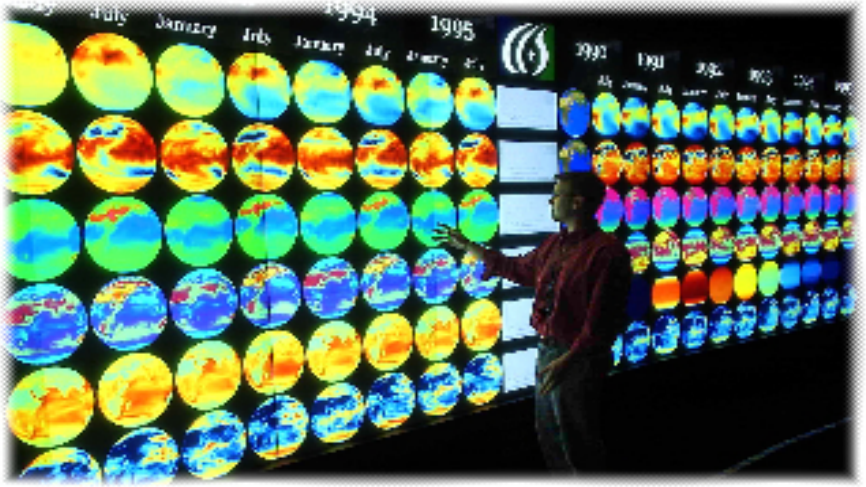
Parallel Computing : An Overview, Supercomputers, MPI, OpenMP, and More..



Kwai Wong

Joint Institute for Computational Sciences

The University of Tennessee, Knoxville

www.nics.utk.edu

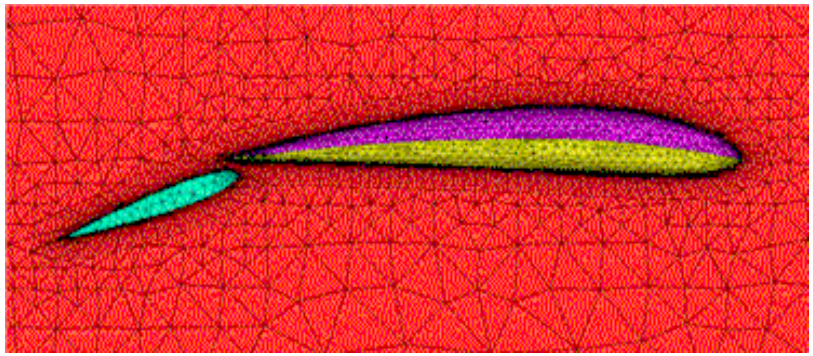
kwong@utk.edu

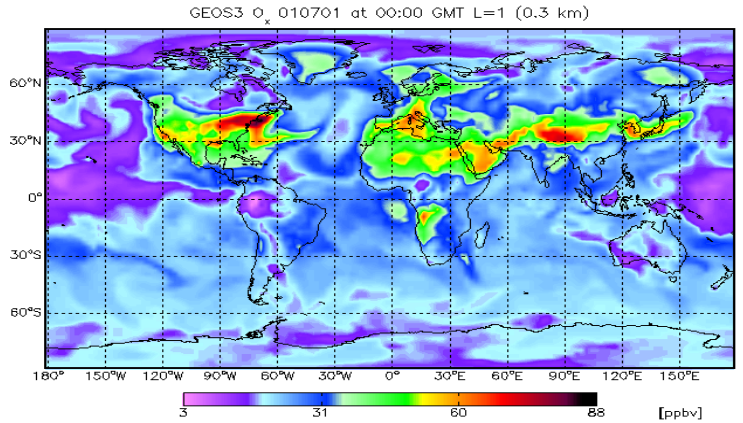
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES

A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

Need of Parallel Computer

- Requirement of computational capacity depends on applications and formulations and what you want to achieve
- Length Scale - resolution of the dimension, e.g. number of grid points
- Time Scale - resolution of duration, e.g. number of time step





- 2D problem :
- grid points 100x100 = 10000 pts
- a vector of 10000 elements ~ 80 KB
- need 10 such vectors ~ 800KB
- Steady State in seconds

- 3D problem :
- grid points 10000x10000 x100 = 10e10 pts
- 10e10 unknowns ~ 80 GB
- need 10 such vectors ~ 800 GB MEMORY
- 100 years simulation !!

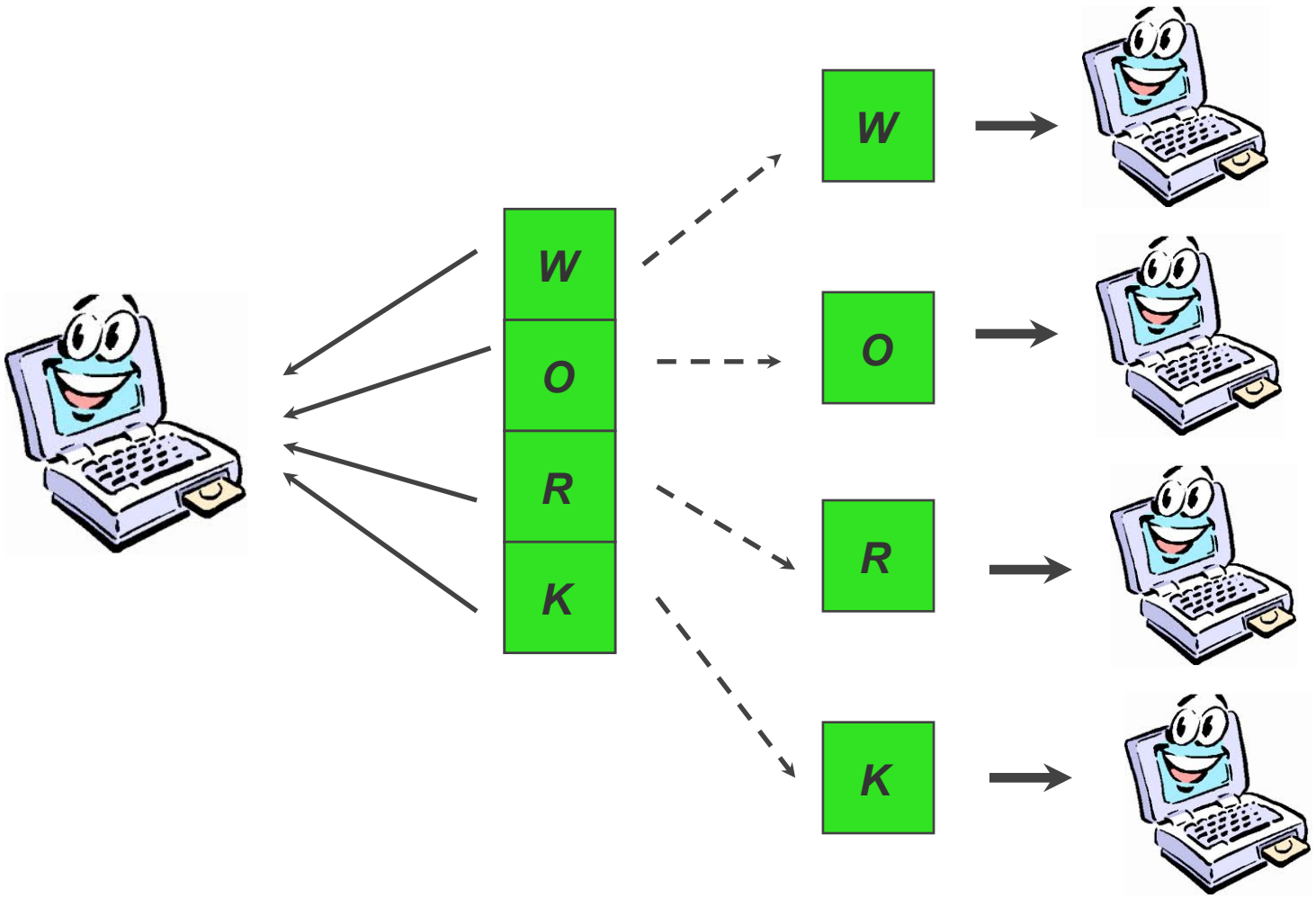
NEED MULTIPLE WORKERS and RESOURCES – PARALLEL COMPUTER

Parallel Computing

Division of work into smaller tasks

Multiple computers work on smaller tasks simultaneously

>> Reduce Wall Clock Time <<



Issues of Parallel Computing

- Pros :
  - decrease wallclock time
  - deliver huge amount of memory
  - Allow realistic simulation
- Cons :
  - Difficult to construct
  - Efficient parallel algorithm may need some thoughts
  - Cost of program development

KEYS:

