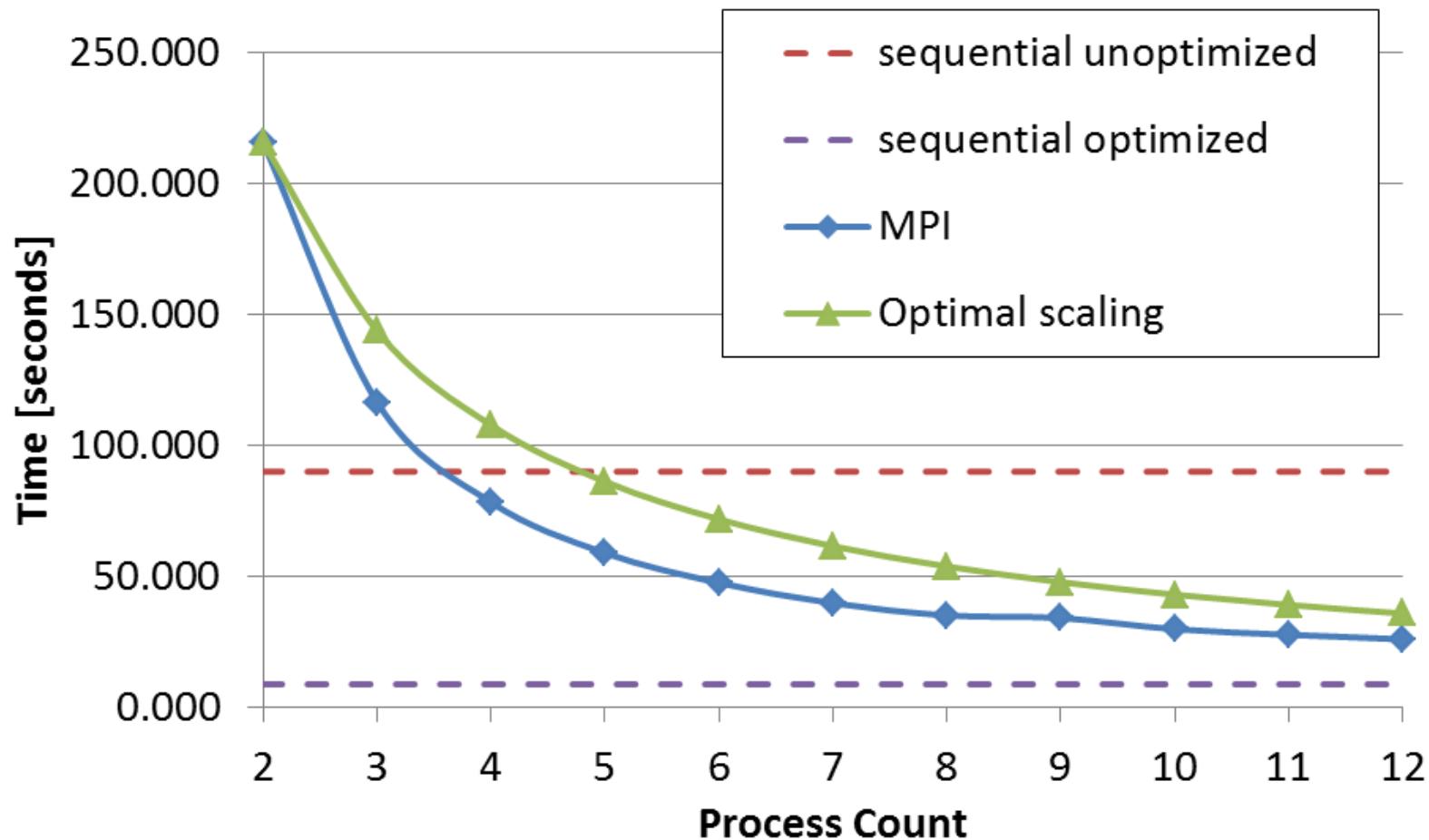




Parallel MMULT – 1D row partitioning

Older experiment!

MPI Matrix Multiply on 6 Dual-Core Nodes





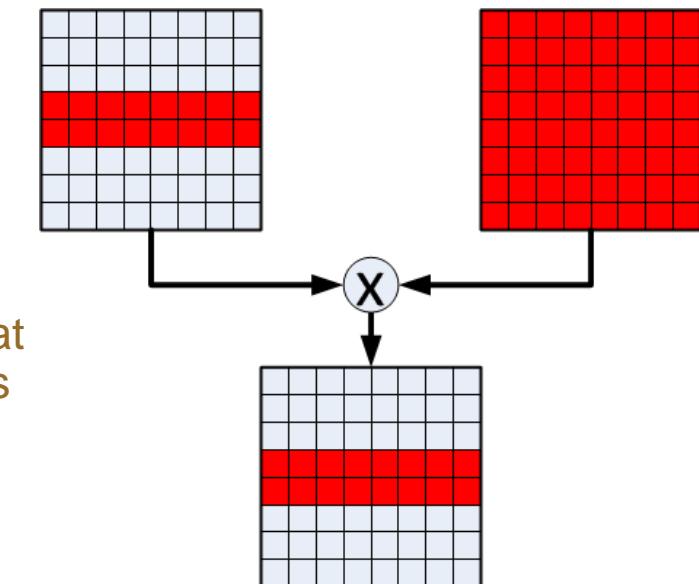
Parallel MMULT – 1D row partitioning

❖ Analysis

- + No communication among workers
- + Nice scalability
- Large memory footprint will sooner or later limit scalability
- Large amounts of memory seldom used, but allocated for complete execution time
- Analogy to caches

❖ More sophisticated partitioning methods

- More messaging overhead
- Basic idea is based on the associativity of the addition
- Operate on blocks (or tiles) of the matrices that you own, then communicate to get new blocks

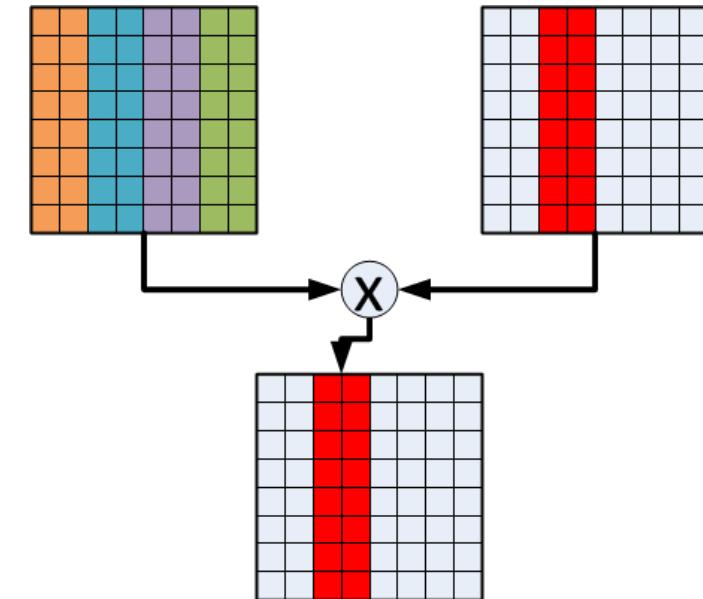




Parallel MMULT – 1D column partitioning

◆ Idea:

1. First, let i compute all it owns:
 - $C[,i] = C[,i] + A[,i] * B[i,]$
 - $i = \text{my_rank}$
2. Then, process i sends $A[,i]$ to process $(i+1) \% p$
 - p is number of processes
3. Repeat for $p-1$ times



```
C[,m] = C[,m] + A[,m]*B[m,]  
S = A[,m]
```

```
For i = 1 to p-1  
    send S to process (m+1)%p  
    recv S from process (m-1)%p  
    // S is now A[, (m-i)%p]
```

```
C[,m] = C[,m] + S*B[(m-i)%p, ]
```



Parallel MMULT – 1D column partitioning

❖ Computation costs per step

- p processes *elements * madd * (vector elements / blocks)*
- Computation costs per step: $f = N^2 * 2 * (N / p) = 2N^3/p$

❖ Timing model for communication

- Communication follows a linear model: $t(s) = t_{lat} + s * t_{BW}$
- s : size (number of elements), t_{lat} : constant overhead in time, t_{BW} : time to transfer one element

❖ Communication costs per step

- $s = N^2/p$ (size of A divided by number of processes)

❖ Total costs:

- $c = f + (p-1) * (f+t(s))$
 - Initial step compute, then $(p-1)$ times repeating
- $c = f * p + (p-1) * t_{lat} + (p-1) * N^2/p * t_{BW}$



Parallel MMULT – 1D column partitioning

❖ Total costs $c_{\text{parallel}} = 2N^3/p + (p-1)*t_{\text{lat}} + (1-1/p)*N^2 * t_{\text{BW}}$

- $t_{\text{new}} = c_{\text{parallel}}$; $t_{\text{old}} = f = 2N^3$

❖ Comments:

- Term 1: Perfect scalability ($O(1/p)$) if no communication costs, resp. no communication
- Term 2+3: Overhead due to parallelization

❖ Scaling the problem size

- $c_{\text{parallel}} = O(N^3/p)$
- $N \rightarrow \infty$: SU $\rightarrow ???$



Parallel MMULT – 1D column partitioning

→ Total costs $c_{\text{parallel}} = 2N^3/p + (p-1)*t_{\text{lat}} + (1-1/p)*N^2 * t_{\text{BW}}$



→ Scalability

- Computation $O(N^3)$
- Communication $O(N^2)$

→ As long as communication costs grow slower than computation costs, system is considered scalable

- One of the strictest definitions
- Rarely used because difficult to verify



Parallel MMULT – 1D column partitioning

❖ Parallel MMULT – 1D column partitioning

- Initialization: distribute $A[:,i]$ to process i , $i=1..P$, broadcast B
 - MPI_SCATTER
 - MPI_BCAST
- During each step: sending and receiving of blocks
 - MPI_SEND & MPI_RECV
 - MPI_SENDRECV
- Final result: gather $C[:,i]$ from process i , $i=1..P$
 - MPI_GATHER

❖ Collective operations can help to simplify a program, like library calls

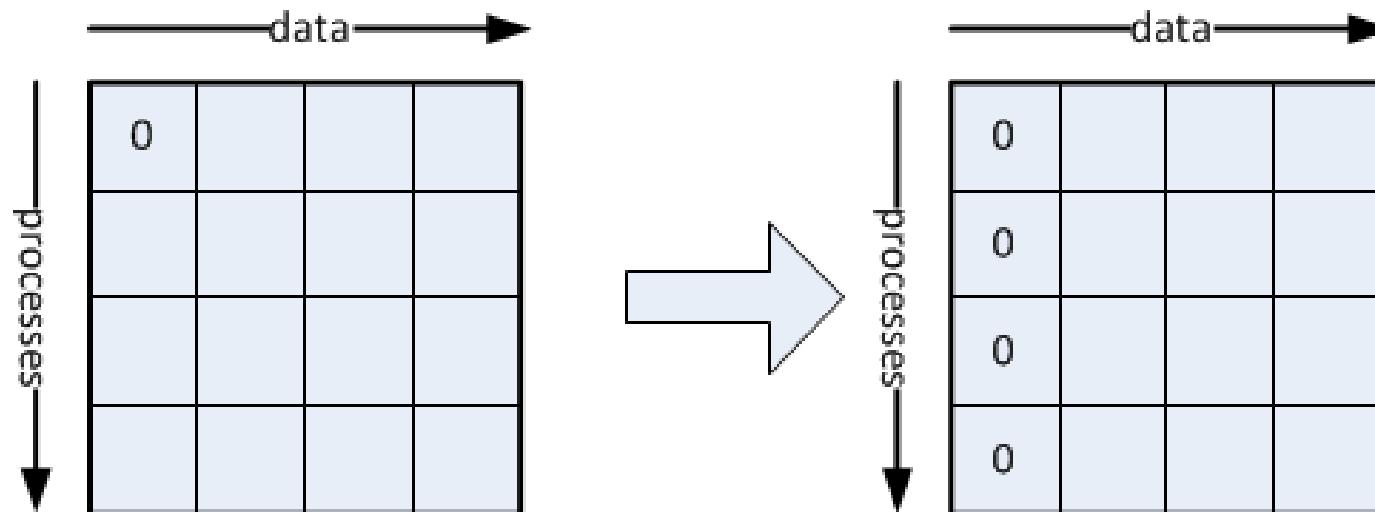
- Every involved process has to make the function call



Parallel MMULT – 1D column partitioning

→MPI_BCAST

- MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- As if process *root* executes n send operations to n destinations

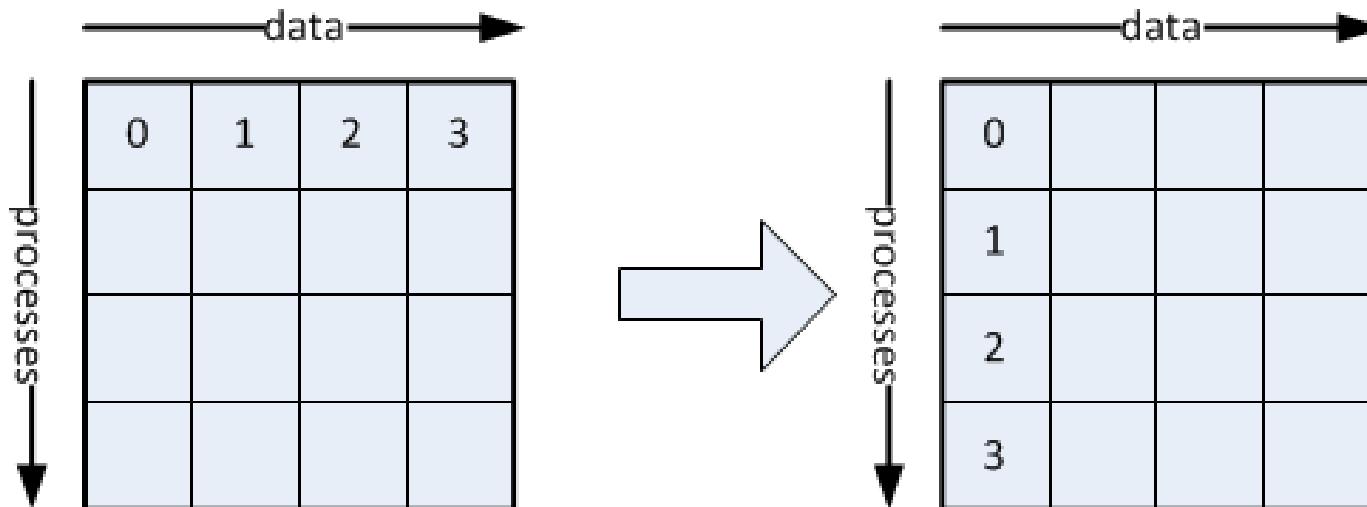




Parallel MMULT – 1D column partitioning

→MPI_SCATTER

- MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- As if process *root* executes n send operations to n destinations
- See also MPI_Scatterv - various

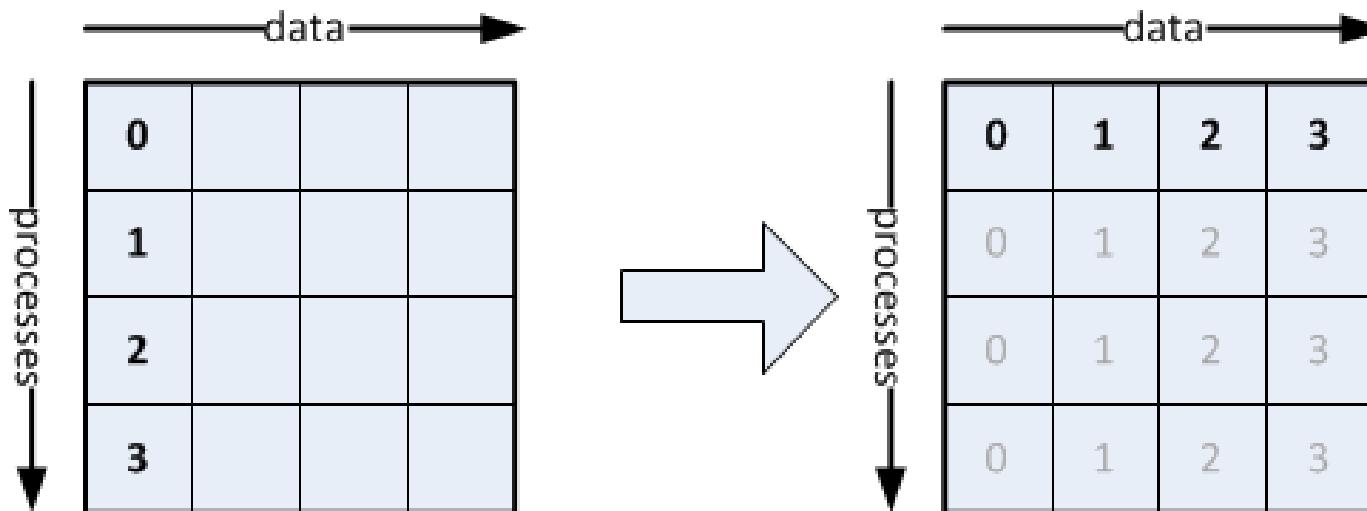




Parallel MMULT – 1D column partitioning

♦MPI_GATHER

- MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- As if process *root* makes n receives from n sources
- See also MPI_Gatherv – various, MPI_Allgather (shown in gray)





Notes on collective operations

- ❖ Plenty of them, to cover all the various patterns
- ❖ Like libraries, others are responsible for optimization
 - E.g., for different topologies
- ❖ Communication cost: between $O(n)$ and $O(n^2)$
 - n =process count
- ❖ Dedicated hardware support for multicasts
 - Only found in advanced communication devices
 - BlueGene, Cray, EXTOLL
 - Typically not covered in specifications like Ethernet or Infiniband
- ❖ Multicast != Ethernet broadcast
 - Ethernet broadcasts are not reliable
- ❖ Currently collectives are always blocking (preventing overlap)
 - See MPI 3.0 for more advanced solutions



Parallel MMULT – 2D partitioning

♦ 1D partitioning

- Maps nicely to ring, also to mesh/tori/hypercube but does not utilize all network resources
- Still huge memory requirements (N^2/P)
- $P \sim N^2$ ⑨ scaling P with N is difficult
- $P \sim N$ would be better

♦ Assume $P=s^2$, s whole number

♦ Then each process computes a block of C:

- For all i,j of block(C); for $k = 0..(s-1)$
 - $C[i,j] = C[i,j] + \sum (A [i, k] * B [k, j])$
- Rotate blocks of A & B circularly

♦ Owner computes, optimize locality



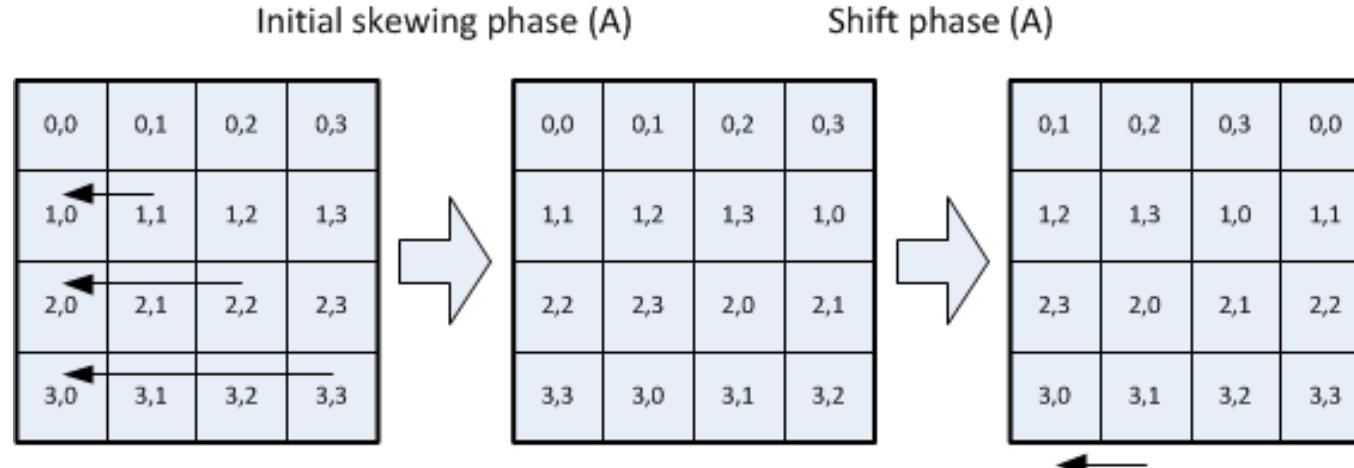
Parallel MMULT – 2D partitioning

❖ Three phases:

1. Initial skewing phase: left shift circular row i of A by i times to the left, up shift circular column j of B by j times
 - For all i : $A[i,j] = A[(i+1)\%s,j]$, B accordingly
2. Computation
3. Shift: {left,up} shift circular of each {row,column} of { A,B } by one

❖ In-place algorithm

❖ Communication costs now by factor \sqrt{p} smaller





Parallel MMULT – 2D partitioning

❖ Broadcast-Multiply-Roll (BMR)

- Using a broadcast instead of the initial skewing process
- Three phases: broadcast, multiply, roll
- Algorithm is not in-place

❖ SUMMA: Scalable Universal Matrix Multiplication Algorithm

- Highly general algorithm
- B small: less memory requirement, lower efficiency; B large: high memory requirement, higher efficiency

❖ Many more

- Recursive layouts



Summary

❖ PCAM used for the MMULT example

- P: easy as domain decomposition is applied
- C: trivial (copies), difficult (1D/2D partitioning)
- A: trivial (copies), difficult (1D/2D partitioning)
- M: easy

❖ Data handling (initial distribution & later movement) is most difficult when writing MPI programs

❖ Sequential versions can help to verify correctness

❖ Collective MPI calls can make the program easier, both for the initial coder and later readers

- Topology agnostic



Introduction to High Performance Computing

Lecture 06 – Messaging

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg



Review and Background

- **Distributed exclusive address spaces**
 - A pointer valid for one process is not valid for another process
 - Hardware running each process can be completely different
 - 32bit vs. 64bit
 - X86 vs. IA64/ARM/SPARC/etc
 - Big-endian vs. Little-endian
- **MPI has to ensure that appropriate conversion takes place**
 - Such heterogenous systems rather seldom today...
 - GPUs are different
- **Finally, in MPI, it's all about copying data...**



Review and Background

- Identifying copies = Tag matching
- Applicable for send/receive messaging scheme
 - Opposed to the Put/Get model
- Tag matching
 - If receive is pre-posted, search list of posted receives
 - If receive is not posted yet, copy message to unexpected queue and upon receive search this queue
- In any case, tag matching = $O(N)$, N number of buffer entries
 - One of the biggest sources of overhead in message passing
 - Besides progress (see later)



Review and Background

- Up to now: Blocking/non-blocking send/receive
 - Returning from the function call guarantees that corresponding operation is complete
 - I.e., message has been sent or received (what about both?)
 - Real life examples
- Messaging in MPI is a little bit more complicated than you might expect
 - Example: P0 sends 16 data words to P2
 - P0 has to tell MPI: destination, count, type, tag (label)
 - P2 has to tell MPI: source, count, type, tag
 - P0 has to use a certain *communication mode* or *send mode*
- What is guaranteed after returning?
- Who is copying from where to where?



Terminology

■ Latency

- Overhead associated with sending a zero-size message
- Between at least two MPI processes
- Hard- and software components, ratio highly implementation dependent
- HPC: usually measured in micro-seconds

■ Bandwidth

- Data rate which can be transmitted between two MPI processes
- Hard- and software components, ratio highly implementation dependent
- Usually measured in (mega-)bytes per second (MB/s)

■ Synchronous/asynchronous communication

- A completing synchronous call guarantees that data has been successfully received by receiving process
- An asynchronous call can return anytime, without any guarantees about the state of the receiving process



Terminology

■ User or application buffer

- User-level address space that holds data to be sent or received
- Passed as pointer to the MPI send/receive calls
- Resides in virtual user-level address space

■ System buffer

- System address space for temporarily storing messages
- Not visible to user/programmer
- Key for asynchronous communication
- **Can** also reside in system-level address space

■ Message envelope

- Meta data like source, destination, tag, size, type, communicator, ... (implementation dependent)
- Messages consists of an envelope and a data (or payload) part



MPI Communication Modes

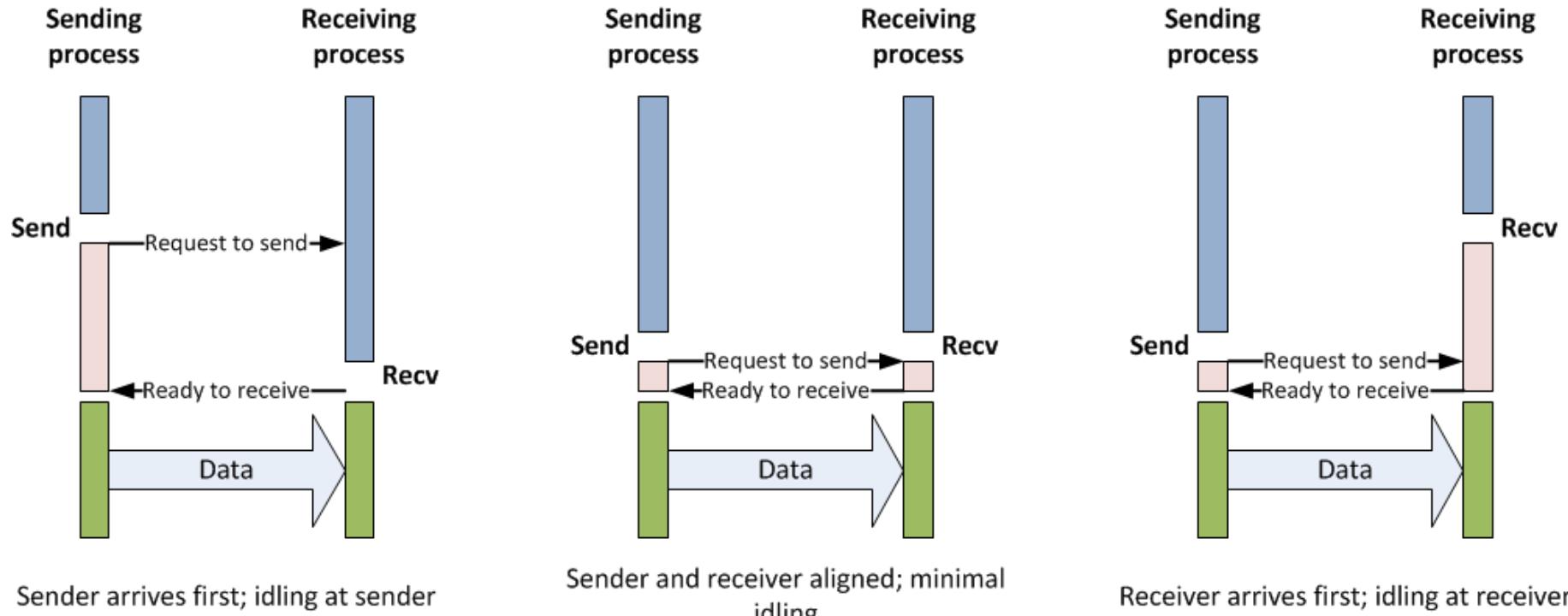


MPI Communication Modes

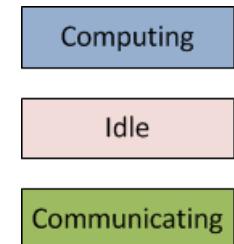
- MPI standard defines four send modes:
 - Standard: MPI_Send
 - Synchronous: MPI_Ssend
 - Buffered: MPI_Bsend
 - Ready: MPI_Rsend
- All in combination with blocking/non-blocking
 - MPI_{I,S,B,R}send
- Receives: only blocking/non-blocking, no other types!
 - Send call determines communication mode



Non-Buffered Blocking Send



- Handshake protocol for non-buffered blocking send/receive
 - „Rendezvous“
- Overheads due to idle times
 - Latency: time required to send a message from A to B





Non-Buffered Blocking Send

- In general: *return only when it's safe to do so*

1. Non-buffered blocking send

- Returns if the matching receive has been encountered
- Major issues: idle times and deadlocks
- Provides synchronization between sender & receiver
- **Non-local**: completion may depend on matching receive

2. Buffered blocking send

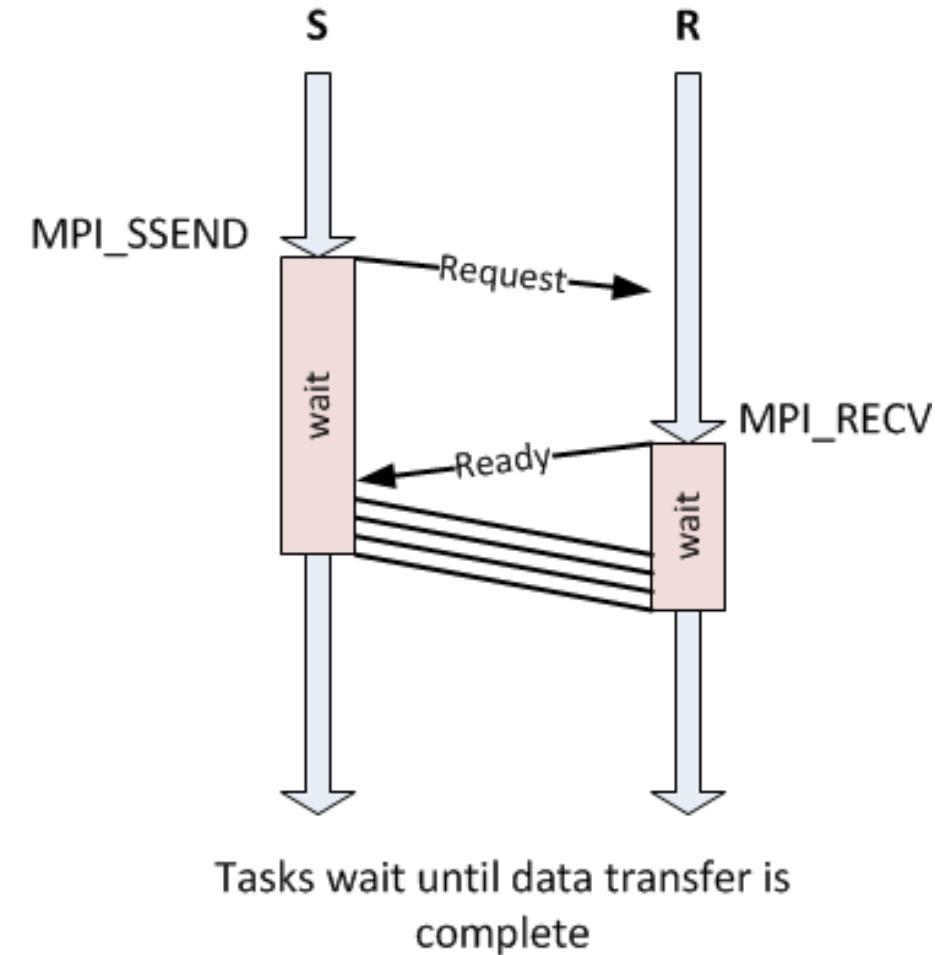
- Returns if the buffer can be used again
 - I.e., buffer has been copied internally
- Reduces idle times, additional costs for copying
- Improved decoupling between sender & receiver
- **Local**: completion does not depend on matching receive

- In any case, after returning the buffer can be reused



Blocking Synchronous Send

- Communication mode is selected while invoking the send routine
- Synchronization overhead at sender
 - Wait for receive to be executed and handshake to arrive
 - Then, transfer message
- Synchronization overhead at receiver
 - Wait for handshake to complete
- Overhead incurred while copying from buffer to the network
- **Non-local, no buffers required**



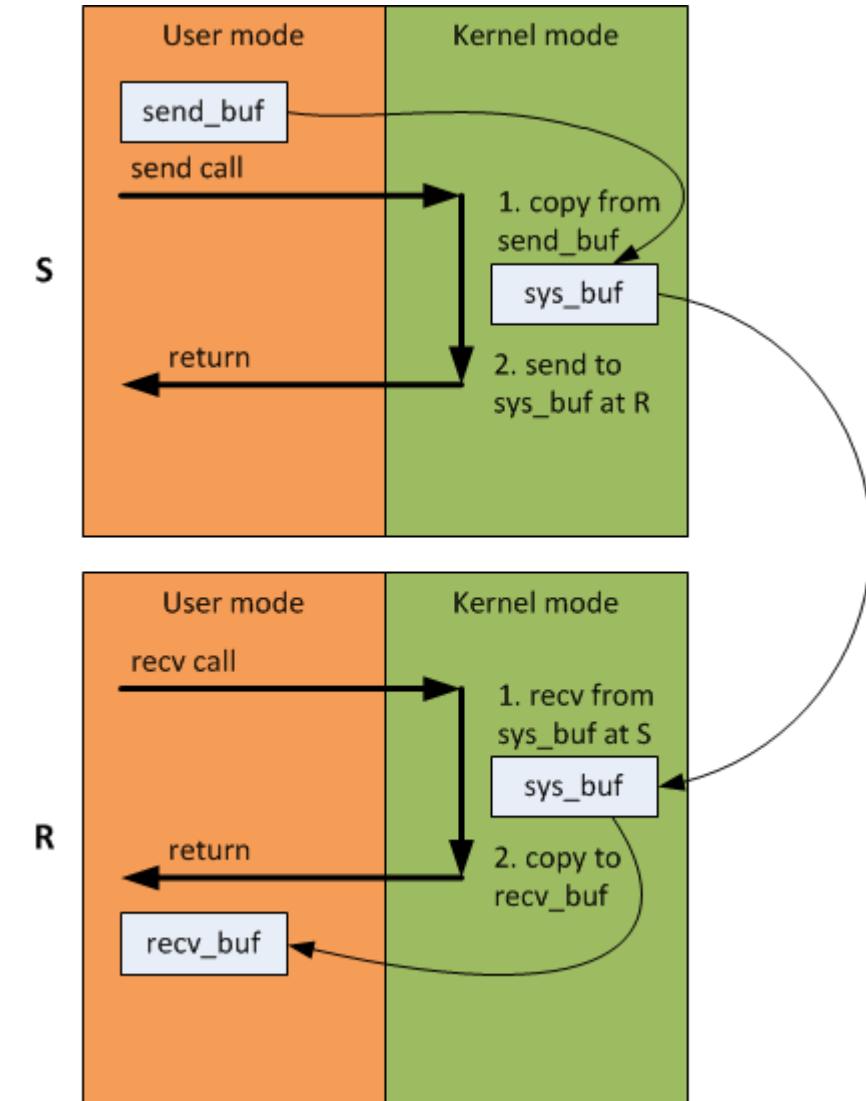


Basic Buffering Concept

1. Sending process copies data to be sent from user buffer to system buffer
2. Data is sent over the network into remote system buffer
3. Receiving process copies data from system buffer to user buffer

Notes

- Return on sender side depends on selected communication mode
- Special communication hardware may provide additional buffers





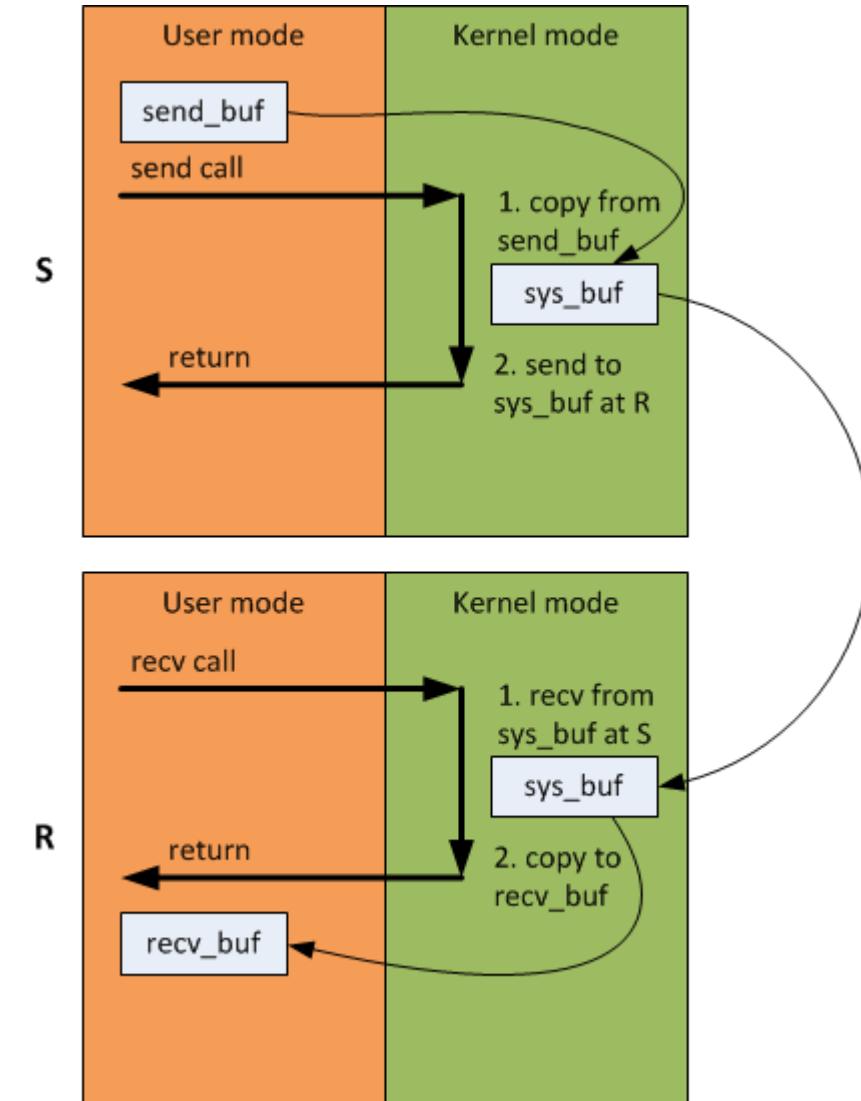
Basic Buffering Concept

▪ Without communication hardware buffers

- Interrupts required
- CPU on-loading
- Example: Ethernet (plain)

▪ With communication hardware buffers

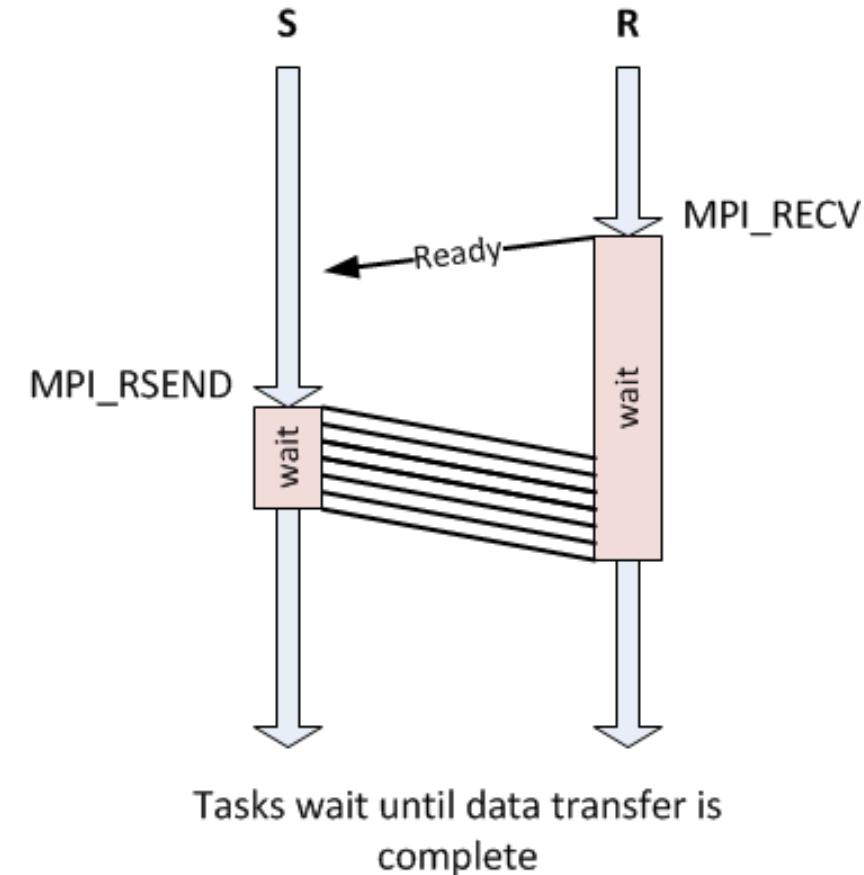
- Minimized overhead
- CPU off-loading
- Example: Infiniband





Blocking Ready Send

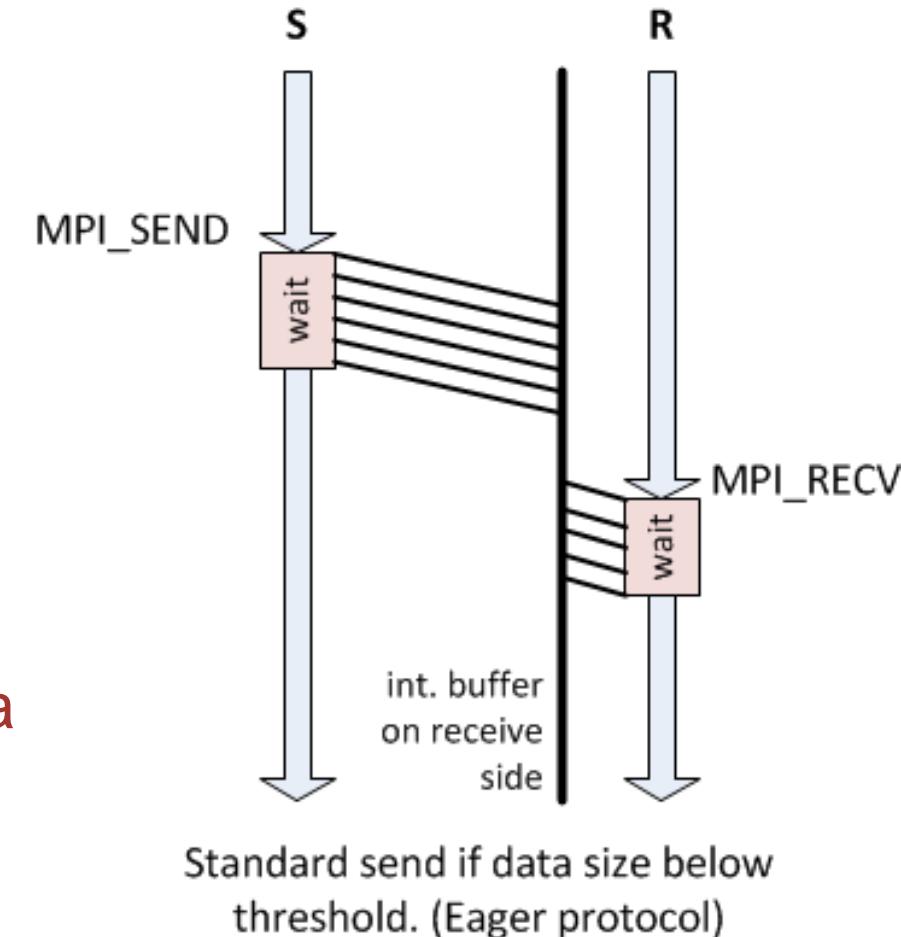
- Ready Send expects pre-posted receive
 - If not, an error will incur and send returns
 - Programmer's responsibility to handle errors
- Overhead on sender side minimized
- Receive side still can incur significant overhead
 - “No free lunch”





Blocking Standard Send – Eager

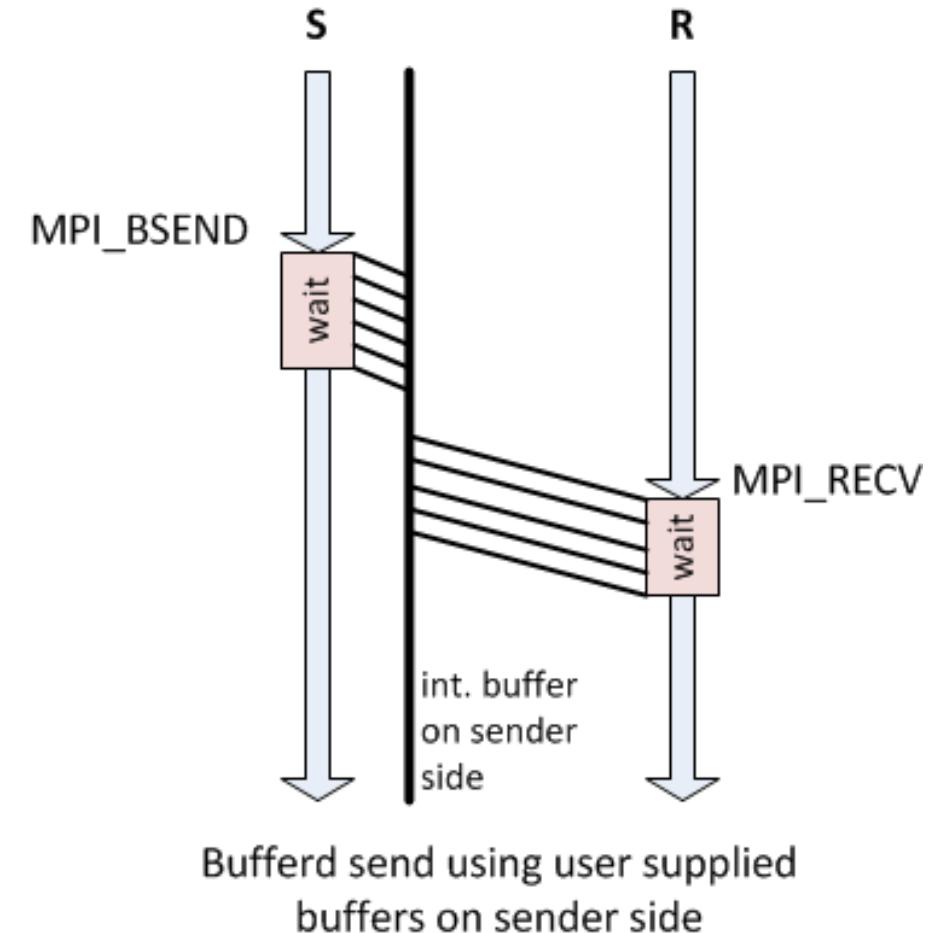
- Standard send is implementation dependent
 - Only typical aspects covered here...
- Typically it includes at least one threshold value
 - If data size is below threshold (small messages), it is copied into a buffer on the receiver side
- Receive call copies the data from system buffer to user buffer





Blocking Buffered Send

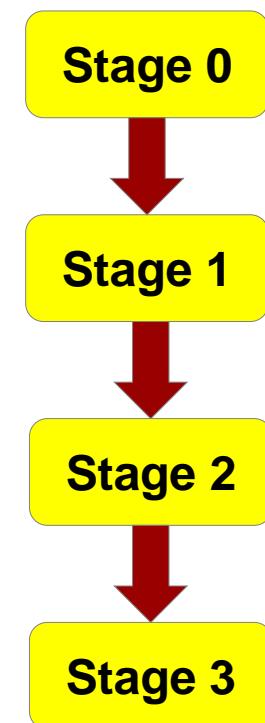
- User supplied buffers on sender side
- Message buffer can be immediately reused
- Low (time) overhead on sender side, large buffering requirements





Note on Bounded Buffers

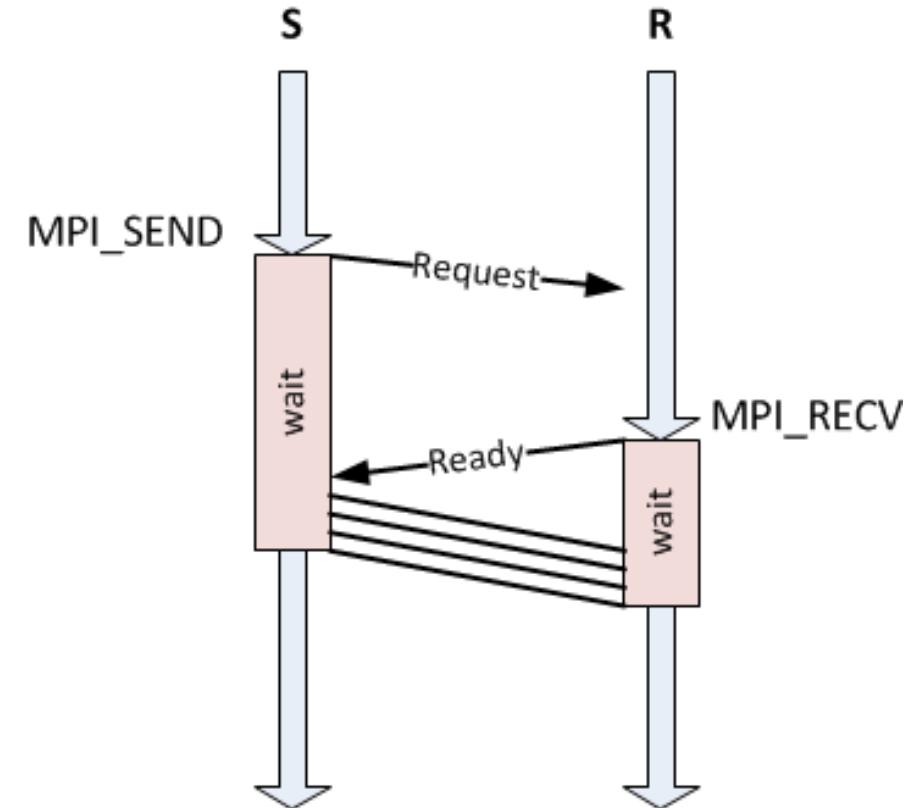
- Severe performance degradation can happen
 - Producer/Consumer with one being slower
 - Buffers are – independently of their size – always full or empty
- Slowest part of a chain limits overall performance
- Independent how large buffers are, there is always an upper bound
- Lower bound: at least one maximum sized message
- Buffer exhaustion or overflow can lead to program failure or stalling
 - Buffering is implementation-dependent





Blocking Standard Send - Rendezvous

- Again: standard send is implementation dependent
- If data size exceeds threshold: **synchronous send**
 - Rendezvous protocol
- Unnecessary copying avoided
- Longer idle times



Standard send if data size above threshold. (Rendezvous protocol)



Eager vs. Rendezvous

| | Eager | Rendezvous |
|---------------|--|---|
| Send call | <ul style="list-style-type: none">• Sender assumes that receiver can store the message | <ul style="list-style-type: none">• No assumptions are made about receiver – send out request containing envelope |
| Receive call | <ul style="list-style-type: none">• System buffers at receiver side• Receiver decides how to handle message | <ul style="list-style-type: none">• Interpret request, search for buffer space (user- and/or system-level)• Send back response |
| Process count | <ul style="list-style-type: none">• Severely limits buffer space | <ul style="list-style-type: none">• Implies no limitations |
| Advantages | <ul style="list-style-type: none">• Reduces synchronization delay, no rendezvous | <ul style="list-style-type: none">• Scalable• No additional copying |
| Disadvantages | <ul style="list-style-type: none">• Not scalable• Additional copies• Buffer space over-provisioned• Behaviour in the case of buffer exhaustion? | <ul style="list-style-type: none">• Inherent synchronization due to handshaking• Best use for non-blocking sends - more programming complexity |
| Primary use | Small messages ($< n$ kB) | Large messages ($\geq n$ kB) |



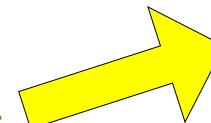
Deadlock issues

- Additional buffers cannot avoid deadlocks!