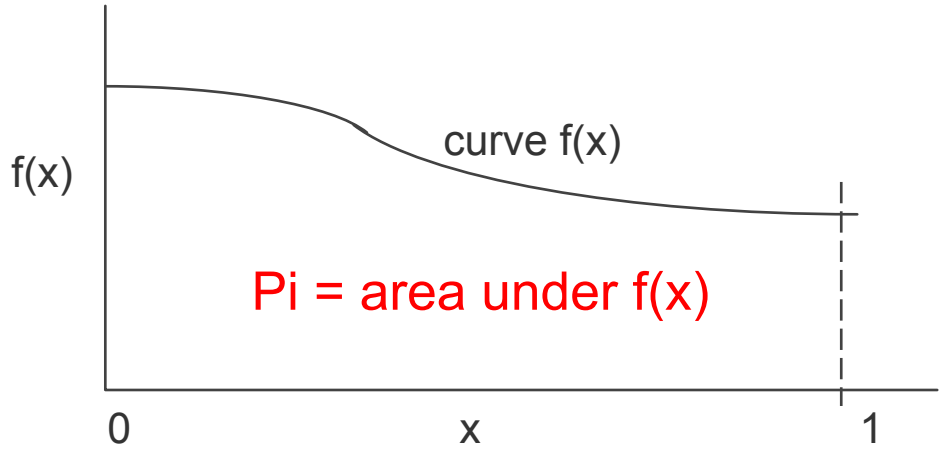
1) LOAD BALANCE - same amount of work for every processor
2) LOCALITY - minimize communications among processors
3) PORTABILITY - work well on different platforms of computers
4) SCALABILITY - can solve larger problem efficiently

Parallel Programming Example: Calculating Pi

- Use numerical integration to compute Pi
- Let $f(x) = 4 / (1+x^2)$ then integrate f(x) from x = 0 to 1
- Using the rectangle rule

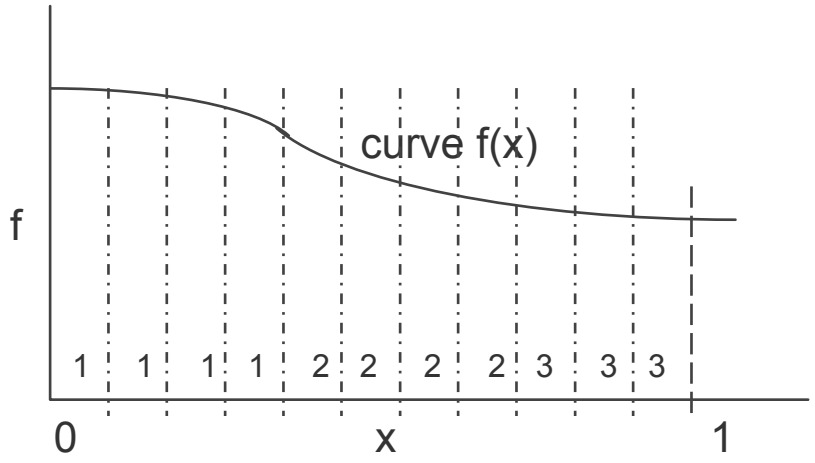
$$R_n(f) = h\sum_{i=1}^{n} f(x_i)$$

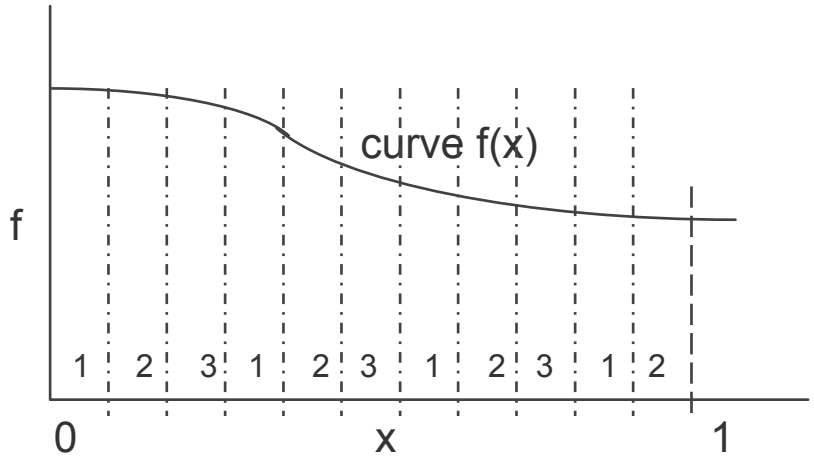
where n = the number of intervals, h = 1/n is the rectangle width and $x_i = h.(n-0.5)$ is the midpoint of each rectangle



Pi Using Rectangles

- **Method: Divide area under curve into rectangles and distribute the rectangles to the processors**
- **Suppose there are 3 processors, how should the distribution be done?**





Parallel Performance Measure

- Using multiple processors you hope your program will go faster
- Observed Speedup using N processors to accomplish a task

$$\text{Speedup} = \frac{T(1)}{T(N)} \quad \frac{\text{Time taken using 1 processor}}{\text{Time taken using N processors}}$$

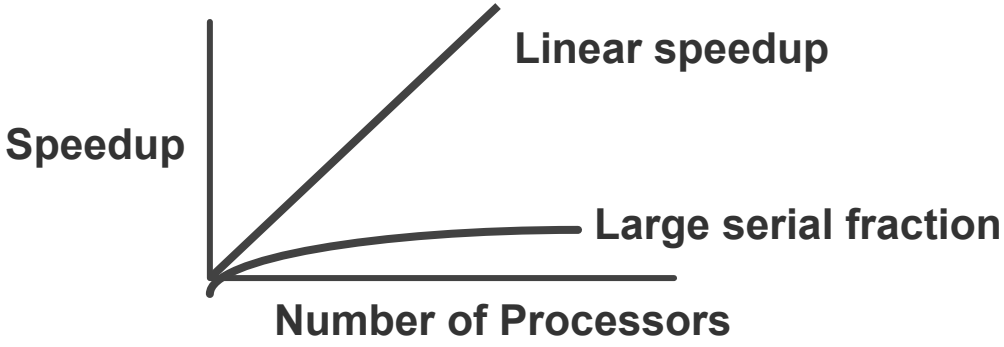
- To be fair, should use the “best” serial algorithm on 1 processor, not the parallel algorithm, simply restricted to 1 processor
- Linear speedup:
  - Two processors take 1/2 the time of 1 processor, so speedup =2
  - N processors take 1/N the time of 1 processor, so speedup =N
- Superlinear speedup
  - May be obtained occasionally, usually due to cache and memory improvements

Amdahl's Law

- Maximum speedup is limited by the serial fraction of a program
- Serial code

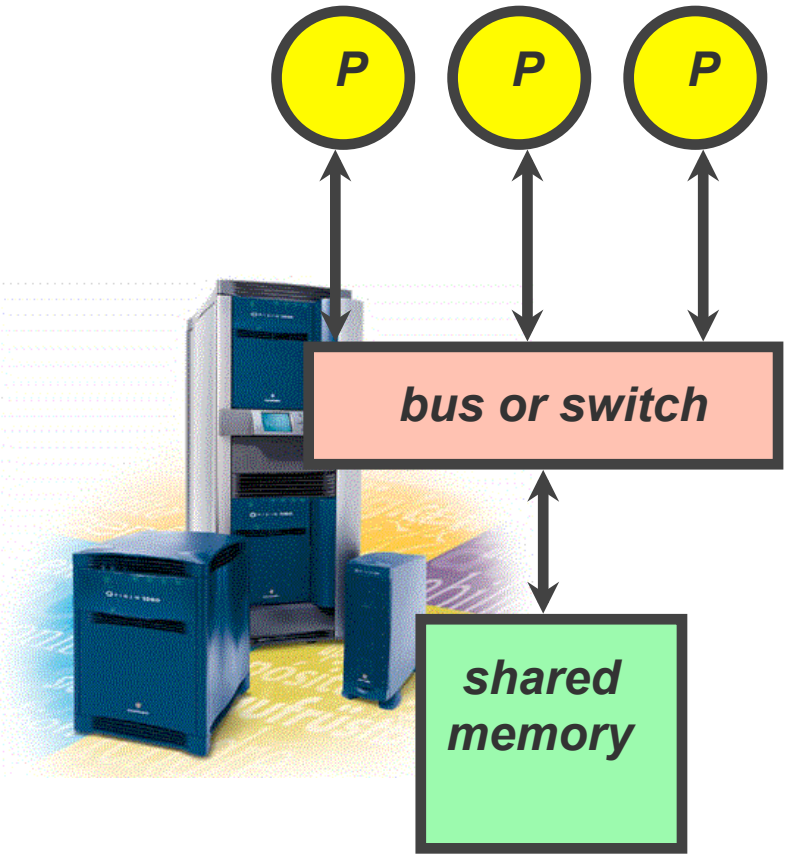
  - Time taken: 100
- Parallel code (using num procs P >> 10)

  - Time taken =10, maximum speedup=10, regardless of P

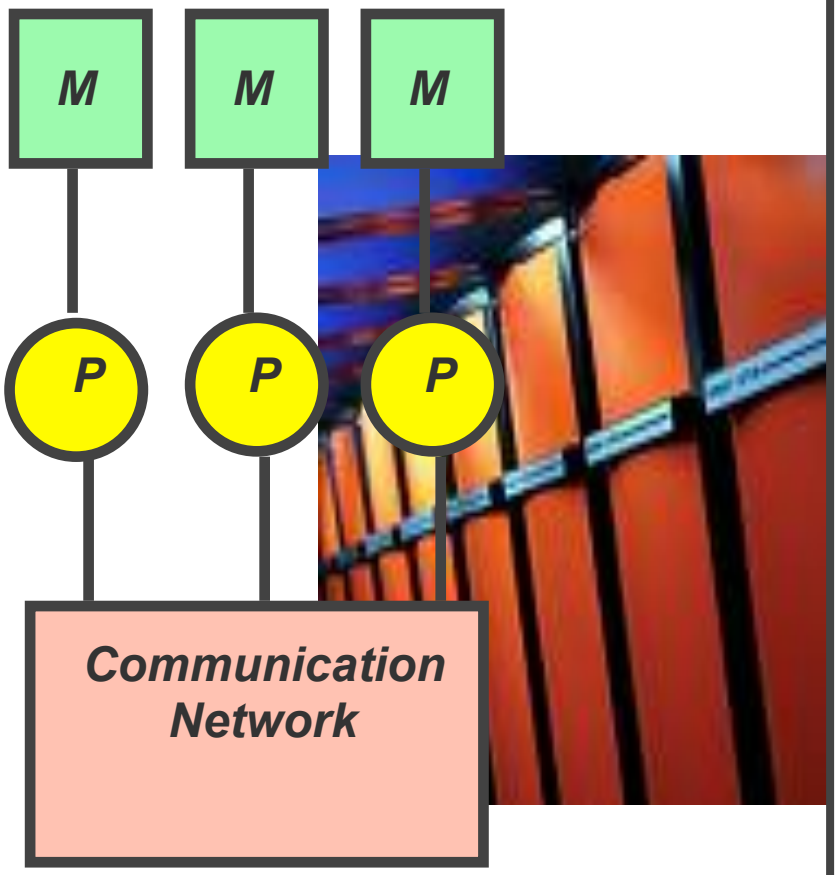


Parallel Computers (simple story)

Shared Memory Systems (SMP)
(SGI, HP, PC, IBM)
(USE OPENMP)



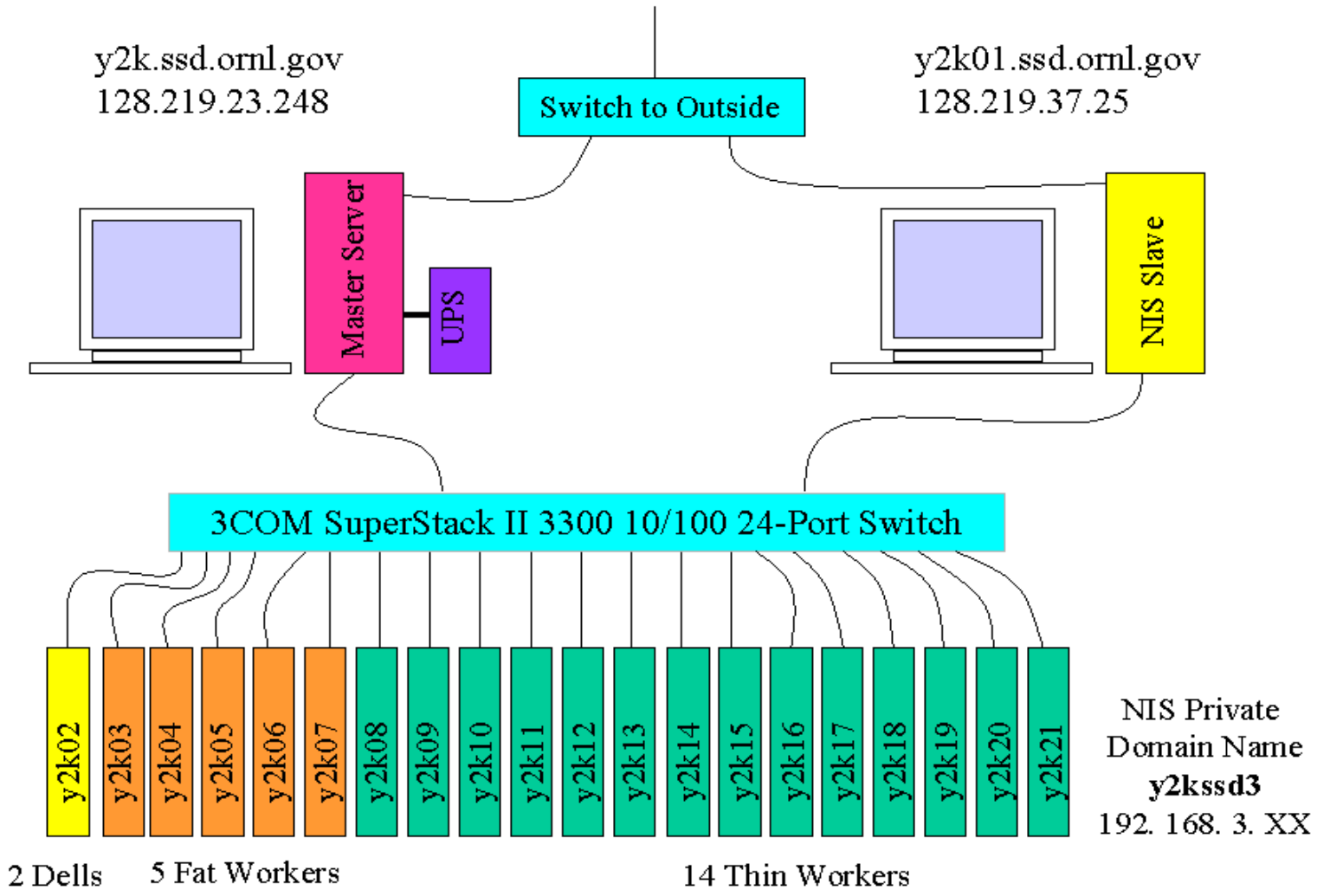
Distributed Memory Systems (MPP)
(IBM SP, Cray XT or PC Cluster)
(USE MESSAGE PASSING)



Simple Parallel Computer

Many commodity units connected by a COS interconnect

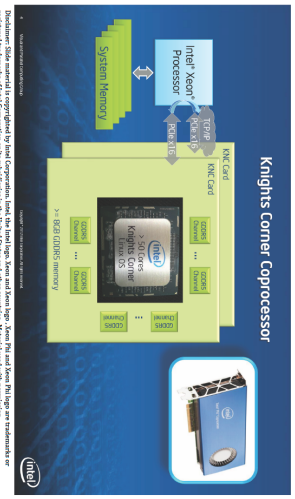
Simple Hardware Schematic



Modern Supercomputers

Commodity plus Accelerator Today

Accelerator (MIC)



Source: George Chrysos, Hot Chips, August 28, 2012

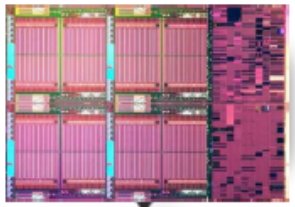
Commodity

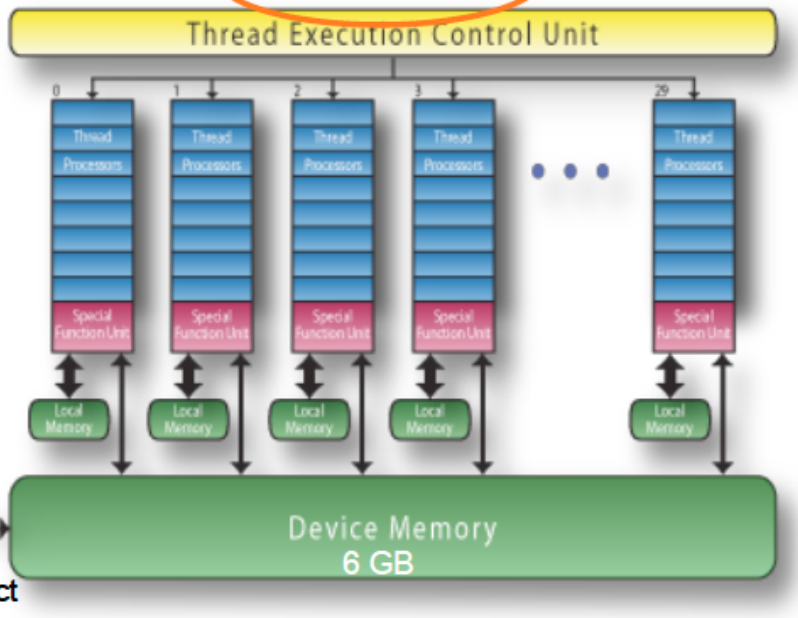
Intel Xeon
8 cores
3 GHz
8*4 ops/cycle
96 Gflop/s (DP)

Accelerator (GPU)

Nvidia K20X "Kepler"
2688 "Cuda cores"
.732 GHz
2688*2/3 ops/cycle
1.31 Tflop/s (DP)