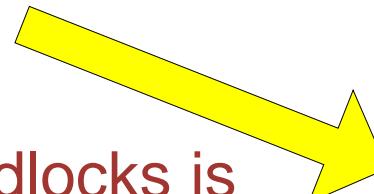
| <u>P0</u> | <u>P1</u> |
|--------------|--------------|
| RECV (P1, B) | RECV (P0, A) |
| SEND (P1, A) | SEND (P1, B) |

- Blocking calls are more likely to deadlock



| <u>P0</u> | <u>P1</u> |
|--------------|--------------|
| SEND (P1, A) | RECV (P0, B) |
| SEND (P1, B) | RECV (P0, A) |

- Solutions:
 - Non-blocking ISENDs
 - Reordering



| <u>P0</u> | <u>P1</u> |
|--------------|--------------|
| SEND (P1, A) | RECV (P0, A) |
| RECV (P1, B) | SEND (P1, B) |

- Producing deadlocks is much easier than one might think



Notes on Blocking Send/Recv

- Blocking: **MPI_SEND & MPI_RECV**
- Blocking in the sense, that the function call won't complete until data has been completely copied from or to message buffer
 - **Send:** nothing is guaranteed about receiving!
 - Obviously a blocking **receive** also implies a corresponding send
- If **MPI_SEND** or **MPI_RECV** finish, then the message buffer is freed respectively valid
 - Nothing is guaranteed for non-blocking calls
 - Use wait or test operation to update corresponding **MPI_STATUS**



More Notes on Blocking Send/Recv

- Beside the high deadlock potential, there are also performance issues
- Make use of overlap between computation and communication!
 - Non-blocking calls as soon as user buffers are ready
 - Use time between first call and following wait for other useful work
 - Computation (pipelined fashion)
 - Other messaging
 - I/O, ...
- Even more beneficial if special communication hardware support exists
 - Off-loading, e.g. DMA engines in the networking device



Notes on non-blocking Send/Recv

- Standard non-blocking send, below threshold (eager):
 - Sender side
 - Eventually: wait until buffer space at receiver available
 - Send & let receiver decide how to handle message
 - Receiver side
 - Copy to user buffer (if suitable receive already posted)
 - Copy to system buffer (otherwise)
 - **Non-local**
- Standard non-blocking send, above threshold (rendezvous):
 - Sender side
 - Enqueue request in local request queue & return
 - **Local**



Progress

- Who guarantees that non-blocking operations are further processed?
- Completion of non-blocking operations might depend on subsequent calls
 - Implementation dependent
- Non-blocking sends typically only enqueue requests
- Non-blocking receives typically also enqueue requests, but also call internally the **progress function**
- All blocking operations call the progress function



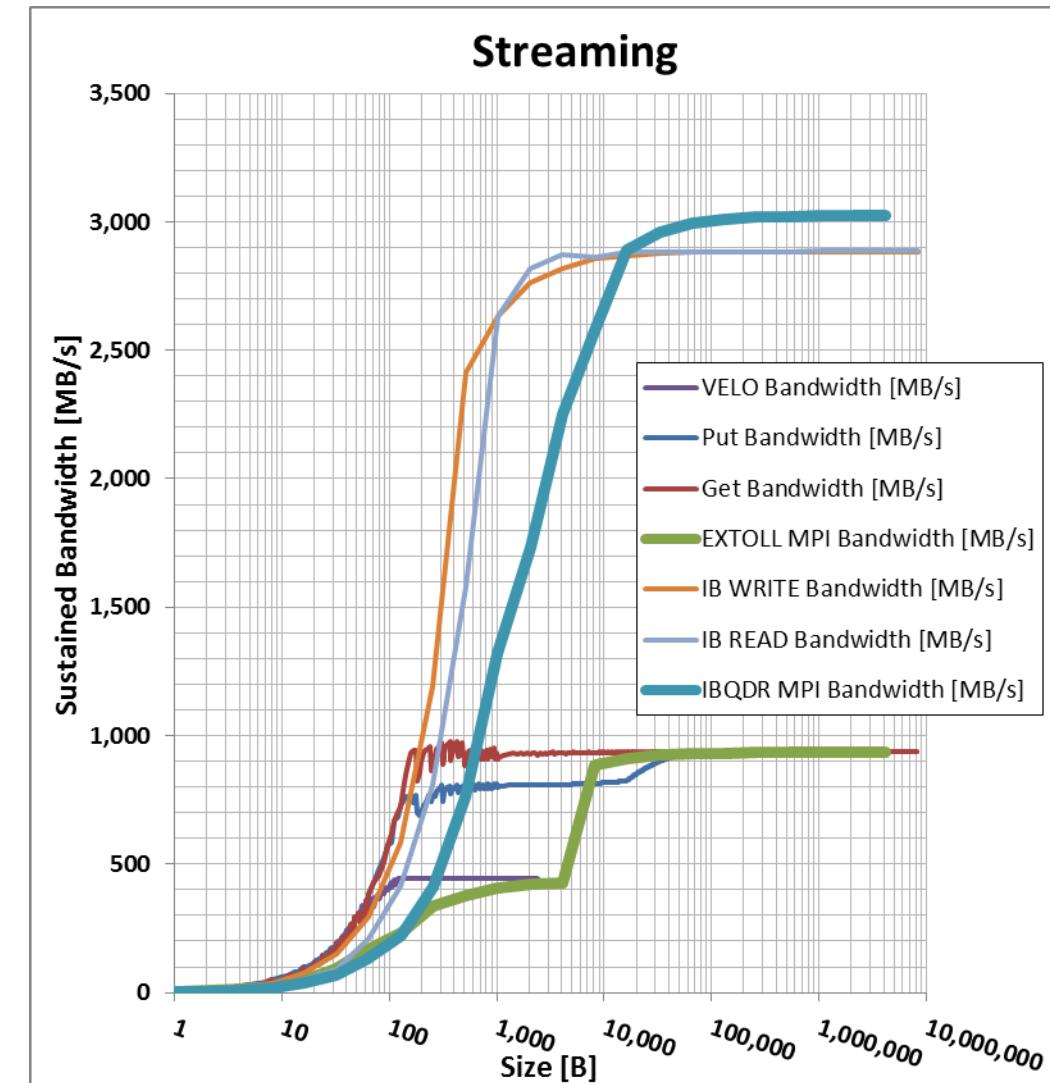
Progress

- Internal request queue (or ring buffer)
 - For efficient decoupling between computation and communication
 - Case of exhaustion is implementation dependent
- Messages have to be processed in-order
 - In the queue there might be older requests with identical criteria (source, tag)
 - Always enqueue, even if a matching send/receive request is available
- Progress is one of the biggest overhead sources in MPI
 - Besides additional copying



MPI Overhead

- Many guarantees
 - Send/receive semantics
- Huge overhead
 - Tag matching
 - Copying
 - Progress
- CPU2CPU example
 - Simple streaming test
 - EXTOLL R2 FPGA & OpenMPI
 - IBQDR & MVAPICH



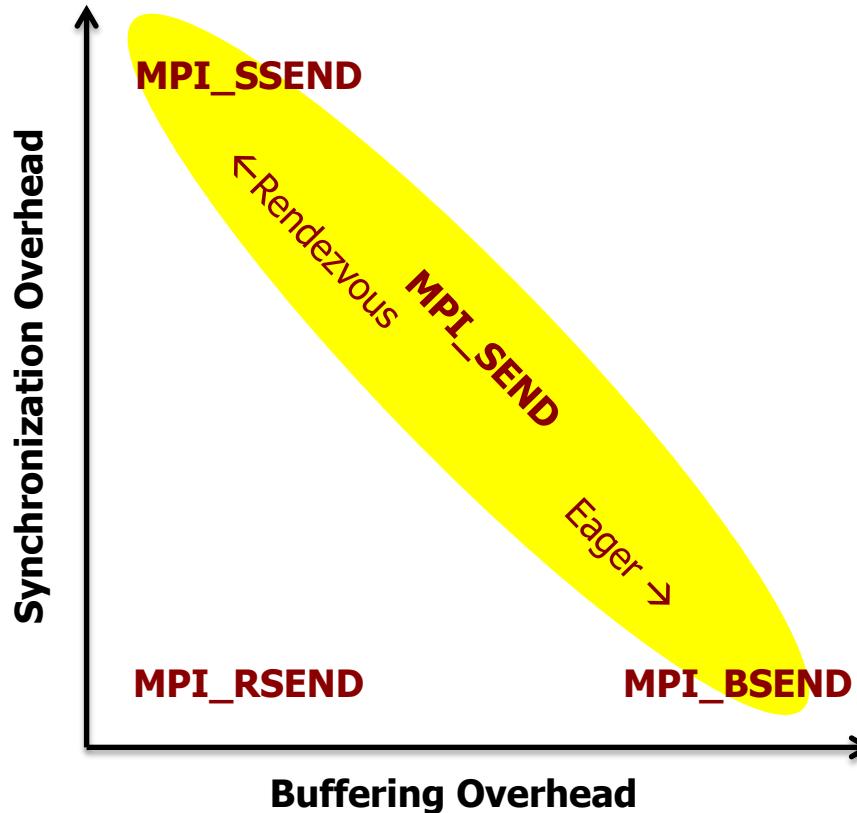


Summary

- MPI send call determines communication mode
 - Standard, Buffered, Synchronous, Ready
- Blocking/non-blocking independent of this
 - Also applies to receive calls
- Typical functionality within communication
 - Rendezvous-protocol
 - Eager-protocol
- Trade-offs between copying and long idle times
 - Both are overhead sources and limit performance
 - Copying is data movement, data movement is extremely expensive
 - So are long idle times ☹
- Many aspects are implementation dependent
 - Allows to optimize for system-specific properties and characteristics



Summary



Communication Modes

1. **MPI_SSEND?**
2. **MPI_RSEND?**
3. **MPI_BSEND?**
4. **MPI_SEND?**



Too difficult?

A short excerpt

“We are probably the 1st generation of message-passing programmers.“

“We are also probably the last generation of message-passing programmers.“

[MPI – The complete reference, Vol. 2, The MPI Extensions, 1998]



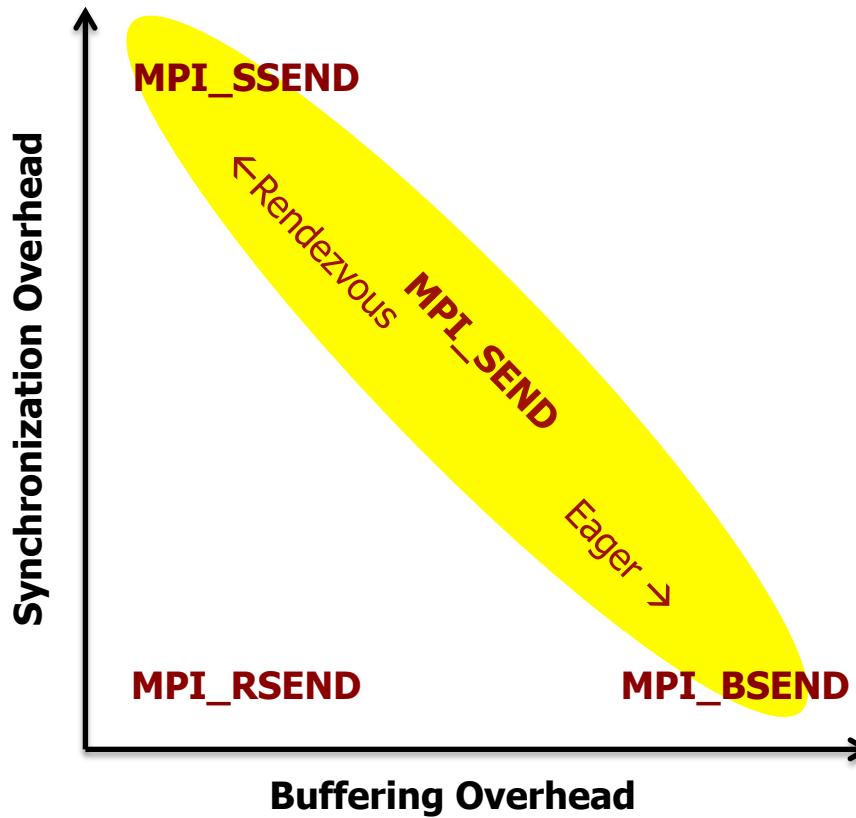
Introduction to High Performance Computing

Lecture 07 – Characteristics

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg



Introduction



- Many ways to send a message
 - Goal: As few overhead as possible
 - Or: as much overlap as possible
- According to Amdahl
 - For scalability, communication must not contribute to the serial fraction
 - E.g., 10% of the execution time for communication, max. 10x SU
- Thus: **large overlap, low overhead** required for scalable parallel computing



Introduction

- So, what can we expect does happen if we perform non-blocking send/recv operations?
 - It all depends on the **network** respectively **network interface**...
 - Hard- and software components contribute to the overhead associated with message passing
 - Usually, SW is responsible to handle HW shortcomings
 - Examples: reliability, ordering, copy operations, ...
- Goal of this lecture
 - The most important performance characteristics of message passing (resp. parallel computing)
 - Their importance for different applications



Latency and Bandwidth Characteristics



Analogy to Shared-Memory Computers

- Standard memory access is characterized using access latency and transfer bandwidth
 - About which of both do you worry more?
- Remember caching effects
 - Caches are used to reduce average access latencies!
 - Leveraging spatial and temporal locality
- Caching for message passing systems?
 - Spatial and temporal locality?
- What is most important for message passing systems?



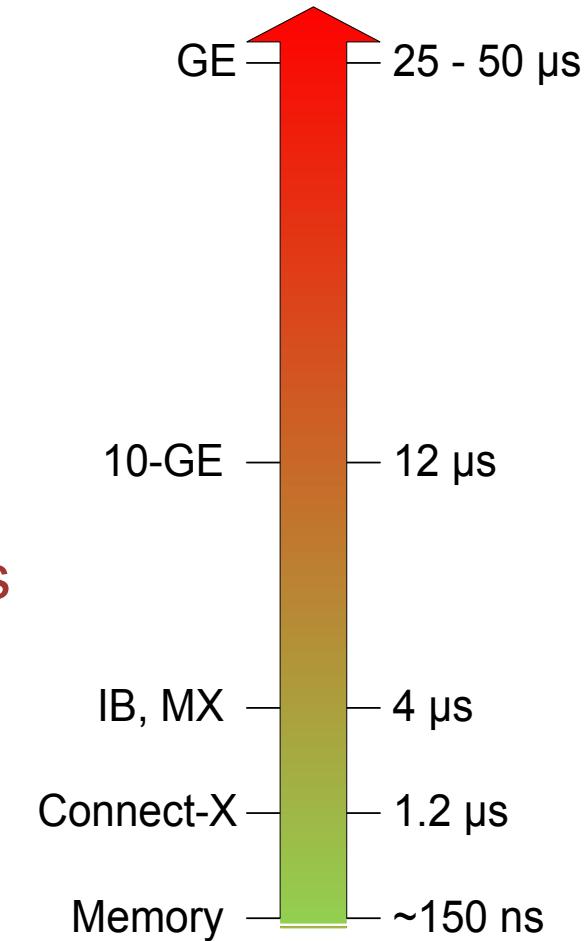
Latency

- Latency is the time between starting a send and completing a receive
 - Typically in micro-seconds (usec)
 - Tends to vary widely across architectures
 - Software latency vs. hardware latency
 - Usually people care about the first one
 - Hardware latency about 10-100 times lower
- Diameter of a network
 - For a pair of two nodes with the highest distance between them, the diameter is the number of hops of the shortest path
- Latency is important for programs with many small messages



Latency

- Patterson stated: “*Latency lags Bandwidth*” (CACM 2004)
 - Bandwidth improves much more quickly than latency: memory, storage, networking
- Every component in between sender and receiver contributes to latency
 - Buffering, queues, pipeline stages, ...
- Also, remember that the speed of light is limited
 - Vacuum: $c_0=300\text{m/usec}$ or 3ns/m
 - PCB (FR4): typ. $c = c_0/2.0$ or about 6ns/m
 - Optical fiber: typ. $c = c_0/1.5$ or about 4.5ns/m



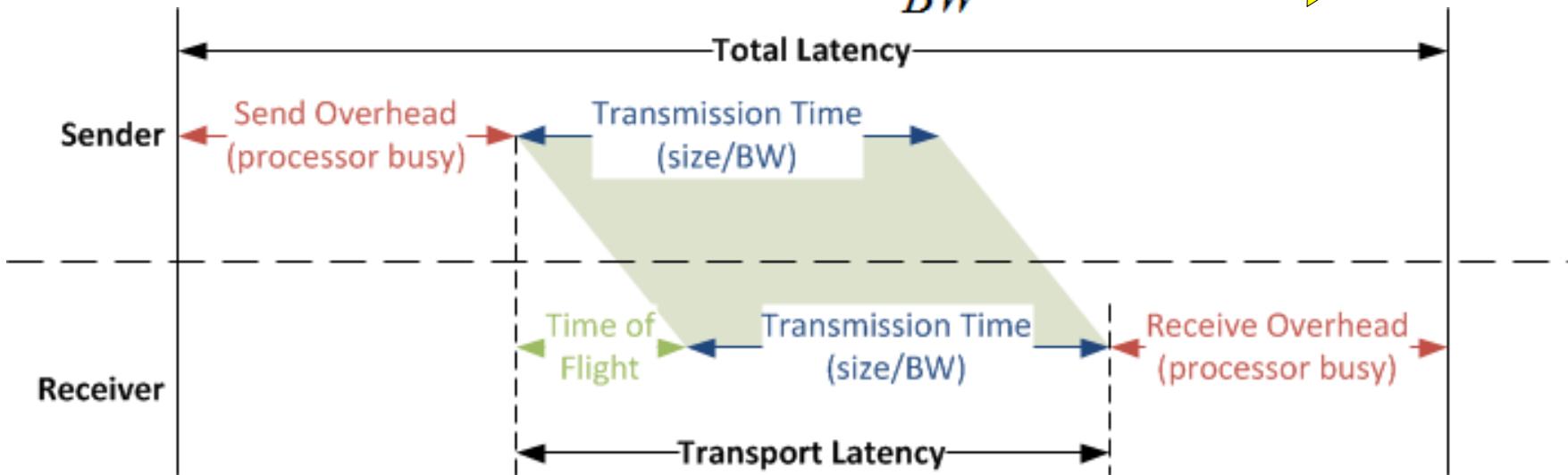


Latency

- Several components contribute to total latency
 - Simplified model here
- Start-up latency: latency of a minimum sized message
 - Zero-sized message
 - Good indicator for overhead

$$t_{lat} = t_{send} + t_{flight} + \frac{size}{BW} + t_{recv}$$

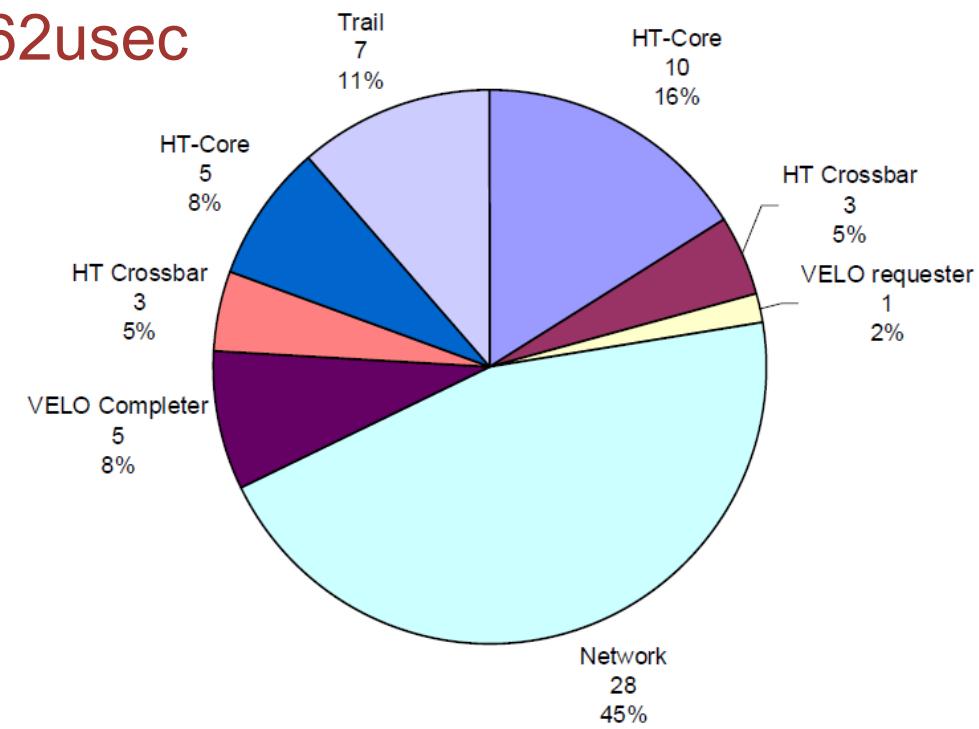
When is the processor free for other tasks?





Latency

- VELO Example
- 62 cycles at 100MHz: 0.62usec
- API latency: 0.97usec
 - Diff = CPU/MC/SW overhead
 - ~0.35usec
- MPI adds 0.2-0.5 usec
 - MTL resp. BTL



Heiner Litz, Holger Fröning, Mondrian Nüssle, Ulrich Brüning, VELO: A Novel Communication Engine for Ultra-low Latency Message Transfers, *37th International Conference on Parallel Processing (ICPP-08)*, Sept. 08 - 12, 2008, Portland, Oregon, USA.



Bandwidth

- Physical or peak bandwidth $BW = \text{data width} / \text{cycle time}$
 - Cycle time = $1 / \text{frequency}$
 - Applies for a network link, internal data path, PCIe subsystem, ...
- Unidirectional vs. bidirectional bandwidth
 - Some old topologies (buses) did not allow bidirectional transfers
 - Not of importance today (except marketing)
- Effective or sustained bandwidth typically lower
 - Protocol overhead
- Bandwidth is maybe the most important characteristic today (Big Data era)
 - Processor/Memory gap → Processor/Network gap

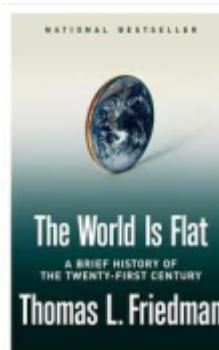
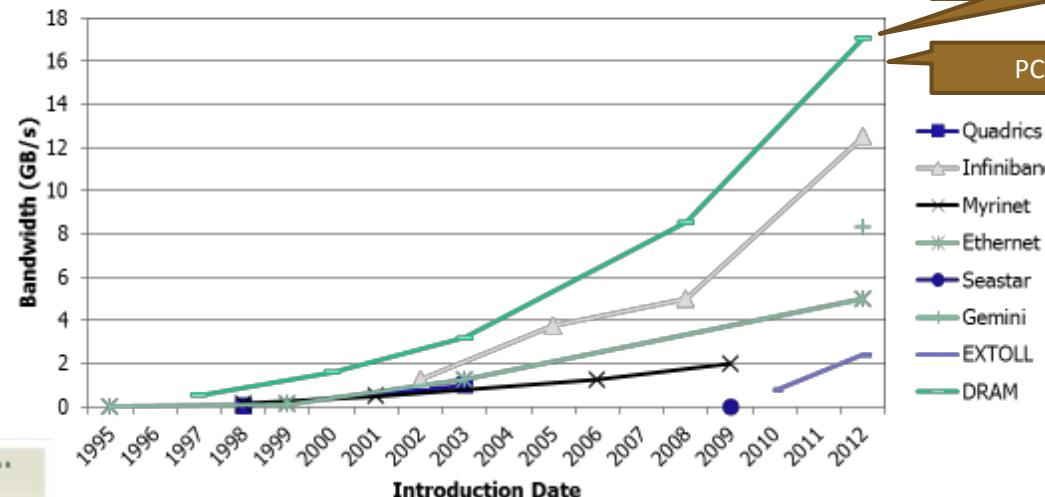


Bandwidth – Current Trends



Future – A Flat World?

**Bandwidth vs. Time for Common
Interconnects**



Similar applies for latency!

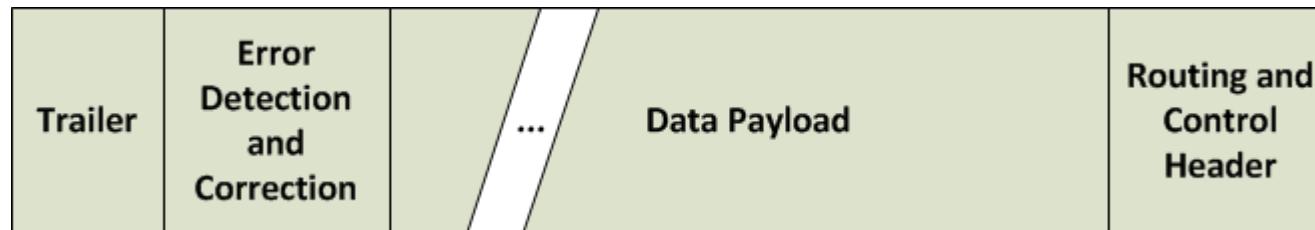
„Everyone is getting closer and we need better sharing [...]“
Sudha Yalamanchili, UCAA Workshop 2012



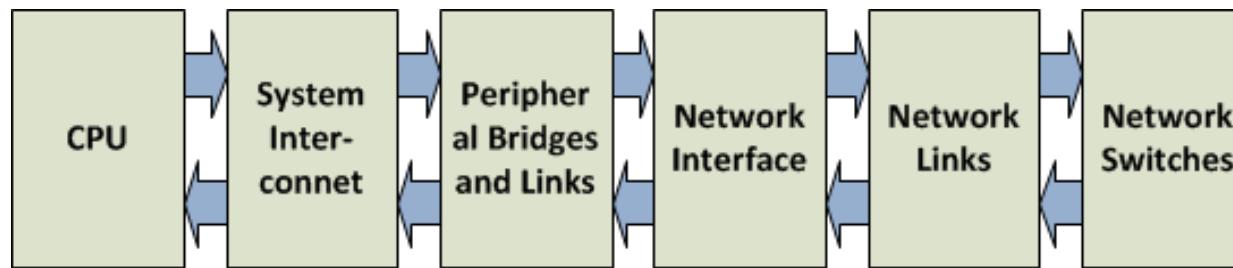
Bandwidth

■ Protocol overhead

- Payload size / Packet size ratio



■ Overall bandwidth limited by individual subsystem bandwidth





Latency Bandwidth Analysis

- Intel MPI Benchmarks (IMB)
 - Easy to use, free benchmark suite
- Benchmarks a large set of MPI functions
 - Point-to-point message passing
 - Single transfer & parallel transfer
 - Global data movement and computation routines
 - Collective
 - One-sided communications & File I/O
- PingPong & PingPing
 - Start-up latency & peak bandwidth
 - No difference for good MPI implementations

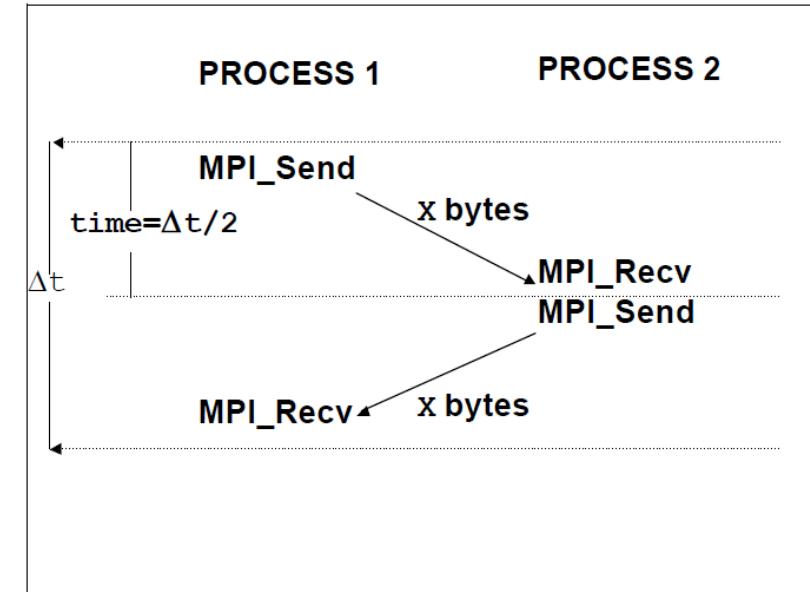


Figure 1: PingPong pattern

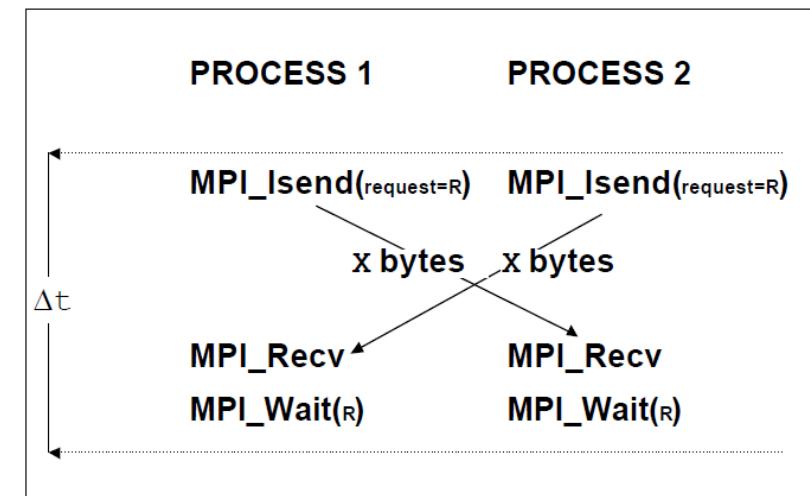
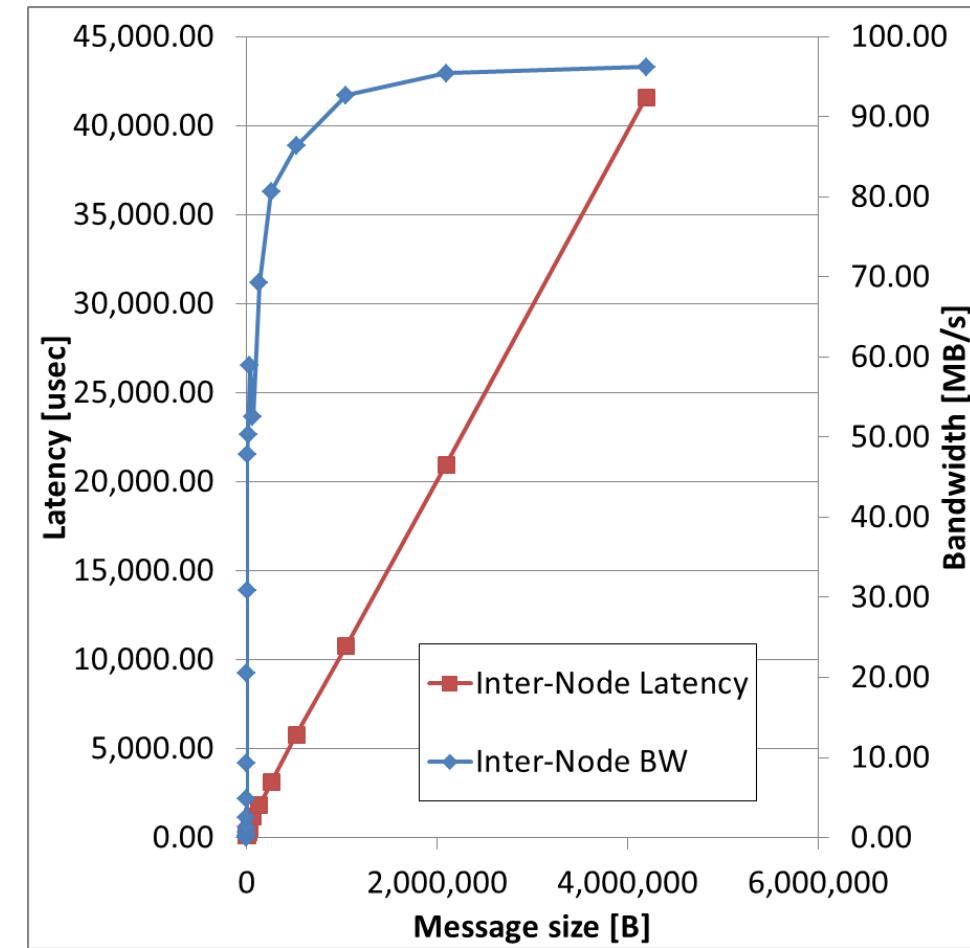


Figure 2: PingPing pattern



Example: IMB & Gigabit Ethernet

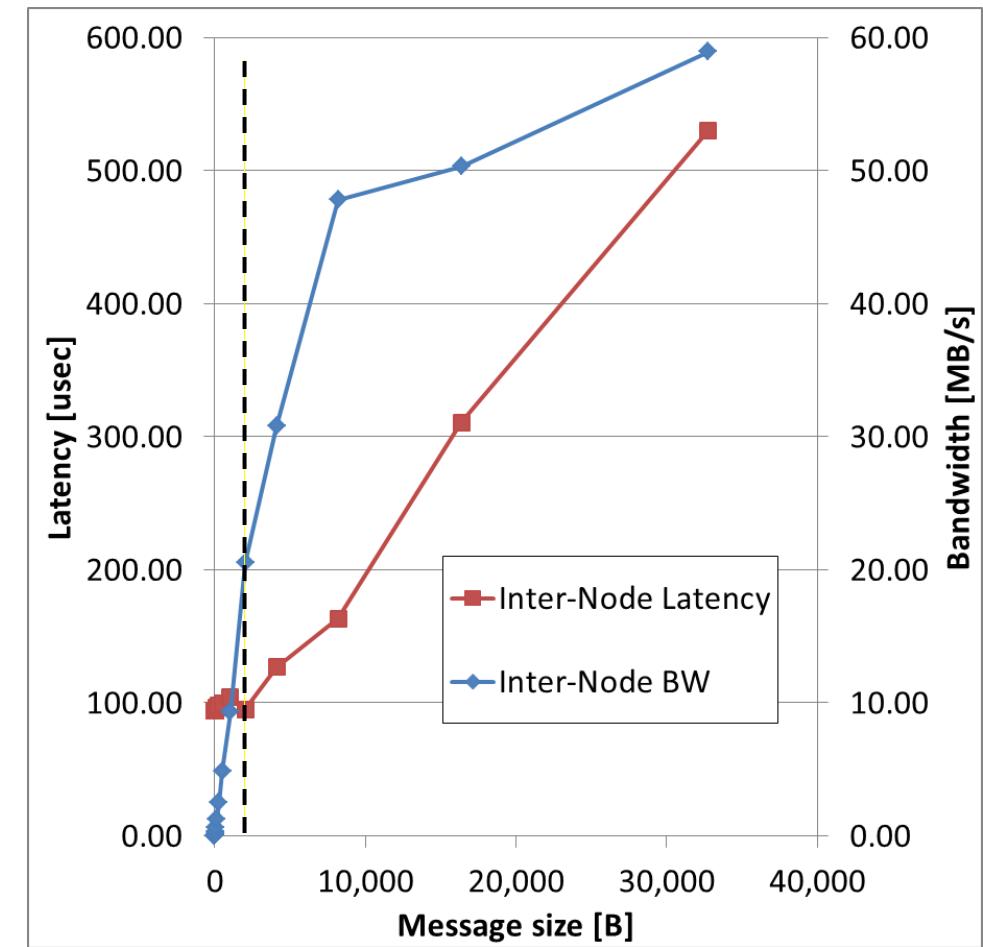
- Pingpong test
- Quiescent system
 - Contention and congestion will dramatically affect performance
- Mapping
 - Inter- vs. intra-node effects
- Observations - BW
 - Low compared to memory BW
 - Reaches saturation asymptotically
- Observations - Latency
 - Linear scaling for large messages





Example: IMB & Gigabit Ethernet

- Pingpong test - zoomed view
- Observations - BW
 - Really low BW for small messages
 - Saturation reached slowly
 - Overheads!
 - Protocol, send, receive
- Observations - Latency
 - For small messages (<2kB) constant
 - Otherwise linear scaling





Latency Bandwidth Analysis

- Intel MPI Benchmarks (IMB) – SendRecv
 - Periodic communication chain, send and receive can overlap
 - Reports 2x peak BW (1 in, 1 out)
 - Double throughput for perfectly bidirectional systems
- Exchange
 - Reports 4x peak BW (2 in, 2 out)

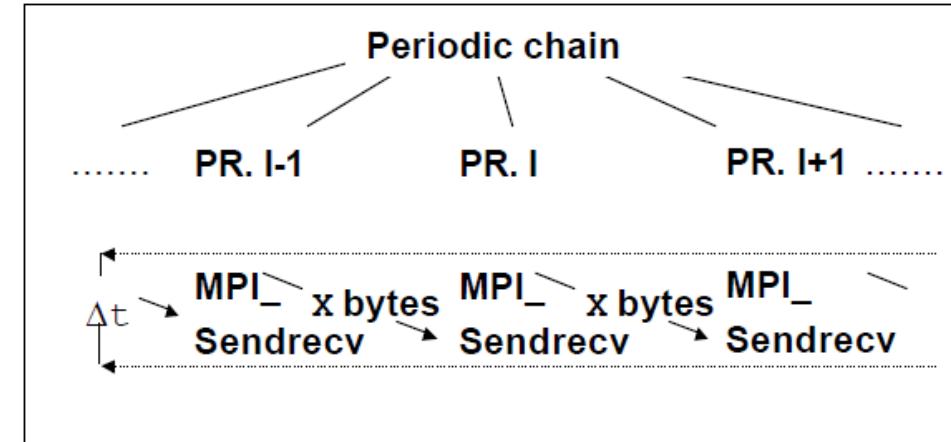


Figure 3: Sendrecv pattern

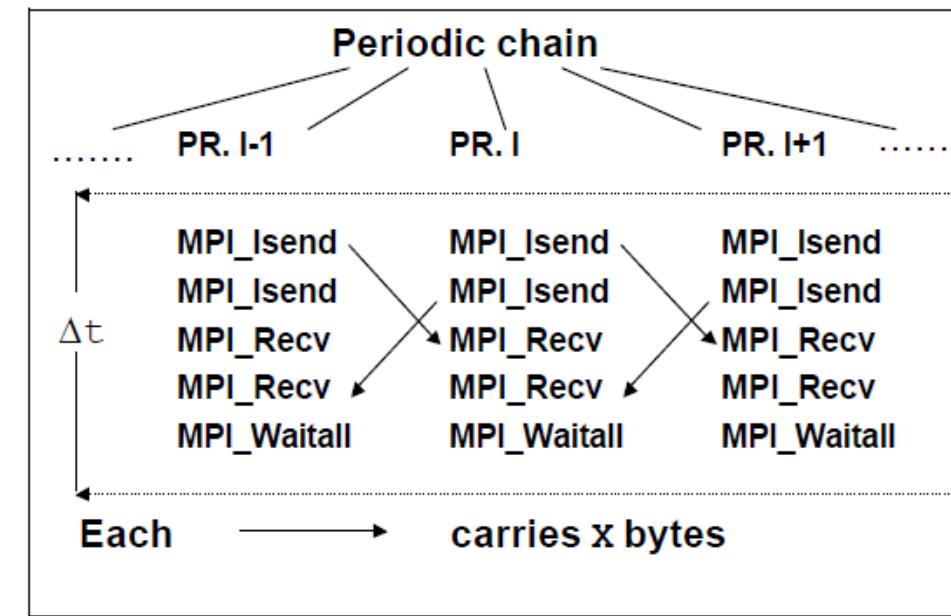
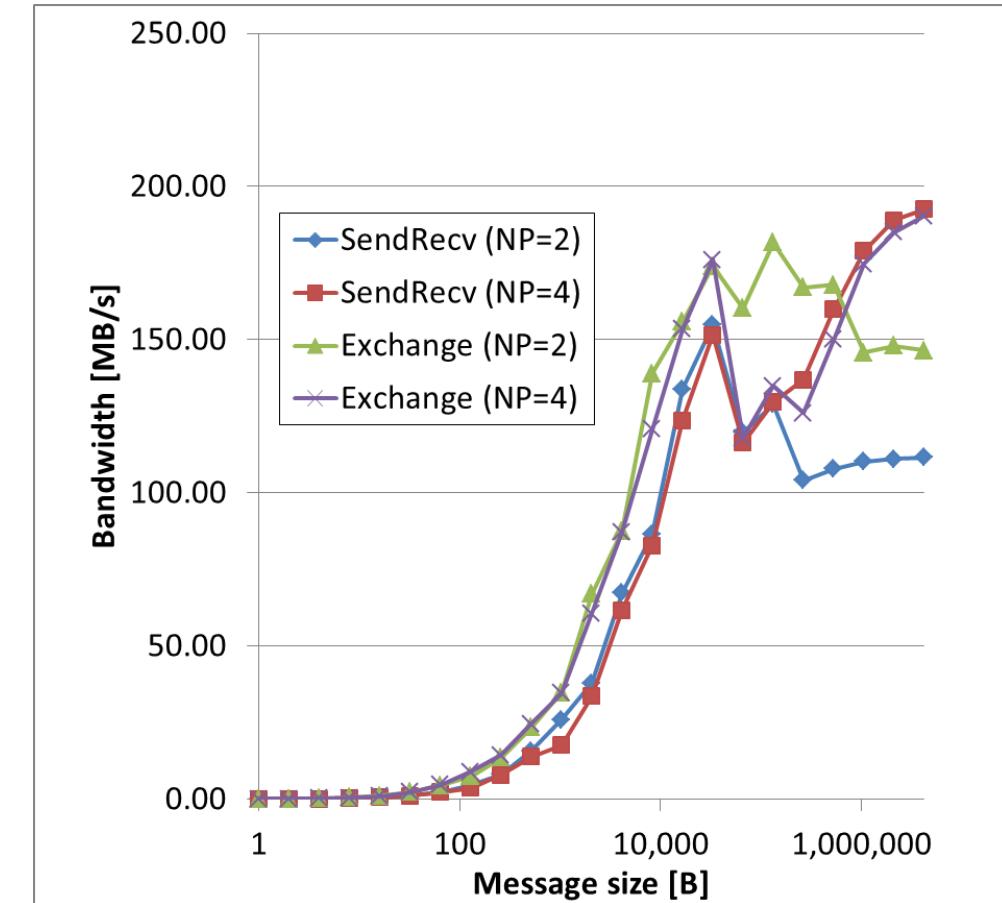


Figure 4: Exchange pattern



Example: IMB & Gigabit Ethernet

- Parallel transfer tests
 - One process per node
- Insights
 - Small messages (<1kB): exchange better
 - Large messages (>64kB): contention for exchange
 - Only NP=2 experiments are able to get close to saturation
 - 4P@64kB: performance downgrades
- Reported BW much lower than expected





Overhead and Availability Characteristics



Overhead and Overlap

- **Overhead** is defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.
 - [Culler et. al, LogP: Towards a Realistic Model of Parallel Computation, *PPoPP*, 1993]
- **Application availability** is defined to be the fraction of total transfer time that the application is free to perform non-MPI related work
 - [Lawry et. al, COMB: A Portable Benchmark Suite for Assessing MPI Overlap, *CLUSTER*, 2002]

$$\text{Availability}[\%] = 1 - (\text{overhead}[usec]/\text{transfer_time}[usec])$$



Overhead and Overlap

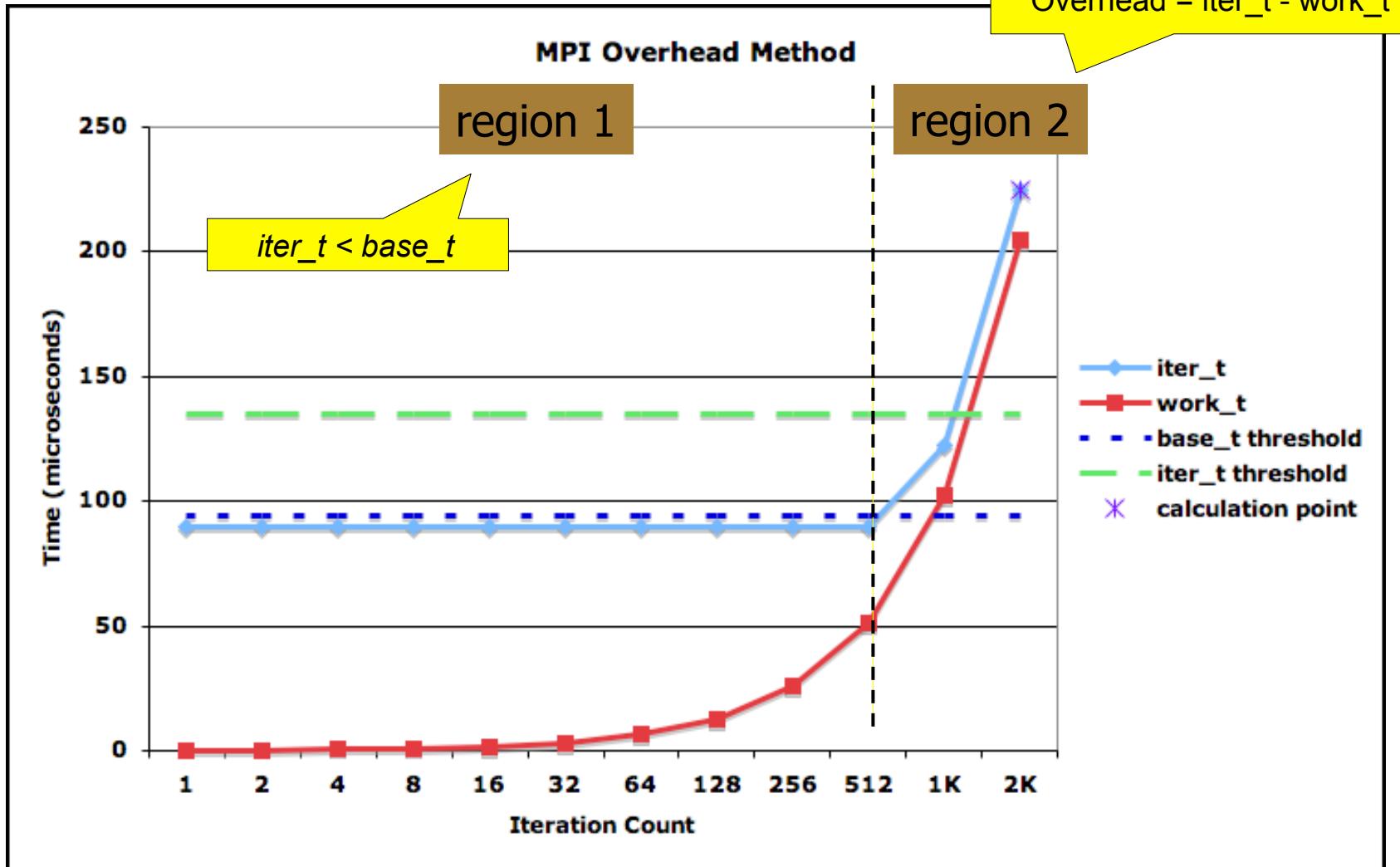
- Measuring overhead: basic idea is a post-work-wait loop
 - For each message size, repeat following steps with increasing `work_t`:
 1. `MPI_Isend`
 2. Work loop
 3. `MPI_Wait`

```
graph LR; A["1. MPI_Isend"] --> B["2. Work loop"]; B --> C["3. MPI_Wait"]; B --- D["work_t"]; D --- E["iter_t"]
```
- Three steps:
 1. Work completes before message transfer is complete (region 1)
 - Derive `base_t` based on first `iter_t` measurement
 - E.g., $base_t = iter_t * 1.05$
 - $iter_t < base_t$
 - Message transfer time equals loop time `iter_t`
 2. Work time exceeds message transfer time (region 2)
 - $iter_t > base_t$
 - Overhead = $iter_t - work_t$
 3. Stop if $iter_t > threshold$



Overhead and Overlap

$iter_t > base_t$
Overhead = $iter_t - work_t$



Source: <http://www.cs.sandia.gov/smb/overhead.html>



Overhead and Overlap Analysis

■ Sandia MPI Micro-Benchmark Suite (SMB)