192 Cuda cores/SMX
2688 "Cuda cores"



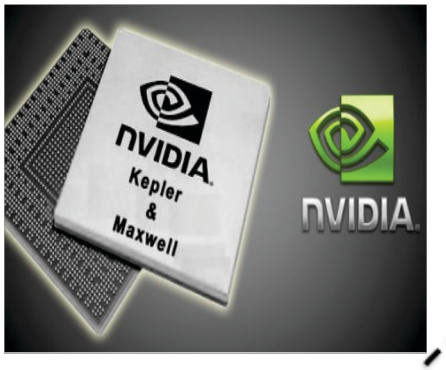


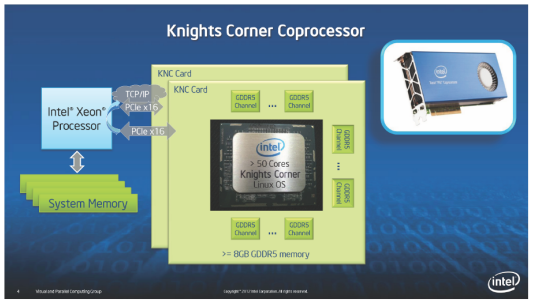
Interconnect
PCI-X 16 lane
64 Gb/s (8 GB/s)
1 GW/s

From ICL Dr. Jack Dongarra : icl.cs.utk.edu

Example of typical parallel machine

Shared memory programming between processes on a board and a combination of shared memory and distributed memory programming between nodes and cabinets





Source: George Chrysos, Hot Chips, August 28, 2012

From ICL Dr. Jack Dongarra : icl.cs.utk.edu

Ride on the Hardware Technology Curve

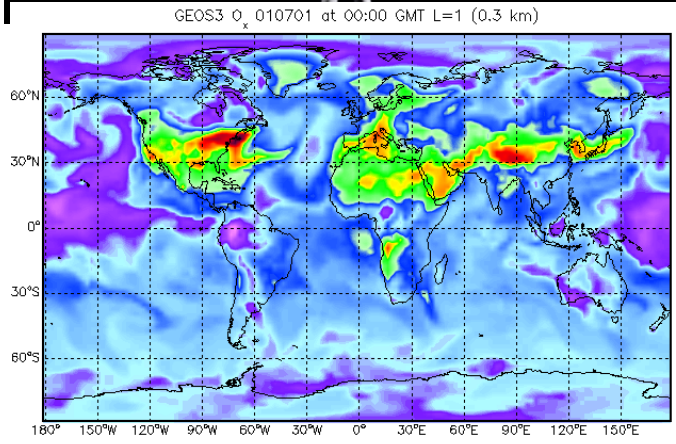
TACC – Stampede
10 PFLOPS

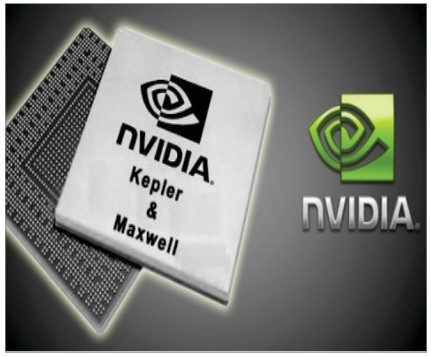
EVEREST facility

- 35 million pixel, 27-tile PowerWall
- Interactive, large-scale, collaborative data analysis
- 27 NVIDIA 8800 GTX GPUs, dedicated Linux cluster
- 30 feet by 8 feet

ORNL – TITAN
20 PFLOPS

64 cores, ~TFLOPS





Kepler (2012), ~TFLOPS

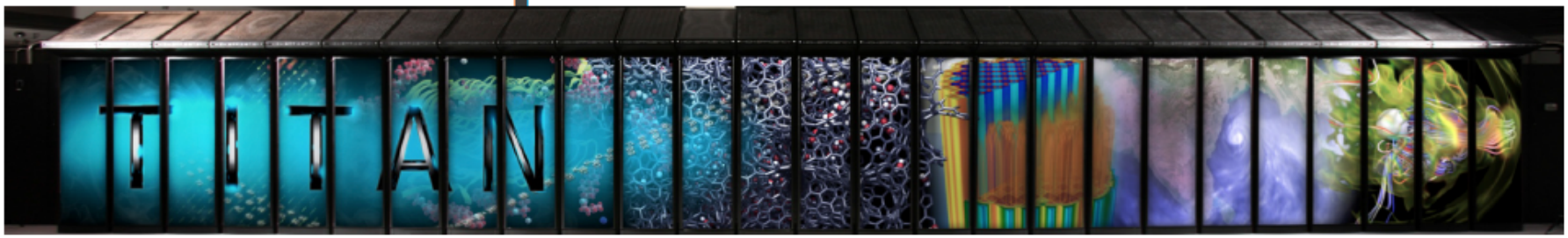
Transformational Science : RT Simulation

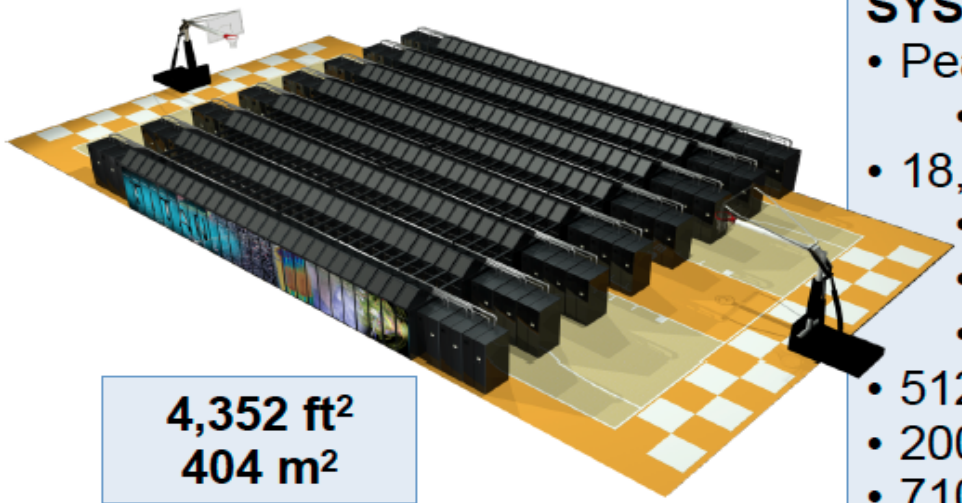
kraken

jaguar

Fermi

ORNL's "Titan" Hybrid System: Cray XK7 with AMD Opteron and NVIDIA Tesla processors





4,352 ft²
404 m²

SYSTEM SPECIFICATIONS:

- Peak performance of 27 PF
  - 24.5 Pflop/s GPU + 2.6 Pflop/s AMD
- 18,688 Compute Nodes each with:
  - 16-Core AMD Opteron CPU
  - 14-Core NVIDIA Tesla "K20x" GPU
  - 32 GB + 6 GB memory
- 512 Service and I/O nodes
- 200 Cabinets
- 710 TB total system memory
- Cray Gemini 3D Torus Interconnect
- 9 MW peak power

OAK RIDGE National Laboratory

- (CPU) 2.26 x 4 x 18688 = 2.392 ; (GPU) PF 1.31 x 18688 x 14 = 24.27 PF
- 17.5 PFLOPS (HPL) 64.8% ; ~ 10 times faster than jaguar; 9 Megawatt,

TOP 500 – www.top500.org

TOP500 List - November 2009 (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

nex

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
2	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM	122400	1042.00	1375.78	2345.50
3	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	98928	831.70	1028.85	
4	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70	2268.00
5	National SuperComputer Center in Tianjin/NUDT China	Tianhe-1 - NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband / 2009 NUDT	71680	563.10	1206.19	

Numbers : Lots of Them:

- Core : computing unit : processor
- Dual core machine (Intel or AMD CPU) : a CPU with 2 cores, each core is a 2.4 GHz computing unit with 2GB of RAM (memory in the processor not disk space)
- Binary bits (b) : "0" or "1" , 1 Byte (B) = 8 bits
- Binary number : 11111111= $(2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^{0)} = (2^8 - 1) = 255$!!
- 32 bits machine or operating system => largest integer (all positive) = $(2^{32} - 1)$ = (4,294,967,296 -1) or range of integer = $-(2^{31})$ to $(2^{31} - 1)$
- 64 bits machine or operating system => range of integer = $-(2^{63})$ to $(2^{63} - 1)$
- Kilo (K) = 10^3 (or 2^{10}) ; Mega (M) = 10^6 (or 2^{20}); Giga (G) = 10^9 (or 2^{30}); Tera (T billion) = 10^{12} (or 2^{40}) ; Peta (P) = 10^{15} (or 2^{50})
- FLoating Point Operation (+, -, / , *) : (10.1 + 0.1) * 1.0 / 2.0 = 5.1 => 3 FLOP
- FLOPS = FLOP per second :: 1 PetaFLOPS (kraken) = 10^{15} FLOP in one second
- FLOPS in a core = (clock rate) x (floating point operation in one clock cycle)
- Peak Rate = (FLOPS in one compute unit, core) x (no. of core)

Jaguar: 2009 World's Most Powerful Computer

www.olcf.ornl.gov

	jaguar XT4	jaguarpf XT5
Peak Performance	263.16 TFLOPS	2.33 PFLOPS
System Memory	61 TB	292 TB
Disk Space	750 TB	10,000 TB
Disk Bandwidth	44 GB/s	240 GB/s
Interconnect Bandwidth	157 TB/s	374 TB/s

Menu

jaguar Rating : World Fastest Computer (2009)

- FLOPS – FLoating Point Operation Per Second
- GFLOPS = 10^9 FLOPS ; TFLOPS = 10^12 ; PFLOPS = 10^15
- FLOPS = (clock rate) x (floating point operation in one clock cycle)
- Peak Rate = (FLOPS in one CPU) x (no. of CPU)
- Cray XT5 one core AMD Opteron :
  - Rpeak : (2.6 GHz) x (4) x (224162 cores) = 2331284 GFLOPS
  - Rmax : 1759000 GFLOPS ➔ 75.4% of peak

jaguar: What does it do?

- Solve a very big system of equations : Ax = b using a standard benchmark C program (HPL)
- Nmax : Size of A for HPL (Solve Ax=b) = 5474272
- Total Memory needed = (Nmax) x (Nmax) x (8 Bytes) = 239741 GB
- Memory needed per core = 1.07 GB
- Elapse Time : 2(Nmax)(Nmax)(Nmax)/3/Rmax ~ = 13 hrs

HPL (High Performance Linpack): Solving Ax = b
http://www.netlib.org/benchmark/hpl/

$$\begin{aligned} 2x_1 + 2x_2 + 2x_3 &= 1 \\ 3x_1 + 4x_2 + 5x_3 &= 2 \\ 4x_1 + 6x_2 + 7x_3 &= 3. \end{aligned}$$

$$A = \begin{bmatrix} 2 & 2 & 2 \\ 0 & 1 & 2 \\ 4 & 6 & 7 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1/2 \\ 3 \end{bmatrix}$$

$$A = \begin{bmatrix} 2 & 2 & 2 \\ 3 & 4 & 5 \\ 4 & 6 & 7 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$A = \begin{bmatrix} 2 & 2 & 2 \\ 0 & 1 & 2 \\ 0 & 2 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1/2 \\ 1 \end{bmatrix}.$$

$$A = \begin{bmatrix} 2 & 2 & 2 \\ 0 & \star & \star \\ 0 & \star & \star \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ \star \\ \star \end{bmatrix}$$

$$A = \begin{bmatrix} 2 & 2 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1/2 \\ 0 \end{bmatrix}.$$

$$x_3 = 0, \quad x_2 = 1/2 - x_3 = 1/2, \quad 2x_1 + 2_x 2 + 2x_3 = 1 \implies x_1 = 0.$$

Total operation count for Gaussian elimination with backward substitution

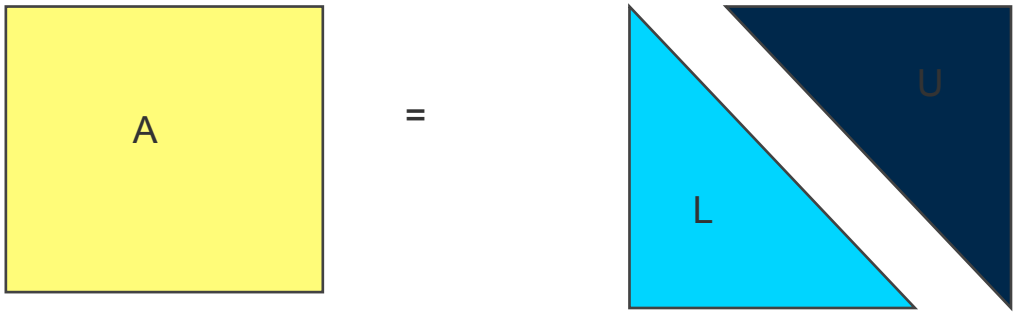
$$\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n.$$

http://wiki.math.msu.edu/index.php/Gaussian_Elimination

HPL - Gausssian Elimination

$$Ax = b$$

change A into $A = L\,U$



so $LUx = b$

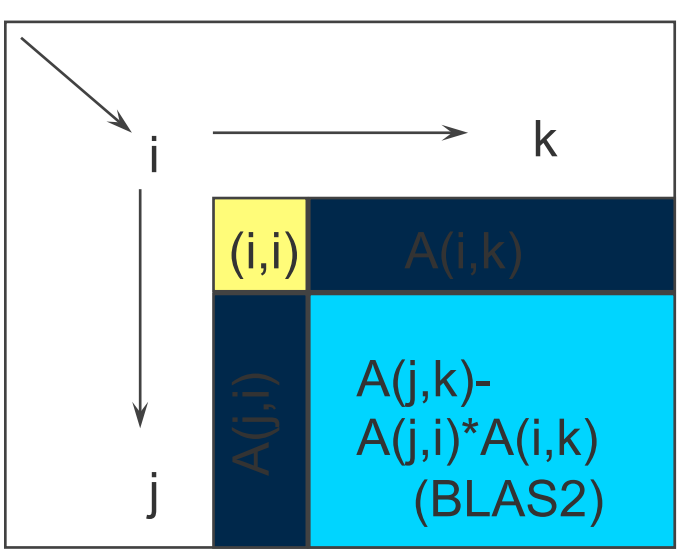
first solve $Ly = b$ by direct downward solve

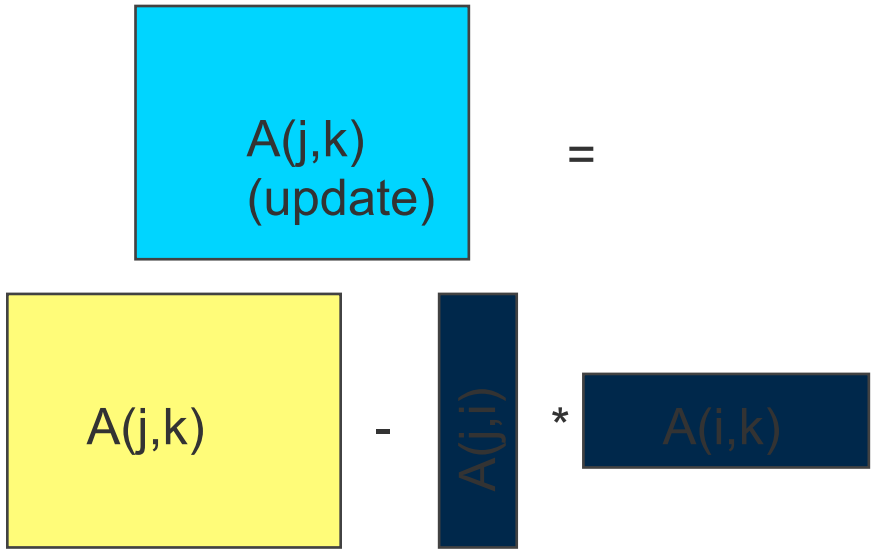
then solve $Ux = y$ by direct upward solve

LAPACK (LU)

- The inner loop consists of BLAS1 and one BLAS 2 operations.

```
for i = 1 to n-1
        for j = i+1 to n
            A(j,I) = A(j,i) / A(i,i)   <------ BLAS1 ( to BLAS2)
            for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i)* A(i,k)   <----BLAS2 ( to BLAS3)
```