- Free benchmark suite

1. Host Processor Overhead

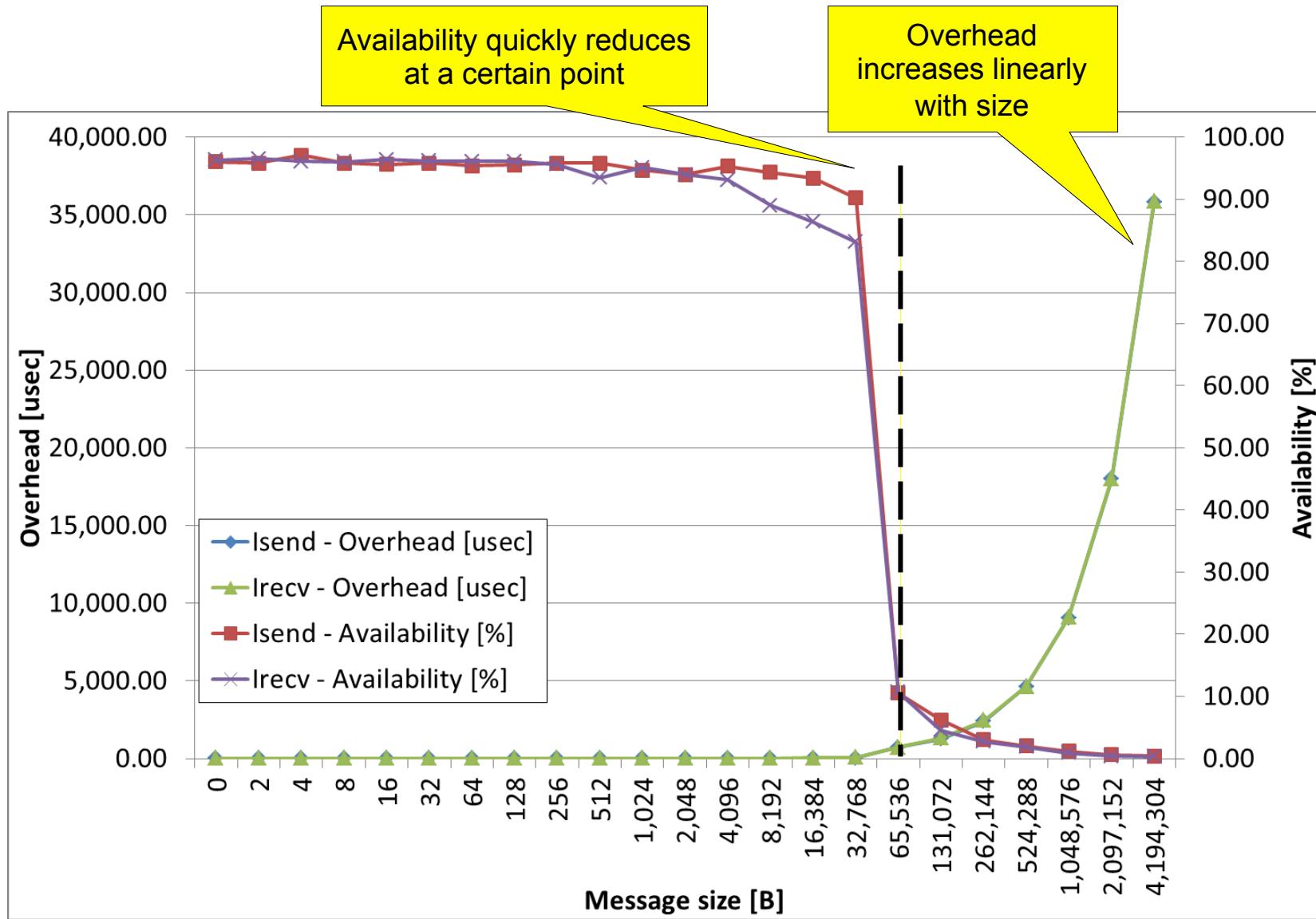
- Measures host processor overhead and availability during non-blocking MPI operations

2. Real World Message Rate Benchmark

- Measures sustained message throughput at scale with multiple peers, as would be expected in real Sandia application scenarios

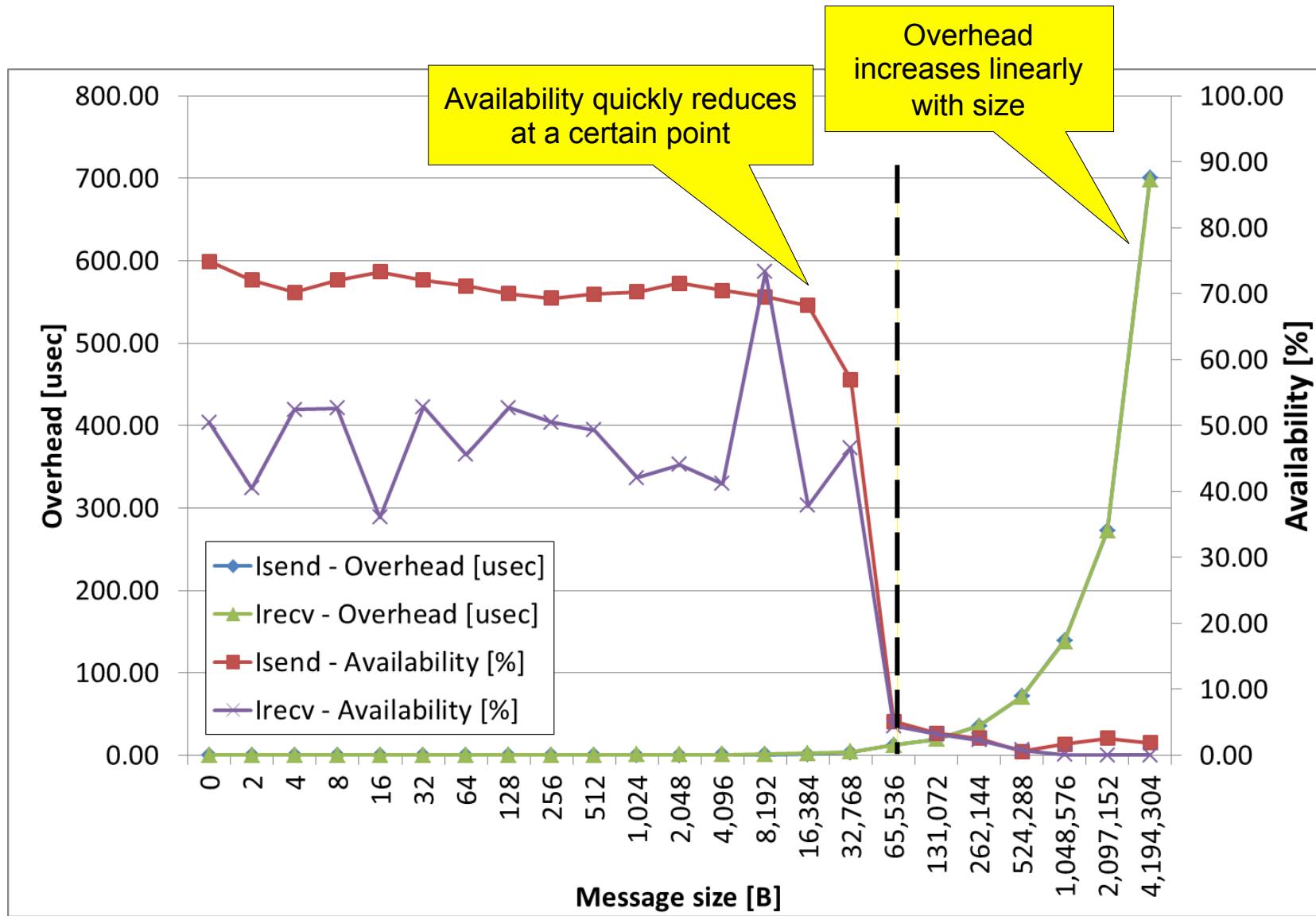


Example: SMB & Gigabit Ethernet





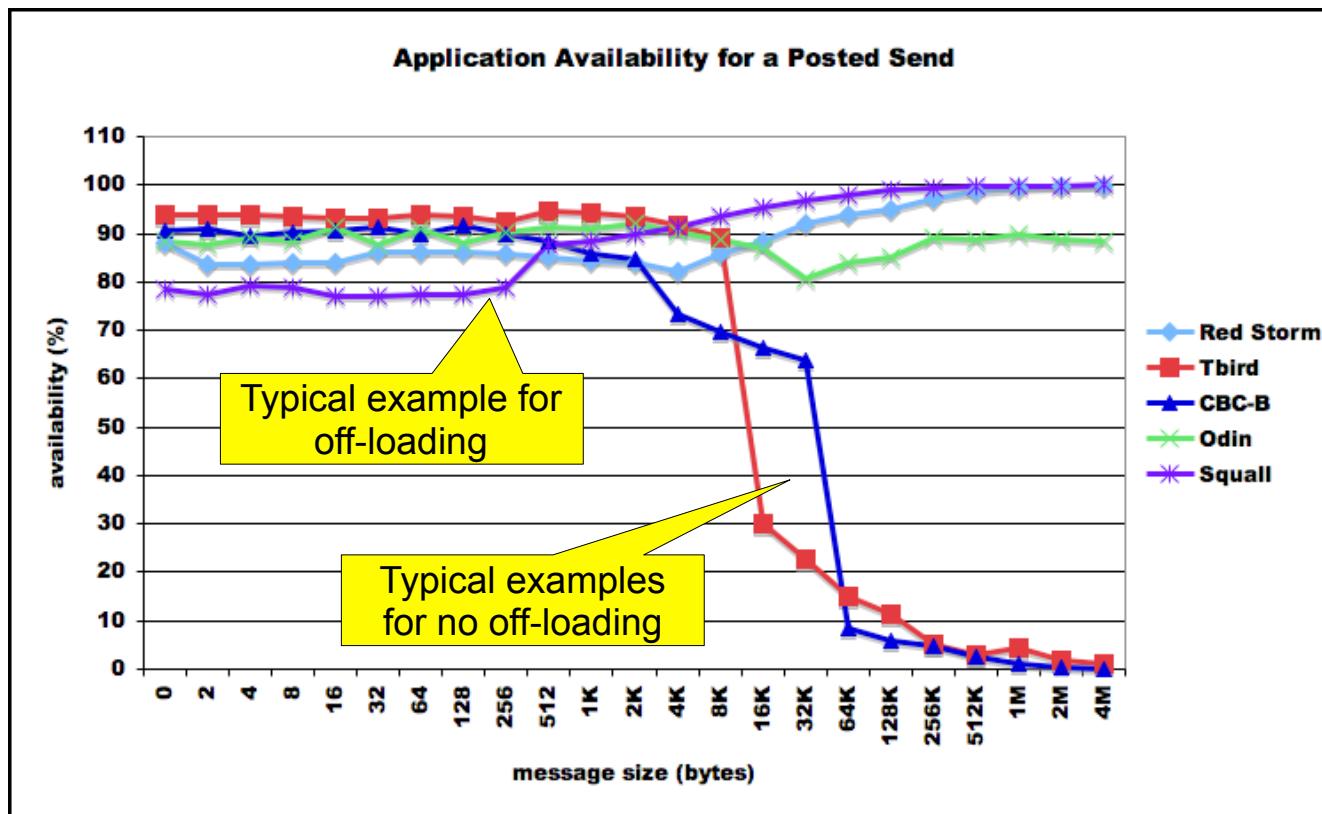
Example: SMB & Infiniband





Example: SMB - various

| | Red Storm | Thunderbird | CBC-B | Odin | Red Squall |
|----------------|-------------|-----------------|------------|-------------|------------|
| Interconnect | Seastar 1.2 | InfiniBand | InfiniBand | Myrinet 10G | QsNetII |
| Adapter | Custom | PCI-Express HCA | InfiniPath | Myri-10G | Elan4 |
| Host Interface | HT 1.0 | PCI-Express | HT 1.0 | PCI-Express | PCI-X |

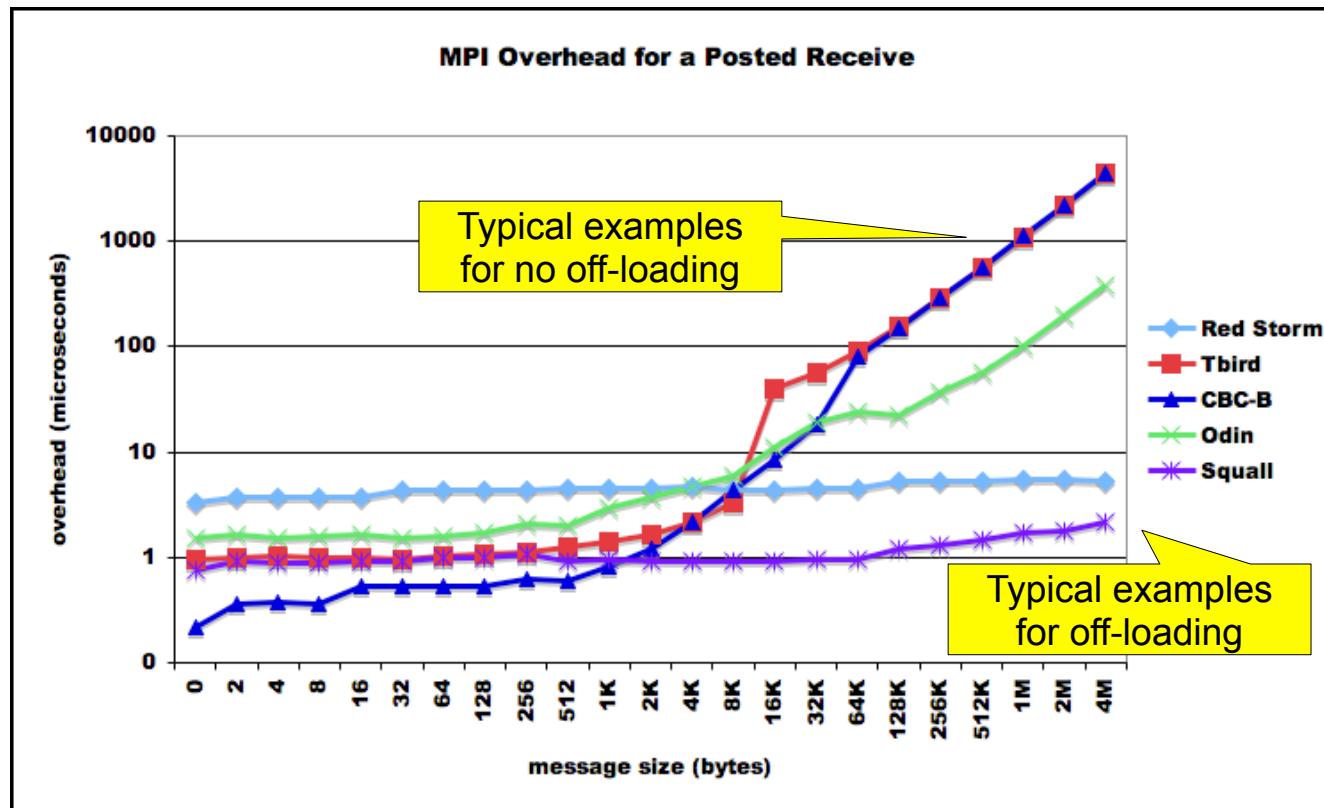


Source: <http://www.cs.sandia.gov/smb/overhead.html>



Example: SMB - various

| | Red Storm | Thunderbird | CBC-B | Odin | Red Squall |
|----------------|-------------|-----------------|------------|-------------|------------|
| Interconnect | Seastar 1.2 | InfiniBand | InfiniBand | Myrinet 10G | QsNetII |
| Adapter | Custom | PCI-Express HCA | InfiniPath | Myri-10G | Elan4 |
| Host Interface | HT 1.0 | PCI-Express | HT 1.0 | PCI-Express | PCI-X |



Source: <http://www.cs.sandia.gov/smb/overhead.html>



Message Rate Characteristics



Message Rate

- Up to now: latency, bandwidth, overhead

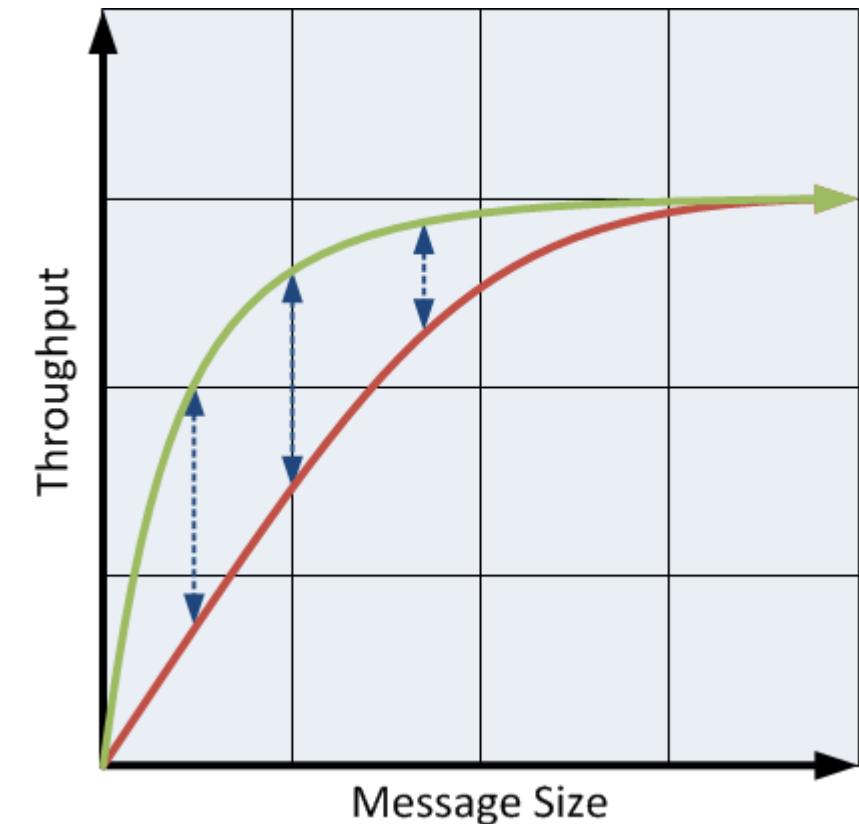
- Basically sufficient for characterization

- However, small messages:

- Latency cannot characterize overlap in this case
 - Bandwidth typically reported for large messages

- More important:

- How many messages per second one can send out?
 - Push-model instead of round-trip communication



→ Message Rate (MR), in messages per second



Message Rate

- Theoretical upper bound: BW/size
 - Sustained bandwidth for given message size
 - I.e., peak bandwidth without protocol overhead for given message size
- Practical upper bound: gaps
 1. Network protocol overhead including framing, headers, CRC, etc
 2. Message passing protocol overhead including tags, source identification, etc
 3. Packet-to-packet gaps caused by network interface
 4. Packet-to-packet gaps caused by switching units
 5. Software overhead for sending and receiving



Message Rate

- Various overhead sources
- Multi-pair tests help overcoming software overhead limitations
 - Multiple end points per node
- Thus:
 - Latency & bandwidth should not (best case) be affected by multi-pair tests
 - MPI message rate typically benefits a lot from multiple end points per node

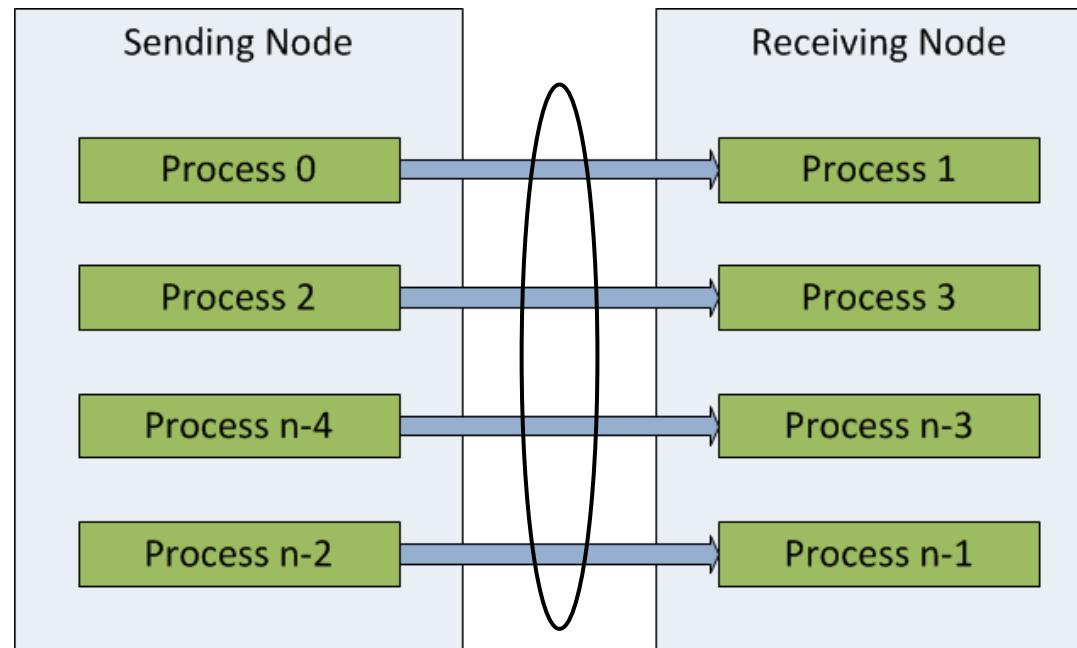
| Network | 10GE ¹ | IB-QDR ² | EXTOLL ³ |
|---|---------------------|---------------------|---------------------|
| Net Speed | 10 Gbps | 32 Gbps | 5 Gpbs |
| Theoretical peak message rate (8B payload) | 156.3 | 500.0 | 78.0 |
| Network protocol overhead | 82 B | 38 B | 32 B |
| MPI protocol overhead | 24 B | 10 B | 16 B |
| Packet-to-Packet gap of switching units | NA | NA | 8 B |
| Packet-to-Packet gap of network interface | NA | NA | 0 B |
| Overhead total (as appropriate) | 114 B (w/o gaps) | 56 B (w/o gaps) | 64 B (total) |
| Sustained Message Rate | 0.66 (0.42%) | 6.67 (1.33%) | 9.73 (12.4%) |
| Calculated overhead derived from sustained MR | 416.67 B | 599.70 B | 64.14 B |

Holger Fröning, Mondrian Nüssle, Heiner Litz, Christian Leber and Ulrich Brüning, On Achieving High Message Rates, 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), May 13-16, 2013, Delft, The Netherlands.



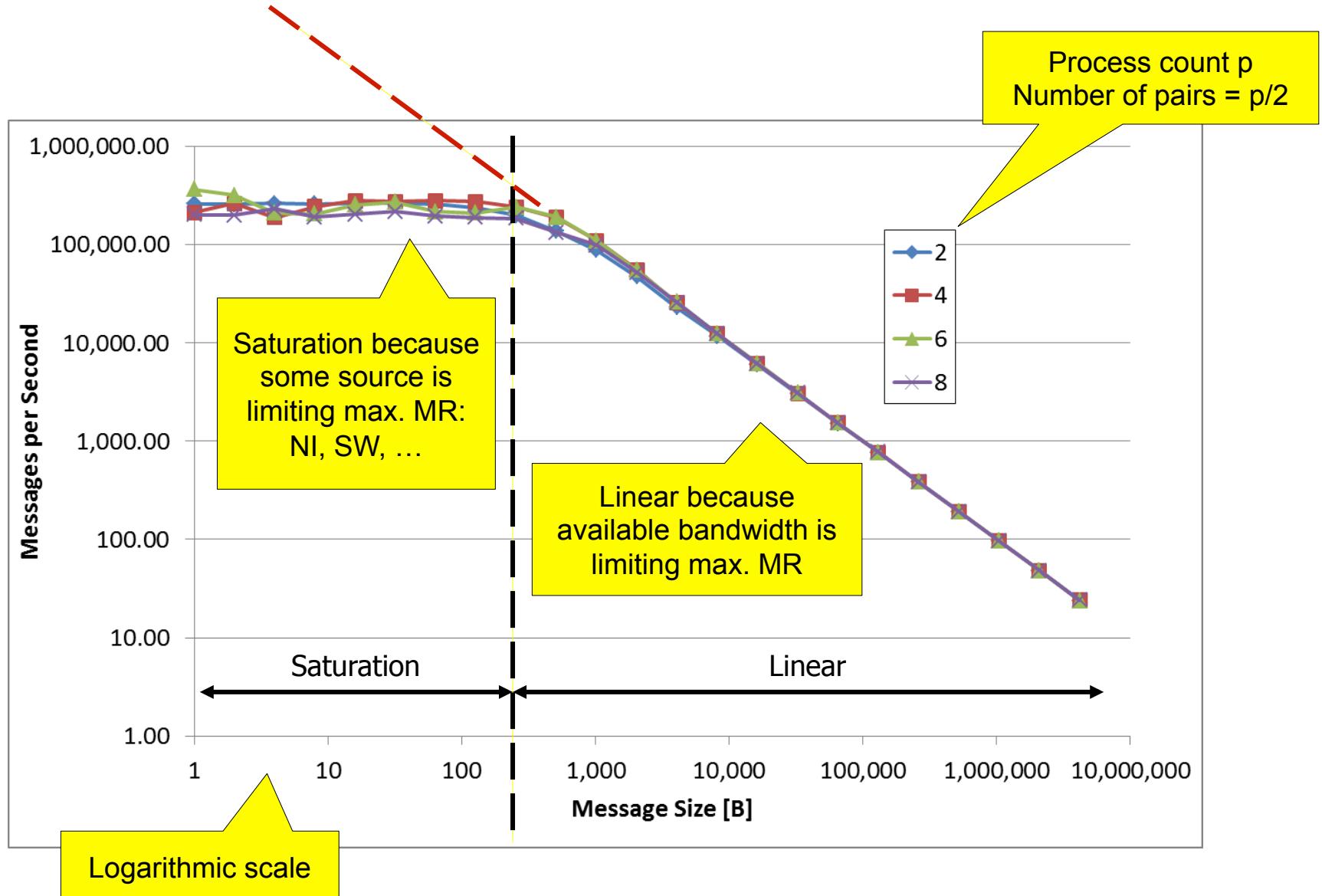
Message Rate Analysis

- Sandia MPI Micro-Benchmark Suite (SMB)
- Ohio State University (OSU) Micro-Benchmarks
 - Both report cumulative messages per second
 - Ensure correct mapping!



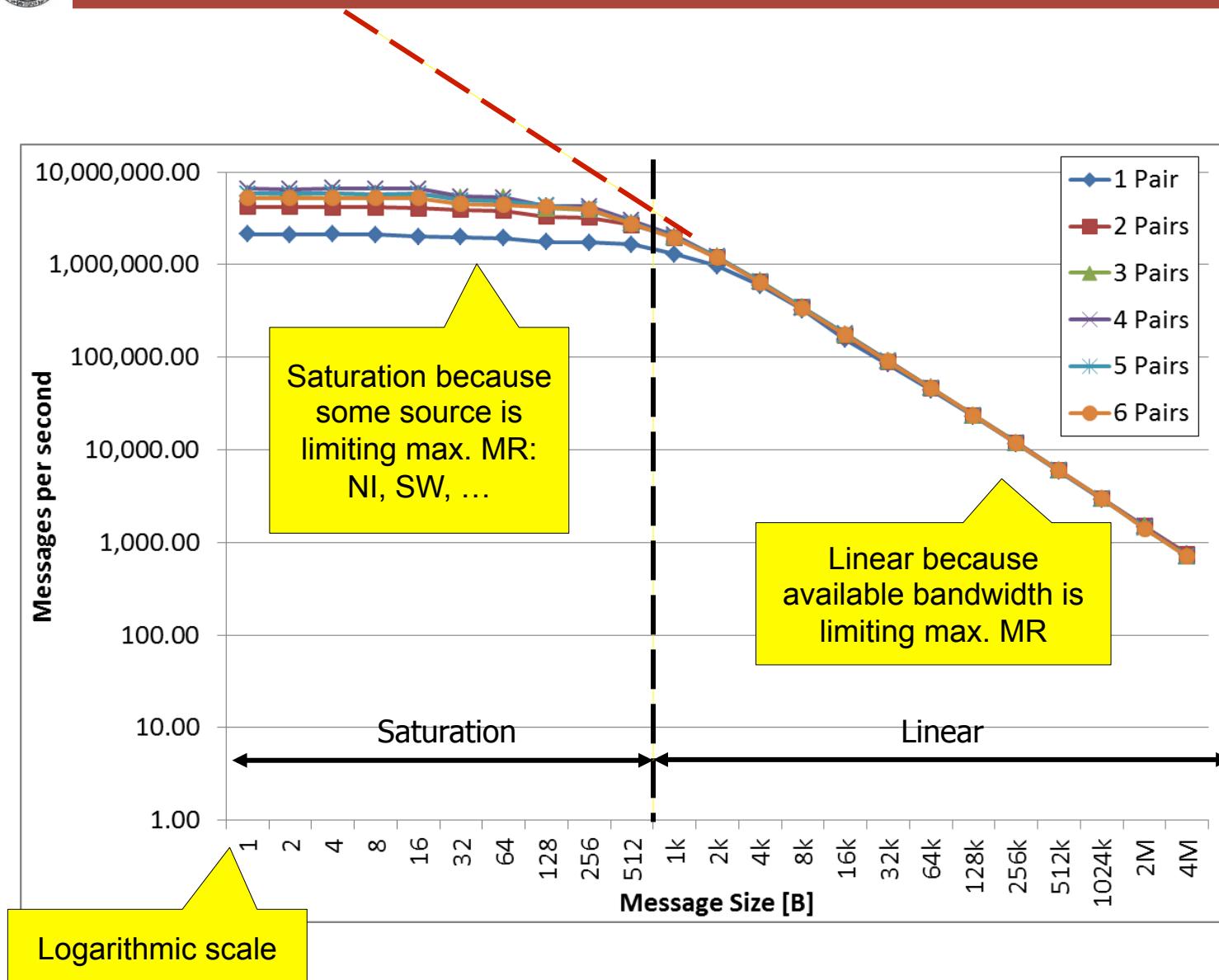


Example: OSU_MBW_MR & Gigabit Ethernet



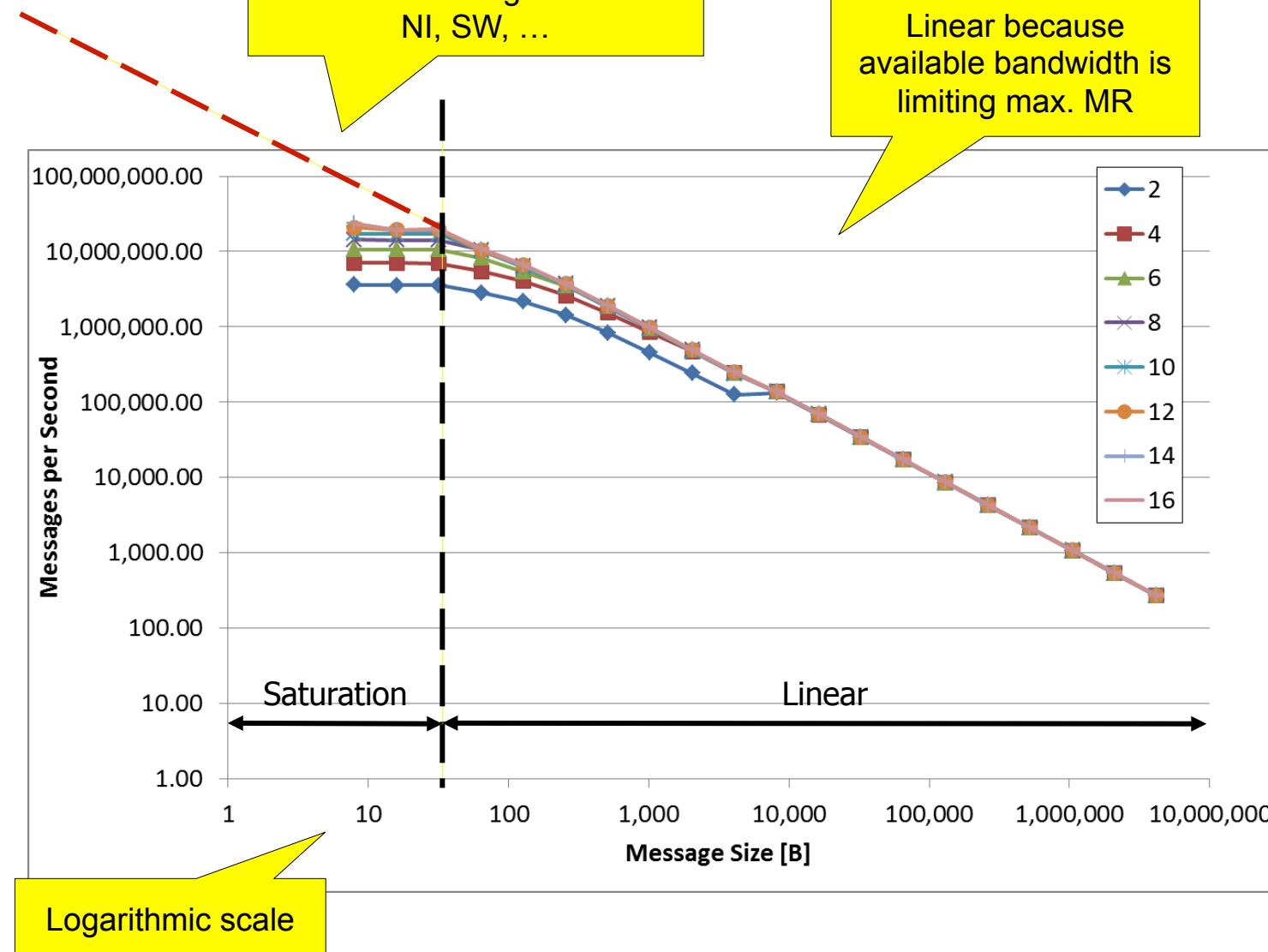


Example: OSU_MBW_MR & Infiniband





Example: OSU_MBW_MR & EXTOLL R2 (FPGA)





Summary

■ Several characteristics

- Latency
- Bandwidth
- Message rate
- Overhead and Overlap

„Right“ characteristic
has to be chosen based
on a certain workload!

■ Complete communication stack contributes

- Software for sending and receiving: API, libraries, drivers
- Network interface architecture
- Network switching resources & links

Optimization
strategy depends
on networking
features

■ Not covered here: contention and congestion

- Can have huge impact on performance
- Characterization of those highly depends on applied workload



Introduction to High Performance Computing

Lecture 08 – Benchmarks

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg



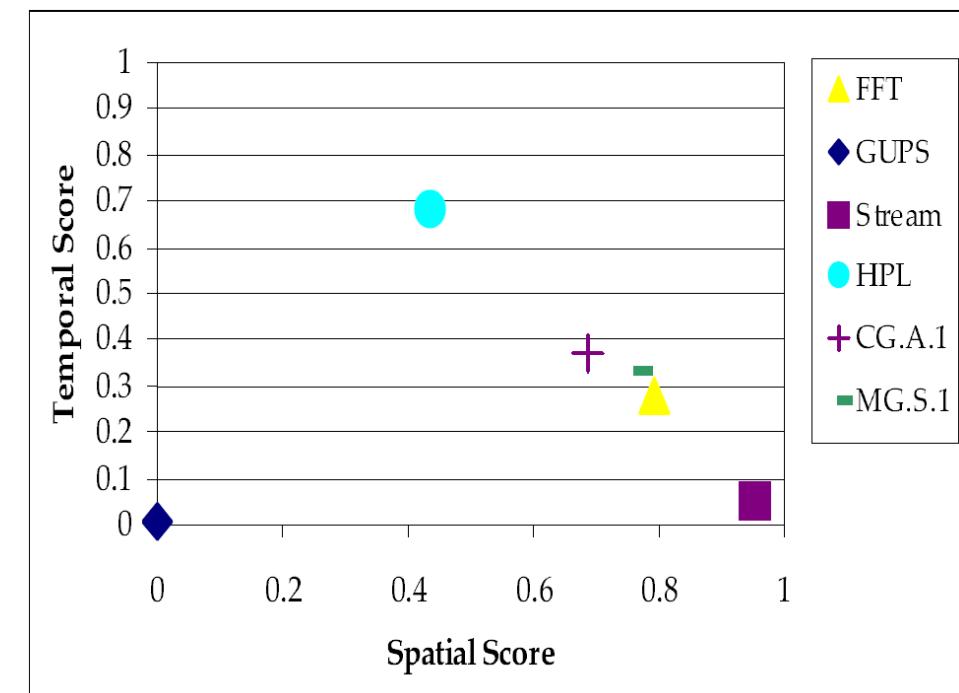
Overview

- Micro-benchmarks
 - Specialized to characterize only a certain aspect
- Latency
- Bandwidth
- Message rate
- Overlap
- Impact towards application performance unclear
- Application-level or complex benchmarks
 - Mimic the behaviour of applications or algorithms
- Examples for algorithms
 - LU decomposition, FFT, ...
- Examples for applications
 - Weather and climate research, crash simulations, molecular dynamics, protein folding, graph computations, ...



Characterization of Benchmarks/Applications

- Temporal and spatial locality for MPI applications
 - FFT: Fast Fourier Transform
 - GUPS: RandomAccess
 - Stream: Stream benchmark
 - HPL: High Performance Linpack
 - CG & MG: part of NAS Parallel Benchmark Suite (NPB)
- Metric how much pressure an application puts on the network?



[Weinberg et. al, Quantifying Locality In The Memory Access Patterns of HPC Applications, Supercomputing 2005]

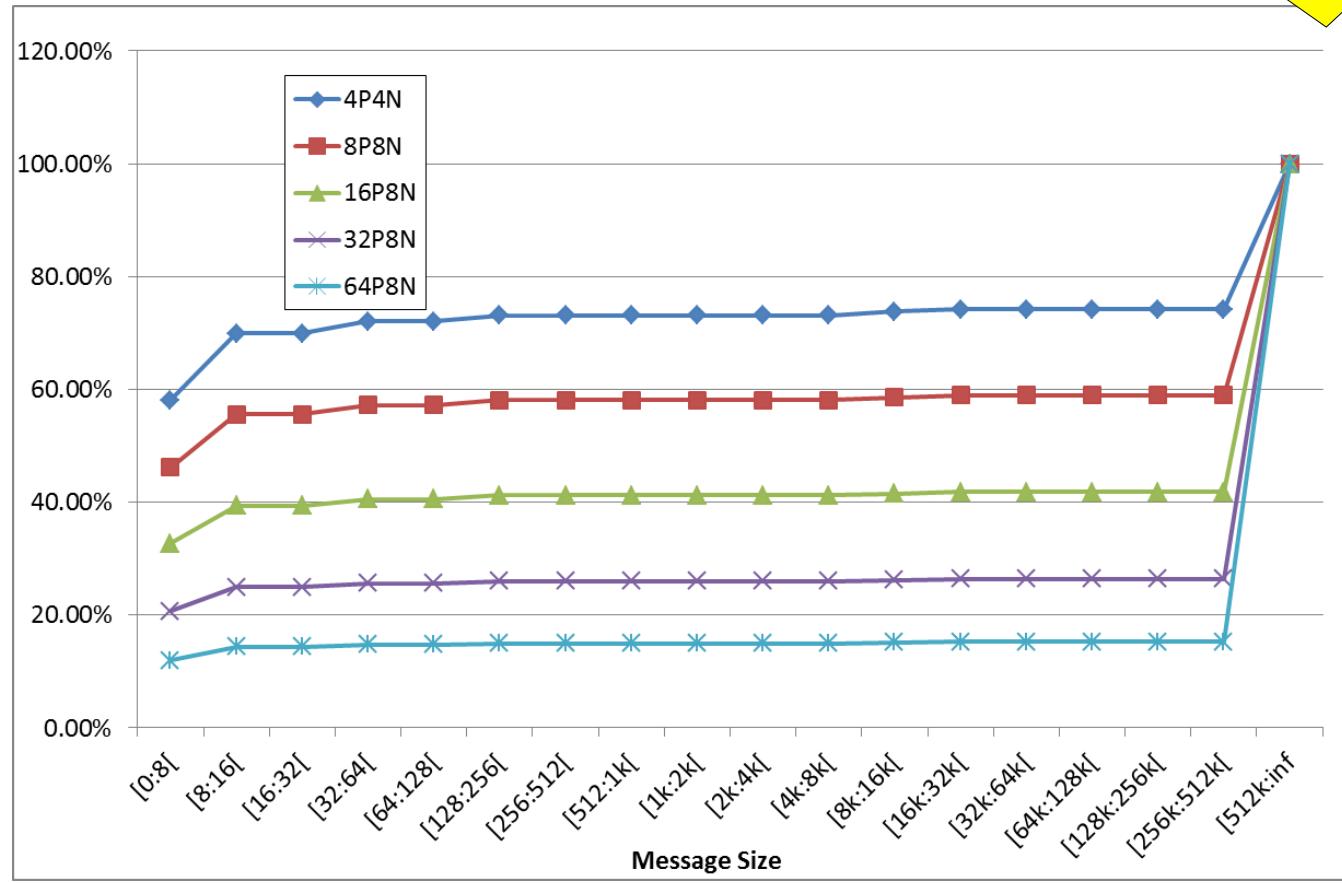


Characterization of Benchmarks/Applications

■ Benchmark: MPIFFT

- Medium temporal locality, high spatial locality

Cumulative distribution function (CDF)

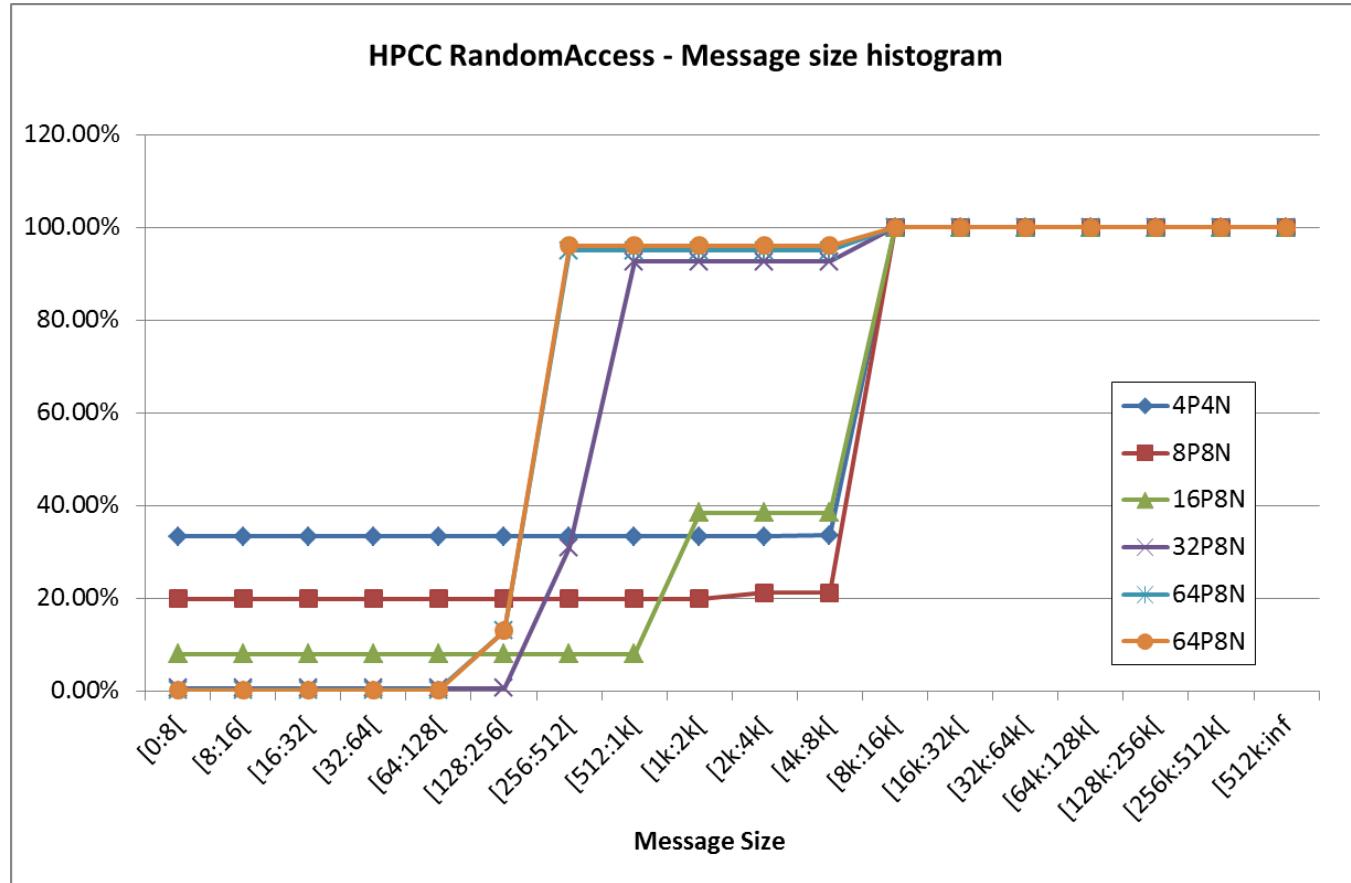


Shift towards **larger messages** with increasing process count!



Characterization of Benchmarks/Applications

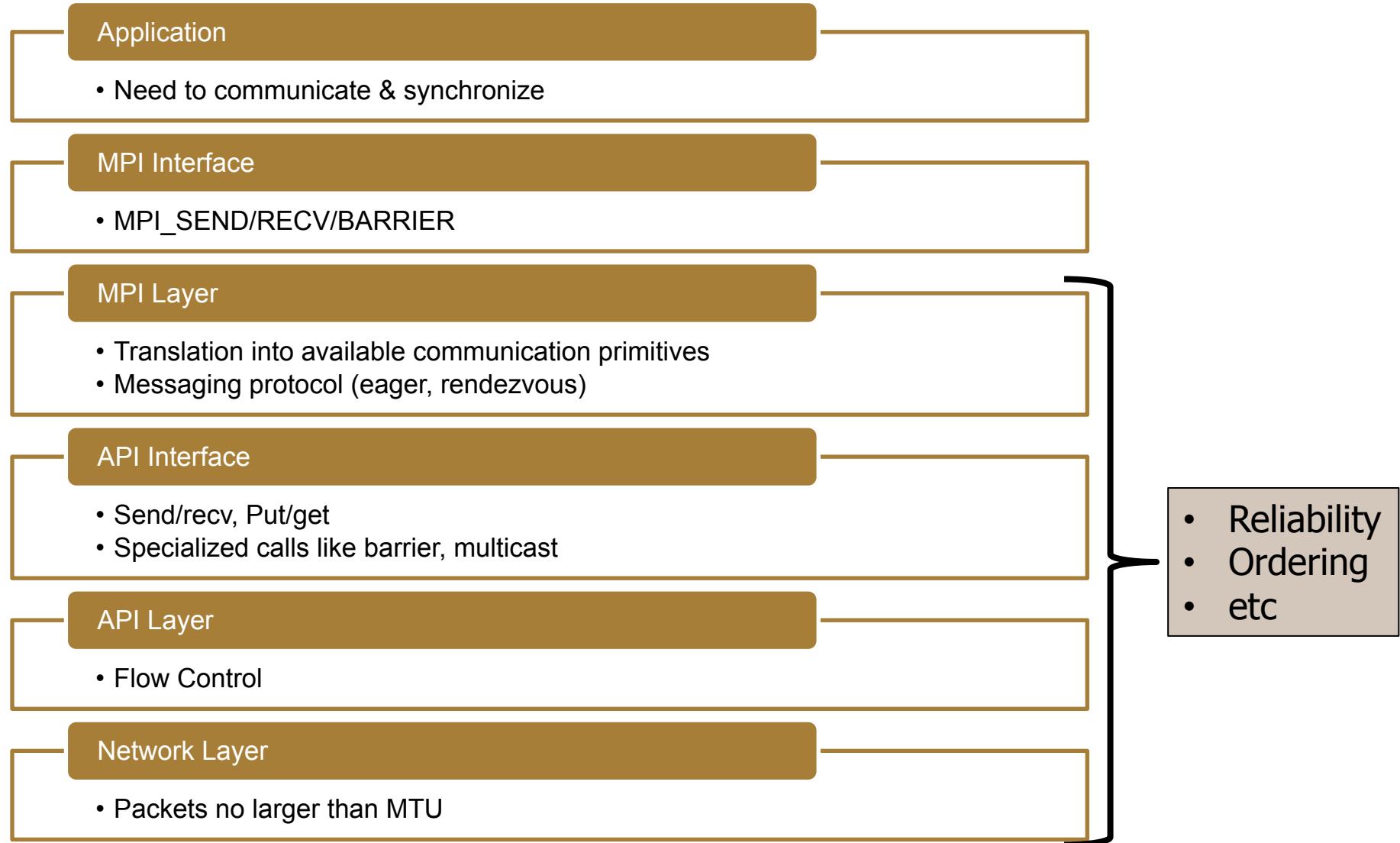
- Benchmark: RandomAccess (GUPS)
 - Both spatial and temporal locality very low



Shift towards **smaller messages** with increasing process count!



Messaging Software Stack





Characterization of Benchmarks/Applications

- Interconnection Network (IN) is the bottleneck
 - Bandwidth & latency mismatch
 - For almost any application/benchmark
- But how big is the pressure?
 - Data volume transferred?
 - Number of messages?
 - Time spent in MPI layers?
- Maybe all of them - everything depends on the application
 - **MPI time** – the execution time fraction spent in MPI layers
 - **Message Density Distribution** – message count between pairs
 - **Data Density Distribution** – data transferred between pairs
 - **Message Size Distribution** – message sizes used



Benchmarks



Overview

- There is no „one fits all“ benchmark
- We will look at:
 - High Performance LINPACK – TOP500
 - NAMD – molecular dynamics
 - NAS Parallel Benchmarks
 - HPC Challenge
 - Weather Research and Forecasting Model
 - GRAPH500
- Many more out there:
 - GROMACS – molecular dynamics
 - SPECMPI2007
 - ANSYS FLUENT CFD
 - See 13 dwarfes in „The Landscape of Parallel Computing Research: A View From Berkeley“
 - HPCG
 - ...



High Performance Linpack (HPL)

- Used to rank supercomputers in the TOP500 list
- Solve a dense NxN system of linear equations ($Ax = b$)
 - DAXPY most important (double precision $\alpha * x + y$; α scalar, x, y vectors)
 - $f = 2/3 \cdot N^3 + 2 \cdot N^2 = O(N^3)$
- Computationally very intensive
- Rather relies on computing performance and cache/memory capacity than on messaging
 - See performance of GE based supercomputers in the TOP500

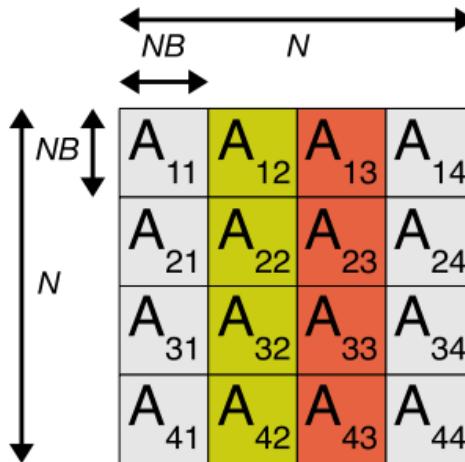
Thom Dunning, director of the National Center for Supercomputing Application, 2011:
"The Linpack benchmark is one of those interesting phenomena -- almost anyone who knows about it will deride its utility. They understand its limitations but it has mindshare because it's the one number we've all bought into over the years."