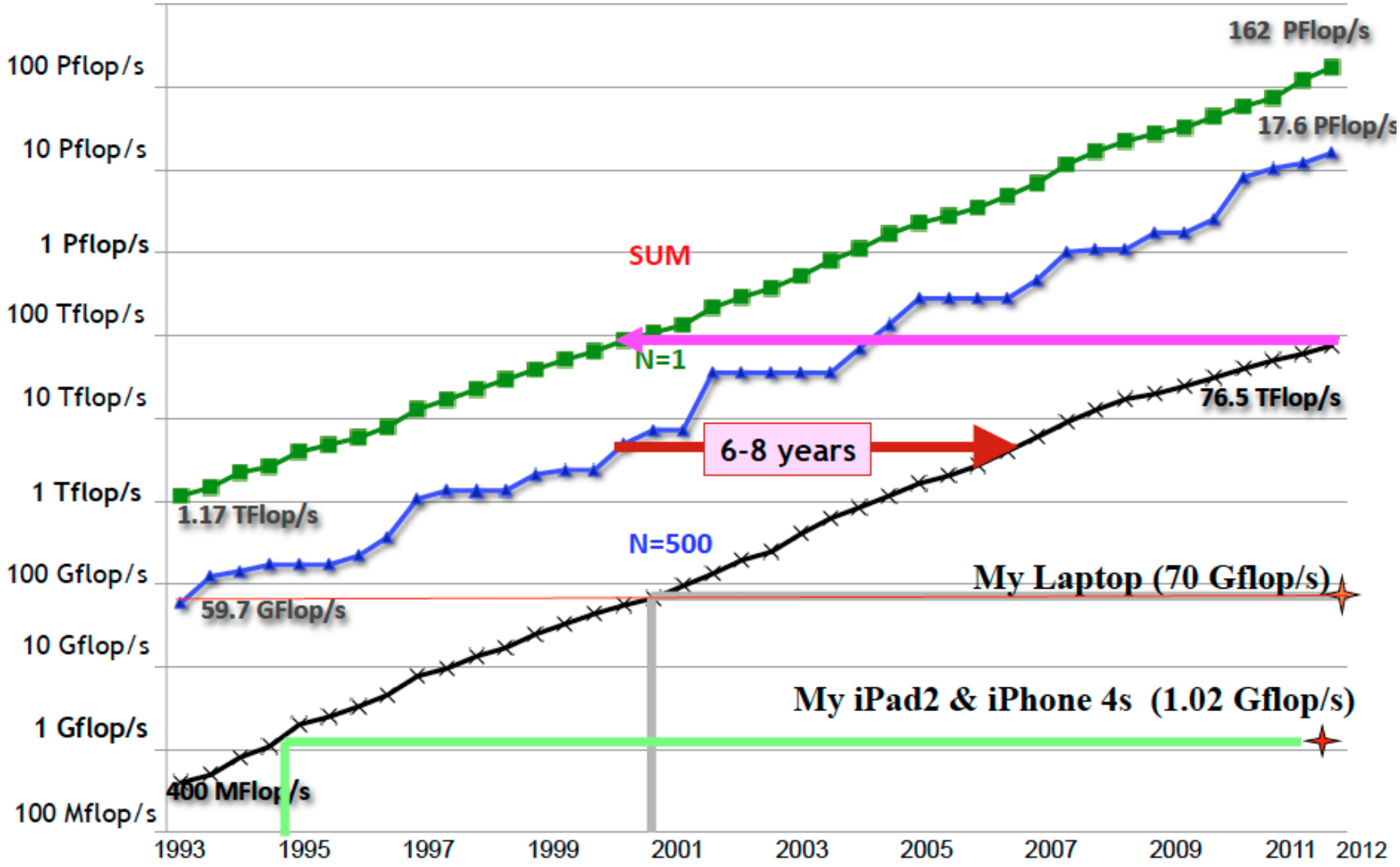




2D Block Cyclic Distribution

- Consider an 8 x 8 system of linear equations using a 2D blocked cyclic data distribution
- Matrix A is first decomposed into 2x2 blocks starting at its upper left corner, bk=2.
- These blocks are then uniformly distributed across a 2x2 processor grid, nprow = npcol =2.
- There are 4 processes in the 2D process grid, nbrow = nbcol = 2.

Performance Development of HPC Over the Last 20 Years



From ICL Dr. Jack Dongarra : icl.cs.utk.edu

JICS and NICS
What's this all about?
And where are the supercomputers?

www.jics.utk.edu
www.nics.utk.edu

Joint Institute for Computational Sciences

- JICS is a collaboration between UT and ORNL since 1991
- Joint Faculty, Research, Education, Outreach
- Kraken (65M), RDAV (10M), Keeneland (12M), Beacon (1M)
- Staffed with 40+ FTEs, multiple projects NSF, DOE, DOE, ..
- Total JICS funding > $100M

National Institute for Computational Sciences

A UT/ORNL PARTNERSHIP
NATIONAL INSTITUTE FOR COMPUTATIONAL SCIENCES
NICS

- NICS is a collaboration project in JICS
- Awarded the NSF Track 2B ($65M)
- Phased deployment of Cray XT systems
- Teragrid, XSEDE center, www.xsede.org (NSF)

Kraken: 1st Academic PetaFLOPS Computer (3rd 2009)

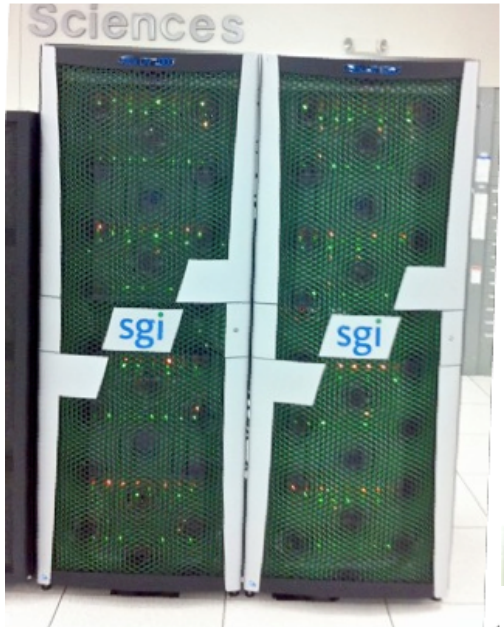
- Cray XT5 – 1.17 PetaFLOPS (Peak)
- 100 cabinets in 4 rows
- 9408 compute nodes (112896 cores)
- Each node has 12 cores - 2.6 GHz AMD (Istanbul) Processor
- 16 GB RAM per node
- 147TB of compute memory
- Scratch disk space, with 2.4PB of usable space
- www.nics.utk.edu

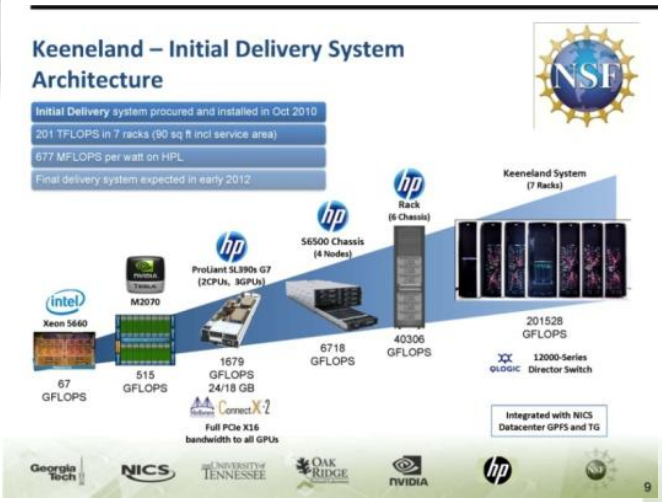
Nautilus & Keeneland

RDAV

- SGI Ultraviolet – 10 TFLOPS (Peak)
- 128 nodes x 8 cores (1024 cores) + 16 GPU
- 4.0 GB per core ; 4 TB Global Addressable RAM SMP
- 1 PB parallel file space; addressable from kraken
- Data Analysis; Pre & Post processing

- HP SL250 – 264 node 615 TFLOPS (Peak), # 74
- 1 node : 2 Sandy Bridge,16 cores + 3 M2090 GPU
- 32 GB per node + 6 GB / GPU
- 4224 cores + 792 GPU
- 1 PB parallel file space; lustre
- Interconnect : infinite band 4 x QDR
- Multi-GPU processing





WORLD RECORD!
"Beacon" at NICS

Intel® Xeon® + Intel Xeon Phi™
Cluster
First to Deliver
2.499 GigaFLOPS / Watt
71.4% efficiency
#1 on current Green500

Beacon - #1 in top green computer (# 253)

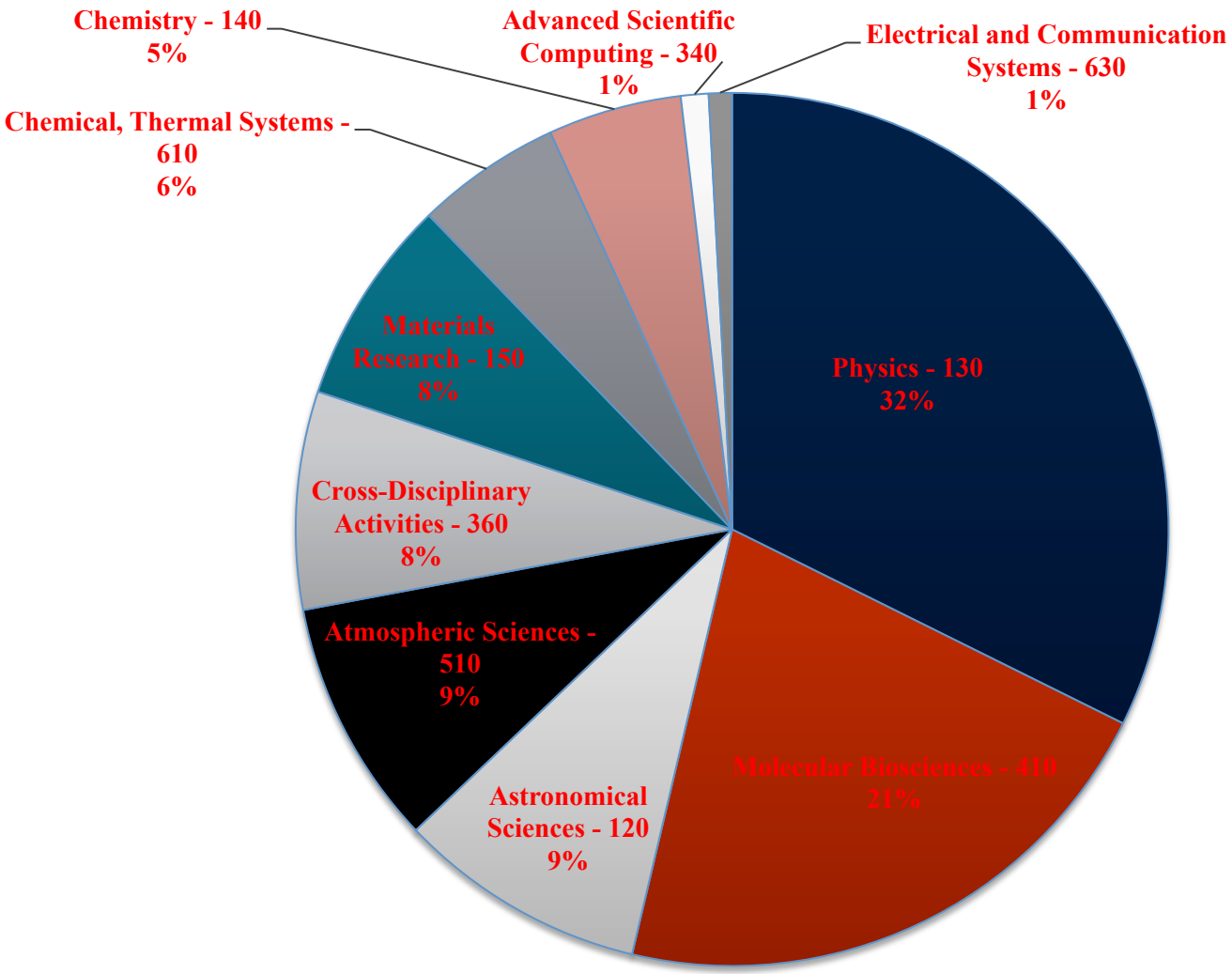
- Cluster by Appro – Beacon, 157.3 TFLOPS (Peak)
- 4 Service, 6 I/O, 48 compute nodes
- Intel Xeon E5-2670 2x 8-core, 2.6 GHz, 256 GB RAM per node
- 4x 5110 Intel Xeon Phi co-processor, 1.054 GHz, 60 core . 8 GB / processor