(<http://www.technologyreview.com/blog/mimssbits/25981/>)



High Performance Linpack (HPL)

- Data is distributed on a $P \times Q$ grid of processes, each owning a block of the size $NB \times NB$
- Plenty of more parameters
 - Ordered by importance on the right
 - Hand optimization!
 - Keep CPUs busy!
- HPL should achieve $\geq 95\%$ efficiency when run on a single node



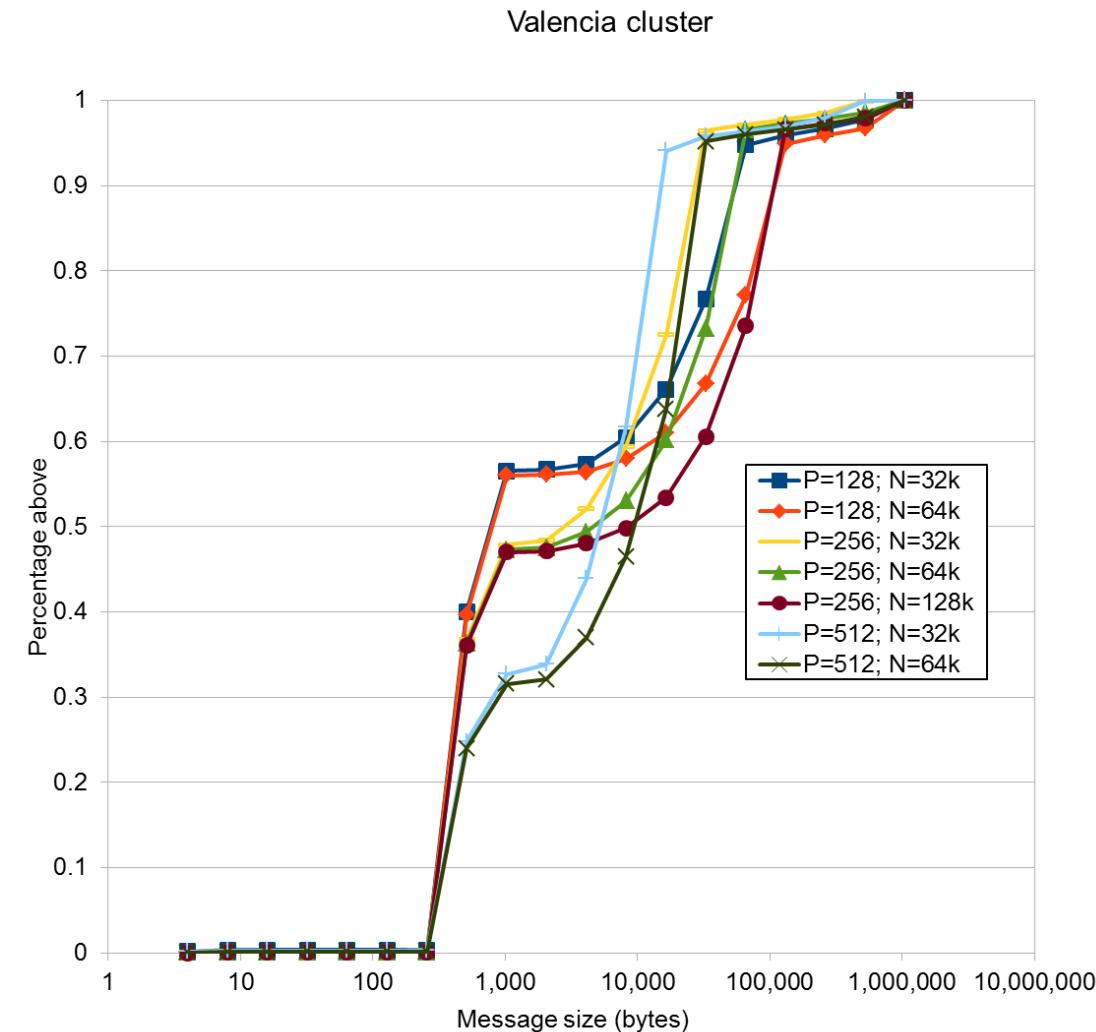
<http://hpc-ga-bench.gforge.uni.lu/>

| Optimization parameter | Description |
|------------------------|--------------------------------------|
| NB | Block size |
| N | Problem size |
| P | Process matrix rows |
| Q | Process matrix columns |
| BCAST | Broadcast algorithm |
| DEPTH | Lookahead depth |
| FSWAP | Swapping algorithm |
| NBMIN | Recursive stopping criterion |
| NDIV | Number of panels in recursion |
| PFACT | Panel factorization algorithm |
| TSWAP | Swapping threshold for mix algorithm |
| ... | ... |



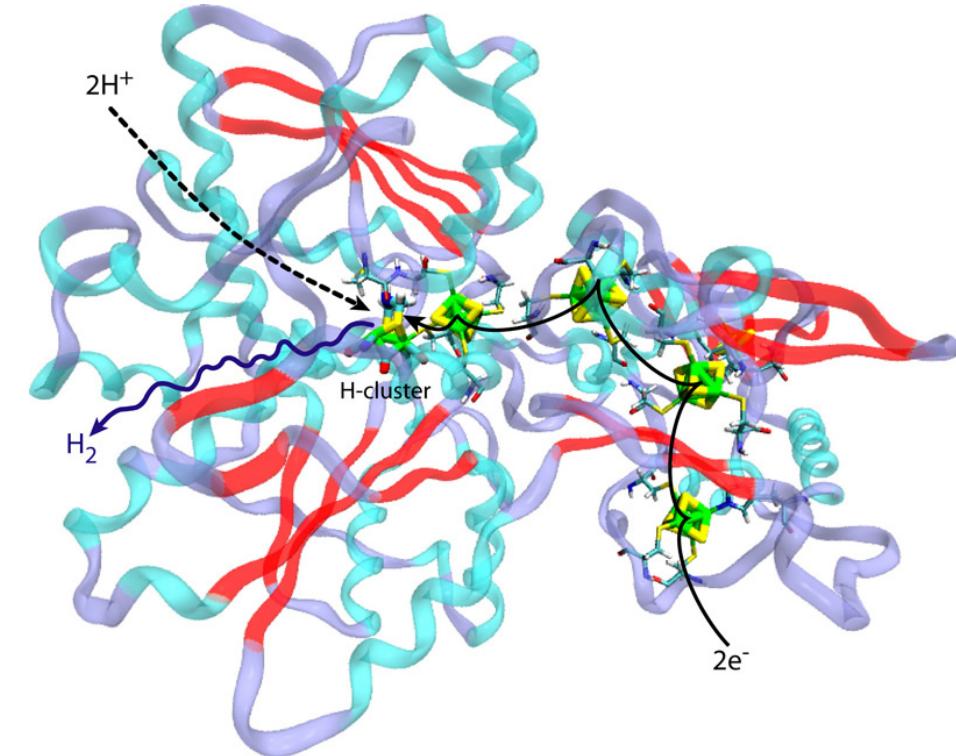
High Performance Linpack (HPL)

- Characteristics
 - Mostly MPI_Recv,
MPI_Send, MPI_Irecv
 - MPI time is about 25%
- Performance numbers
 - See TOP500





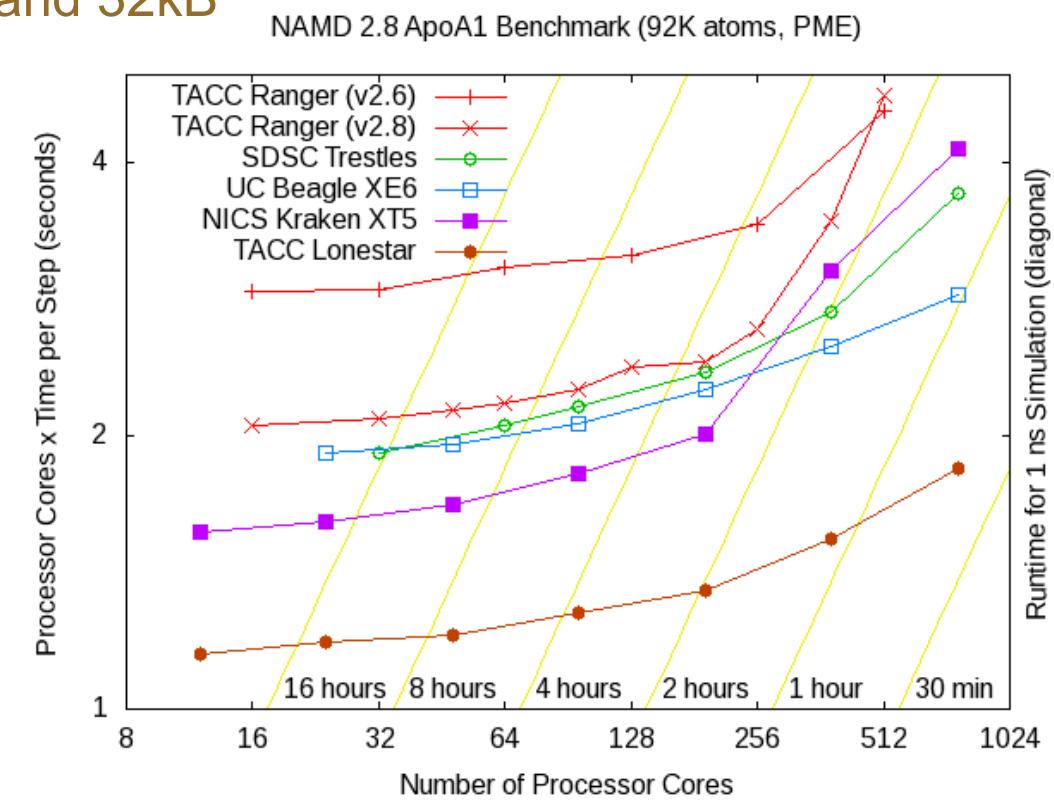
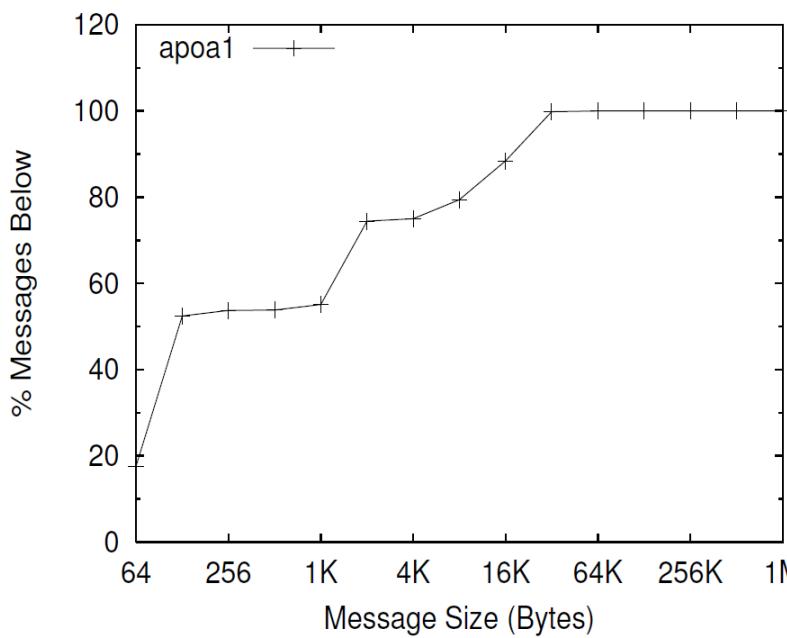
- Fully featured, production molecular dynamics program for high performance simulation of large bio-molecular systems
 - Electrostatic and Van der Waals forces
 - Millions of objects
 - Open-source
- N-Body problem, $f = O(N^2)$
- Time scale of 1fs (10^{-15} s)
- Example workload: Satellite Tobacco Mosaic Virus (STMV)
 - 100M atoms, 160 genes
 - Petascale-class: 100ns/day of simulation time
- Complex structures like parasites:
~ 6k genes
 - Estimation: 1400x runtime increase



Source: <http://www.ncsa.illinois.edu>



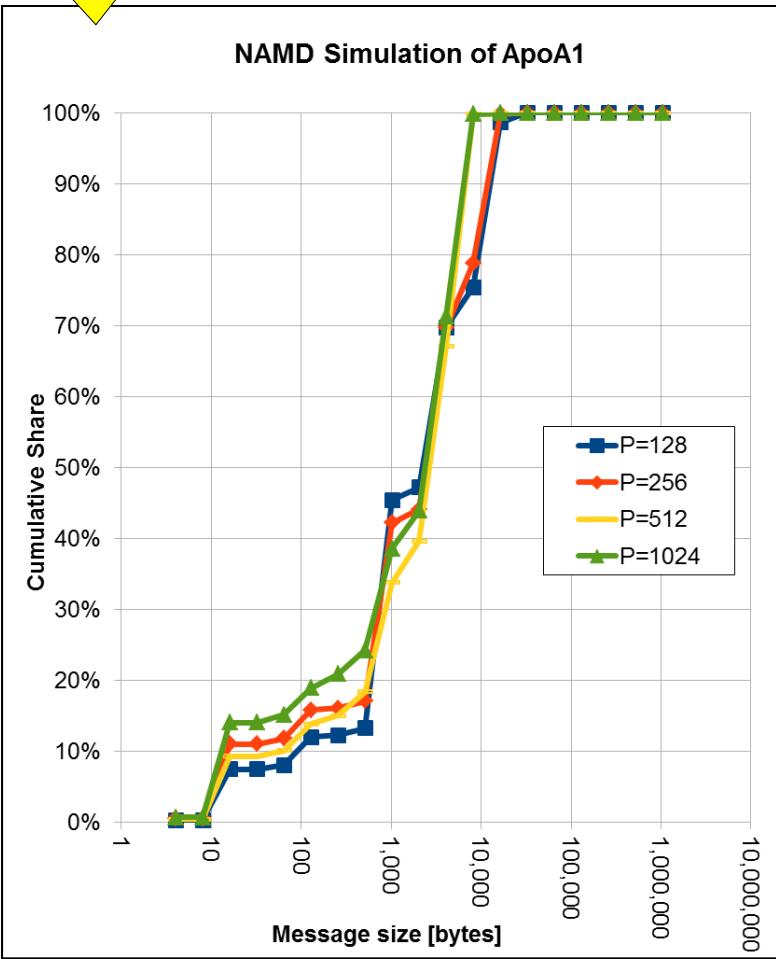
- ApoA1: protein with 92K atoms of lipid, protein, and water
 - Mostly MPI_Isend, MPI_Send, MPI_Recv, MPI_Barrier
 - 50% below 128B
 - Other 50% between 128 and 32kB
- MPI time is about 24%



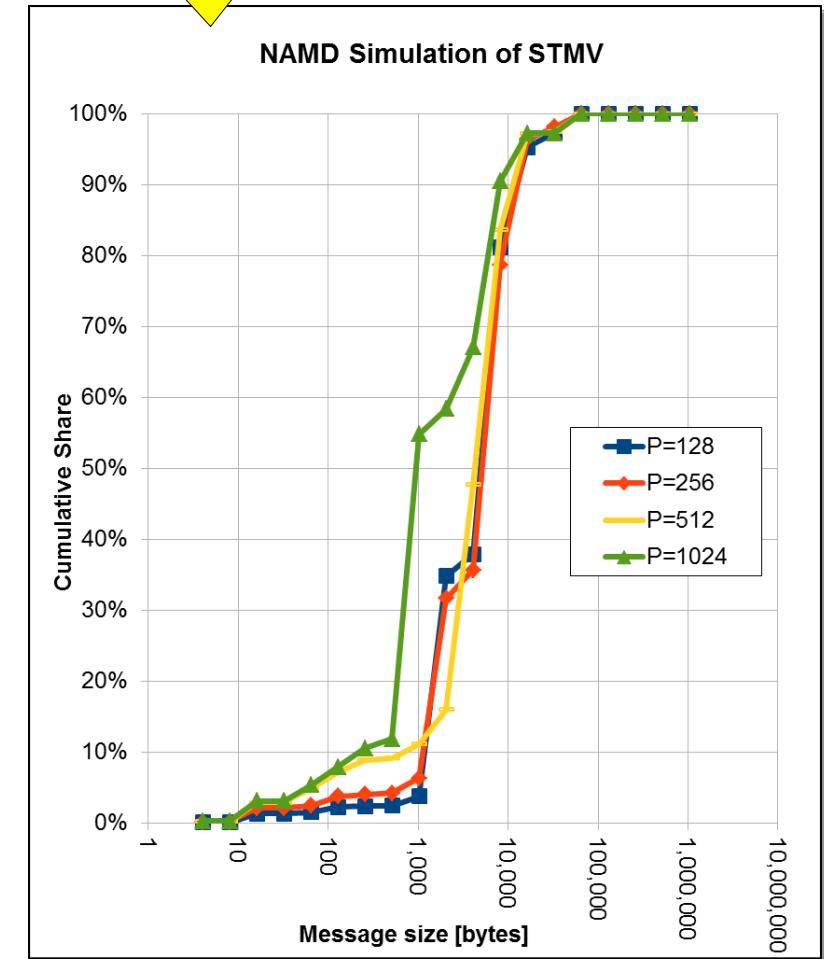
<http://www.ks.uiuc.edu/Research/namd/performance.html>



92k atoms



1M atoms





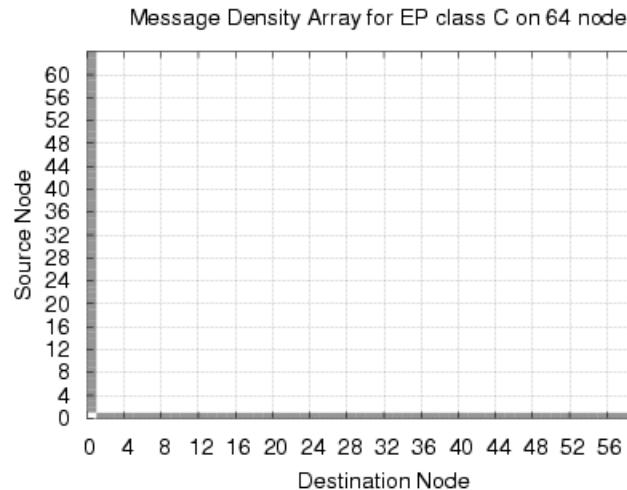
NAS Parallel Benchmarks (NPB)

- EP: „Embarrassingly Parallel“ (0% MPI time)
 - Generates pairs of random values typically used in Monte Carlo simulations
 - Upper achievable limit of floating point performance, no significant communication
- MG: „Multi-grid kernel“ (9% MPI time)
 - Approximate solution to the discrete Poisson problem (used in CFDs)
 - Highly structured long distance communication, both short and long data communication
- CG: „Conjugate Gradient“ (34% MPI time)
 - Approximation to the smallest eigenvalue of large, sparse, symmetric matrix
 - Unstructured grid communication, irregular long distance communication, unstructured matrix-vector multiplication
- FT: „3D-FFT PDE“ (37% MPI time)
 - Solve 3D partial differential equation (PDE) using forward and inverse FFT
 - Heavily relies on long distance communication
- IS: „Integer Sort“ (47% MPI time)
 - Sort N integer keys in parallel, often used in particle methods
- BT: „Block Tridiagonal“ (10% MPI time)
 - Solve nonlinear PDE using block tridiagonal matrix algorithms

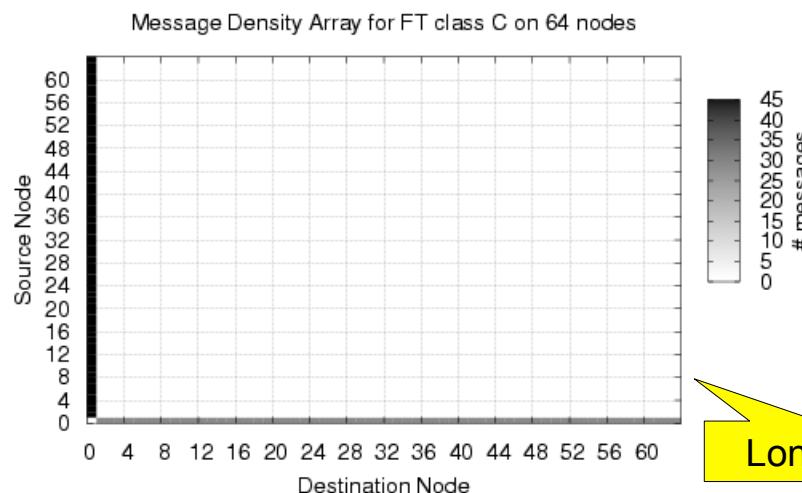
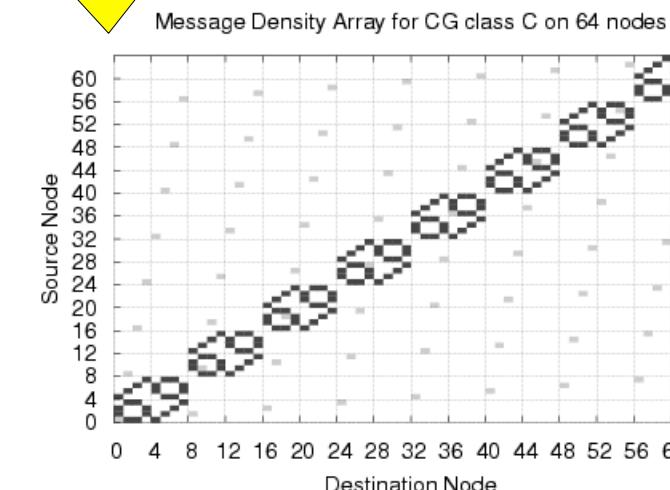
Numbers from [S.Sur, M. J. Koop, D. K. Panda, High-Performance and Scalable MPI over InfiniBand With Reduced Memory Usage: An In-Depth Performance Analysis, 2006 ACM/IEEE conference on Supercomputing]



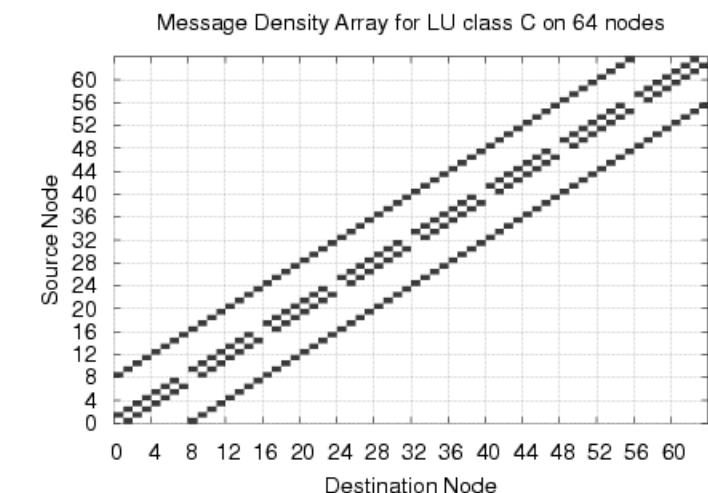
NPB - Message Density Distribution



Unstructured



No significant communication

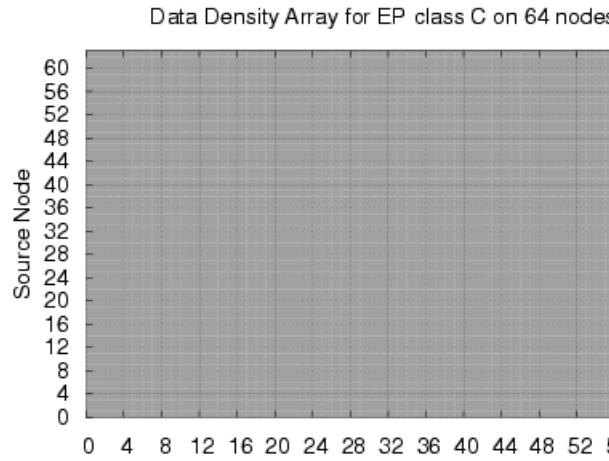


Long range

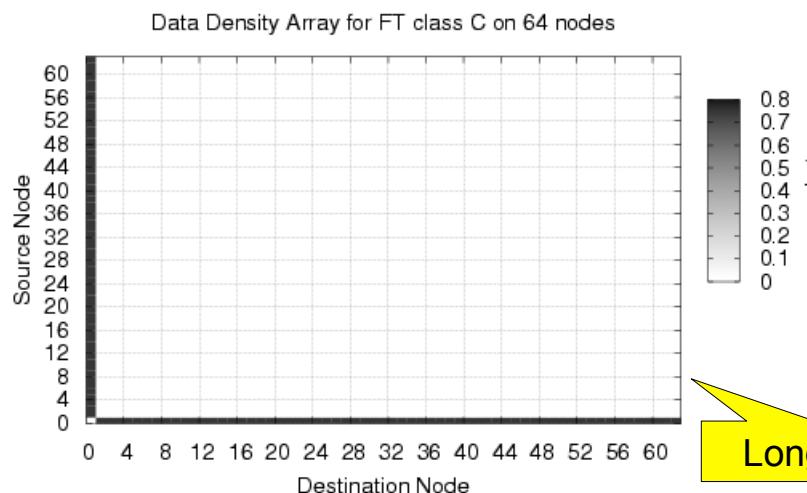
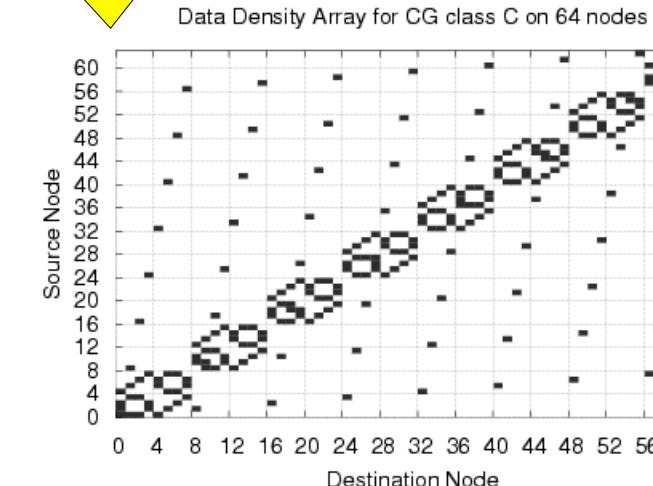
<http://www.cs.sandia.gov/~rolf/NAS/index.html>



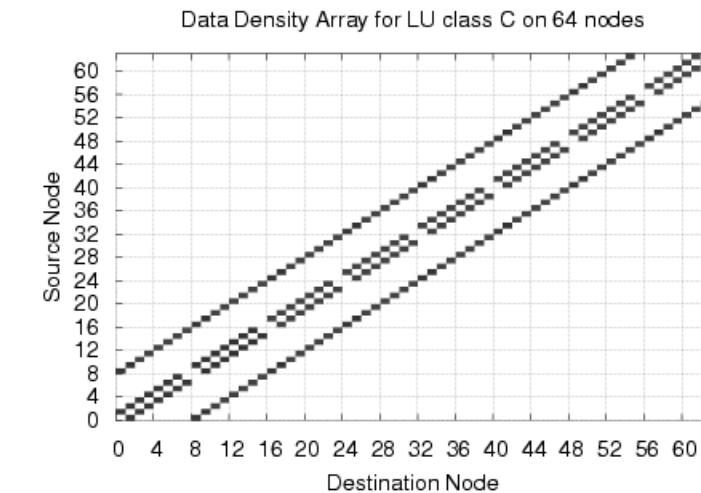
NPB – Data Density Distribution



Unstructured



No significant communication



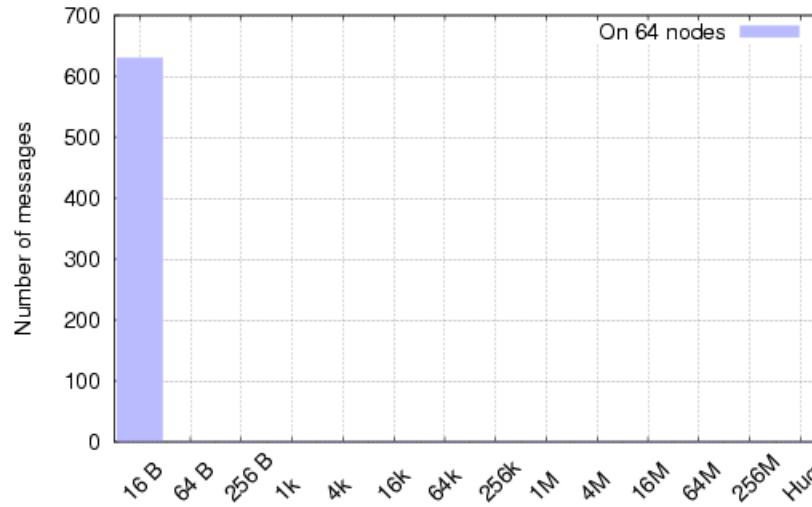
Long range

<http://www.cs.sandia.gov/~rolf/NAS/index.html>

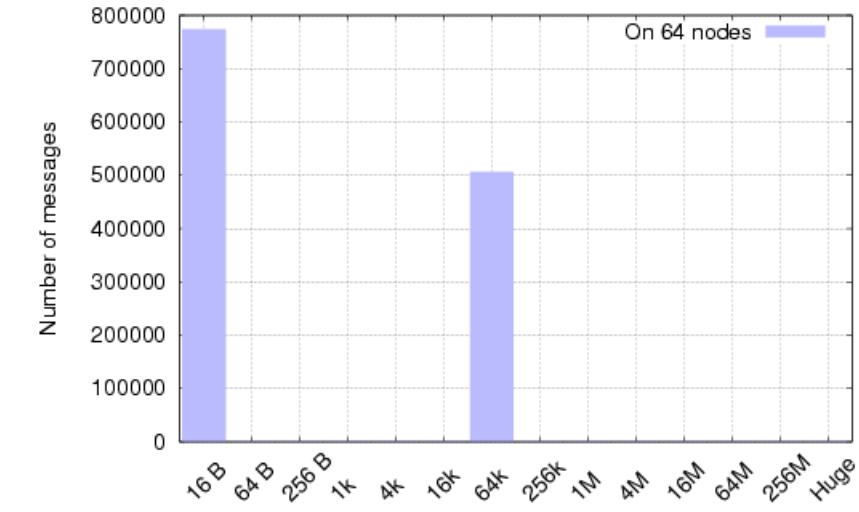


NPB - Message Size Distribution

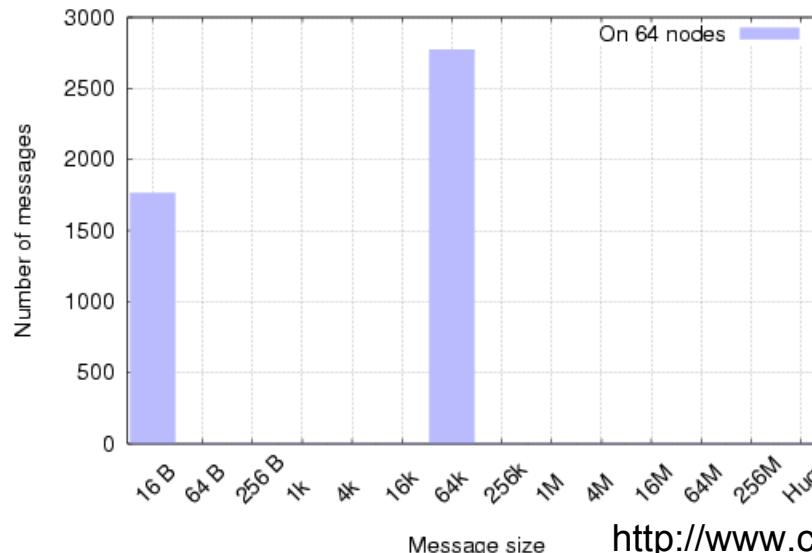
Message sizes used by EP class C on 64 nodes



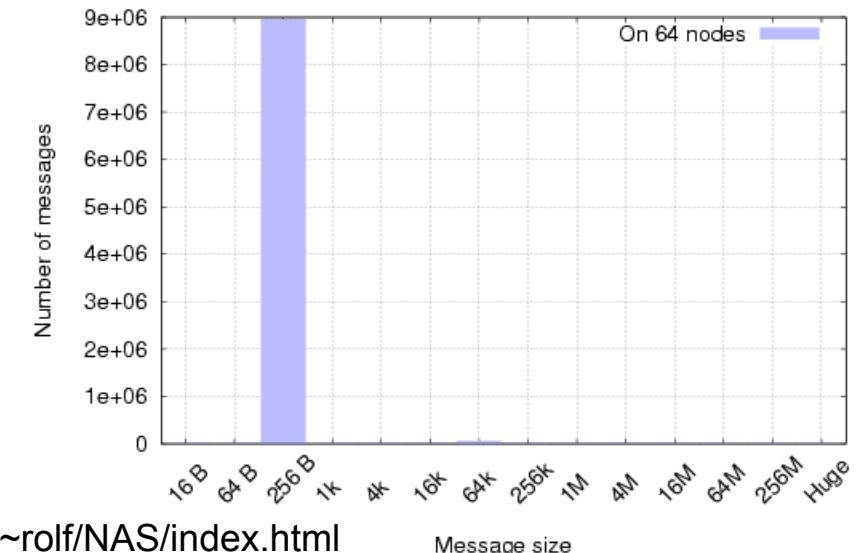
Message sizes used by CG class C on 64 nodes



Message sizes used by FT class C on 64 nodes



Message sizes used by LU class C on 64 nodes

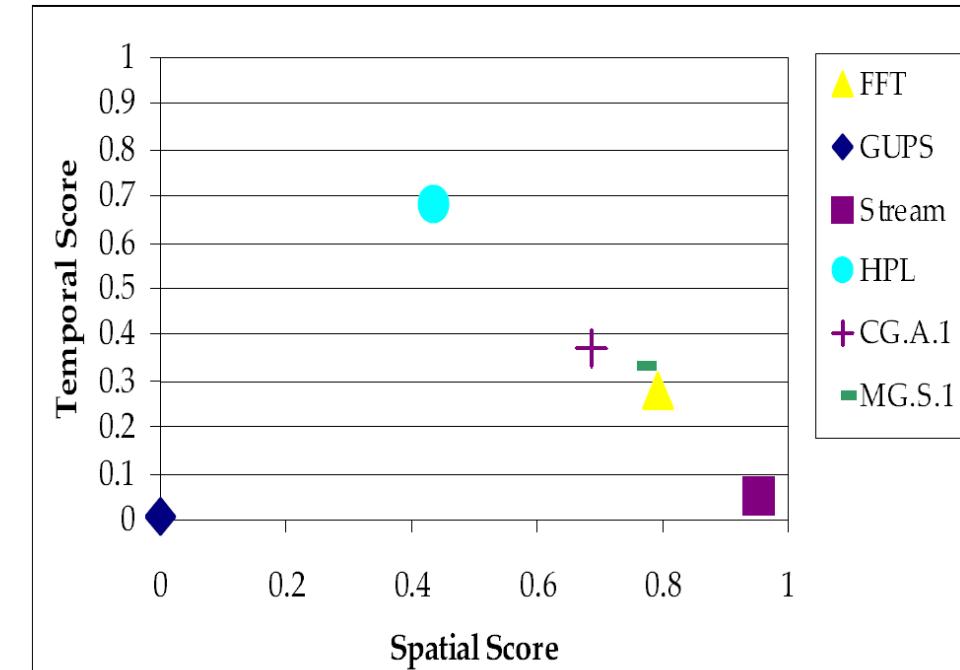


<http://www.cs.sandia.gov/~rolf/NAS/index.html>



HPC Challenge (HPCC)

- <http://icl.cs.utk.edu/hpcc/>
- Suite consisting of 7 tests
 1. **HPL**: well-known
 2. **DGEMM**: matrix multiply
 3. **STREAM**: synthetic benchmark that measures sustainable memory bandwidth (local!)
 4. **PTRANS**: parallel matrix transpose, good indicator for total network capacity
 5. **RandomAccess**: integer random updates
 6. **FFT**: one-dimensional discrete fourier transform
 7. Communication bandwidth and latency



[Weinberg et. al, Quantifying Locality In The Memory Access Patterns of HPC Applications, SC 2005]

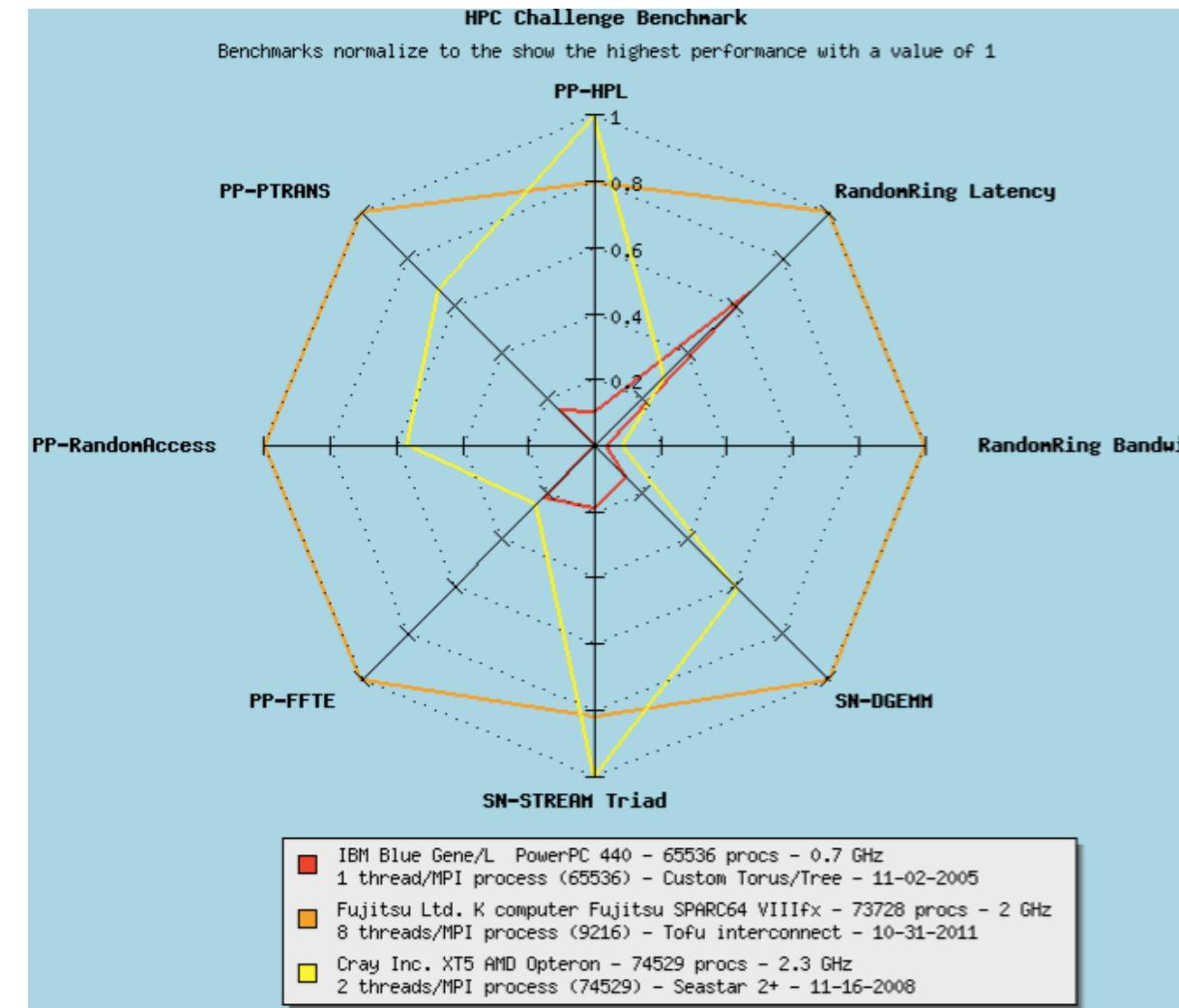
| Benchmark | Spatial locality | Temporal locality | MPI time fraction (64P8N) |
|------------------------|------------------|-------------------|---------------------------|
| HPCC RandomAccess [19] | Low | Low | 89.00% |
| HPCC MPIFFT [19] | Low | High | 77.18% |
| WRF [12] | High | High | 45.00% |



HPC Challenge (HPCC)

■ Result database

- Blue Gene P=64k
- K computer P=74k
- Cray XT5 P=74k





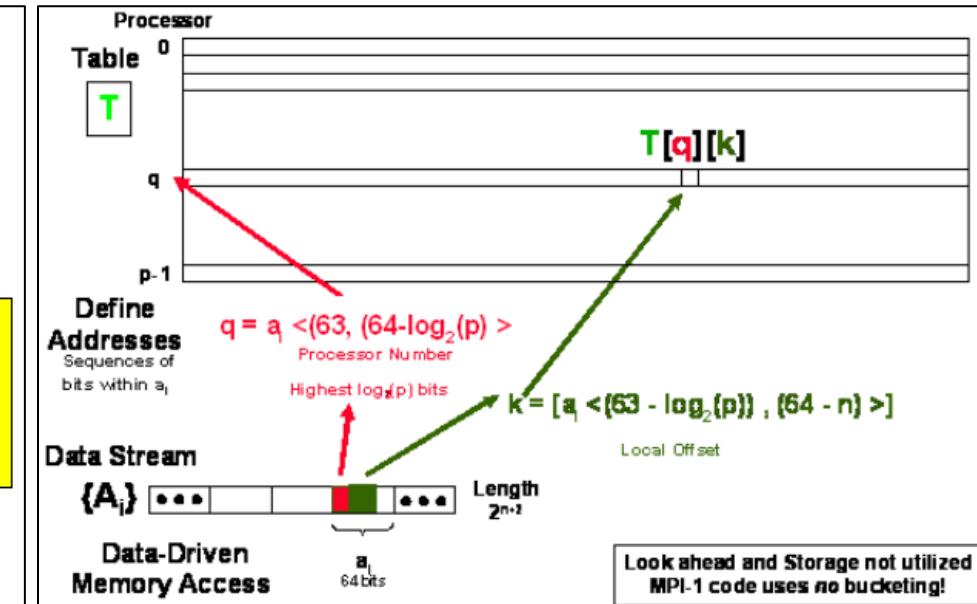
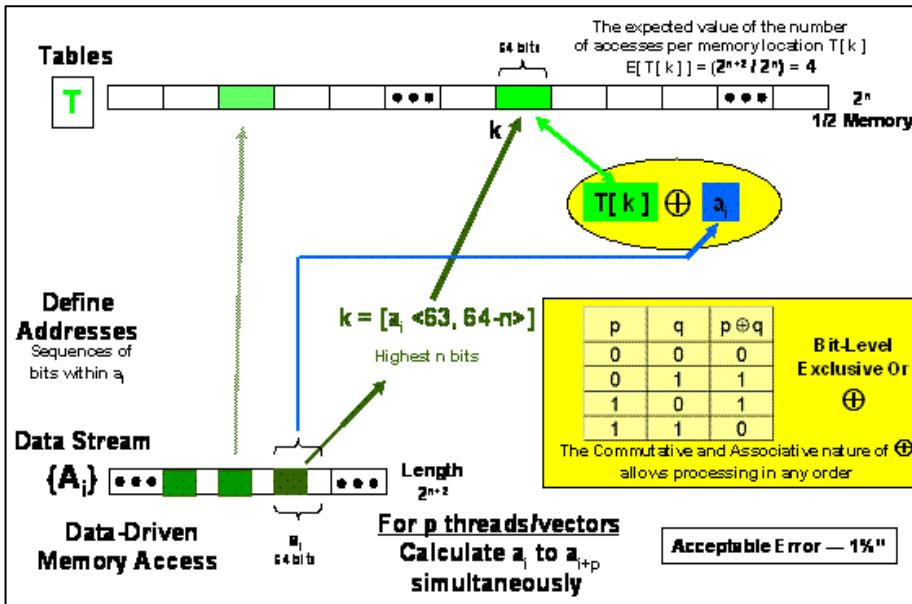
HPC Challenge (HPCC) - RandomAccess

- Random location memory performance

- Current trends favor stride performance at the expense of random access performance

■ Test

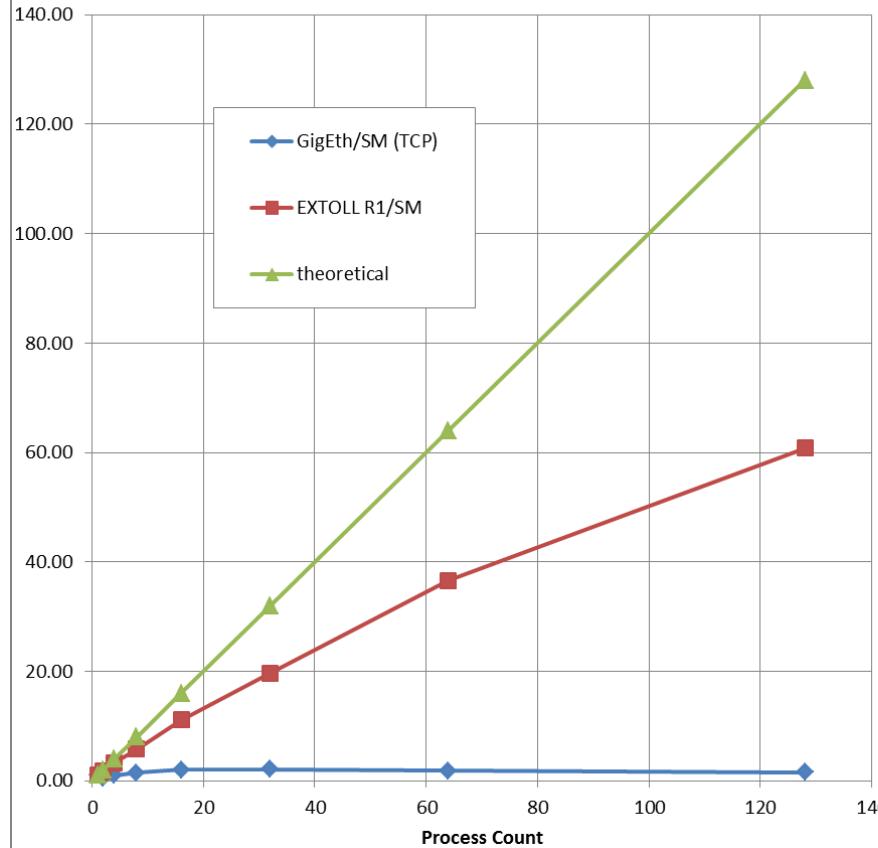
- Distribute table T over p processes
 - Random stream of integers $\{A_i\}$ describes with addresses to update with which value
 - Update is commutative and associative!