Power:
• 2x Intel® Xeon® E5-2670 @ 115W TDP
• 4x Intel® Xeon PhiTM Coprocessor 5110Ps @ 225W TDP • 256GBRAM:16DIMMs@6Wea.=96W
• *Chassis: 120W (cooling + other)*
• Approximate total power: 1346 W (theoretical)
• Peak performance:
• 4x Intel® Xeon PhiTM coprocessor 5110P: 4040 GFLOPS
• 2x Intel® Xeon® E5-2670: 330 GFLOPS
• Approximate total performance: 4370 GFLOPS
• Assume 70% ef[iciency for HPL: • 4370 * 0.7 = 3059 GFLOPS
• Approximate MFLOPS/W:
• 3059 / 1458.7 = 2097 MFLOPS/W before optimizations

So much computing power! What's being done with these supercomputers?

The Science of it all!

Kraken Actual Usage by Discipline



Kraken Stats by Discipline (in Hours)

Field of Science	Allocation Hours	Charge	Usage
Physics - 130	92,675,730	48,535,391	49,489,972
Atmospheric Sciences - 510	49,252,000	17,204,796	18,294,744
Molecular Biosciences - 410	38,874,680	25,146,406	28,125,072
Astronomical Sciences - 120	37,258,960	18,928,404	23,590,527
Chemical, Thermal Systems - 610	21,248,300	14,065,003	25,962,112
Earth Sciences – 520	17,293,500	9,054,470	10,147,609
Staff Accounts – 940	13,510,000	5,847,086	6,176,245
Chemistry – 140	11,511,000	9,969,956	10,301,938
Advanced Scientific Computing - 340	10,918,440	2,977,647	3,168,694
Materials Research – 150	9,820,000	3,352,978	4,184,680
Cross-Disciplinary Activities - 360	4,252,500	237,359	237,359
Design and Manufacturing Systems - 640	2,250,000	2,759,092	3,122,916
Computer and Computation Research - 310	1,155,230	474,689	474,689
Electrical and Communication Systems - 630	886,200	810,759	1,000,441
Environmental Biology - 430	600,000	0	0
Mechanical and Structural Systems - 620	475,000	304,925	304,925
Training - 950	240,000	0	0
Microelectronic Information Processing Systems - 330	200,000	25,079	25,079
Information, Robotics, and Intelligent Systems - 320	50,000	0	0
Integrative Biology and Neuroscience - 470	50,000	0	0
Mathematical Sciences - 110	50,000	0	0
Networking and Communications Research - 350	40,000	0	0
Total	312,611,540	159,694,040	184,607,002

Billed usage differs from actual usage in that it contains credits for failed jobs, discounts for running large core count jobs, etc.

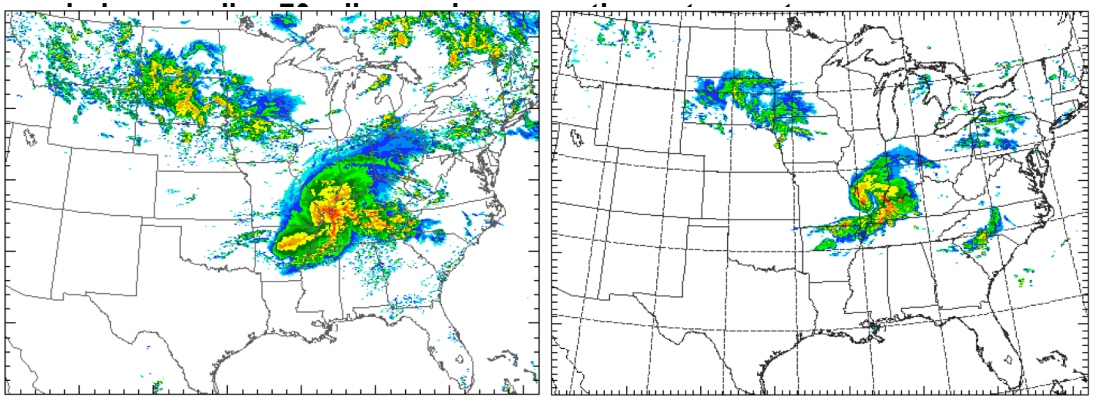
Applications at Scale on Kraken

Science Area	Code	Contact	Max Cores	Par Eff at scale
Astrophysics	ENZO	Harkness	98,304	
Astrophysics	Gadget	di Matteo	99,000	
Astrophysics	VH1	Messer	65,536	
Astrophysics	RadHyd3D	Lentz	32,768	
Atmos Sci.	H3D	Daughton	98,304	
Atmos Sci.	VPIC	Daughton	98,304	
Biomedical	NektarG	Grinberg	65,536	92%
CFD	PHASTA	Jansen	98,304	94%
CFD	[MPM]ICE	Luitjens	98,304	50%
Chemistry	NWChem	Harrison	65,536	
Climate	CCSM	Dennis	32,148	
Combustion	DNS	Yeung	65,536	
Earth Sciences	AWM-Olsen	Maechling	96,000	75%
Materials	OMEN3D	Luisier	96,768	98%
Molecular Biosciences	NAMD Gromacs	Smith	12,288	
Physics/QCD	HMC	Joo	98,304	23%
Weather	WRF	Xue	14,400	

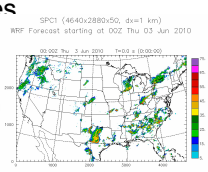
Center for Analysis and Prediction of Storms (CAPS), U. Oklahoma

Predicting Continental US Scale Weather at up to 1 km Grid Spacing in Realtime with Full-Scale Radar Data Assimilation

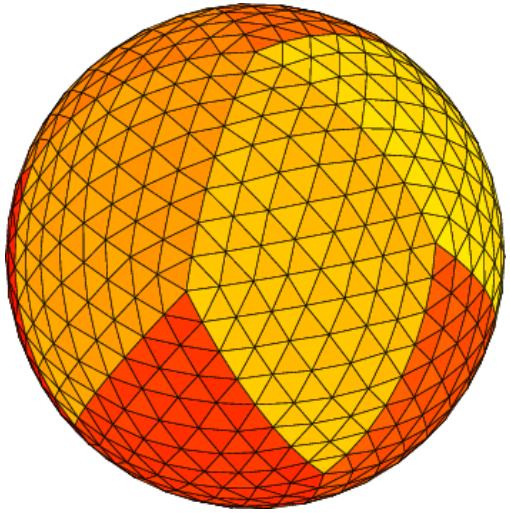
- **26 x 4km and one 1km forecasts were performed on the XT4 Athena using all 18000 processor cores in dedicated mode 5 nights each week during April – June 2010**
  - **the 30-hour long forecasts typically take 5 hours to complete during the overnight hours.**
  - **The same machine was also used to run the two ARPS 4 km forecasts as part of the ensemble.**
  - **Running realtime forecasts at such a high resolution for a continental-scale domain was a first in this line of research, while direct assimilation of data from over 120 operational weather radars at such a high resolution had never been done before.**
- **The figure shows an example forecast at 1 km grid spacing for a case where widespread wind, hail and flooding damages occurred over southern Missouri and southern Illinois**
  - **Nearly 30 tornadoes were reported in the same region. The severe weather was caused by an intense mesoscale convective vortex that contained a large bow echo.**
  - **Figure shows a comparison between the 18-hour forecast of the mesoscale vortex on the 1 km grid, as compared to the radar observation. The model reproduced the mesoscale vortex very well and predicted a large area of intense surface**