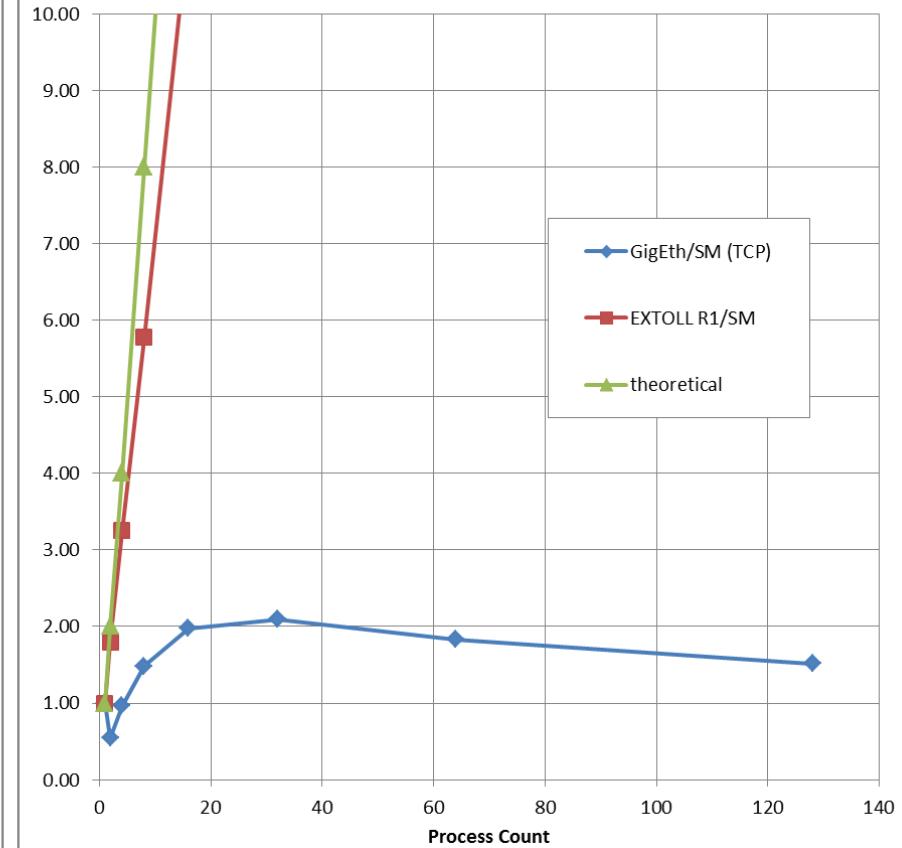


HPC Challenge (HPCC) - RandomAccess

Speed-up of HPCC RandomAccess



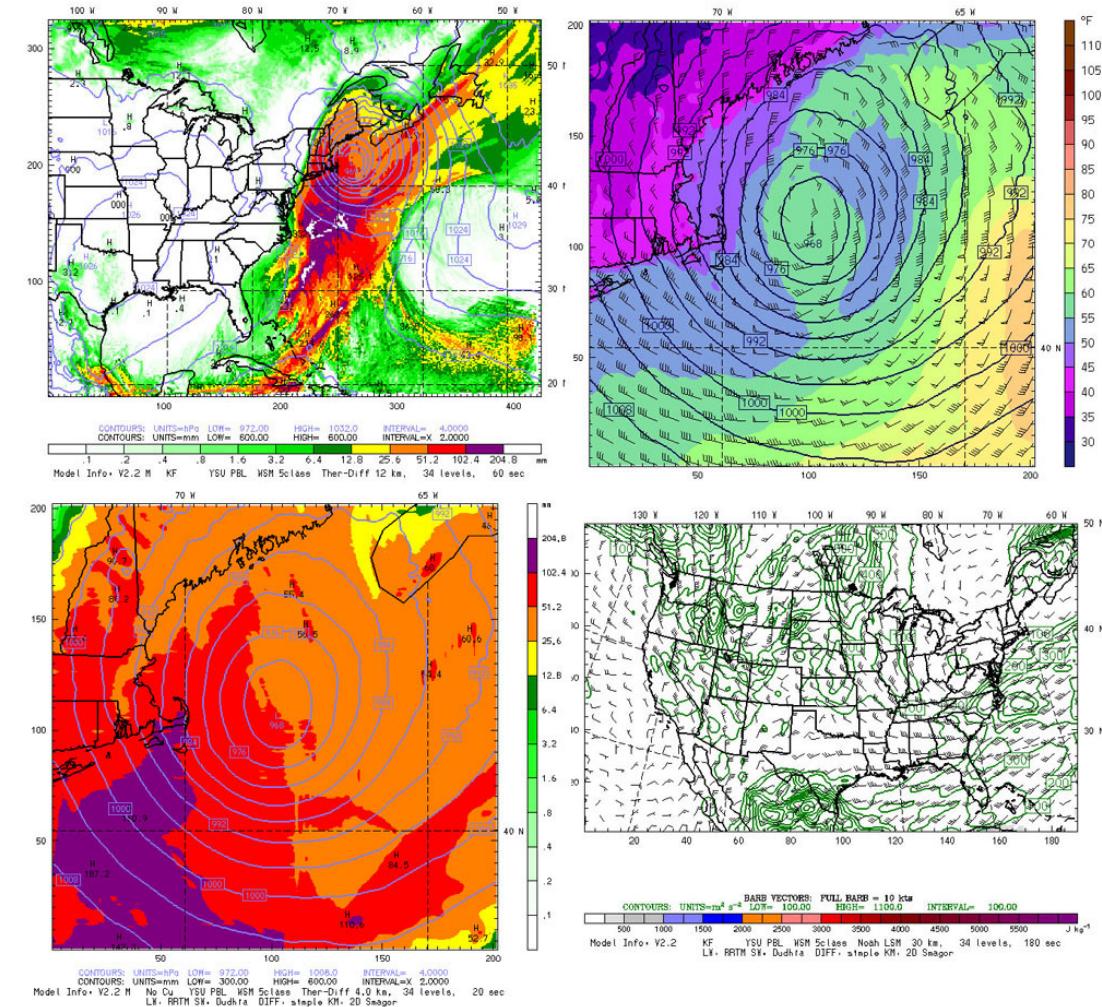
Speed-up of HPCC RandomAccess





Benchmark: Weather Research Forecast (WRF)

- Numerical weather prediction system
 - Operational forecast
 - Atmospheric research
- Weather and climate modeling
- Application and benchmark
 - Standard input data available
 - <http://wrf-model.org>

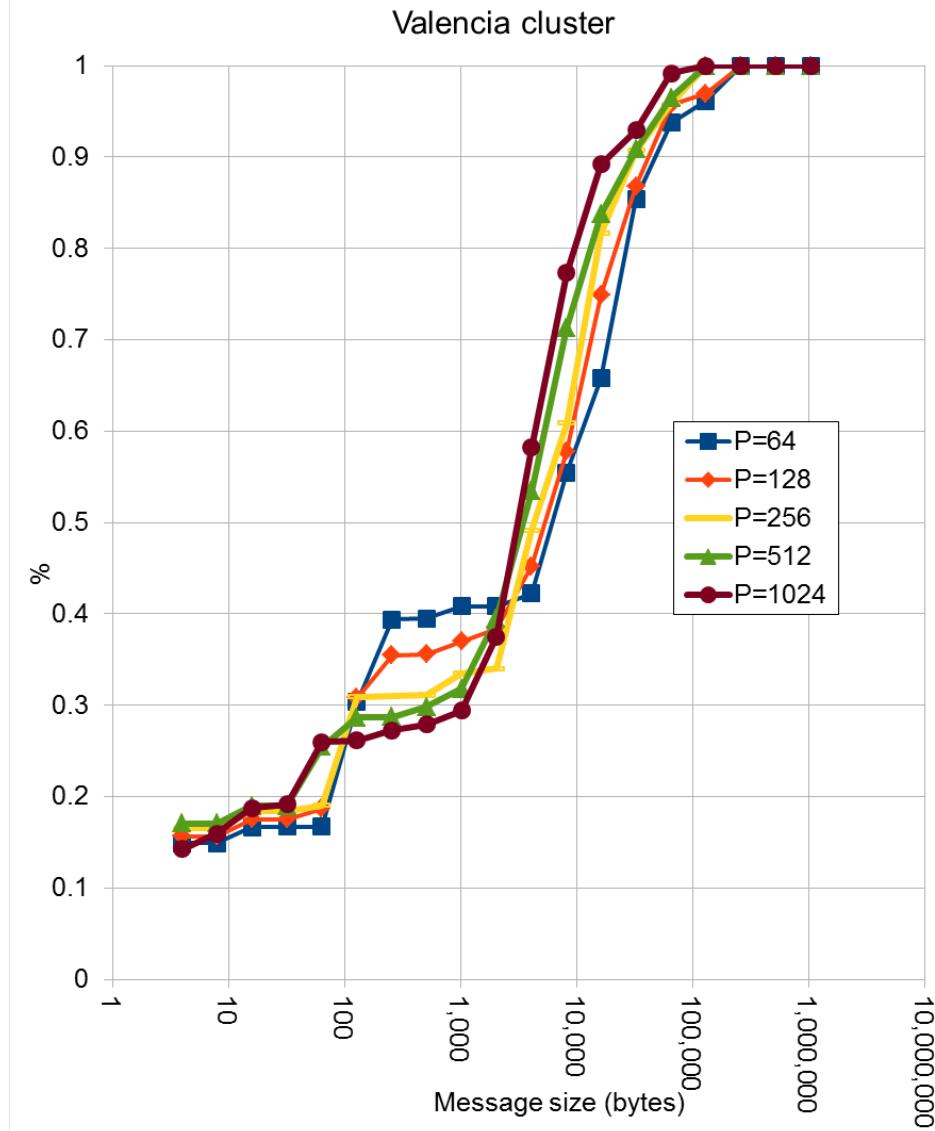


Courtesy: WRF



Benchmark: Weather Research Forecast (WRF)

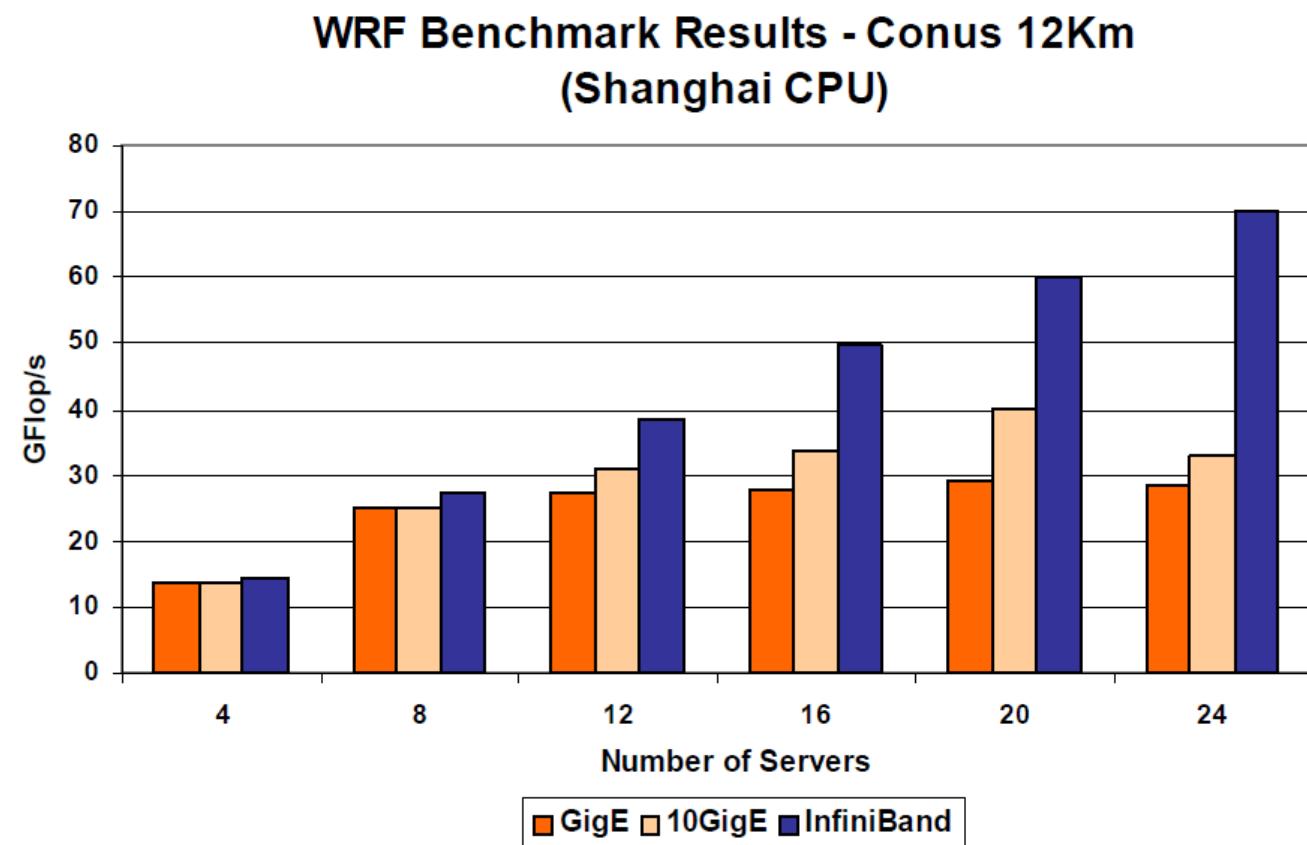
- CONUS input data
- Peak at 256B and 64kB
 - For P=64 about 30% of all messages are smaller than 512B
- Communication dominated by MPI_Bcast (mostly 4B)





Benchmark: Weather Research Forecast (WRF)

- Infiniband DDR
 - 16/20Gbps
- Gigabit Ethernet
 - 1Gbps
- 10 GE
 - 10Gbps
- AMD Shanghai
 - 2 CPUs/node
 - 8 cores/node

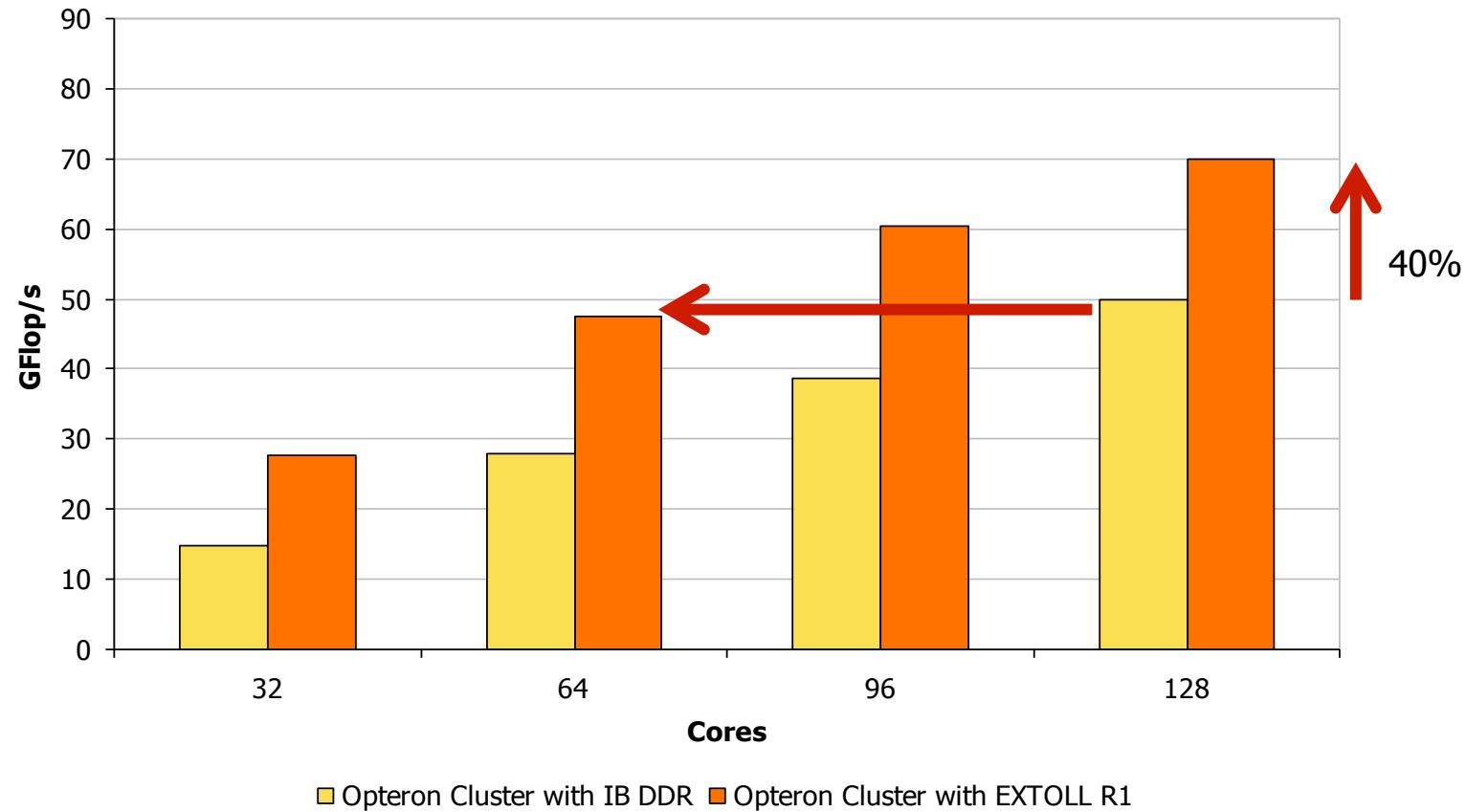


Courtesy: HPC Advisory Council



Benchmark: Weather Research Forecast (WRF)

■ Infiniband DDR vs. EXTOLL R1



IB Cluster: Opteron Shanghai 2358 2.6GHz

EXTOLL Cluster: Opteron Shanghai 2380 2.5GHz



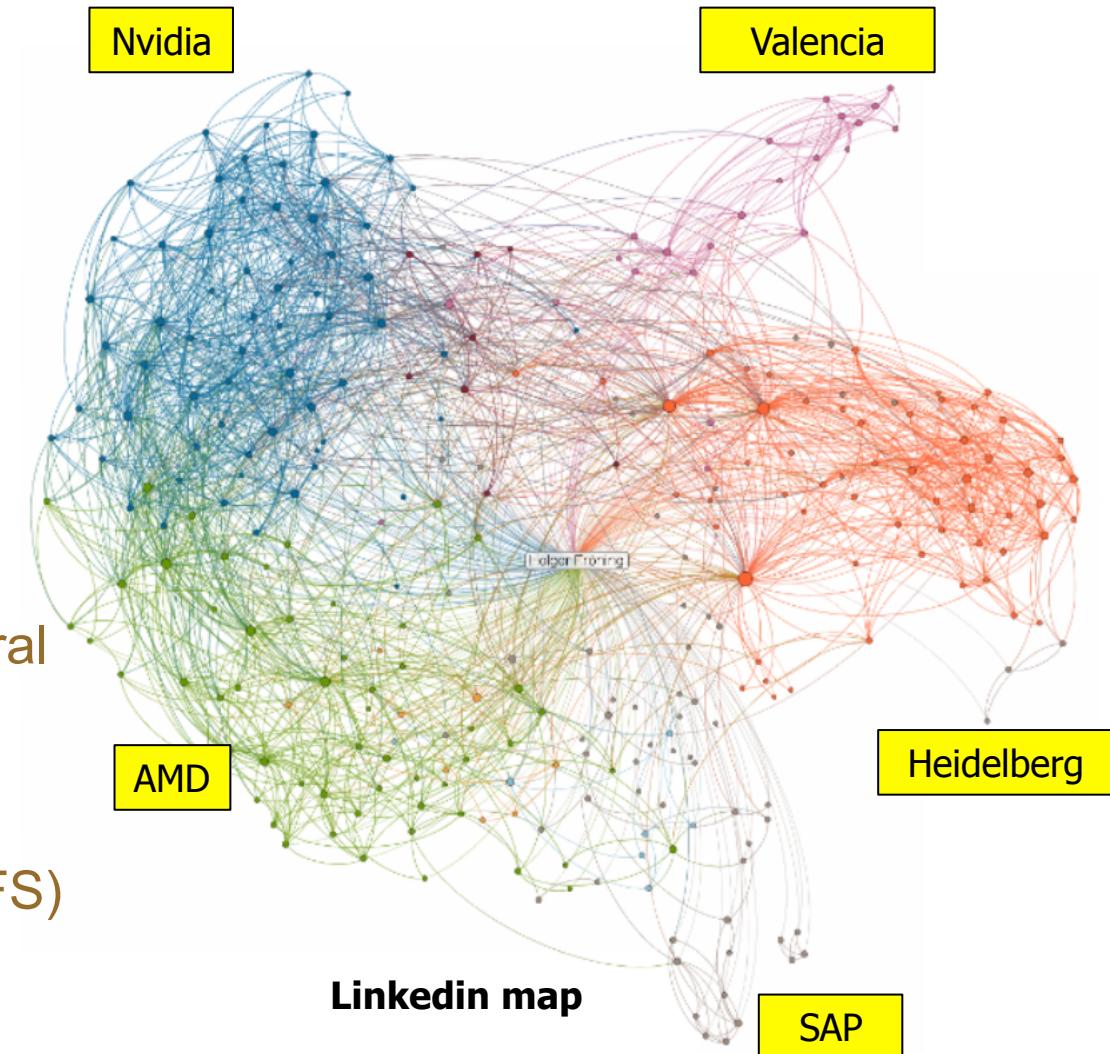
Graph Computations

- **Graphs:** natural representation of unstructured data

- Biology, transportation, internet & power grids, communication data, social media
- Relational data in general

- **Search most common operation**

- Breadth-first search (BFS)





GRAPH500

- Breadth-first search (BFS)
 - Large data sets
 - Irregular access patterns
 - Little reuse and spatial locality
 - Low computational intensity
 - Pointer-chasing requires heavy overlap
 - Prefetching almost useless
- GRAPH500 benchmark
 - Graph data generation (un-timed)
 - Data structure creation (timed)
 - BFS iterations (timed)
 - Result validation (un-timed)

November 2013

| No. | Rank | Machine | Installation Site | Number of nodes | Number of cores | Problem scale | GTEPS |
|-----|------|--|--|-----------------|-----------------|---------------|---------|
| 1 | 1 | DOE/NNSA/LLNL Sequoia (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz) | Lawrence Livermore National Laboratory | 65536 | 1048576 | 40 | 15363 |
| 2 | 2 | DOE/SC/Argonne National Laboratory Mira (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz) | Argonne National Laboratory | 49152 | 786432 | 40 | 14328 |
| 3 | 3 | JUQUEEN (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz) | Forschungszentrum Juelich (FZJ) | 16384 | 262144 | 38 | 5848 |
| 4 | 4 | K computer (Fujitsu - Custom supercomputer) | RIKEN Advanced Institute for Computational Science (AICS) | 65536 | 524288 | 40 | 5524.12 |
| 5 | 5 | Fermi (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz) | CINECA | 8192 | 131072 | 37 | 2567 |
| 6 | 6 | Tianhe-2 (MilkyWay-2) (National University of Defense Technology - MPP) | Changsha, China | 8192 | 196608 | 36 | 2061.48 |
| 7 | 7 | Turing (IBM - BlueGene/Q, Power BQC 16C 1.60GHz) | CNRS/IDRIS-GENCI | 4096 | 65536 | 36 | 1427 |
| 8 | 7 | Blue Joule (IBM - BlueGene/Q, Power BQC 16C 1.60 GHz) | Science and Technology Facilities Council - Daresbury Laboratory | 4096 | 65536 | 36 | 1427 |

<http://www.graph500.org>



Summary

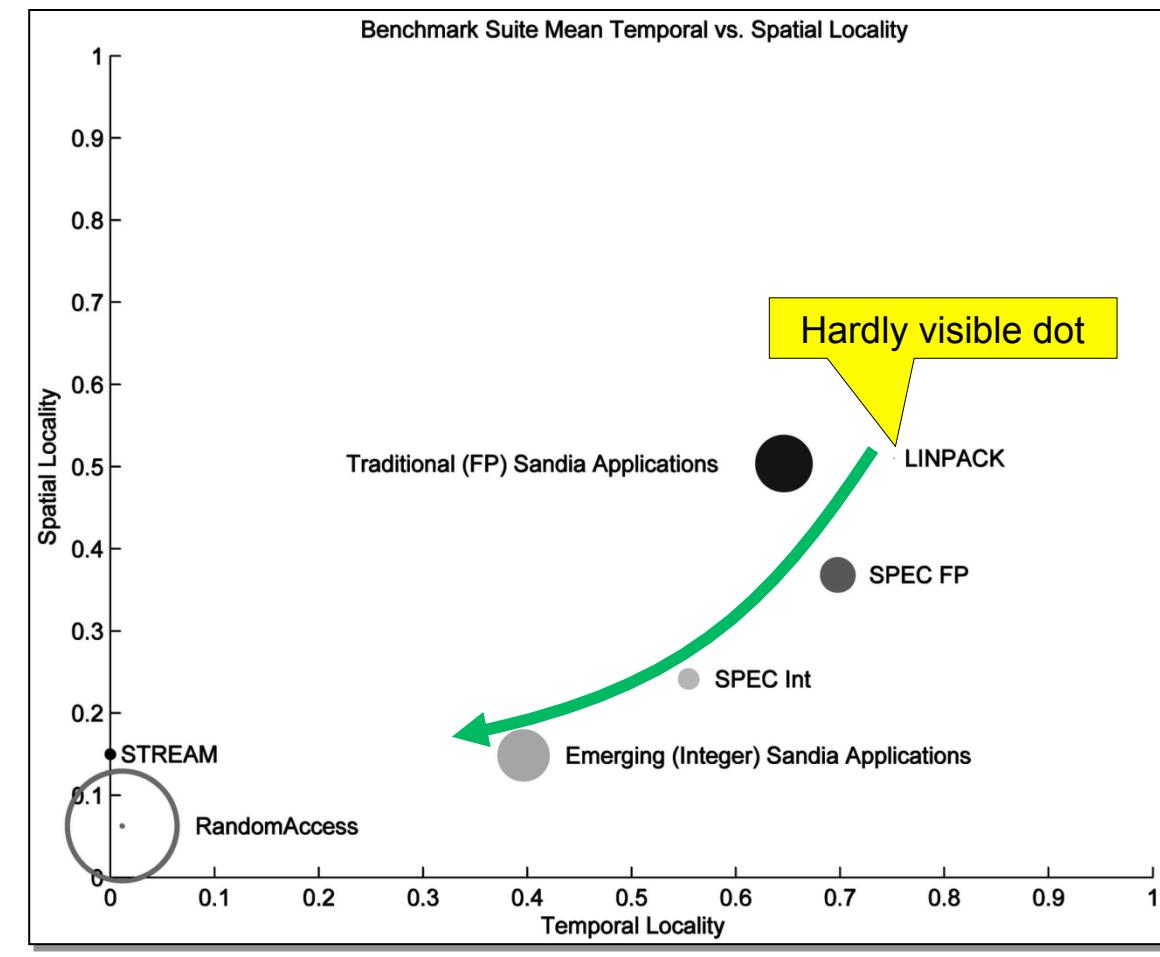
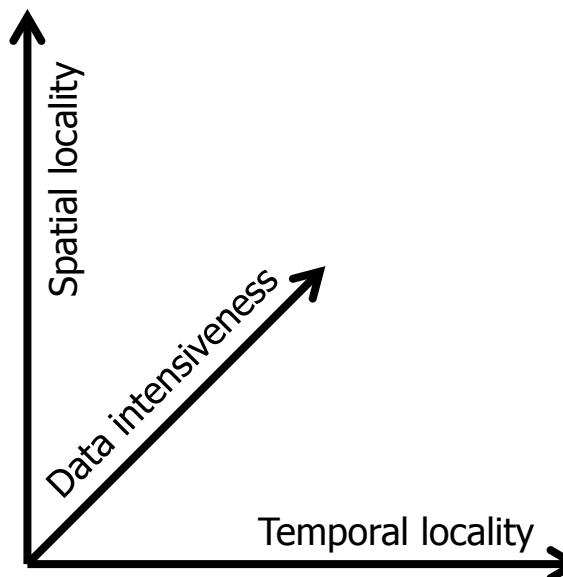
- Benchmarking is complex...
- The best choice always depends on the application
 - MPI time is a nice measure for a first impression, but highly depends on IN/implementation
- Micro-benchmarks help to classify systems, but they cannot substitute real assessment
- Application characterization: payload size, communication pattern, ...
 - Differences between messaging layer and functional layer
- Big Data?

| Test | MPI Time |
|--------------|----------|
| NPB-EP | 0% |
| NPB-MG | 9% |
| NPB-BT | 10% |
| NAMD | 24% |
| HPL | 25% |
| NPB-CG | 34% |
| NPB-FT | 37% |
| NPB-IS | 47% |
| WRF | 45% |
| MPIFFT | 77% |
| RandomAccess | 89% |



Future – Big Data

- Data intensiveness = number of unique bytes accessed
 - Size of the circles





Message size/frequency trend

frequent small transfers

few bulk transfers

- Parallelism degree dramatically increasing
 - Data distribution
 - Synchronization
- Trend towards graph algorithms
 - Complex, unstructured patterns
- Optimization of MPI programs
 - Need to rely on bulk transfers to expose overhead
 - Not native to the problem
- Big Data era
 - Massive data set sizes



Introduction to High Performance Computing

Lecture 02 – CUDA Programming

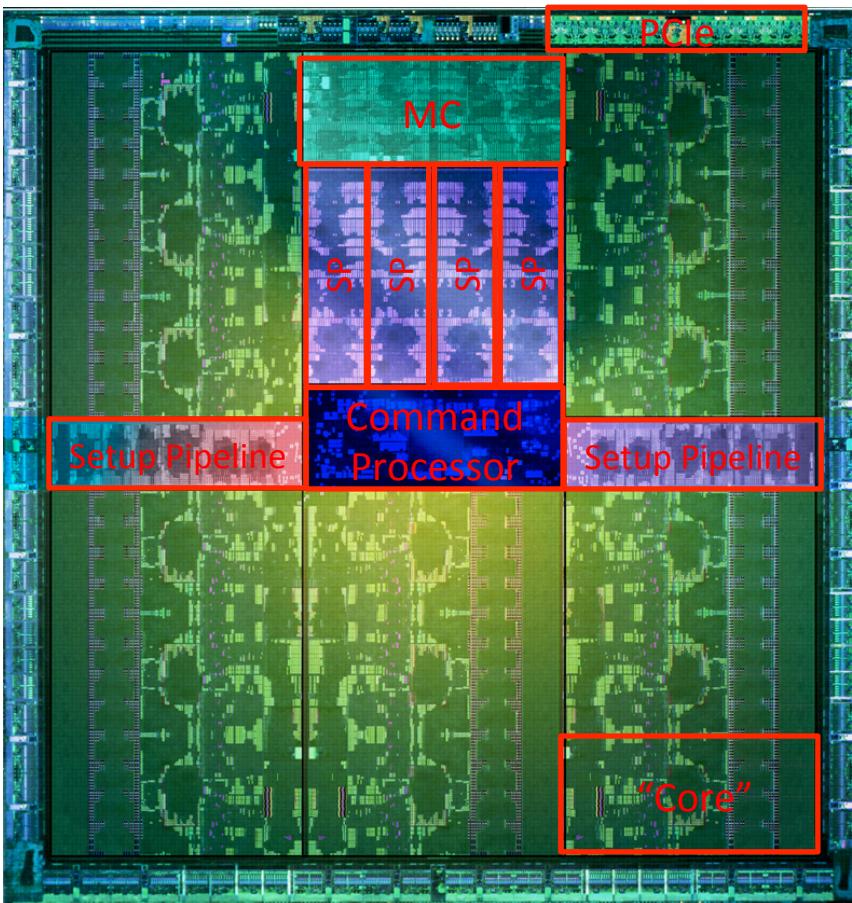
Holger Fröning
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

*With material from D. Kirk, W. Hwu („Programming Massively Parallel
Processors“)*

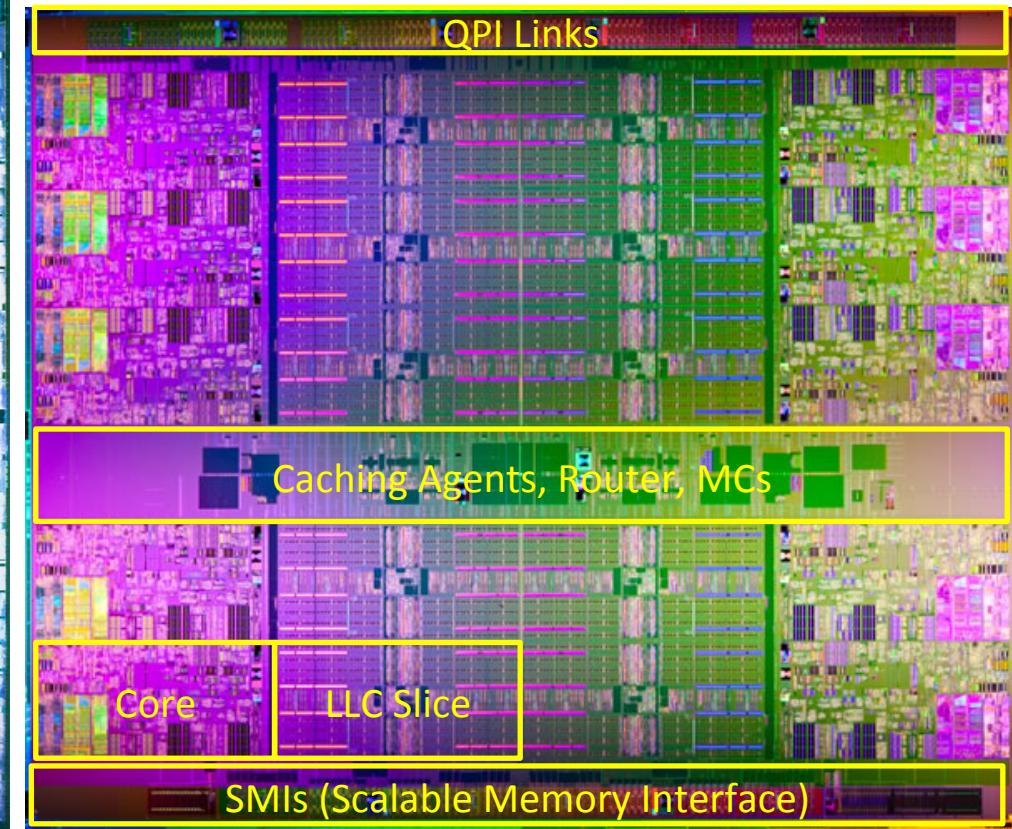


Introduction

GK110



XEON-E7





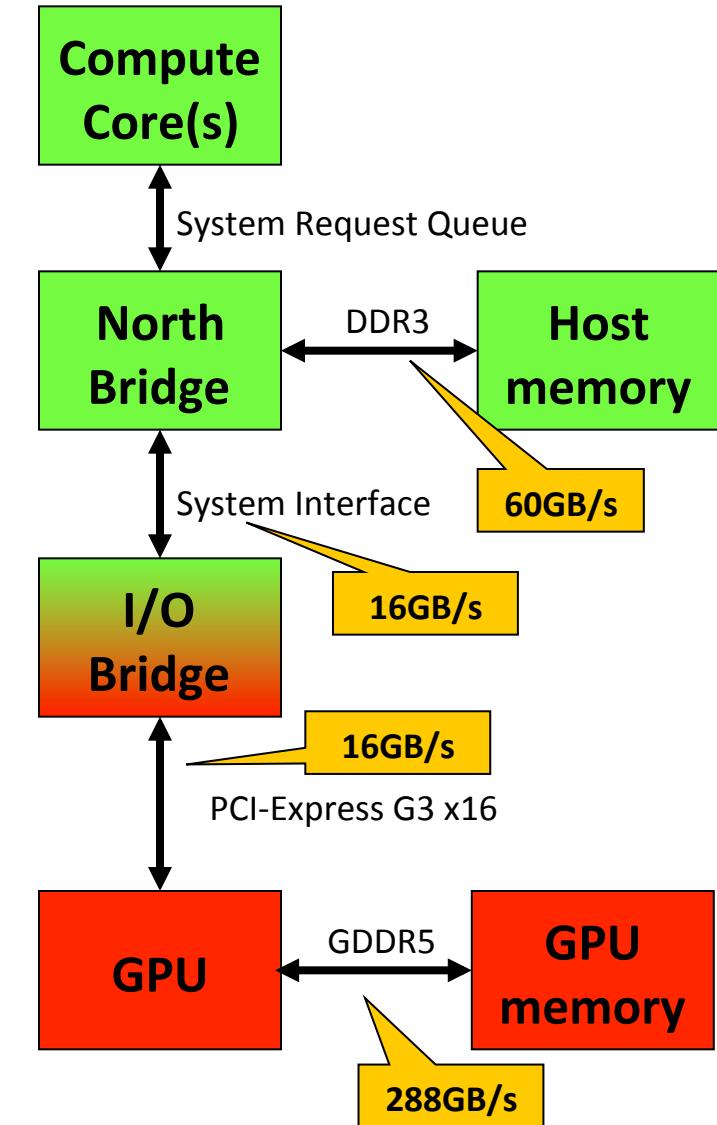
GPUs vs. CPUs

| | Tesla K20 | Xeon E7-4800 4P |
|---------------------------------|--|----------------------------|
| Core count | 13 SMs 64/832 (DP), 192/2,496 (SP) | 10 Cores 2 FP-ALUs/core |
| Frequency | 0.7GHz | 2.4GHz |
| Peak Compute Performance | 1,165 GFLOPS (DP) 3,494 GFLOPS (SP) | 96 GFLOPS (DP) |
| Use model | throughput-oriented | latency-oriented |
| Latency treatment | toleration | minimization |
| Programming | 1000s-10,000s of threads | 10s of threads |
| Memory bandwidth | 250 GBytes/sec | 34 GByte/s (per P) |
| Memory capacity | <= 8 GB | up to 2TB |
| Die size | 550mm ² | 684 mm ² |
| Transistor count | 7.1 billion | 2.3 billion |
| Technology | 28nm | 32nm |



Introduction

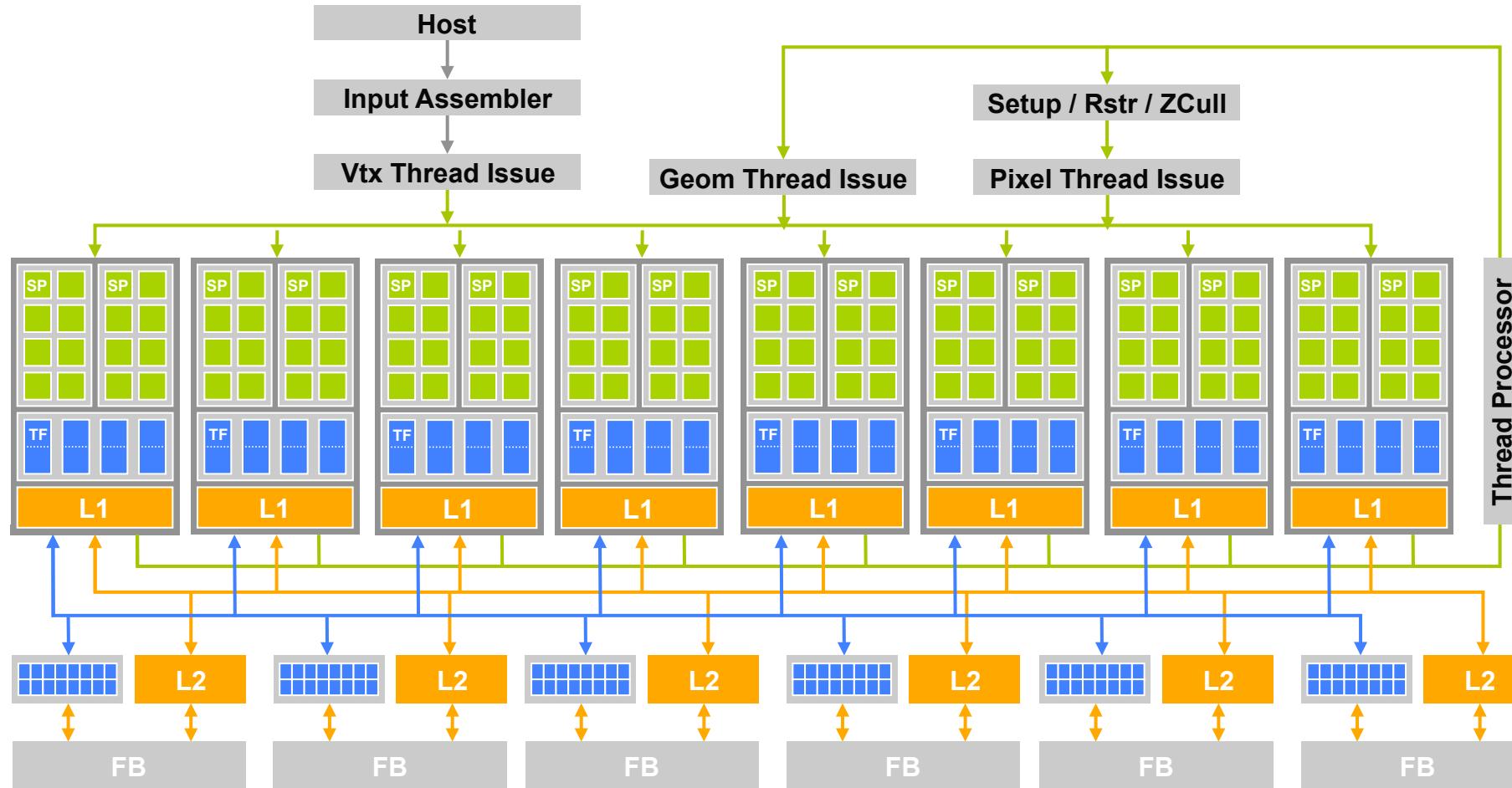
- NVidia CUDA
 - Compute kernel as C program
 - Explicit data- and thread-level parallelism
 - Computing, not graphics processing
 - Host communication
- Memory hierarchy
 - Registers at thread level
 - Shared memory at thread block level
 - GPU memory
 - Host memory
- More HW details exposed
 - Use of pointers
 - Load/store architecture
 - Barrier synchronization of thread blocks
- Requires more detailed understanding of underlying HW





Introduction

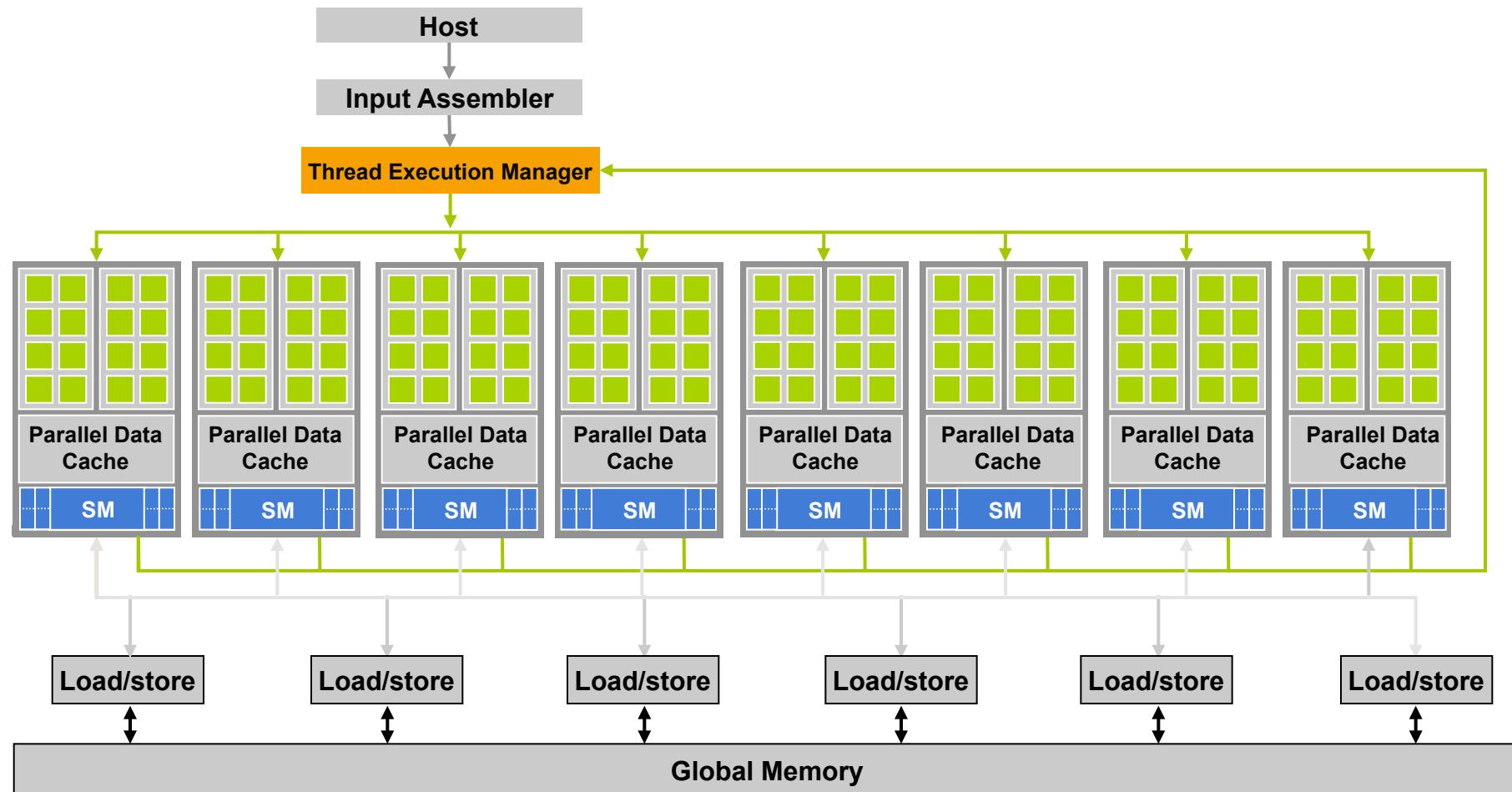
■ G80 architecture for graphic processing





Introduction

- G80 architecture for general-purpose computing



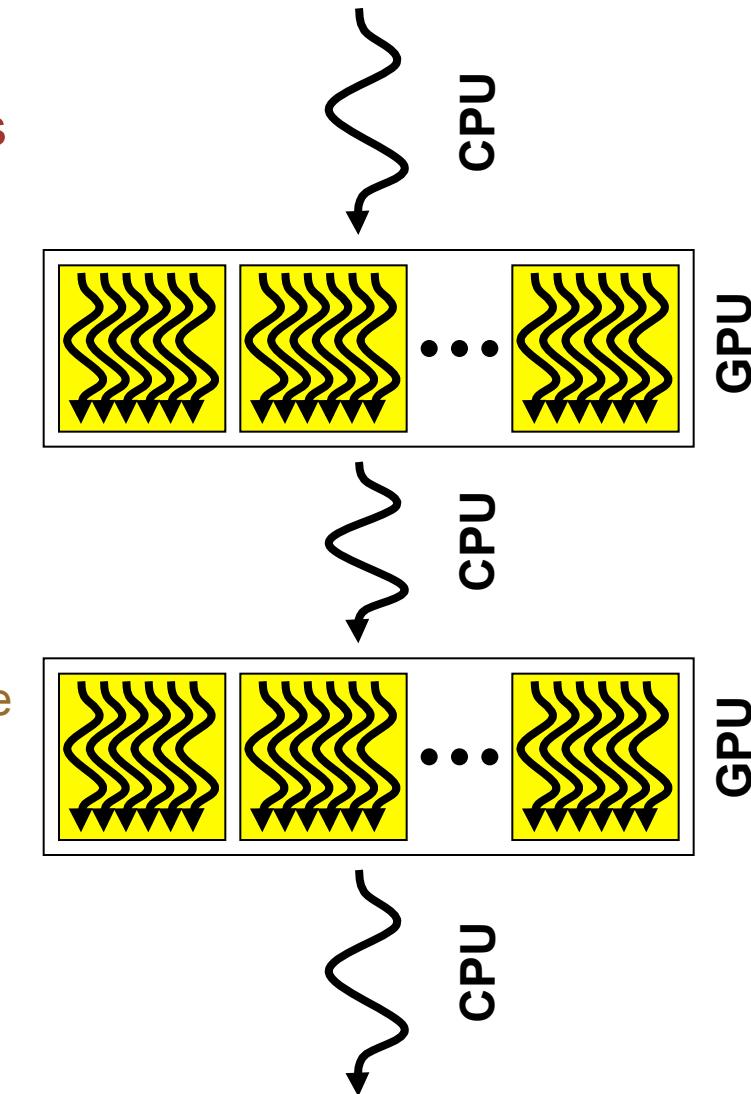


CUDA Programming Model



Programming Model

- C Extension with three main abstractions
 1. Hierarchy of threads
 2. Shared memory
 3. Barrier synchronization
- Exploiting parallelism
 - Fine-grain DLP
 - TLP
- CUDA program consists of CPU & GPU part
 - CPU part: part of the programm with no or little parallelism
 - GPU part: high parallel part, SPMD-style
- Concurrent execution
 - Non-blocking thread execution
 - Explicit synchronization





Programming Model

```
__global__ void matAdd (float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation
    dim3 dimBlock ( N, N );
    matAdd <<< 1, dimBlock >>> ( A, B, C );
}
```



▪ Kernels: n-fold execution by N threads

- Definition: keyword `__global__`

▪ Execution: `kernel <<<NumBlocks, threadsPerBlock>>> (args)`

▪ (Unique) ID: `threadIdx`

- Control flow for SPMD programs
- Memory access orchestration





Programming Model

```
__global__ void matAdd (float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation
    dim3 dimBlock ( N, N );
    matAdd <<< 1, dimBlock >>> ( A, B, C );
}
```

- `threadIdx` has up to 3 dimensions: `threadIdx.{x, y, z}`
- ➔ Each thread block has up to 3 dimensions
- Number of threads per block is limited
 - $512 \times 512 \times 64 \rightarrow 1024 \times 1024 \times 64$ (implementation dep.)
- ➔ Additional hierarchy level: grid = blocks of threads
 - Unique ID `blockIdx`, up to 3 dimensions
 - Blocks are executed independently and implementation-dependent
 - Number of blocks limited (typ. $64k-1$ per dimension)



up to 3D



Programming Model

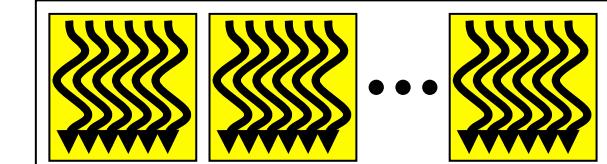
```
__global__ void matAdd (float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if ( i < N && j < N )
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation
    dim3 dimBlock ( 16, 16 );
    dim3 dimGrid  ( ( N + dimBlock.x - 1 ) / dimBlock.x,
                    ( N + dimBlock.y - 1 ) / dimBlock.y );
    matAdd <<< dimGrid, dimBlock >>> ( A, B, C );
}
```

- Operator “/” rounds down, so add block size to round up!
- E.g. N=50:
- grid size = $(50+16-1)/16=4.0625 \Rightarrow 4$

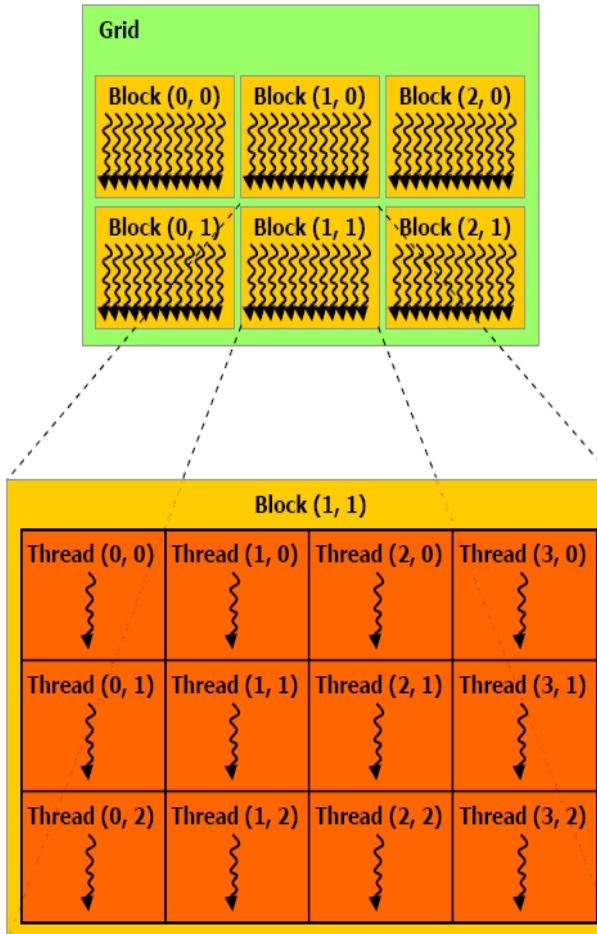
Block size

Fine grain PCAM
One thread computes
one element





Programming Model



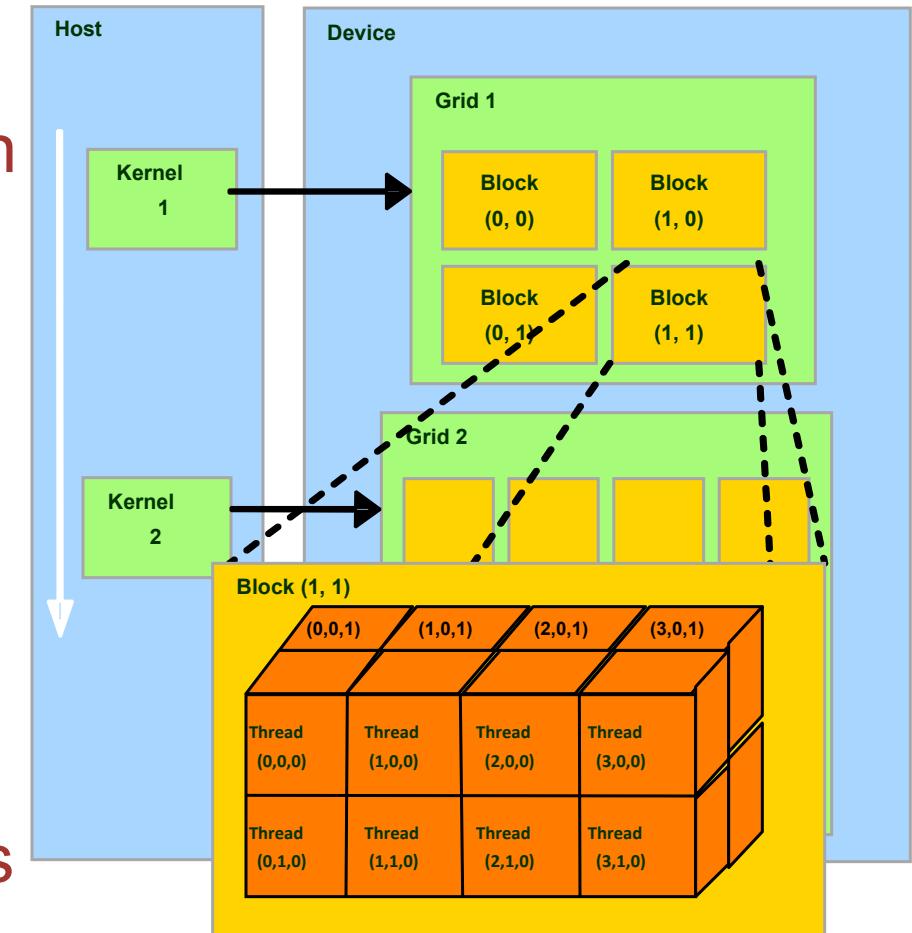
- **Thread hierarchy**
 - Grid of thread blocks
 - Blocks of equal size
- Given problem size N, how to choose parameters threads per block respectively blocks per grid?
- **Recommendations wrt block count**
 - >2x number of SMs
 - Optimal: 100 – 1000 (max. 64k-1)
- **Recommendations wrt threads/block**
 - Parallel slackness vs. number of registers per thread
 - $R / (B * \text{ceil}(T, 32))$

R = total registers, B = active blocks / SM, T = threads per block, ceil = round up to next multiple of 32
R = 8k/SM .. 32k/SM (implementation dependent)



Programming Model

- Communication and synchronization only within one thread block
 - Shared memory
 - Atomic operations
 - Barrier synchronization
- Threads from different blocks cannot interact
 - Exception: global memory
 - Very weak coherence & consistency guarantees
- Iterative kernel invocations





Memory Hierarchy – Global Memory

▪ Global memory

- Communication between host and device
- Accessible from all threads
 - Read/write
- High associated latency (no caching)
- Lifetime exceeds thread lifetime

▪ Sensitive to coalescing

▪ Allocation

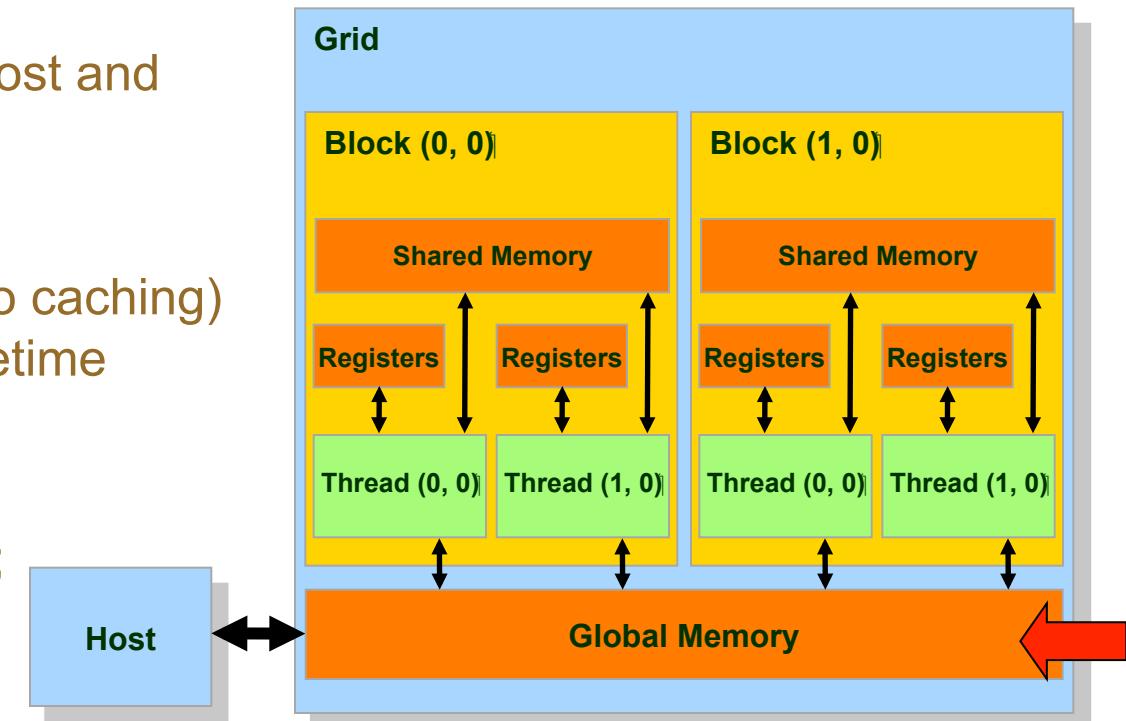
- `cudaMalloc (&dmem, size);`

▪ De-allocation

- `cudaFree (dmem);`

▪ Data transfer (blocking)

- `cudaMemcpy (*dst, *src, size, transfer_type);`
- `cudaMemcpyAsync (...)`





Memory Hierarchy – Global Memory

Variable scope annotation

```
void *dmem = cudaMalloc ( N*sizeof ( float ) ); // Allocate GPU memory

void *hmem = malloc ( N*sizeof ( float ) ); // Allocate CPU memory

// Transfer data from host to device
cudaMemcpy ( dmem, hmem, N*sizeof ( float ), cudaMemcpyHostToDevice );

// Do calculations
kernel1 <<< numBlocks, numThreadsPerBlock >>> ( dmem, N );
...
kernel2 <<< numBlocks, numThreadsPerBlock >>> ( dmem, N );

// Transfer data from device to host
cudaMemcpy ( hmem, dmem, N*sizeof ( float ), cudaMemcpyDeviceToHost );

cudaFree ( dmem );           // Free device buffer
free ( hmem );      // Free host buffer
```

Only references
to device
memory



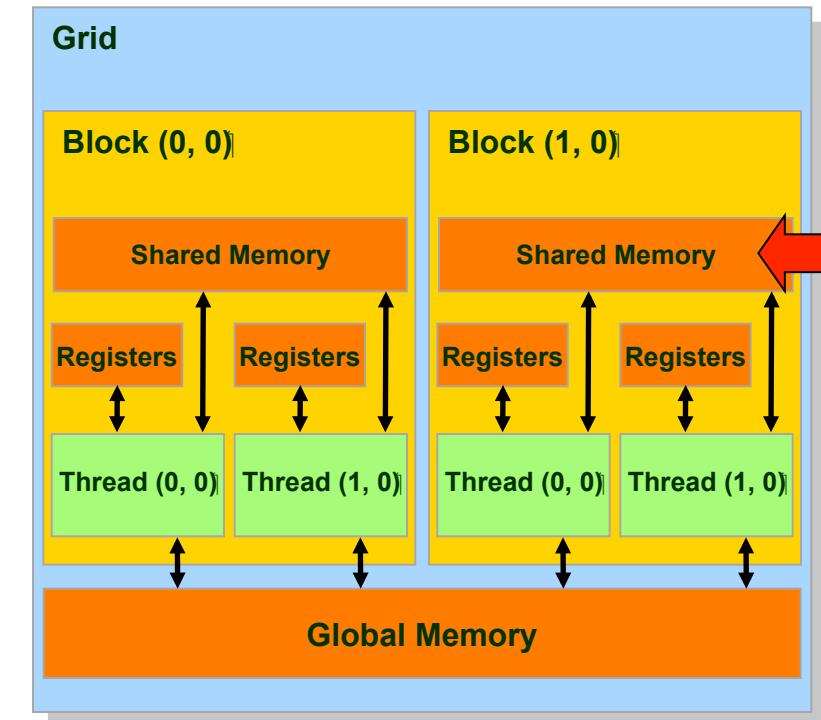
Memory Hierarchy – Shared Memory

■ Shared Memory

- On-chip memory
- Lifetime: thread lifetime

■ Access costs in the best case equal register access

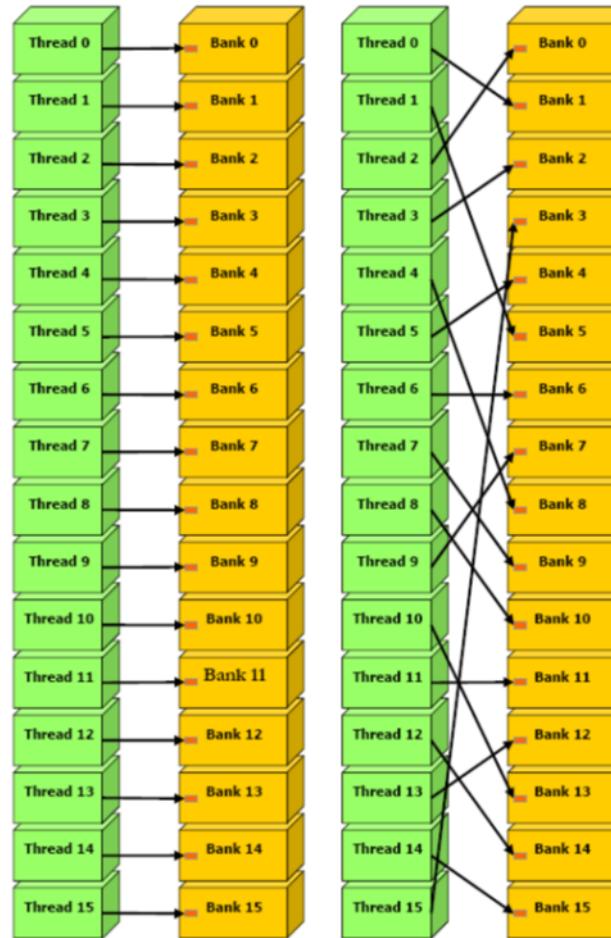
- Organized in n banks
 - Typ. 16-32 banks with 32bit width
 - Low-order interleaving
- Parallel access if no conflict
- Conflicts result in access serialization



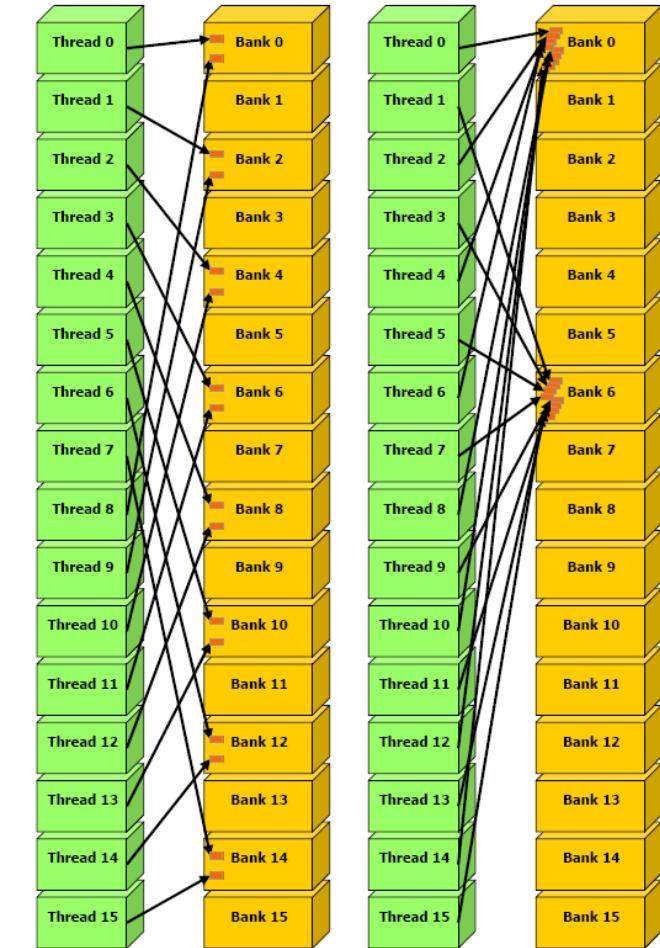


Memory Hierarchy – Shared Memory

Shared Memory
Bank Access without Blocking



Shared Memory
Bank Access with Blocking





Memory Hierarchy - Declaration

| | Location | Access from |
|--|-----------------|--------------------|
| <code>__device__ float var;</code> | global mem | device/host |
| <code>__constant__ float</code> <code>var;</code> | constant mem | device/host |
| <code>__shared__ float var;`</code> | shared mem | block |

- **device** can be combined with one additional type
 - Otherwise placed in global memory, application lifetime
- **constant** implies **device**, see above
- **shared** implies **device**
 - Lifetime of a block, only available within this block
 - **syncthreads** to wait for outstanding write operations
 - Unconstrained completion of read/write operations (exception: **volatile**)



■ Vector types

- char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1, float2, float3, float4, double2
- Derived from basic types (int, float, ...)

■ Dimension type: dim3

- Based on uint3
- Unspecified components are initialized with 1



Function Declarations

| | Executed on | Callable from |
|---|-------------|---------------|
| <code>__device__ float</code> | device | device |
| <code>DeviceFunc()</code> <code>__global__ void KernelFunc()</code> | device | host |
| <code>__host__ float HostFunc()</code> | host | host |

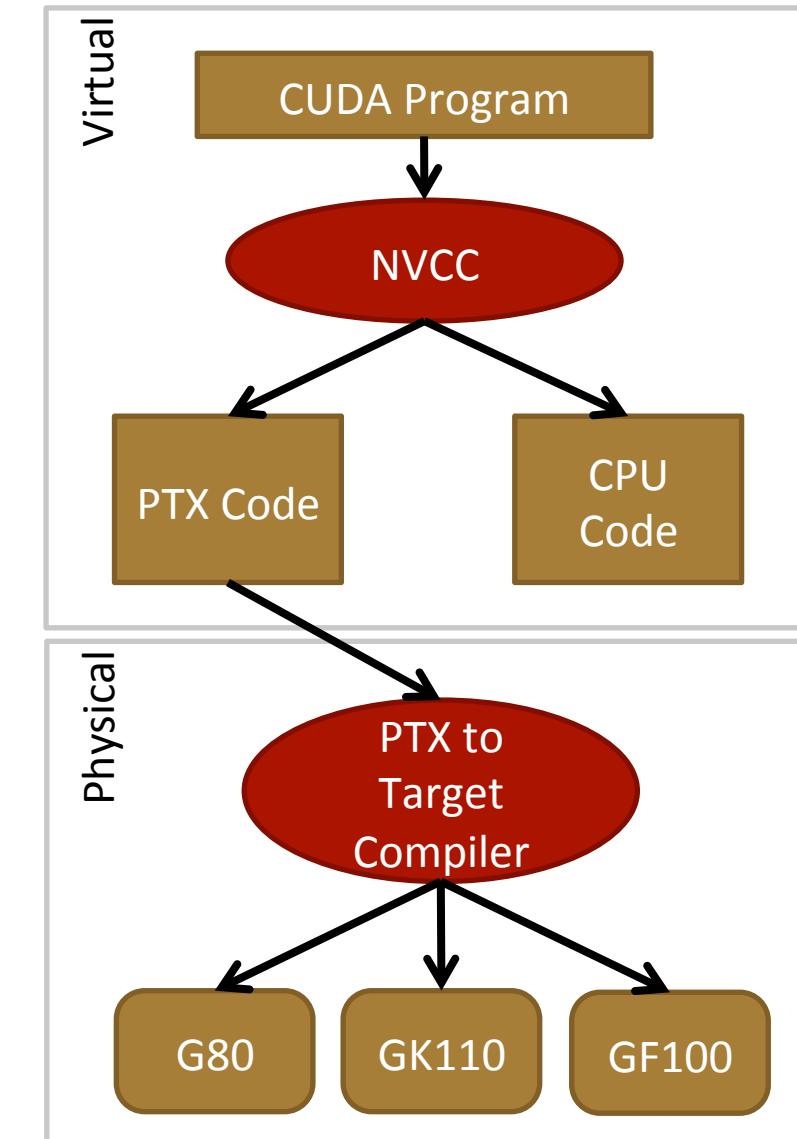
- `__global__` defines a kernel (return type: `void`)
 - Rückgabewert vom Typ `void`
- `__host__` is optional
- `__host__` and `__device__` can be combined
- No pointers to `__device__` functions
 - Exception: `__global__` functions
- For functions that are executed on the GPU:
 - No recursions, only static variable declarations, no variable parameter count



Just-in-time Compilation

- Device code only supports C-subset of C++
- Compile with nvcc
 - Compiler Driver
 - Calls other required tools
 - cudacc, g++, cl, ...
 - Debugging:
command line parameter -deviceemu
- Output
 - C code (host CPU Code)
 - Either PTX object code, or source code for run time interpretation
- PTX (Parallel Thread Execution)
 - Virtual Machine and ISA
 - Execution resources and state
- Linking
 - CUDA runtime library cudart
 - CUDA core library cuda

```
export LD_LIBRARY_PATH=/usr/local/cuda/lib
export PATH=$PATH:/usr/local/cuda/bin
nvcc --help
nvcc foo.cu -o foo
```





Brief Property Survey (deviceQuery)

| Model | Concurrent copy and execution | Clock rate [GHz] | Max. memory pitch [bytes] | Max. dimension of a grid | Max. dimension of a block | Threads per block | Warp size | Registers per block | Shared memory per block [bytes] | Total constant memory [bytes] | Cores | Multiprocessors | Total global memory [bytes] | Revision number |
|-----------------|-------------------------------|------------------|---------------------------|--------------------------|---------------------------|-------------------|-----------|---------------------|---------------------------------|-------------------------------|-------|-----------------|-----------------------------|-----------------|
| GeForce GTX 280 | Y 1 | 1.3 | 256k | 65535 x 65535 x 1 | 512 x 512 x 64 | 512 | 32 | 16k | 64k | 16k | 240 | 30 | 1G | 1.3 |
| GeForce GTX 480 | Y 1 | 1.4 | 2G | 65535 x 65535 x 65535 | 1k x 1k x 64 | 1k | 32 | 32k | 48k | 48k | 480 | 15 | 1.5G | 2.0 |
| Tesla K20c | Y 2 | 0.7 | 2G | 2G x 65535 x 65535 | 1k x 1k x 64 | 1k | 32 | 64k | 48k | 64k | 2496 | 13 | 5G | 3.5 |



CUDA Example: saxpy



SAXPY Example

- SAXPY: Scalar Alpha X Plus Y
- Simple test to compare the GPU and the CPU
 - Objective: runtime reduction
 - Max. Gridsize * threadsPerBlock elements
 - $65535 \times 1k \rightarrow \sim 64M$ elements
 - Memory requirement = $32M$ elements * 2 arrays * 4 Byte/element = $256MB$
- Source code contains kernels for the GPU and the CPU

Scalar Alpha X Plus Y
 $y[i] = alpha*x[i] + y[i]$



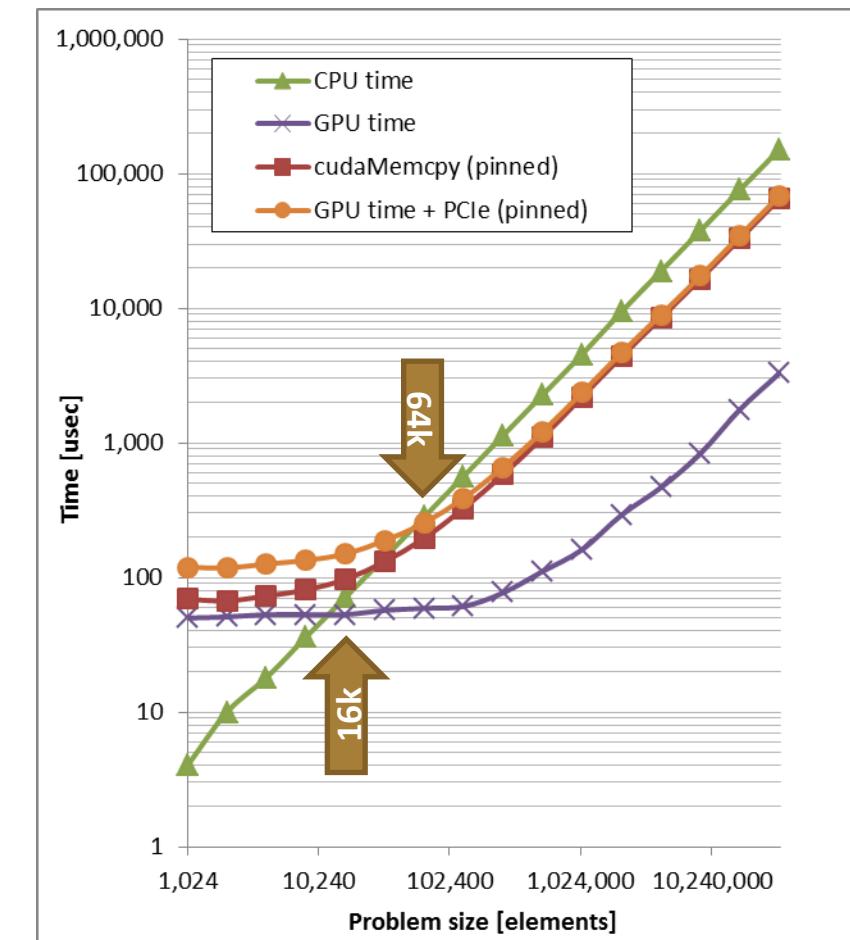
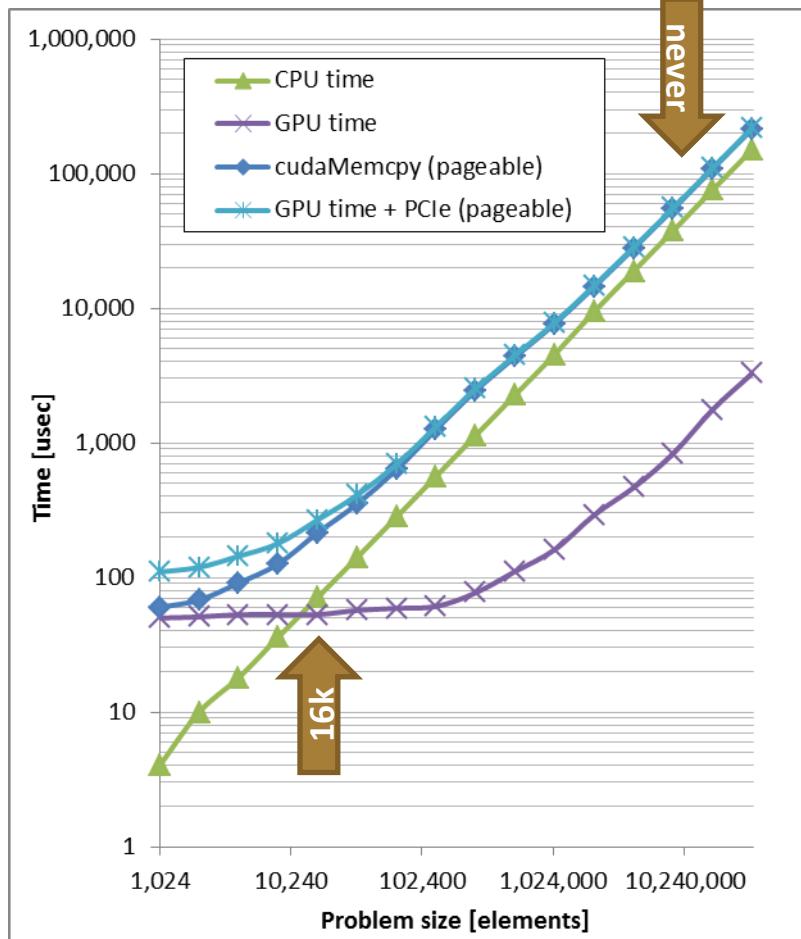
SAXPY Example

```
//////////  
// kernel function (CPU)  
//////////  
void saxpy_serial(int n, float alpha, float *x, float *y)  
{  
    int i;  
    for (i=0; i<n; i++) {  
        y[i] = alpha*x[i] + y[i];  
    }  
}
```

```
//////////  
// kernel function (CUDA device)  
//////////  
__global__ void saxpy_parallel(int n, float alpha, float *x, float *y)  
{  
    // compute the global index in the vector from  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
  
    // avoid writing past the allocated memory for the vector y.  
    if (i<n) {  
        y[i] = alpha*x[i] + y[i];  
    }  
}
```



SAXPY Example (2)



- Tesla K10, PCIe 2.0 x16 connection, Intel Xeon E5, single-threaded
- Break-even at 16k
- Huge advantage for GPU when no data movements
- Pageable memory vs. pinned memory



SAXPY Example (3)

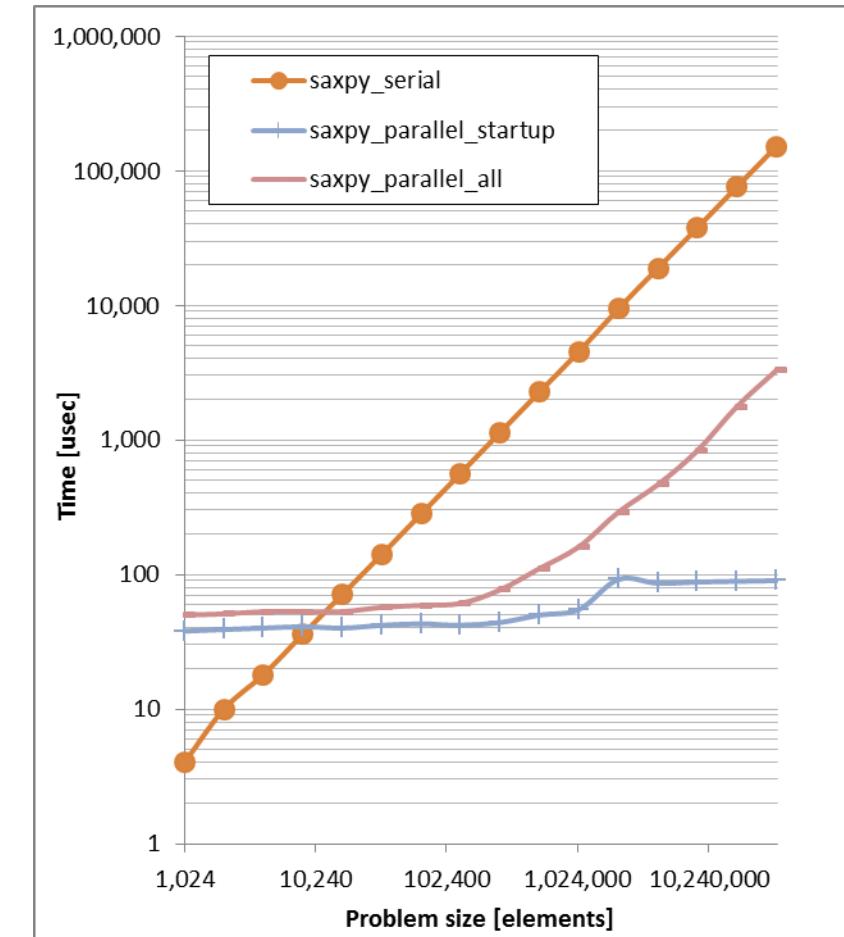
- Replace malloc with cudaMallocHost
 - Significant reduce data movement costs
- Pinned memory is a scarce resource!

```
///////////
// (2) allocate memory on host (main CPU memory) and device,
//      h_ denotes data residing on the host, d_ on device
/////////
float *h_x;
float *h_y;
float *d_x;
float *d_y;
if (USE_PINNED_MEMORY) {
    cudaMallocHost ( (void**) &h_x, N*sizeof(float) );
    cudaMallocHost ( (void**) &h_y, N*sizeof(float) );
} else {
    h_x = (float*) malloc( N*sizeof(float) );
    h_y = (float*) malloc( N*sizeof(float) );
}
cudaMalloc((void**)&d_x, N*sizeof(float));
cudaMalloc((void**)&d_y, N*sizeof(float));
checkErrors("memory allocation");
```



SAXPY Example

- Successful measures ☺
 - Use of pinned memory
 - (Increase computational intensity)
- Unsuccessful 😞
 - Precompilation: 40usec kernel startup time is maintained, independent of pre-compiled code or not
- SAXPY is a trivial example
 - See dependency analysis





Summary – CUDA & GPU Computing

■ Introduction to CUDA

- Pretty unusual concept compared to CPU programming
- Pretty easy programming model
- Pretty challenging for upmost speed-ups

■ Direct control over hardware

- Plenty of opportunities for the (experienced) user
- Increases the burden

■ Main differences to CPU programming

- Sophisticated resource planning and usage
- Scratch pad: use shared memory as explicitly controlled cache
- Data movement over PCIe
- Limited memory

■ Key characteristics for good GPU applications

- High degree of DLP
- Low data reuse
- High computational intensity
- High bandwidth



Common Errors

- Runtime problems:

CUDA Error: the launch timed out and was terminated
→ Stop X11

CUDA Error: unspecified launch failure
→ Typically a segmentation fault

CUDA Error: invalid configuration argument
→ Too many threads per block, too many resources per thread (shared memory, register count)

- Compile problem:

mmult.cu(171): error: identifier "__eh_curr_region" is undefined
→ Non-static shared memory, use static allocation of shared memory



Introduction to High Performance Computing

Lecture 10 – GPU Computing II

Holger Fröning

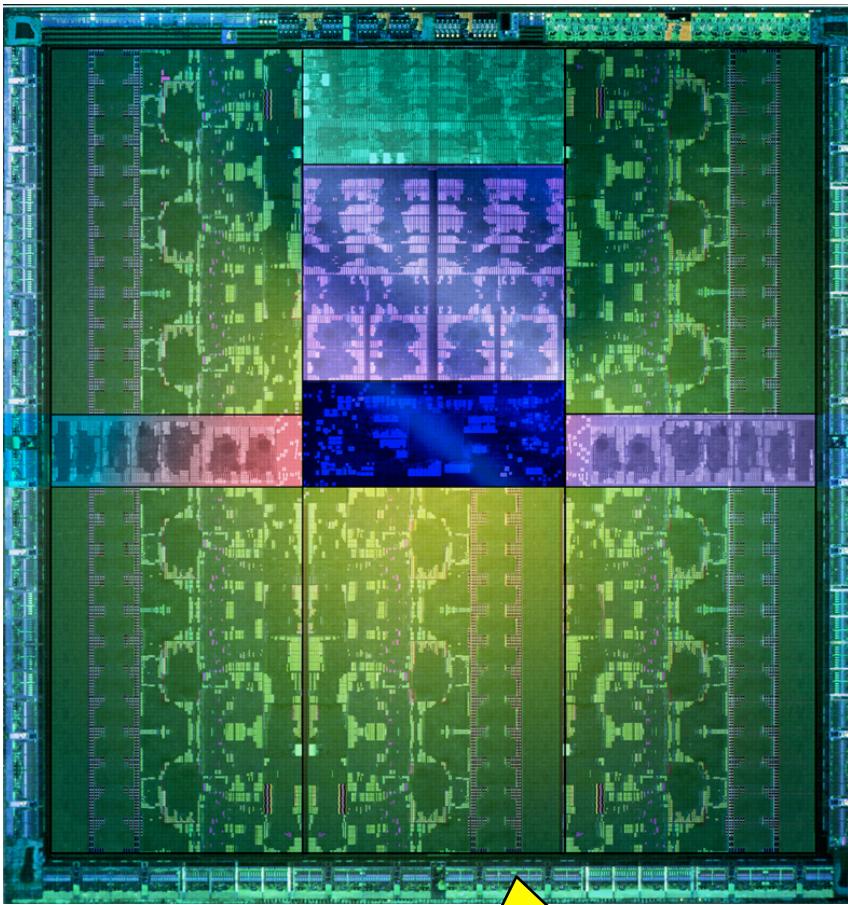
Institut für Technische Informatik

Universität Heidelberg

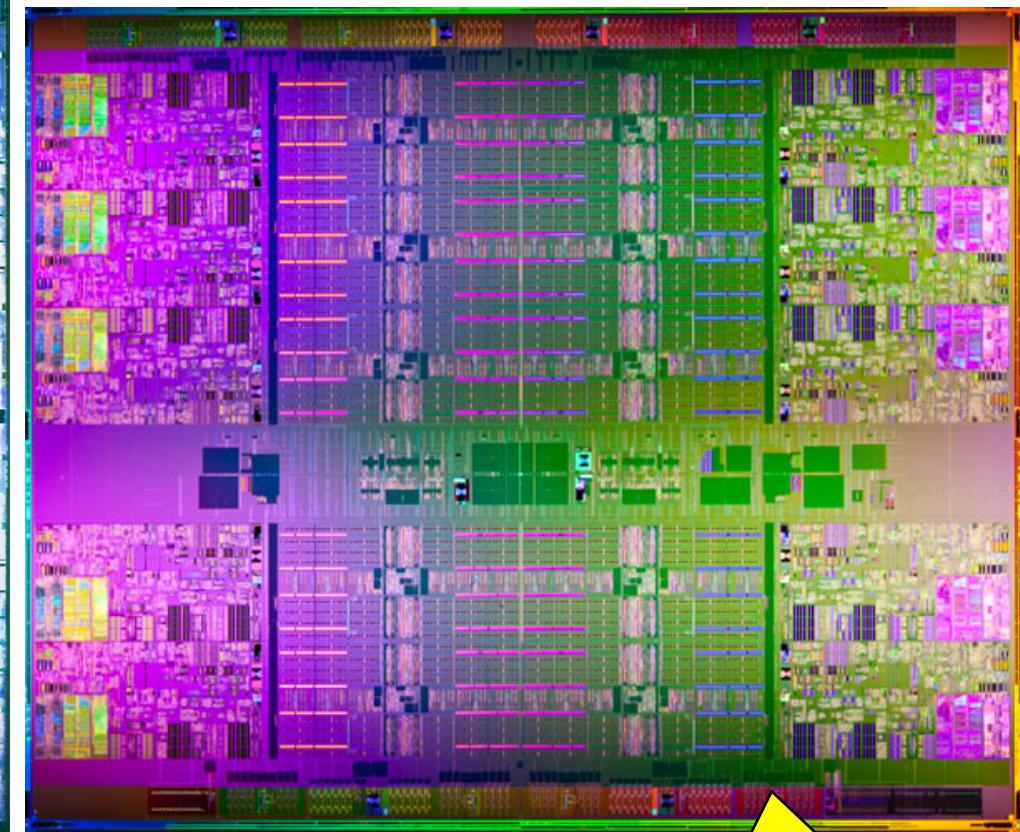


Introduction

GK110



XEON-E7





CUDA Thread Scheduling

src:flickr.com

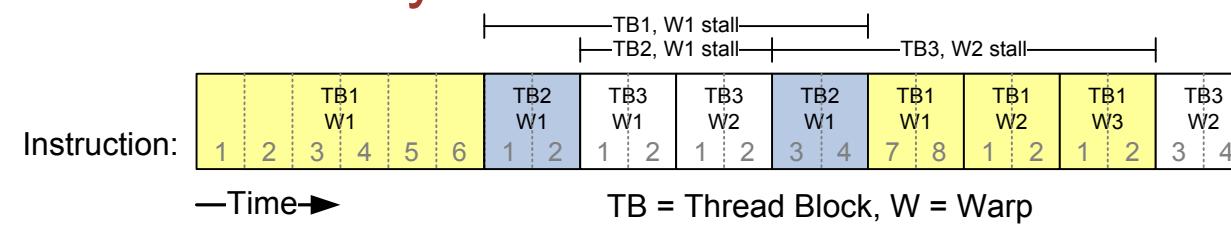


- Up to 1k threads per block
 - One block executes on one SM
- Each thread block is divided in 32-thread **warps**
 - Implementation decision, not CUDA
- Warps are the units for the scheduler
- Example:
 - 4 blocks being executed on one SM, each block 1k threads
 - How many warps?
 - Each block here has 32 warps, thus 128 warps in total
- Scheduler: select one out of these 128 warps for instruction fetch and execution



Thread Scheduling

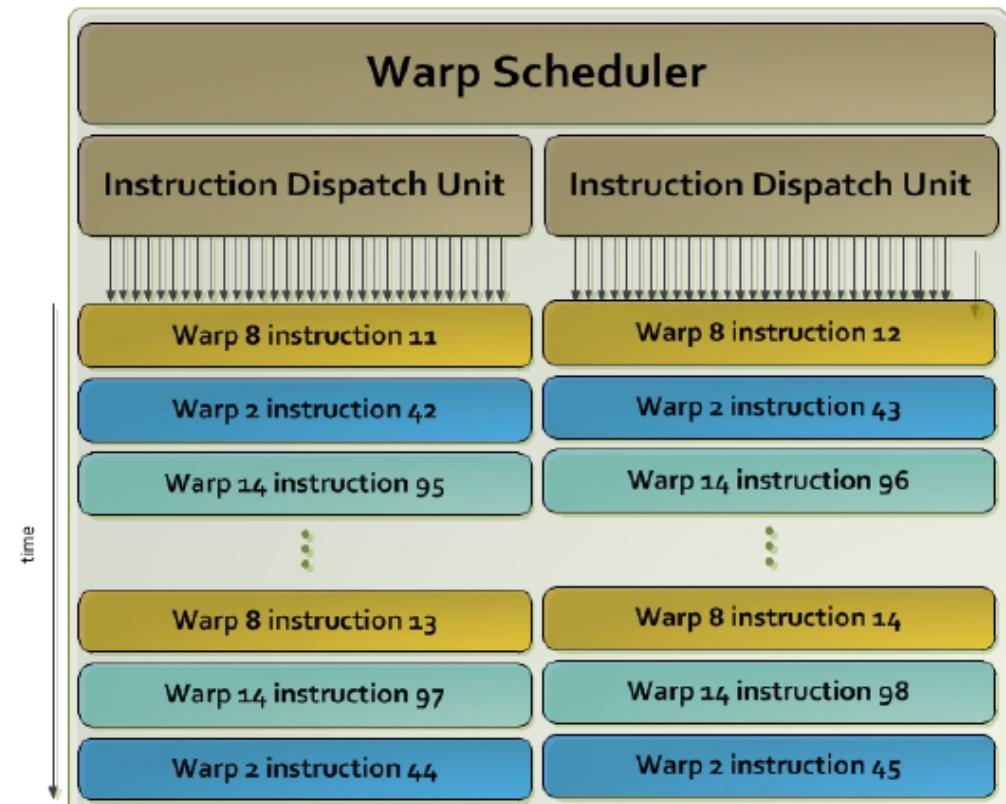
- Fine-grained multi-threading (FGMT)
 - Switch context (i.e., warp) every cycle
 - A warp that has the operands ready for its next instruction is ready for execution
 - All threads in a warp execute the same instruction
- Goal of FGMT: latency hiding
 - Global memory access latency: ~400-600 cycles
 - Sufficient number of warps can keep all functional units busy
- Warp count for maximum utilization depends on computational intensity





Thread Scheduling

- Fetch one instruction per cycle
 - From I\$
- Determine dependencies (operands)
- Scoreboard checks if dependencies are resolved
 - Prevent data hazards
- Issue: select one warp based on prioritized round-robin
 - Priority: warp age
- Scheduler broadcasts the instruction to all 32 threads in a warp
 - Dedicated control paths
 - Branch divergence problem
 - Write-masks



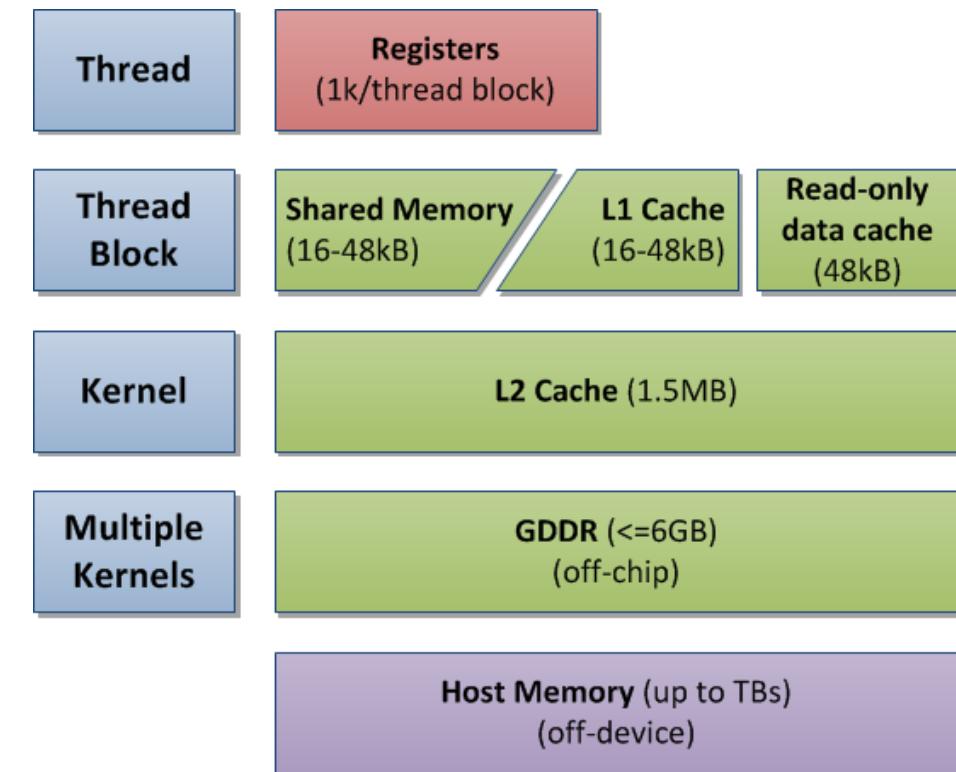


Memory Subsystem



GK110 – Memory Hierarchy

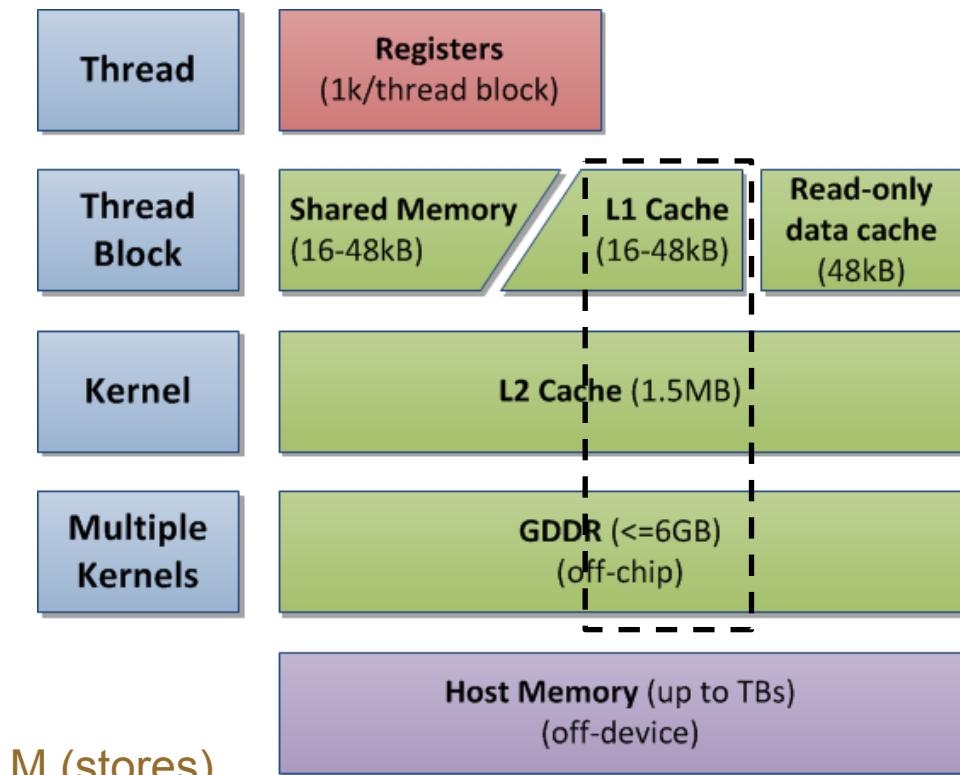
- **Registers at thread level**
 - Registers/thread depends on run-time configuration
 - Max. 255 registers/thread
- **Shared memory / L1\$ at block level**
 - Variable sizes
 - L1\$ can serve for register spilling
 - L1\$ not coherent, write-invalidate
 - Compiler controlled RO data\$
- **L2\$ / GDDR at device level**
 - GDDR: ~400-600 cycles access latency
 - L2\$ as victim cache for all upper units, write-back
 - Purpose: reducing contention





Local Memory

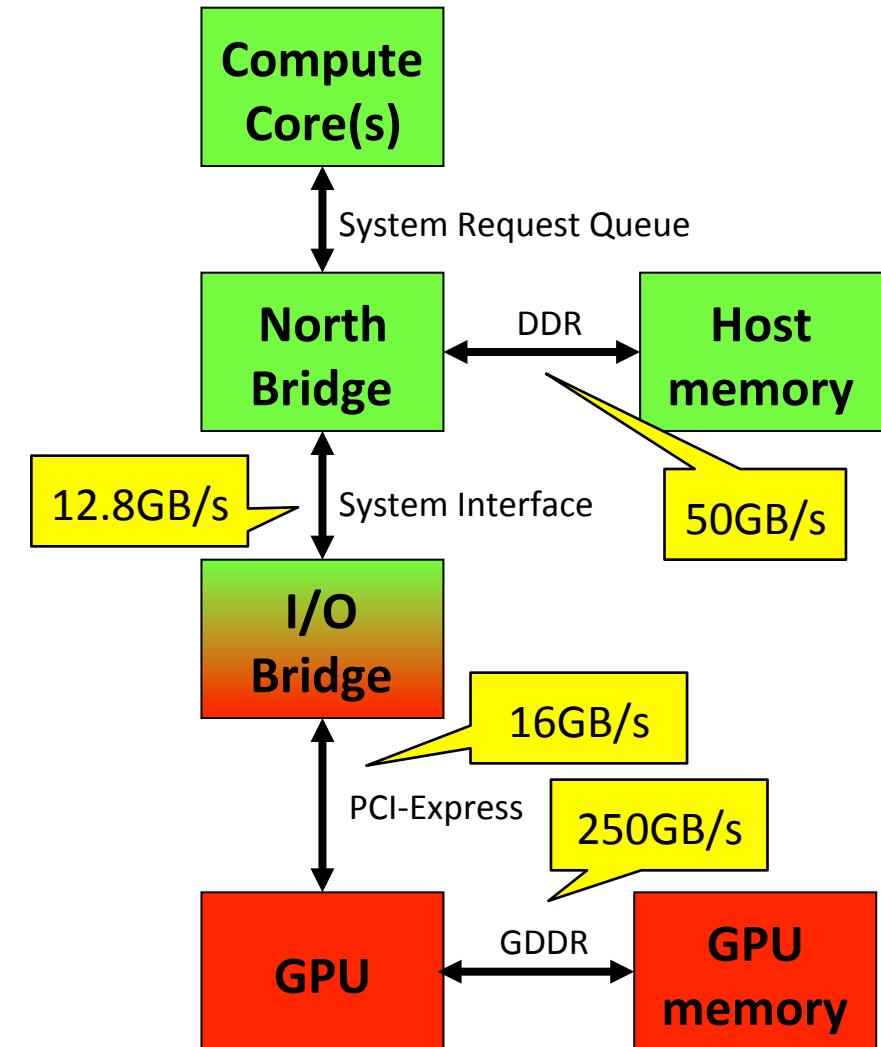
- Local memory: part of global memory, but thread-local
 - Register spilling: when SM runs out of resources
 - Limited register count per thread
 - Limited total number of registers
 - LM is used if the source code exceeds these limits
 - Local because each thread has its private area
- Differences from global memory
 - Stores are cached in L1\$
 - Addressing is resolved by compiler
- Store always happens before load
 - Per thread: move data from GM to LM (stores)
 - Subsequent load accesses





Host Memory

- Pinned/unpinned host memory
 - Unpinned host memory: probability of demand paging
 - Pinned host memory: autonomous device access possible
- cudaMemcpy
 - GPU DMA engine(s)
- Zero copy
 - GPU threads (CC ≥ 2.0)
 - For initial shared memory fills etc.





- **High bandwidth, high latency**
 - GPU knows how to overcome latency issues
- **Coalesced access**
 - Avoid access with offset or stride
 - Combine fine-grain accesses by multiple threads into single GDDR operations
 - Coalesced thread access should match a multiple of L1/L2 cache line sizes
 - Cache line sizes: L1: 128B, L2: 32B
- **Misaligned accesses: one warp is scheduled, but accesses misaligned addresses**
 - GPUs use caches for access coalescing



Global Memory – Access Penalties

- Main problem: thread scheduling does not result in coalesced accesses
- Solution: manually control data movement in memory hierarchy
 - Caches = transparent, implicit hierarchy
 - Scratchpad = opaque, explicit hierarchy
- Collaborative loads from global memory to shared memory
 - Common case: one thread is not moving the data it requires (at least not immediately)
 - Main GPU advantage is memory bandwidth (together with multi-core): **coalescing of utmost importance!**



Matrix Multiply for a CPU (Reference)



Matrix Multiply – CPU naive

■ CPU sequential version

- No big surprises
- Can be called directly

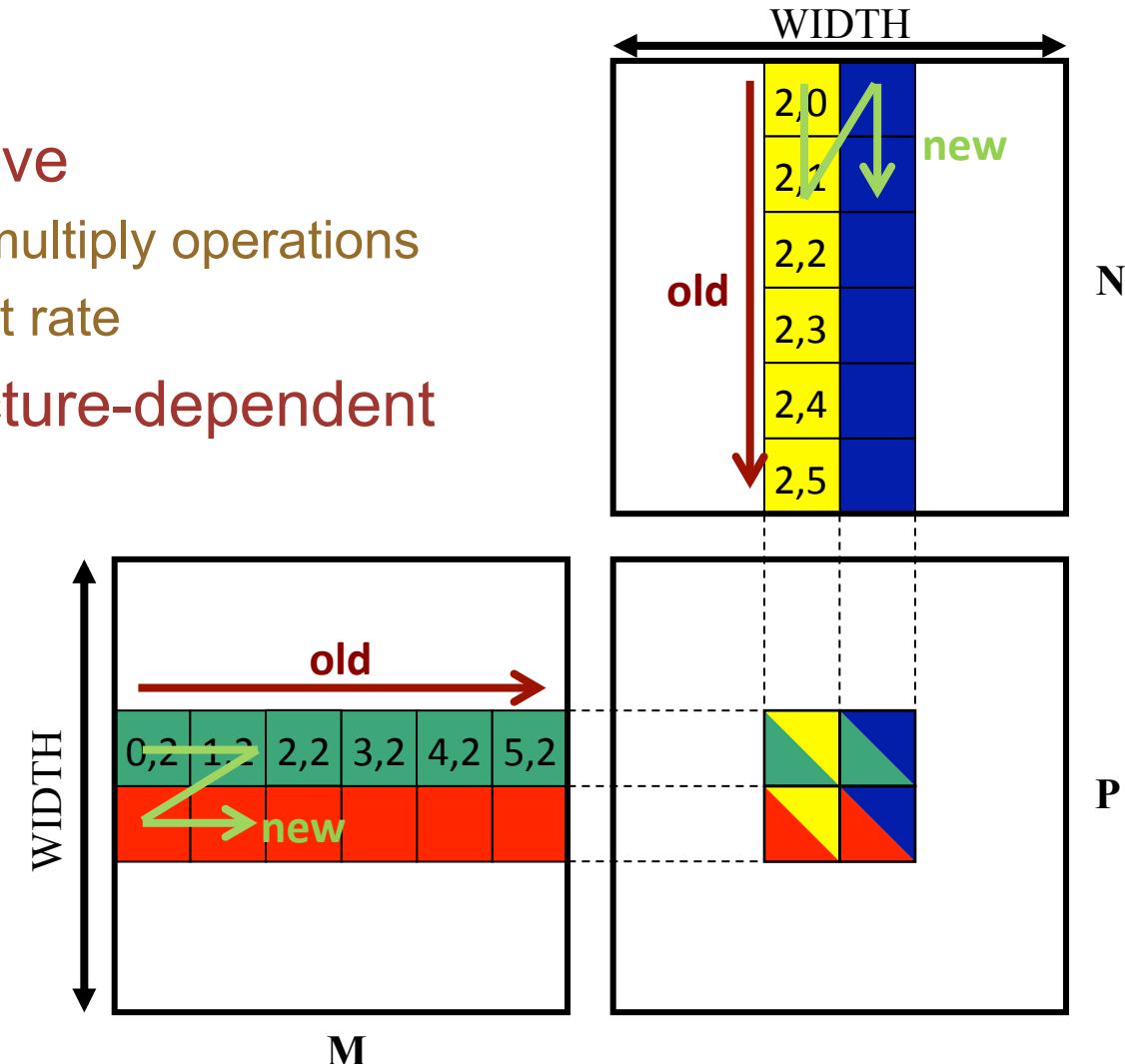
```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i) {
        for (int j = 0; j < Width; ++j) {
            float sum = 0;
            for (int k = 0; k < Width; ++k) {
                float a = M[i * width + k];
                float b = N[k * width + j];
                sum += a * b;
            }
            P[i * width + j] = sum;
        }
    }
}
```



Matrix Multiply – CPU blocked

- **Multiply is commutative**
 - So feel free to reorder multiply operations
 - Goal: increase cache hit rate
- **Block size is architecture-dependent parameter**
 - Cache size

$$P_{i,j} = \sum_k M_{i,k} \times N_{k,j}$$





Matrix Multiply – CPU blocked

```
void MatrixMulOnHostBlocked ( float* M, float* N, float* P, long matWidth,
long blockSize )
{
    // Matrix multiply of P = M * N
    // assume P to be initialized to zero
    // assume matrices to be square ones

    float temp;
    for ( long ii = 0; ii < matWidth; ii += blockSize ) {
        for ( long jj = 0; jj < matWidth; jj += blockSize ) {
            for ( long kk = 0; kk < matWidth; kk += blockSize ) {
                for ( long i = ii; i < findMin ( ii+blockSize, matWidth ); i++ ) {
                    for ( long j = jj; j < findMin ( jj+blockSize, matWidth ); j++ ) {
                        temp = 0;
                        for ( long k = kk; k < findMin ( kk+blockSize, matWidth ); k++ )
                            temp += M[i * matWidth + k] * N[k * matWidth + j];
                        P[ i * matWidth + j] += temp;
                    }
                }
            }
        }
    }
}
```



Matrix Multiply – CPU blocked

Trace for naive implementation

```
..  
<snip>  
..  
P[2] [3] += M[3] [0] * N[0] [2]  
P[2] [3] += M[3] [1] * N[1] [2]  
P[2] [3] += M[3] [2] * N[2] [2]  
P[2] [3] += M[3] [3] * N[3] [2]  
..  
P[3] [3] += M[3] [0] * N[0] [3]  
P[3] [3] += M[3] [1] * N[1] [3]  
P[3] [3] += M[3] [2] * N[2] [3]  
P[3] [3] += M[3] [3] * N[3] [3]
```

No locality - RED

Where is spatial locality? - GREEN

Where is temporal locality? - BLUE

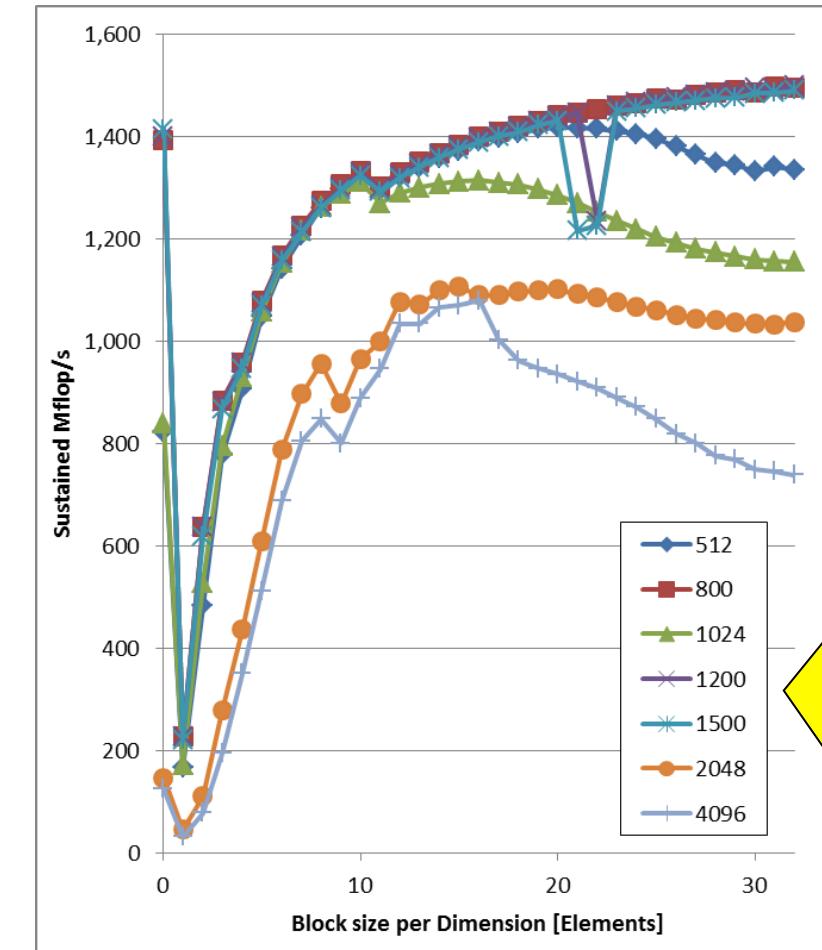
Trace for blocks of two-by-two

```
..  
<snip>  
..  
P[2] [3] += M[3] [0] * N[0] [2]  
P[2] [3] += M[3] [1] * N[1] [2]  
P[3] [3] += M[3] [0] * N[0] [3]  
P[3] [3] += M[3] [1] * N[1] [3]  
..  
P[2] [3] += M[3] [2] * N[2] [2]  
P[2] [3] += M[3] [3] * N[3] [2]  
P[3] [3] += M[3] [2] * N[2] [3]  
P[3] [3] += M[3] [3] * N[3] [3]
```



Matrix Multiply – CPU blocked

- Performance for single-threaded CPU run
 - Xeon E5 Sandy Bridge
 - 4 cores @ 2.4GHz
- Single precision
- Varying matrix sizes [elements per dimension]
- Block size 0 = non-blocked (reference)
- Huge drop for block size of 1?
 - Control flow overhead
- Non-blocked better than blocked?
 - Cache size!
- Factor of 10x for blocked vs. non-blocked typical



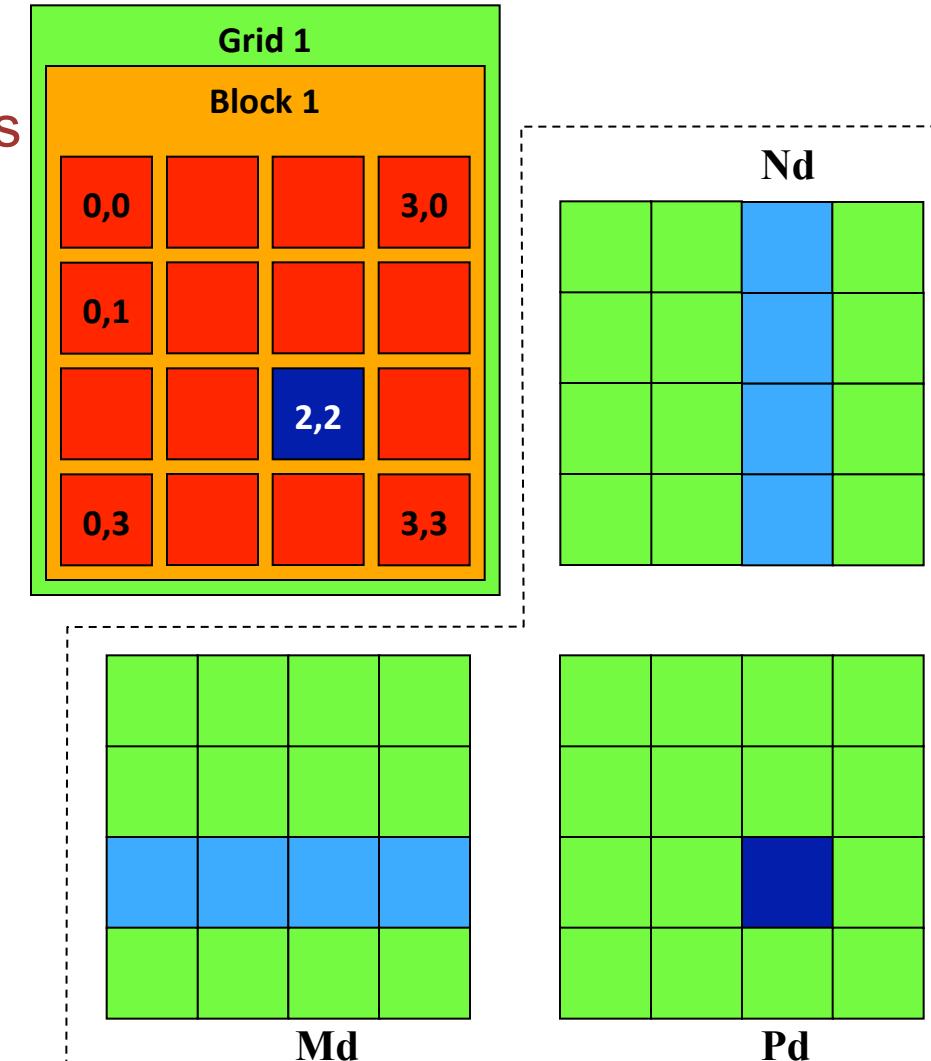


Optimizing Matrix Multiply for a GPU



Matrix Multiply – Quick analysis

- A single thread block computes P_d
 - Each thread computes a single element of P_d
 - Load a row of M_d
 - Load a column of N_d
 - Per element: one mult., one add
 - Write P_d
- Issue 1: Matrix size limited by threads/block
- Issue 2: Compute:Memory ratio
 - Computational intensity
 - $\sim 2^{\text{WIDTH}}:2^{\text{WIDTH}}$ or 1:1 (very low)
 - Or 1 FLOP/4B





Matrix Multiply – Multiple Thread Blocks

■ Kernel using multiple blocks

```
__global__ void MatrixMulKernel ( float* Md, float* Nd, float* Pd, int Width )
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

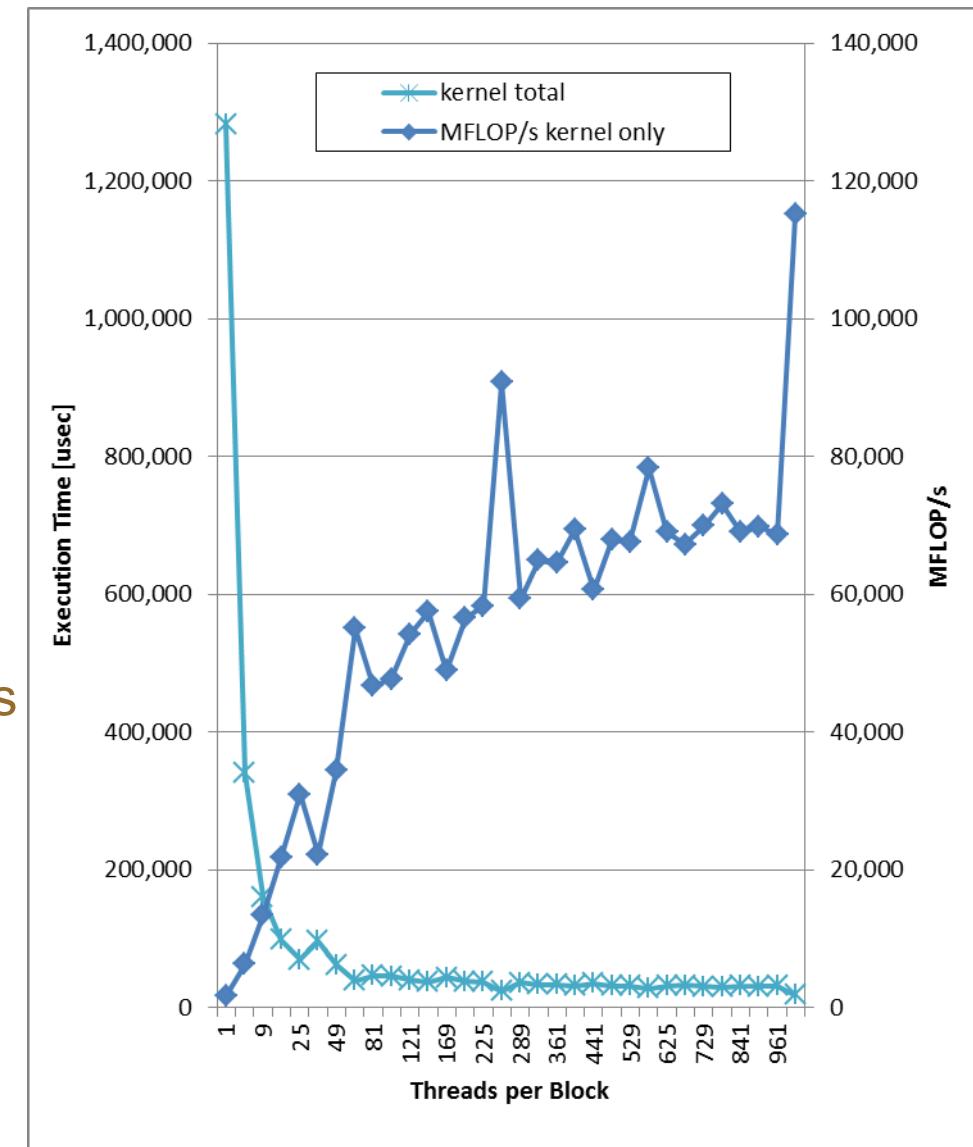
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for ( int k = 0; k < Width; ++k )
        Pvalue += Md [ Row * Width + k ] * Nd [ k * Width + Col ];

    Pd [ Row * Width + Col ] = Pvalue;
}
```



Matrix Multiply – Single Precision Results

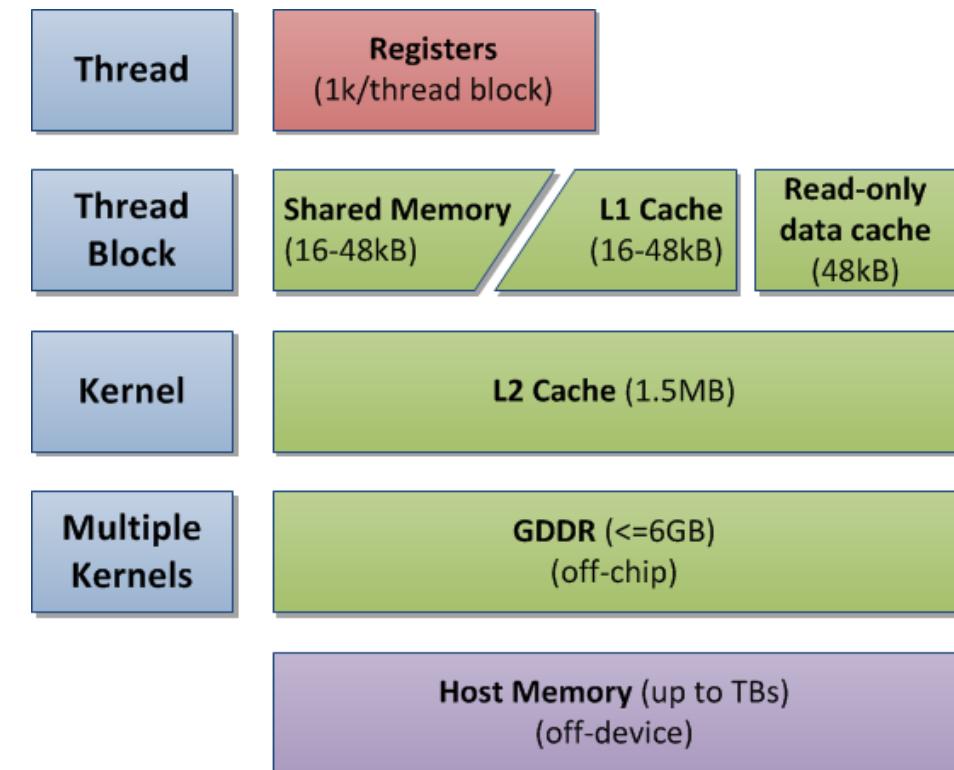
- Tesla K20c GPU, Kepler-class
- Scheduling: varying the number of threads per block
 - 1k x 1k matrix size
 - Match block count
 - Nice example for performance increase of large thread counts
 - Not always optimal
- Resulting GFLOP/s
 - $N^2=1M$ elements, each 2N FLOP = 2.14 GFLOP
 - Without data movement





GK110 – Memory Hierarchy

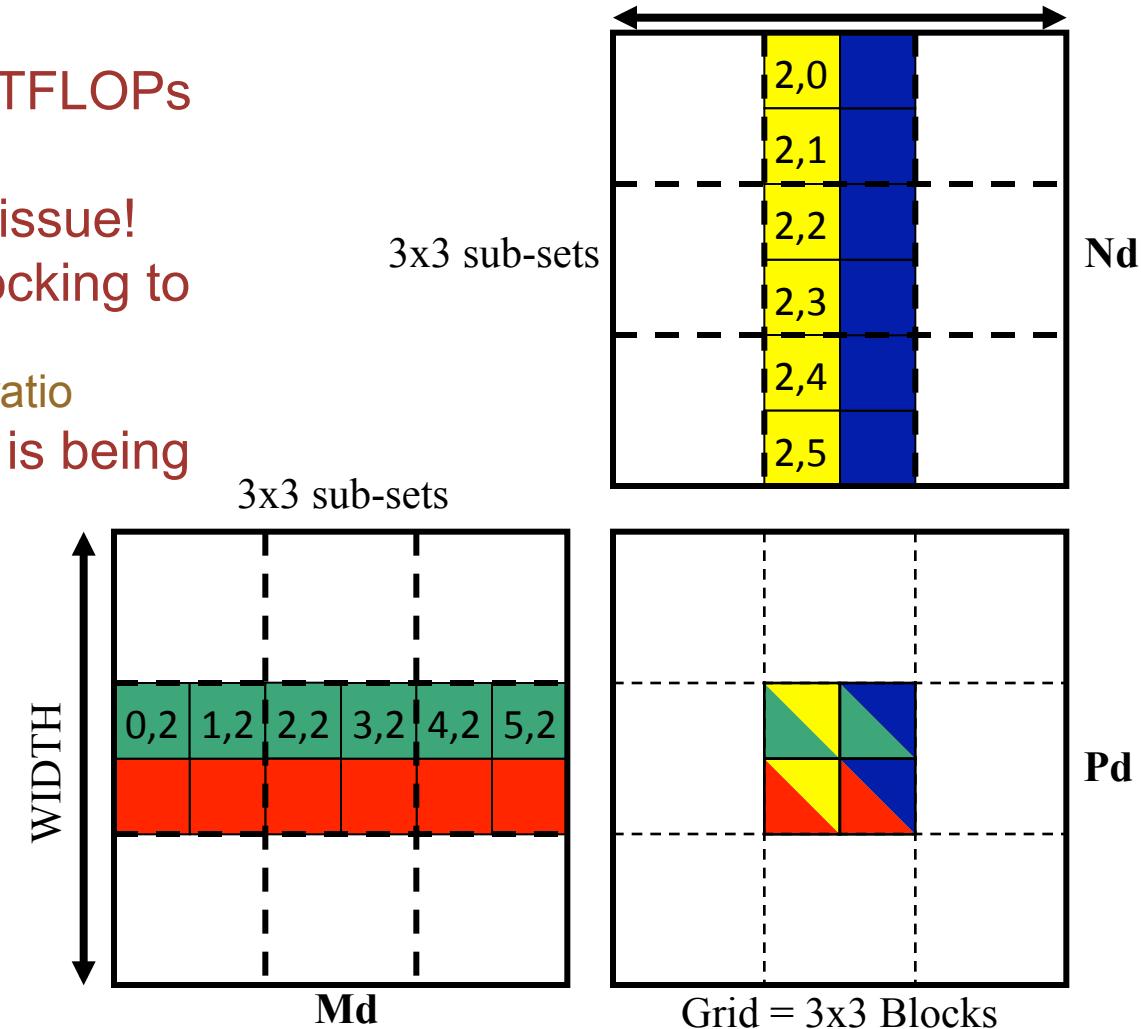
- Shared memory is (yet) not used
 - Unlike caches for CPUs, it is manually controlled
 - I.e., the programmer is responsible to move data from/to shared memory
- If it's used, same access costs as any cache





Matrix Multiply – Shared Memory

- ~140GFLOPs out of 3.5TFLOPs
- ➔ Optimizations required!
- Compute intensity is an issue!
- Similar to CPUs, use blocking to increase data reuse
 - And thus the Flop-to-byte ratio
- Old: each input element is being read by **WIDTH** threads
- New: is read by one thread, but used by multiple threads
- Separate kernel execution into phases
 - Size of a sub-set should match a tile size





Matrix Multiply – Shared Memory

Time →

| | Phase 1 | | | Phase 2 | | |
|------------------|--|--|---|--|--|---|
| T _{0,0} | Md _{0,0} ↓ Mds _{0,0} | Nd _{0,0} ↓ Nds _{0,0} | PdValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1} | Md _{2,0} ↓ Mds _{0,0} | Nd _{0,2} ↓ Nds _{0,0} | PdValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{1,0} *Nds _{0,1} |
| T _{1,0} | Md _{1,0} ↓ Mds _{1,0} | Nd _{1,0} ↓ Nds _{1,0} | PdValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1} | Md _{3,0} ↓ Mds _{1,0} | Nd _{1,2} ↓ Nds _{1,0} | PdValue _{1,0} += Mds _{0,0} *Nds _{1,0} + Mds _{1,0} *Nds _{1,1} |
| T _{0,1} | Md _{0,1} ↓ Mds _{0,1} | Nd _{0,1} ↓ Nds _{0,1} | PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1} | Md _{2,1} ↓ Mds _{0,1} | Nd _{0,3} ↓ Nds _{0,1} | PdValue _{0,1} += Mds _{0,1} *Nds _{0,0} + Mds _{1,1} *Nds _{0,1} |
| T _{1,1} | Md _{1,1} ↓ Mds _{1,1} | Nd _{1,1} ↓ Nds _{1,1} | PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1} | Md _{3,1} ↓ Mds _{1,1} | Nd _{1,3} ↓ Nds _{1,1} | PdValue _{1,1} += Mds _{0,1} *Nds _{1,0} + Mds _{1,1} *Nds _{1,1} |

Per thread: TILE_WIDTH MADDs

Phase change with
offset = TILE_WIDTH



Matrix Multiply – Shared Memory

```
__global__ void MM_SM ( float* Md, float* Nd, float* Pd, int Width )
{
    __shared__ float Mds [TILEWIDTH] [TILEWIDTH];
    __shared__ float Nds [TILEWIDTH] [TILEWIDTH];
    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the Pd element to work on
    int Row = by * TILEWIDTH + ty;
    int Col = bx * TILEWIDTH + tx;
    float Pvalue = 0;

    if !(Row > Width || Col > Width) {
        // Loop over the Md and Nd tiles required to compute the Pd element
        for ( int m = 0; m < Width / TILEWIDTH; ++m ) {
            // Collaborative loading of Md and Nd tiles into shared memory
            Mds [ty] [tx] = Md [ Row * Width + ( m * TILEWIDTH + tx ) ];
            Nds [ty] [tx] = Nd [ Col + ( m * TILEWIDTH + ty ) * Width ];
            __syncthreads();

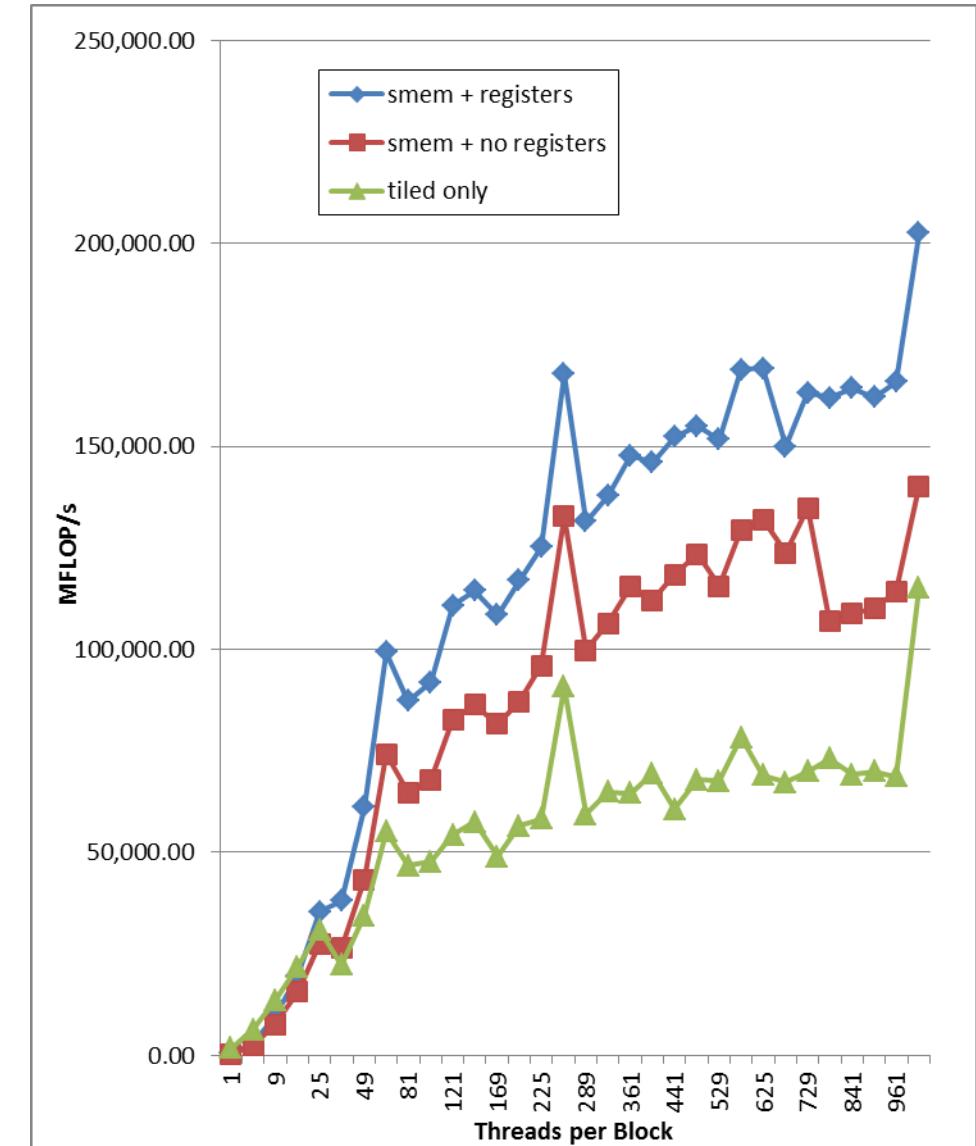
            for ( int k = 0; k < TILEWIDTH; ++k )
                Pvalue += Mds [ty] [k] * Nds [k] [tx];
            __syncthreads();
        }
        Pd[Row * Width + Col] = Pvalue;
    }
}
```

Synchronization
needed due to
dependencies



Matrix Multiply – Shared Memory Results

- Performance comparison for Tesla K20c:
 - Tiled only
 - Use of shared memory, but no register for intermediate value
 - Use of both shared memory and register
- Block size <= 1k
 - Both code optimizations and configuration matters!





Matrix Multiply – Shared Memory Dependencies

```
<snip>
    for ( int m = 0; m < Width / TILEWIDTH; ++m ) {
        // Collaborative loading of Md and Nd tiles into shared memory
        Mds [ty] [tx] = Md [ Row * Width + ( m * TILEWIDTH + tx ) ];
        Nds [ty] [tx] = Nd [ Col + ( m * TILEWIDTH + ty ) * Width ];
        __syncthreads();

        for ( int k = 0; k < TILEWIDTH; ++k )
            Pvalue += Mds [ty] [k] * Nds [k] [tx];
        __syncthreads ();
    }
<snip>
```

1

2

■ Three types of dependencies

- RAW: true or data dependency
 - True dependency, so not solvable, see (1)
- WAR: anti dependency
 - Is a name dependency, so can be solved using renaming, see (2)
- WAW: output dependency
 - Is a name dependency, so can be solved using renaming, n.a. here



Matrix Multiply – Shared Memory Allocations

```
__global__ void MM_SM ( float* Md, float* Nd, float* Pd, int Width )
{
    __shared__ float Mds [TILEWIDTH] [TILEWIDTH];
    __shared__ float Nds [TILEWIDTH] [TILEWIDTH];
    ...
}
```



```
__global__ void MM_SM ( float* Md, float* Nd, float* Pd, int Width )
{
    extern __shared__ float mem_ds [];
    float *Mds = & ( mem_ds [0] );
    float *Nds = & ( mem_ds [size_of_Mds] );
    ...
}

int main ()
{
    ...
    MM_SM <<< dimGrid, dimBlock, sharedSize >>> ( Md, Nd, Pd, matWidth );
    ...
}
```

Manual memory management

Declare total shared memory



Summary

- Matrix multiply as a good example to leverage locality using the shared memory (scratch pad)
- Mind the synchronization within the thread block
 - Dependencies = race conditions
 - Threads are scheduled in warps, threads per warp might not match the scratchpad use model
- Scratch pad about 10x faster than global memory in terms of bandwidth
 - Leverage that for data reuse!
 - Collective memory access, so mind dependencies!
 - Usually one thread will fetch data for other threads to maximize coalescing
- Further candidates for matrix multiply optimizations
 - Improving coalescing, reducing possible bank conflicts



GPUs in MPI Environments

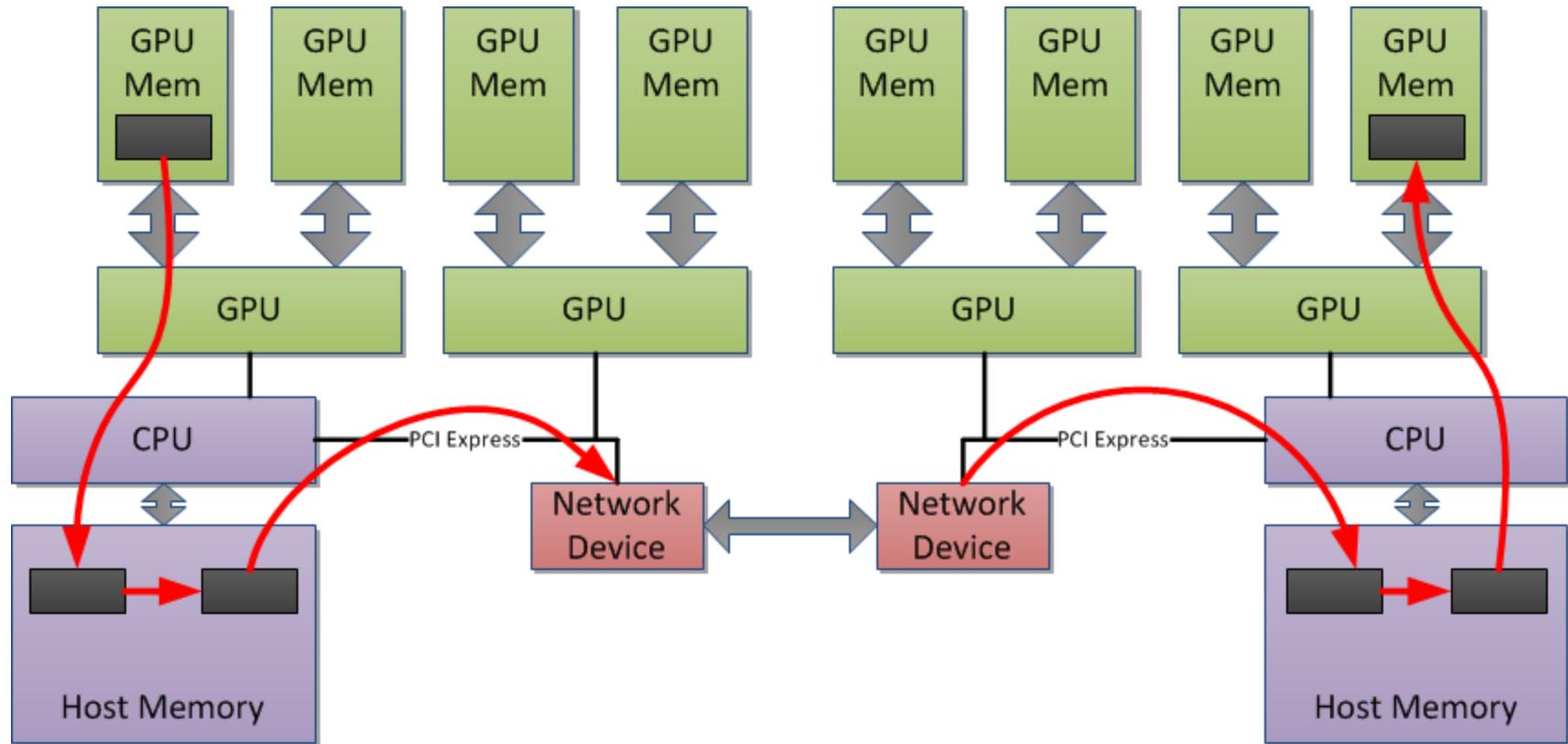


Communication Among Distributed GPUs

- GPUs are the „working horse“ of accelerated HPC clusters
 - Only operate on special on-device memory (global, shared, ...)
 - Question is how to move data efficiently between distributed special memory
 - Control flow for communication is a separate issue

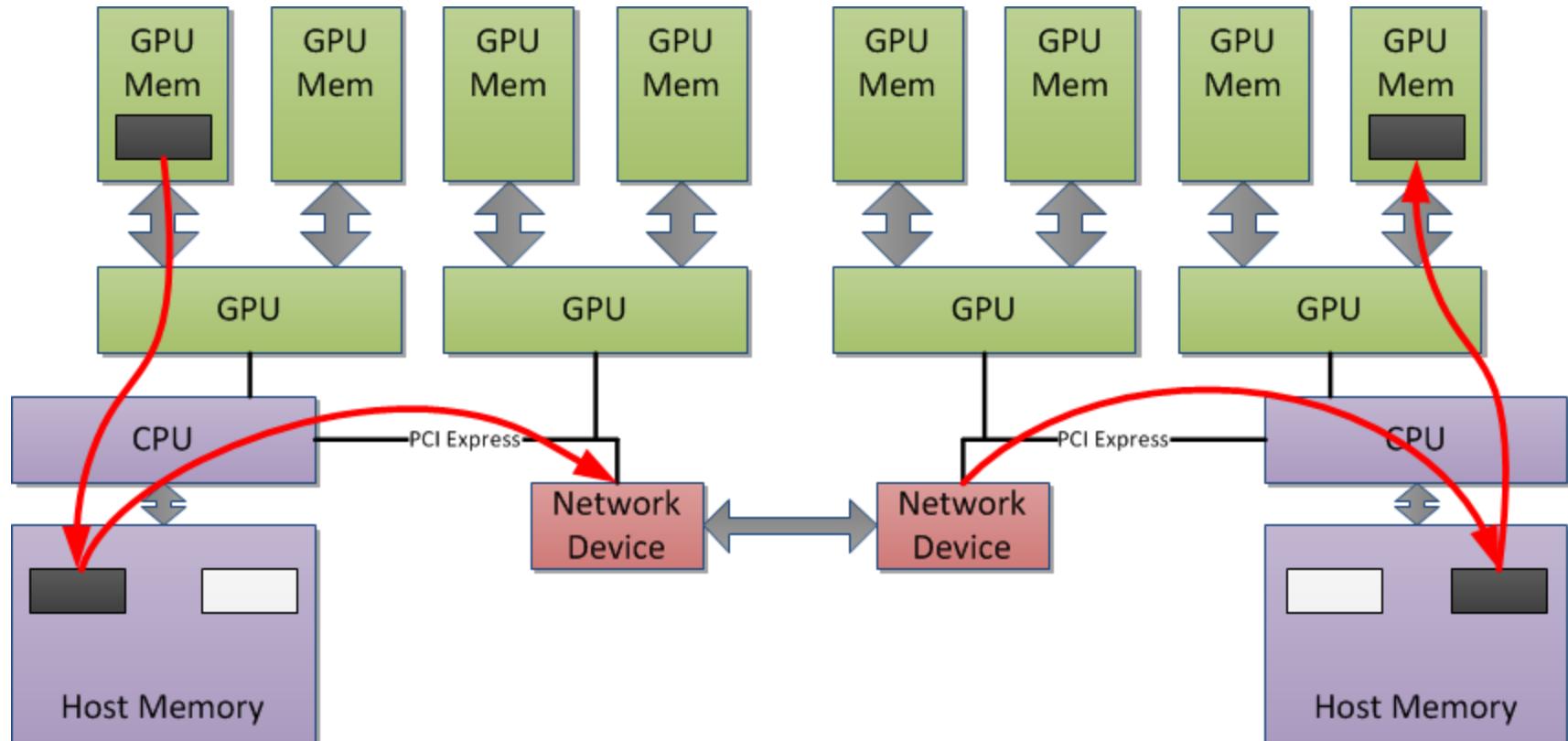


First Approach: 4 copies





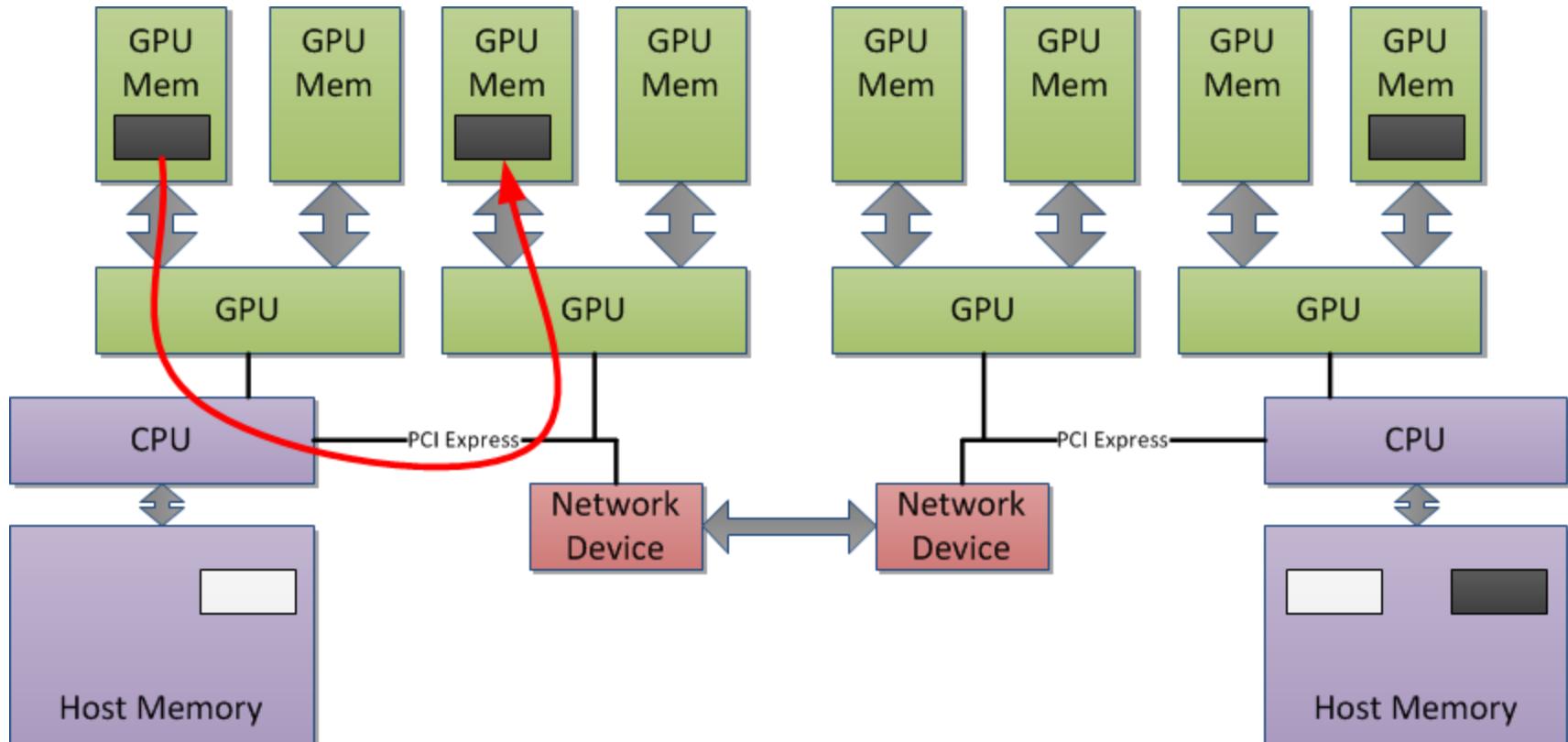
Second Approach: GPUDirect 1.0



- GPU and network device can read/write the same pinned host memory regions
 - Introduced by NVidia in 2010, 2 copies: Up to 33% speed-ups reported
 - G. Shainer et al., “The development of Mellanox NVIDIA GPUDirect over Infiniband - a new model for GPU to GPU communications”, *Comput. Sci. Res. Dev.* (2011) 26:267-273.



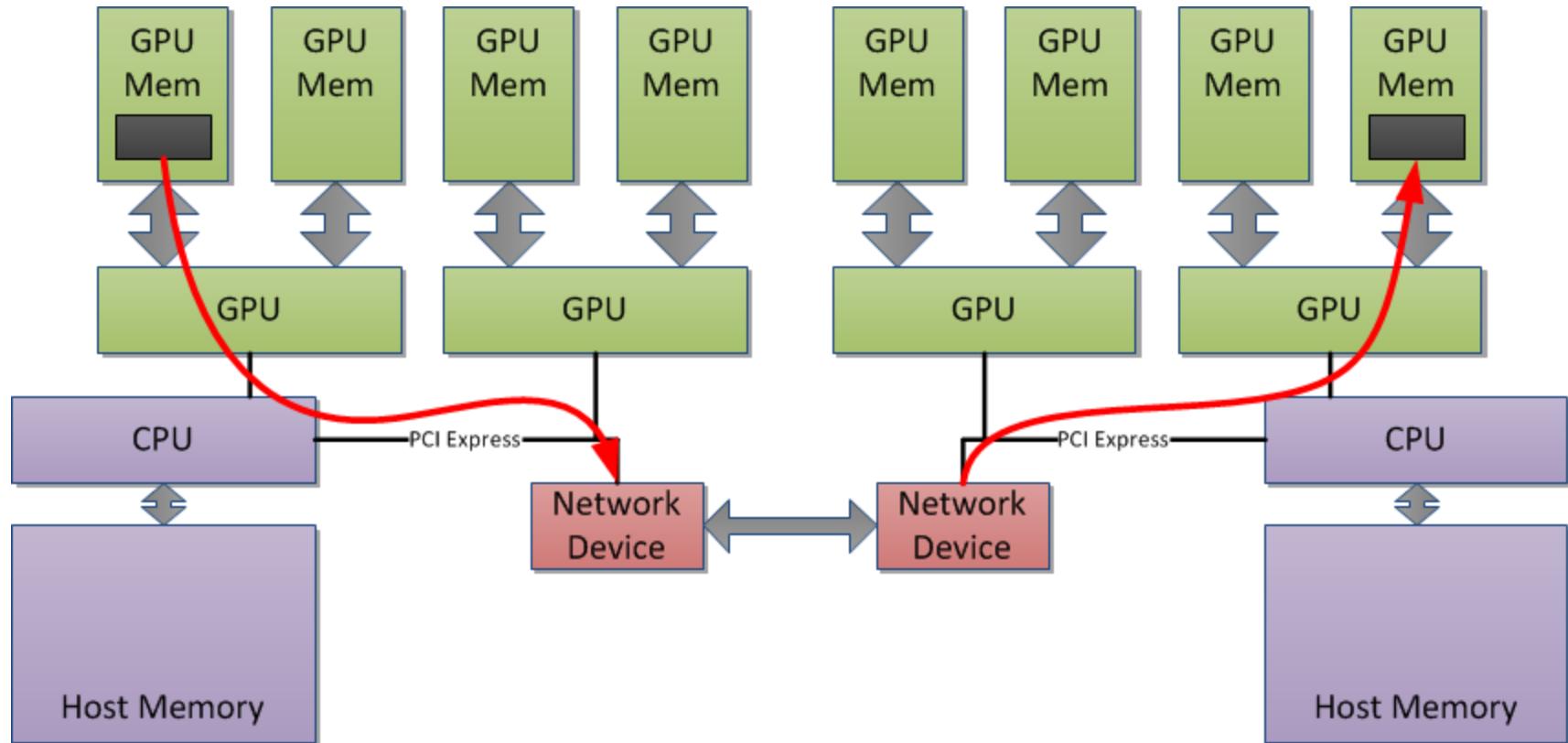
Note on GPUDirect 2.0



- Inter-GPU communication, but intra-node (peer-to-peer)
- Zero-copy scheme



Third Approach: GPUDirect RDMA

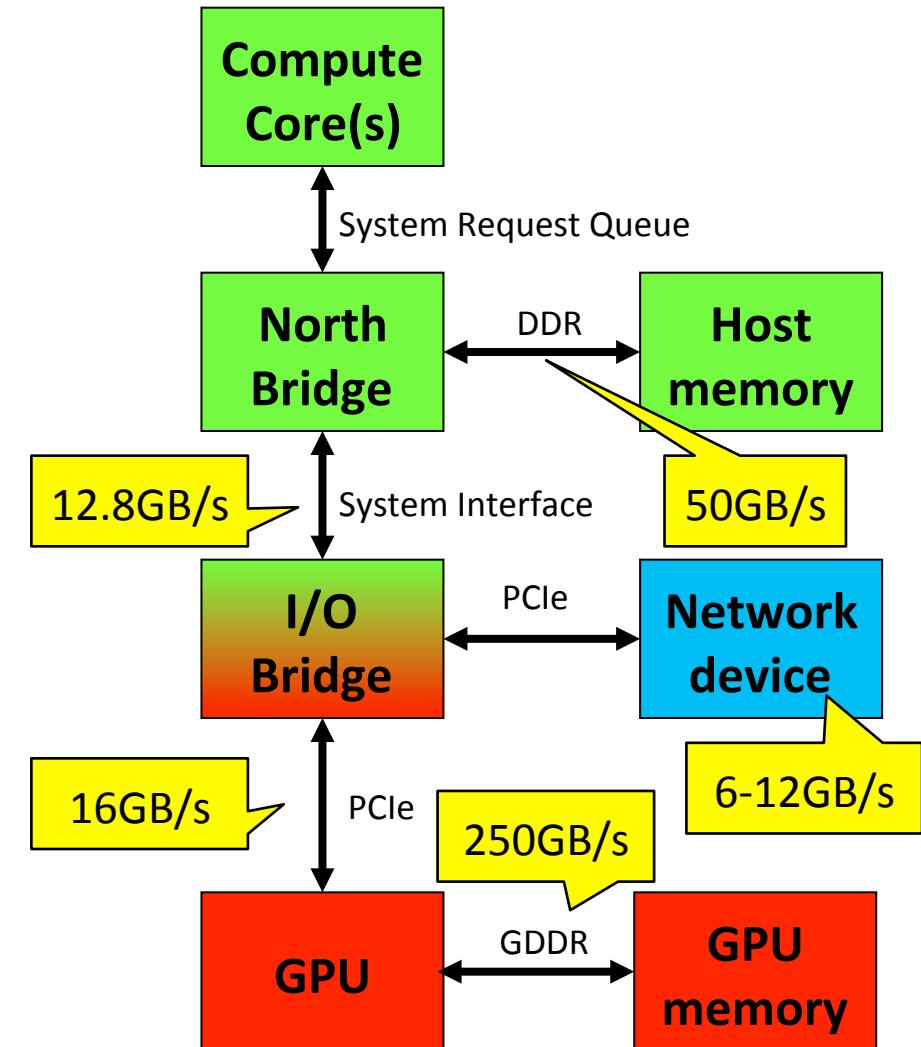


- Zero-copy scheme for inter-node GPU communication
- Figure only depicts data movement, not control paths
- Most network devices require host CPU supervision



Outlook

- CPU as offload engine for communication purposes
- Requires to return control flow to the CPUs
 - Associated overhead acceptable for bulk transfers, but not for fine grain communication
- Another possibility:
 - GPU Global Address Spaces (GGAS) & GPU-initiated Put/Get commands





- GPUs have manually-controlled memory hierarchies (the GPU world is flat)
 - Caches in GPUs not used to reduce latency, but to reduce memory contention and to coalesce accesses
 - Matrix Multiply Example: optimization of similar complexity, but manual data movement for shared memory
- Scheduling
 - Hierarchy: thread block & thread warps
 - Instruction stream == thread warp, != single thread (as for CPUs)
- Latency hiding
 - Hide GM latencies using TLP
 - Also ILP!
 - Less threads → more registers per thread!
- Inter-GPU communication
 - Still some way to go



Introduction to High Performance Computing

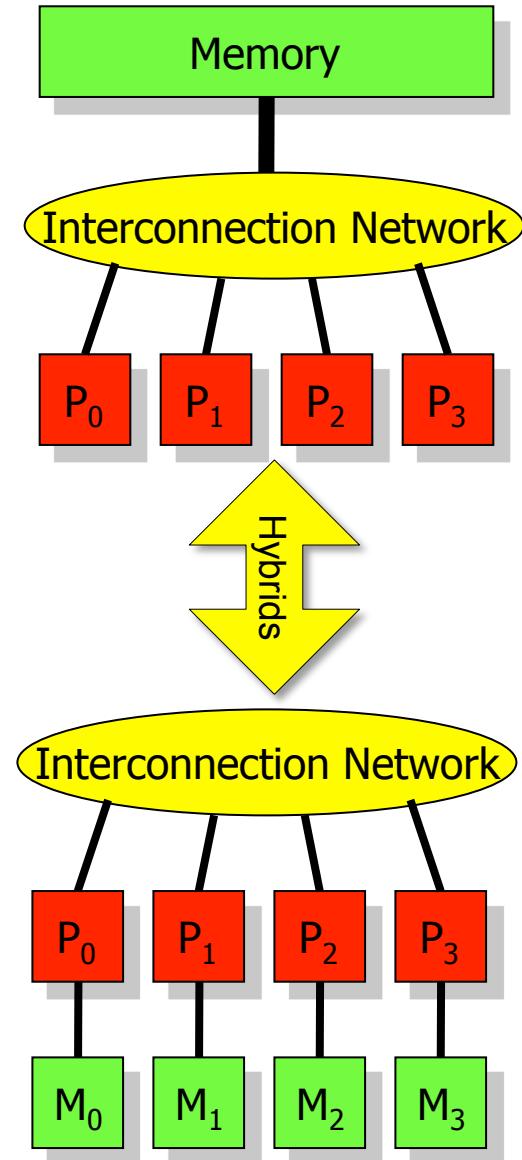
Lecture 11 – Basics of Interconnection Networks I

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg



Introduction

- Up to now: Interconnection Network (IN) as a black box
 - Turning into the key component of HPC systems
 - Exact behavior is crucial to overall performance
- INs are found everywhere
 - On-Chip Networks (different modules or cores)
 - Intra-Node (CPU, memory, graphics, devices)
 - Inter-Node (multiple nodes)
 - SAN, LAN, WAN
- Different requirements/workloads!
 - Here: focus on HPC and its demands





Types of INs in a computer system

| Type | Description | Length |
|----------------------------------|---|-----------|
| Processor or system interconnect | Connections between processors, memory controllers, ... (HyperTransport, QPI, FSB) | 10..30cm |
| Memory network | Connections between memory controller and memory modules | 10cm |
| I/O bus (better: interconnect) | Connection from device to system using connectors (PCI-Express) | 30cm..1m |
| System-Area-Network | Connections within a cluster or parallel computer | 5-25m |
| Storage-Area-Network (SAN) | Connection from processing nodes to storage modules | 5-25m |
| Local-Area-Network (LAN) | Connection between loosely coupled workstations and/or servers (*Ethernet) | 25-500m |
| Metropolitan-Area-Network (MAN) | Connections within the scope of city limits (ATM, FDDI) | ~25km |
| Wide-Area-Network (WAN) | Connections without any length restrictions, worldwide, multiplexing of a large number of connections, typically using fiber optics (SONET) | unlimited |



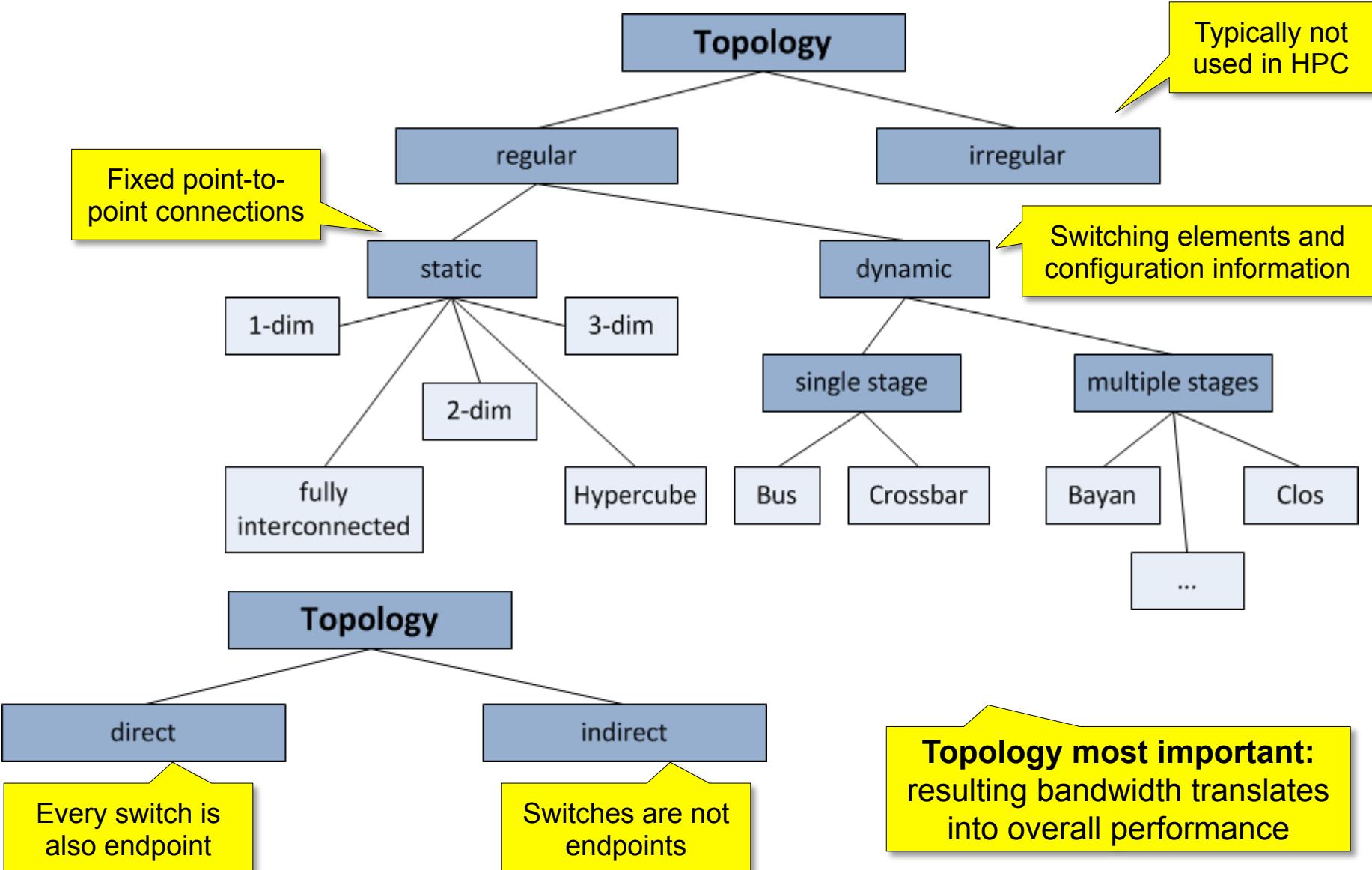
User requirements

- Costs
- Bandwidth
- Max. supported transmission length
- Scalability
- Latency
- Blocking behaviour
- Lossy/loss-less (reliability)
- In-order/out-of-order delivery

**Order of importance
application-
dependent**

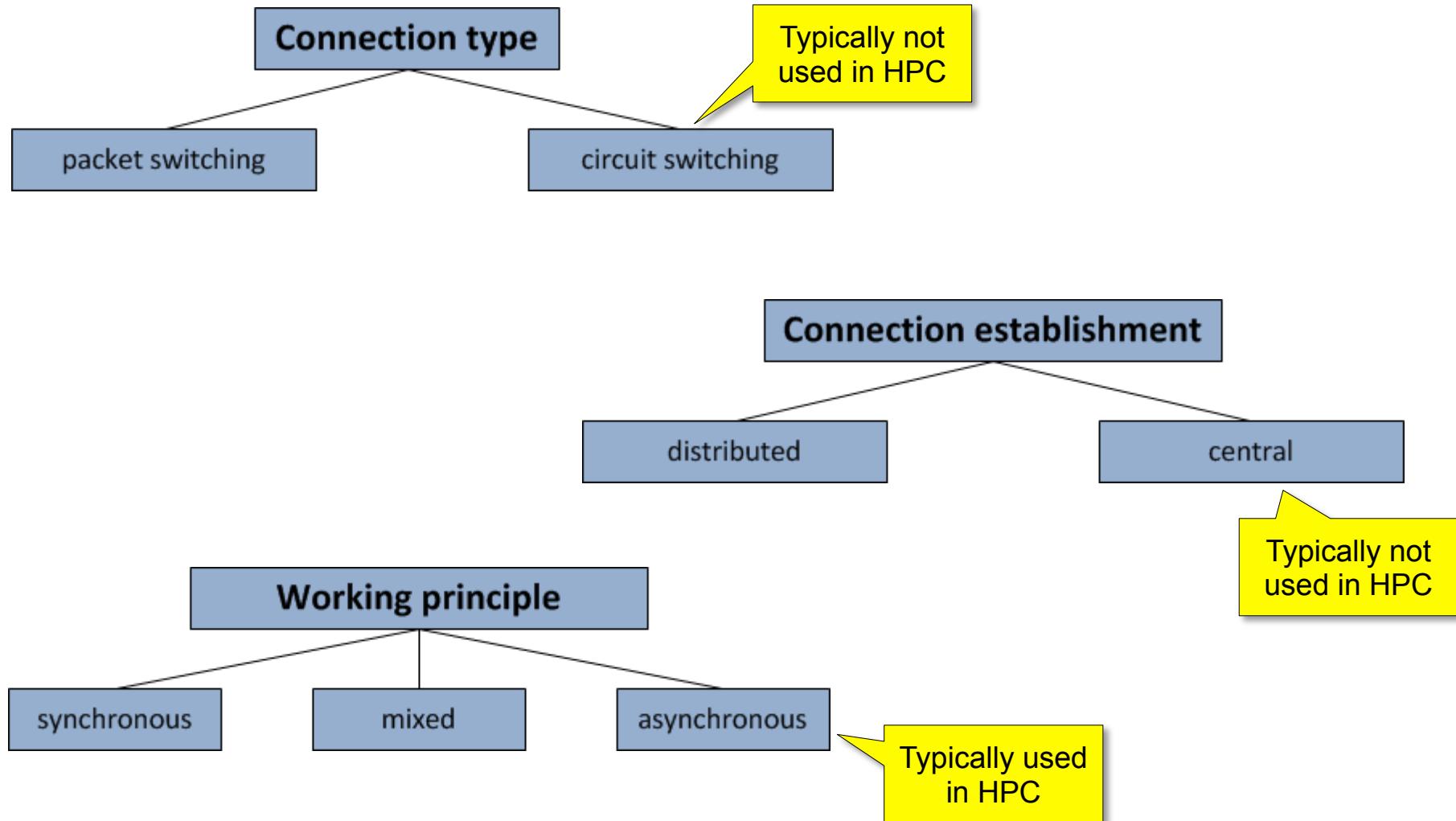


Classification





Classification



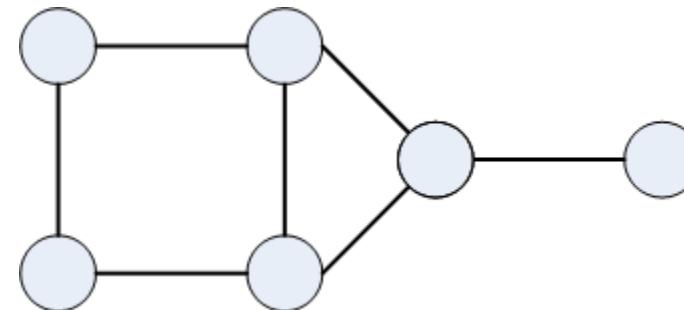


Static Topologies



Static Topologies

- Mainly used in massively parallel processors (MPP)
 - Fixed communication structure
 - Based on point-to-point connections between processing nodes
 - Node, processor, ...
- Representation
 - Node as node, connection as edge
 - Directed or undirected graph

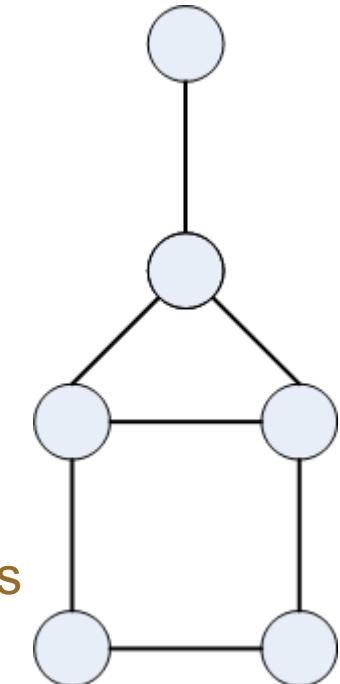


Representation of a static IN as graph



Properties of (Static) Topologies

- Topological and functional properties
- **Node degree:** Number of connections per node
 - As few as possible due to costs
 - Fixed degree mandatory for scalability
- **Diameter**
 - Maximum distance in hops between any two node pairs
- **Symmetry**
 - IN is symmetric if the view of the IN from each node is identical
- More:
 - Scalability, blocking behaviour, costs, latency, fault-tolerance, max. expansion



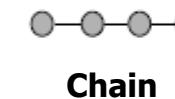


Static Topologies - Examples

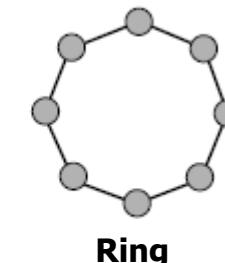
■ Trivial ones

- Chain
- Ring
- Star
- (binary) Tree

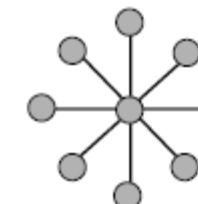
**Node degree?
Diameter?
Symmetry?**



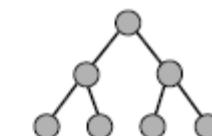
Chain



Ring



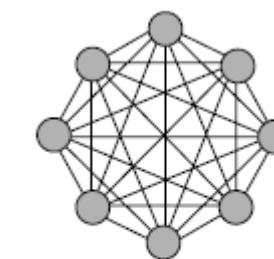
Star



(Binary) Tree

■ Completely interconnected

- Every node is connected to every other node
- Not used in practise
- Max. distance from one node to another = 1
 - „1 hop“



Completely interconnected

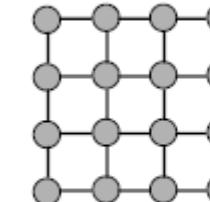


Static Topologies - Examples

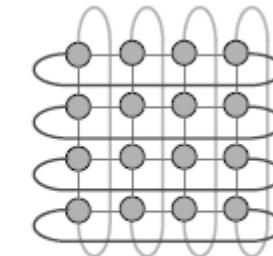
■ Grid or Mesh

- Nearest neighbor mesh
- N-dimensional mesh suitable for n-dimensional problems

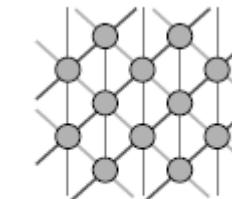
Node degree?
Diameter?
Symmetry?



Grid



2D Torus



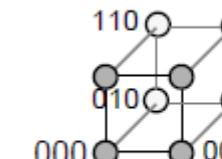
Hexagonal grid

■ Hexagonal grid

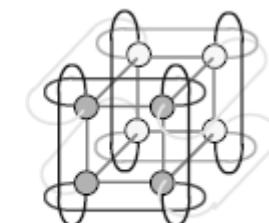
- 2D, but maps nicely to 3D problems
- Systolic algorithms

■ Torus

- Based on grid with wrap-around links
- Better connectivity



Cube



3D Torus



Static Topologies - Examples

■ Hypercube

- Given: n dimensions
- 2^n nodes,
- $n \cdot 2^{n-1}$ connections,
- n connections per node
- diameter = n

■ Better properties than a grid

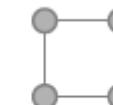
- Limited scalability (node degree)

■ Mainly used in the beginning of MIMD-based parallel computing: nCube

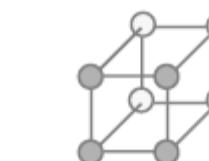
Construction:
double each
node per
additional
dimension



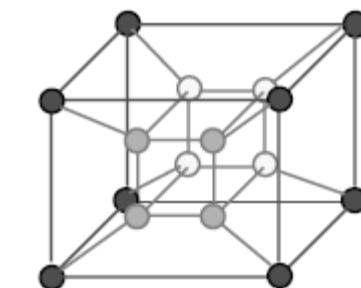
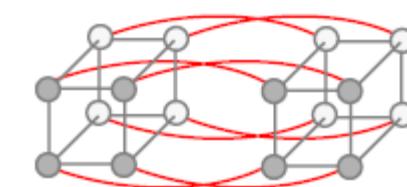
1D Hyperwürfel



2D Hyperwürfel



3D Hyperwürfel



4D Hyperwürfel



Properties of Static Topologies

Translates
into scalability

Only scalability with
regard to topology

| Topology | Node degree | Diameter | Number of connections | Scalable | Symmetric |
|---------------------------|-------------|------------------|-----------------------|----------|-----------|
| 1D grid (chain) | 2 | N-1 | N-1 | Yes | No |
| 1D torus (ring) | 2 | (N-1)/2 | N | Yes | Yes |
| 2D grid | 4 | $2(N^{1/2}-1)$ | $2N-2N^{1/2}$ | Yes | No |
| 2D torus | 4 | $N^{1/2}-1$ | $2N$ | Yes | Yes |
| 3D grid | 6 | $3(N^{1/3}-1)$ | $3N-3N^{1/3}$ | Yes | No |
| 3D torus | 6 | $3/2(N^{1/3}-1)$ | $3N$ | Yes | Yes |
| Hypercube | $\log_2 N$ | $\log_2 N$ | $N \log_2(N/2)$ | No | Yes |
| Binary Tree | 3 | $2(\log_2 N-1)$ | N-1 | Yes | No |
| Completely interconnected | N-1 | 1 | $N(N-1)/2$ | No | Yes |



Properties of Static Topologies

- Filtering for scalability and symmetry:
only n-dimensional tori
- Torus vs. Mesh
 - Basically only advantages for tori: highly reduced diameter, symmetric
 - Slightly higher connection count is not relevant in practise
- Side note: Binary tree
 - Disadvantage: root is bottleneck
 - Typically only used for specialized tasks: synchronization and collective communication (barrier, multi-/broad-cast)



Dynamic Topologies



Dynamic Topologies

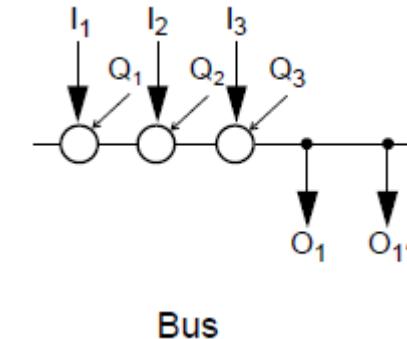
- Dynamic INs are based on **configurable switching elements**

- Different number of stages

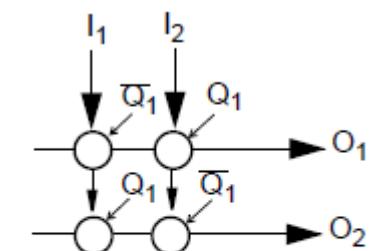
- Single stage

- Building blocks
- Shuffle, crossbar, bus
- Representation as graph: switching elements are nodes, connections are edges
- Control signals Q_i
- Inputs I_i
- Outputs O_i

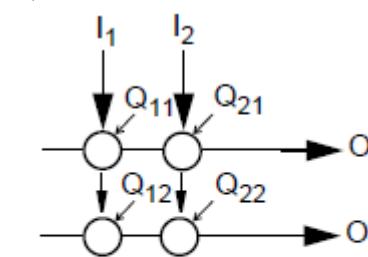
- Today basically only crossbar used



Bus



shuffle

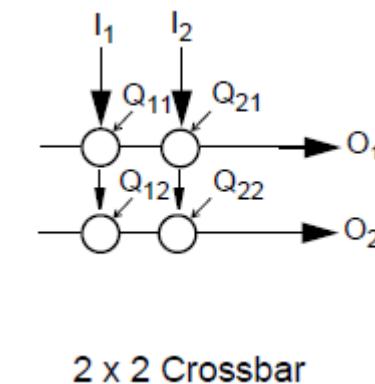


2 x 2 Crossbar



Dynamic Topologies: Crossbar

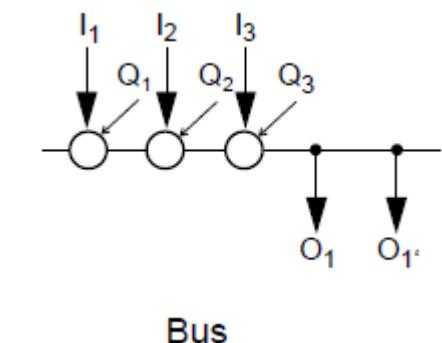
- Most universal element
- Can connect arbitrary combinations of inputs and outputs
 - Broadcast
- Conflicts avoided by arbiter
 - For all i : only one $Q(i,y)=1$
- Logical complexity is $O(N^2)$
 - For N inputs and N outputs
 - Due to VLSI technology basically no limitation
 - Most limiting today is pin count
 - Number of pins for a certain package
 - Pin is several orders of magnitude larger than a transistor!
 - Micrometers vs. nanometers
 - Pin limitation





Dynamic Topologies: Bus

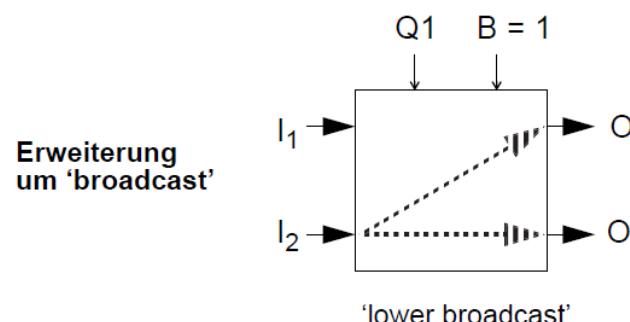
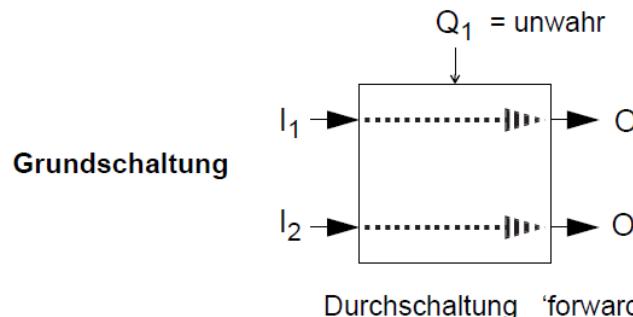
- A bus is basically a crossbar with a $1 \times m$ configuration
 - Only one driver at a time
 - High blocking potential
 - Arbiter required
 - Limited operation frequency
 - Length of connection, capacities, signal levels
 - Limited number of nodes
- Advantages: implicit broadcasts
 - Simplicity
 - Snooping protocols for cache coherency
- Today: almost vanished
 - Except human I/O and other peripherals





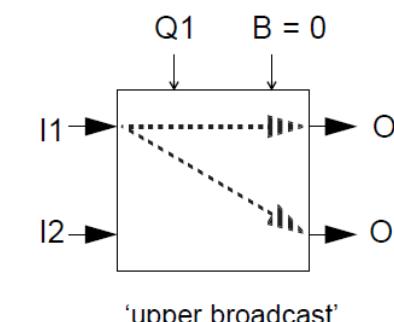
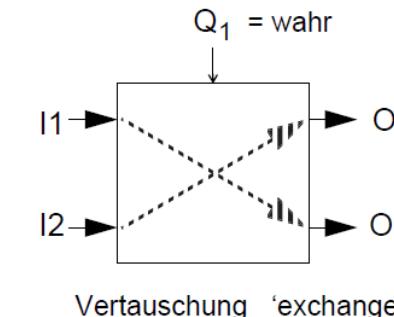
Dynamic Topologies: Shuffle

- A shuffle is basically a crossbar with a restricted set of configurations
 - „Forward“ or „exchange“
 - Possible extensions: upper and lower broadcast



- Shuffle only as 2x2 element available

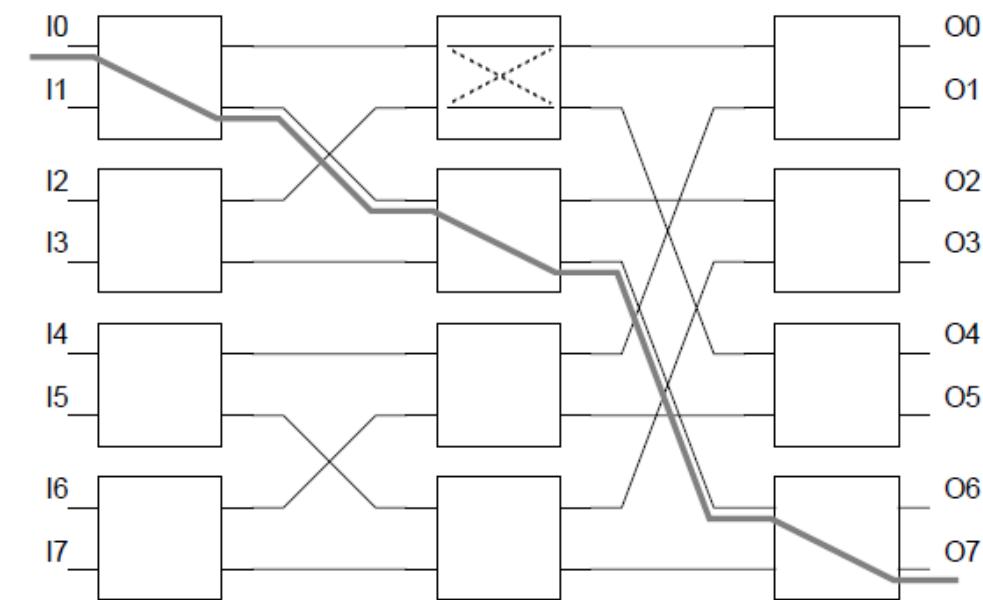
- Larger structures based on shuffle as building block





Dynamic Topologies: multi-stage

- **No single-stage element scales!**
- Multiple stages with one-staged elements as building blocks
 - (Bus,) crossbar, **shuffle**
- Examples
 - Banyan, Baseline, Cube, Delta, Flip, Indirect Cube, Omega
- Basically identical, only connectivity differs



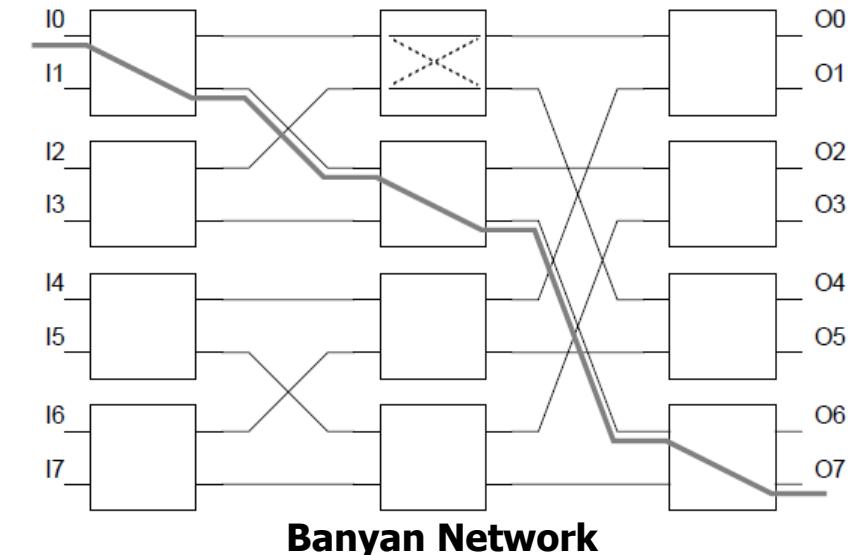
Banyan Network

Note that the shaded connection may block other connections

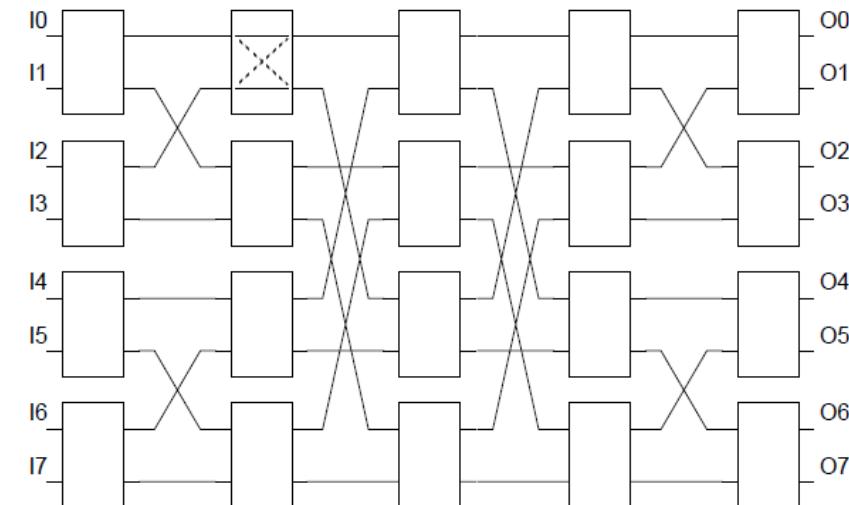


Dynamic Topologies: multi-stage

- **Unidirectional**
 - N inputs and N outputs
- **Properties**
 - $\log_2 N$ stages
 - $N/2 * \log_2 N$ shuffle elements
- **Blocking!**
 - Unlike crossbar
- **Improved blocking behaviour**
 - Additional stages
 - Benes network, composed of two banyan networks
- ➔ Nonblocking



Banyan Network

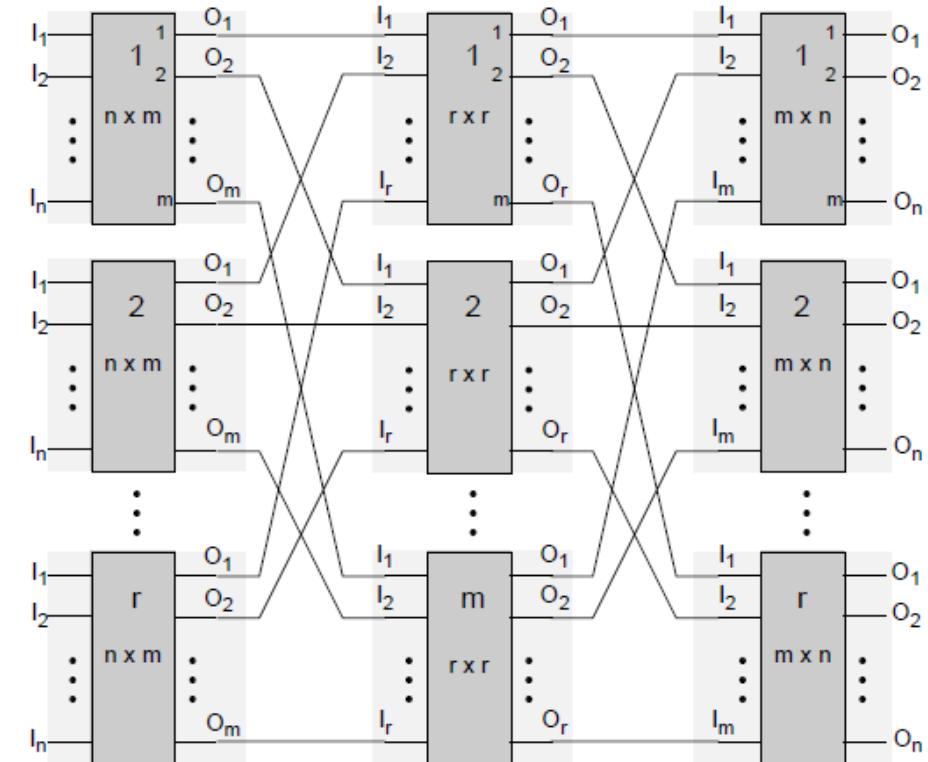


Benes Network



Dynamic Topologies: multi-stage

- Use of crossbars instead of shuffles: CLOS network
 - Advantages of crossbars (no blocking) and of multi-staged INs (reduced complexity)
- 1-stage CLOS: identical to crossbar
- 2-stage CLOS: blocking
- 3-stage CLOS: nonblocking (see Banyan-Benes)
- Each CLOS can be seen as a crossbar with higher degree



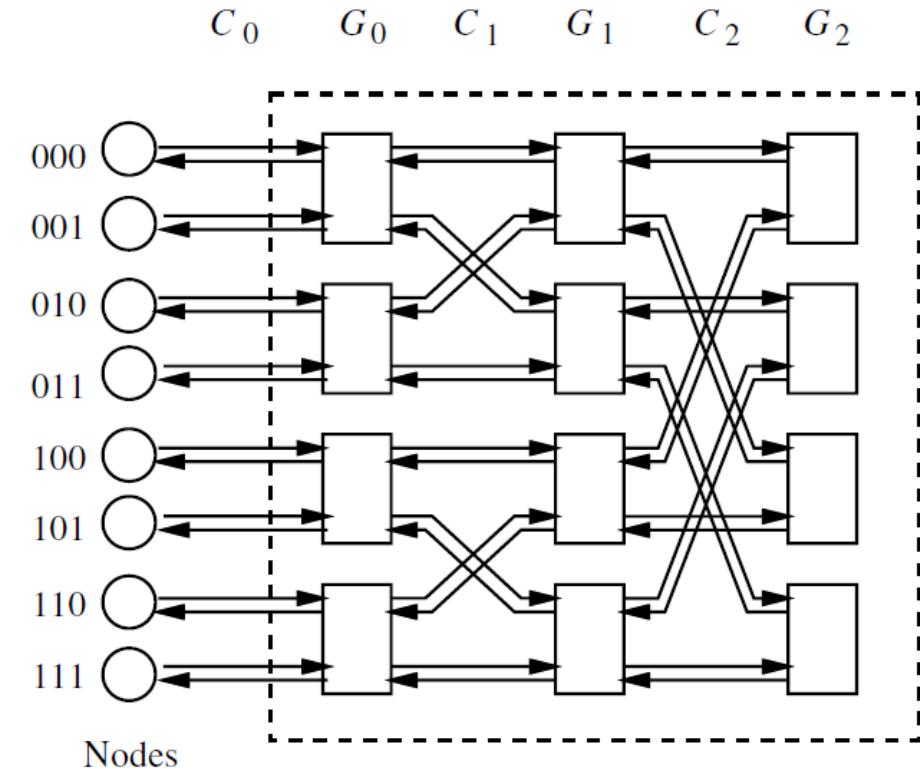
CLOS Network

Notice the 3 different types of XBARS used
Assuming $n=m=r$ and a 16x16 building block:
256x256 CLOS



Dynamic Topologies: BMIN

- BMIN: bi-directional multi-stage IN
 - Similar to before, but inputs/outputs all on the left
- Switching elements are extended
 - Forward
 - Backward
 - Turnaround
- Alternate paths
 - Path diversity



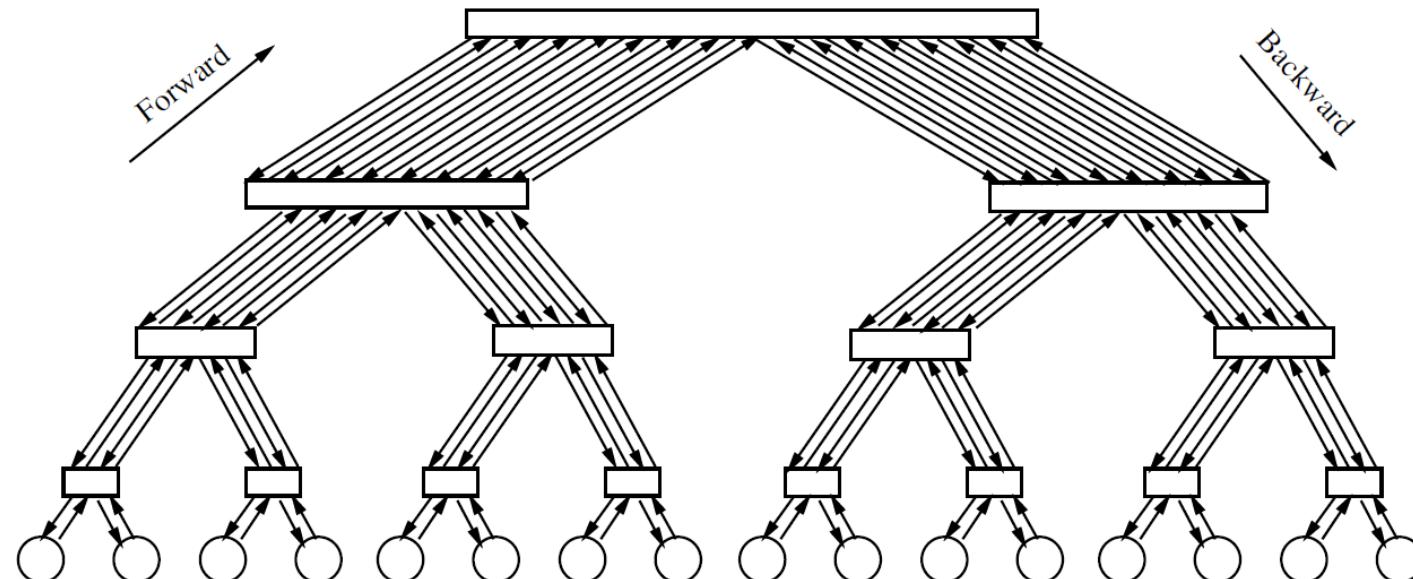
8-node butterfly BMIN

[Duato et al., Interconnection Networks, 2003]



Dynamic Topologies: BMIN

- Remember the nice scalability of binary trees
 - Replace graph nodes with switching elements and increase number of connections accordingly
- Fat Tree
 - Bottleneck at root is avoided by appropriate provisioning
 - Typically uses crossbars
 - Main disadvantage: heterogeneity, different elements required

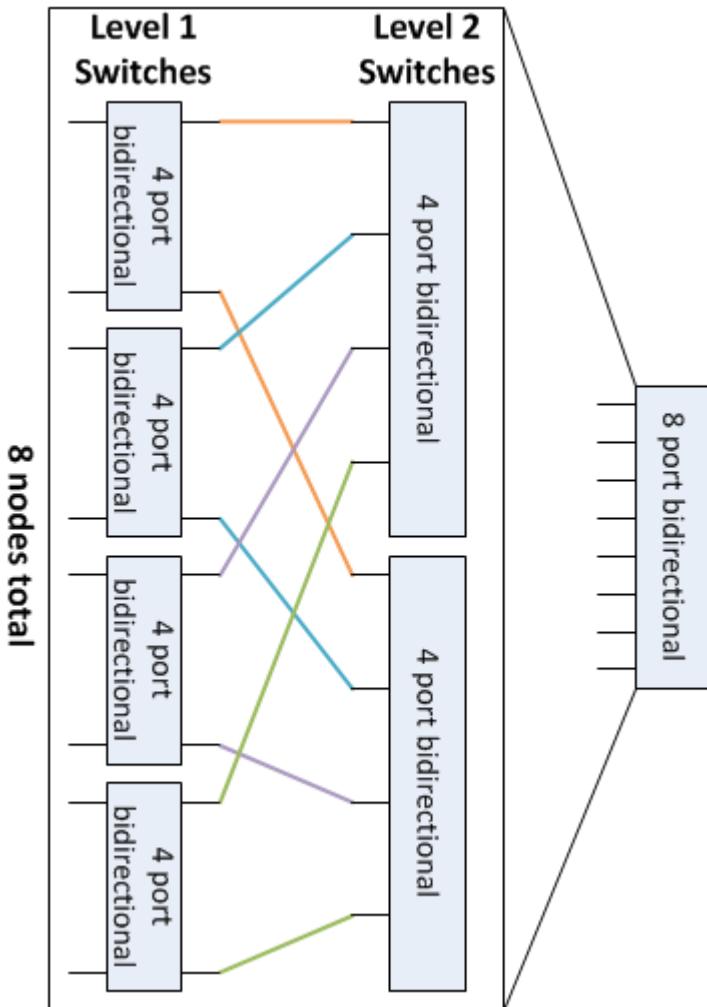


BMIN with turnaround viewed as **Fat Tree** – switching elements are typically crossbars or CLOS



Dynamic Topologies: BMIN

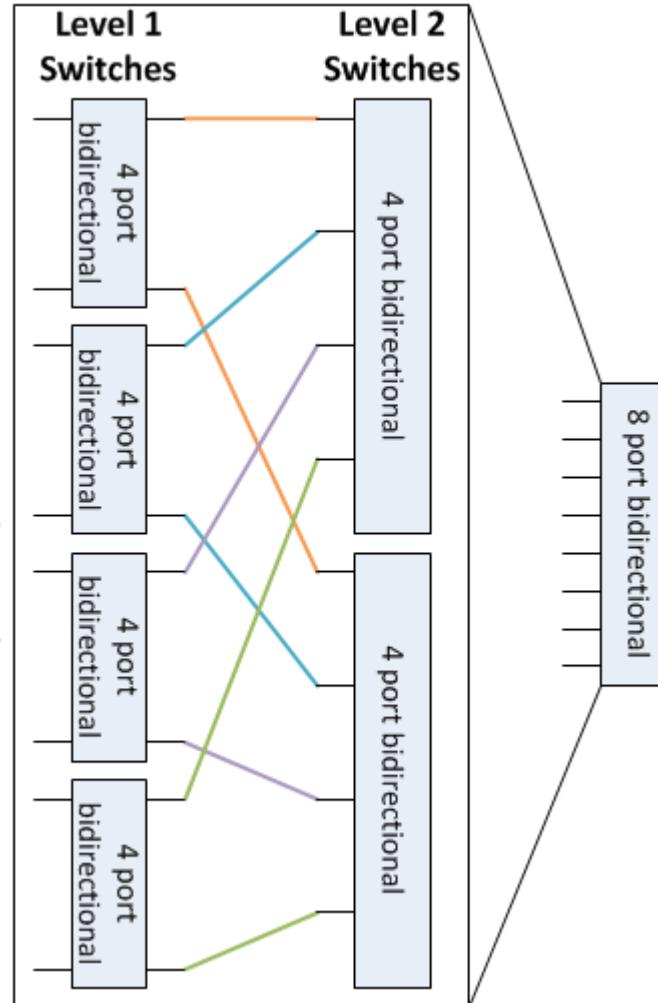
- Typically complete Fat Tree is based on one building block
- n -port crossbar switch (single chip)
- „Fatter“ switches constructed out of this switch in a CLOS fashion
 - 2 stages: max. $(n^2/2)$ end points
 - 3 stages: max. $(n^3/4)$ end points
- User point of view:
 - Non-blocking fat crossbar
 - But number of internal stages may increase → hop latency increases!



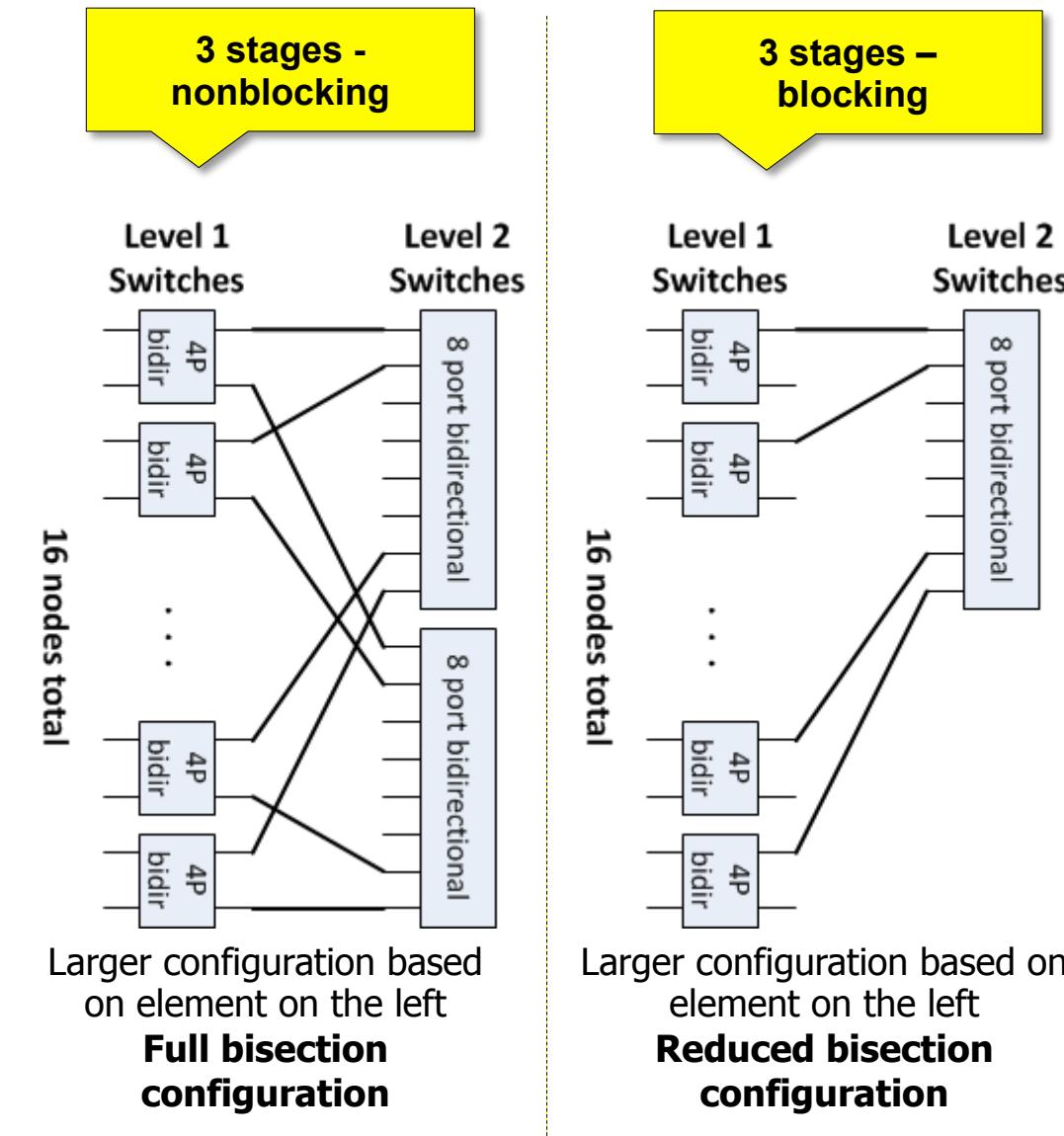
Constructing a 8 port BMIN with 4-port building blocks



Dynamic Topologies: BMIN



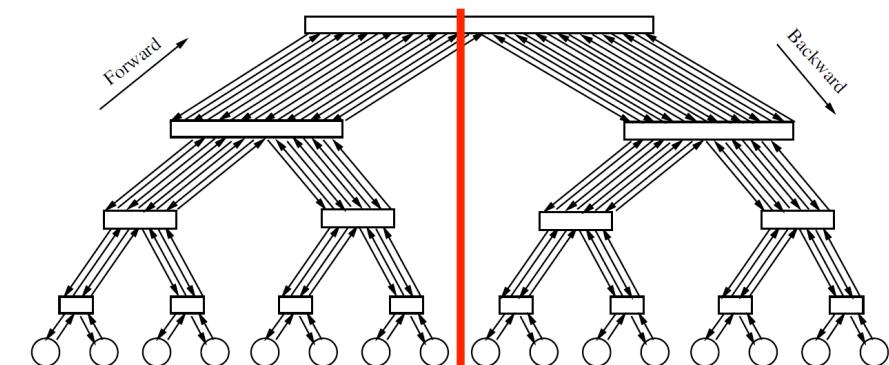
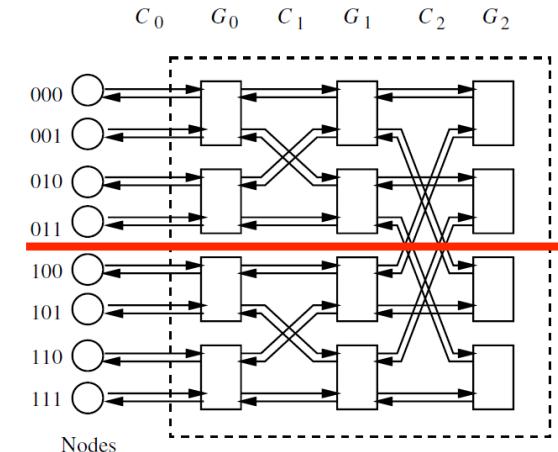
Constructing a 8 port BMIN with 4-port building blocks





Bisection bandwidth

- Bisection: Segmentation of an IN into two equal parts
 - As few cuts as possible
- Bisection BW: sum of the data rate of all cutted links
- The higher the bisection BW is, the lower is the blocking potential
 - Uniform traffic: $\frac{1}{2}$ of traffic crosses bisection





Overview of Properties

| Topology | Node degree | Diameter | Number of connections | Scalable | Symmetric | Bisection |
|---|-------------|----------------------------|---|----------|-----------|--------------------|
| 2D grid | 4 | $2(N^{1/2}-1)$ | $2N-2N^{1/2}$ | Yes | No | $N^{1/2}$ |
| 2D torus | 4 | $N^{1/2}-1$ | $2N$ | Yes | Yes | $2N^{1/2}$ |
| 3D grid | 6 | $3(N^{1/3}-1)$ | $3N-3N^{1/3}$ | Yes | No | $N^{2/3}$ |
| 3D torus | 6 | $3/2(N^{1/3}-1)$ | $3N$ | Yes | Yes | $2N^{2/3}$ |
| Hypercube | $\log_2 N$ | $\log_2 N$ | $N \log_2(N/2)$ | No | Yes | $2^{(\log_2 N)-1}$ |
| Crossbar | 1 | 1 | N^2 | No | Yes | $N/2$ |
| CLOS | 1 | 3 | $r(2n+2m)$ $(4N^2 \text{ for } r=n=m)$ | Yes | Yes | $N/2$ |
| Fat Tree, S <i>is number of stages</i> | 1 | $2(S-1);$ $S=O(\log N)$ | $N*S$ | Yes | Yes | $N/2$ |



Summary

- Interconnection networks as key in HPC
 - INs are pervasive today: from smartphones to microcontrollers to large computing facilities
- Topologies and their properties
 - Direct vs. indirect
 - Static vs. dynamic
 - Node degree, diameter, number of connections, symmetry, scalable, (non-)blocking
- Bisection bandwidth
- Many more topologies possible
 - Regular but hierarchical
 - Irregular



Credits & Further Reading

- Duato, Yalamanchili, Ni:
Interconnection Networks -
An Engineering Approach.
2002
- Dally, Towles: Principles
and Practices of
Interconnection Networks.
2003