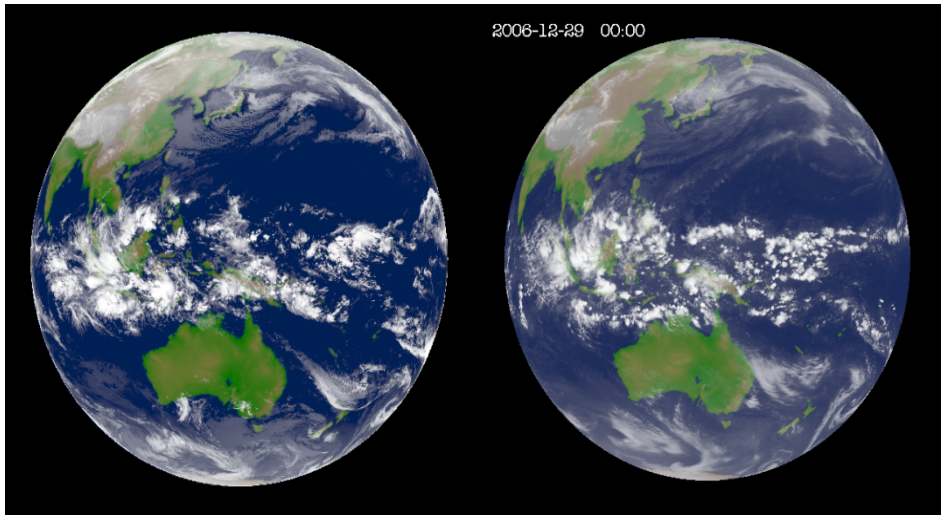
Radar reflectivity field produced by the CAPS 1-km forecast on NICS/ UTK Cray XT5, using 9,600 cores (left), as compared to radar observation of the same quantity (right). The forecast length is 18 hours and the fields are valid1t 18UTC, 8 May 2009.



COLA Dedicated Athena Project



MTSAT-1R, IR NICAM 3.5km, OLR



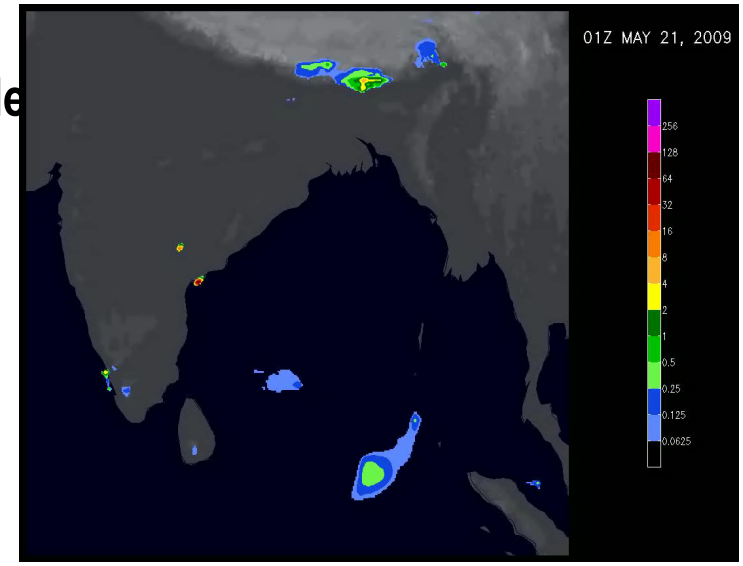
- **NICAM computational model COLA team in Maryland**

  Nonhydrostatic ICosahedral Atmospheric Mode for Global Cloud-Resolving Simulations

- **IFS computational model – ECMWF team in Europe**

6 months dedicated time on 18048 cores of Athena, Cray XT4 == ~70 M hrs computing time

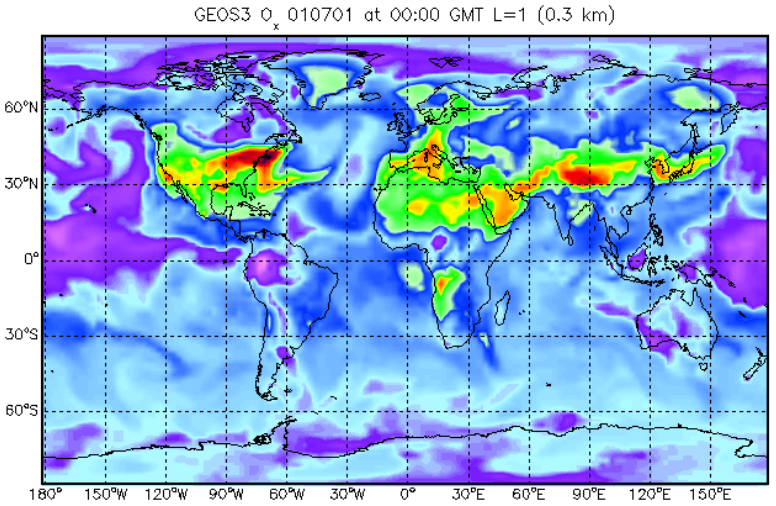
Miura et a.(2007,Science)



Multi-dimensional Climate and Air Quality Study,

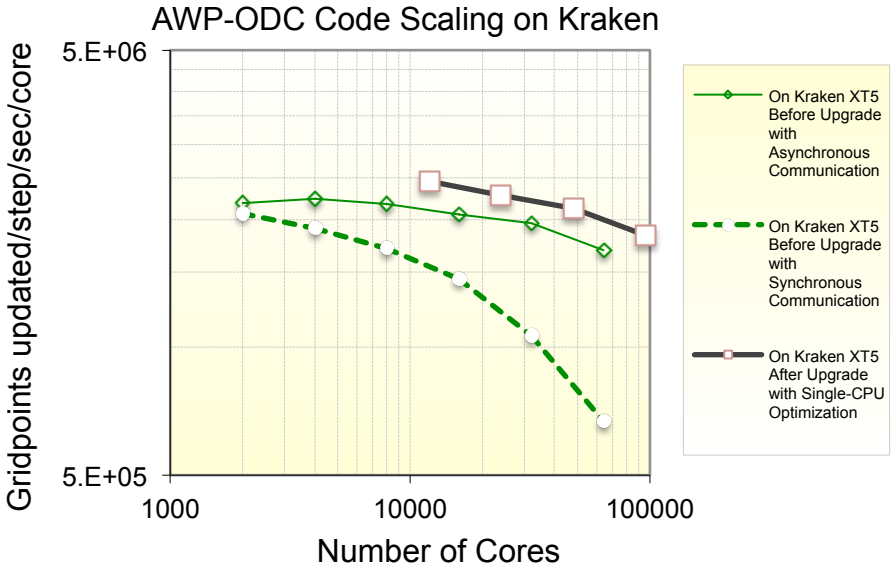
Joshua Fu

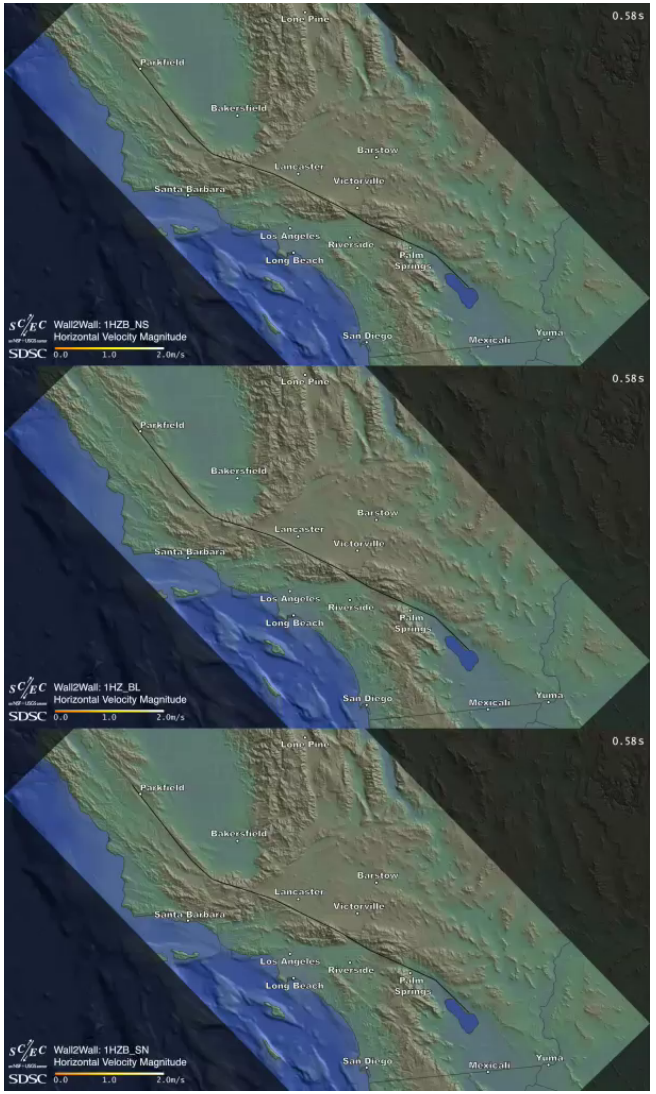
- Predict U.S. air quality in 2050 for future air quality planning
- Evaluate the effect of U.S. climate changes in 2050
- Pollutants source and receptor study for United Nations
- Downscaling applications coupling climate and air quality model, CCSM to WRF, CMAQ
- Challenges : petacale computing, 2 TB data per yearly simulation per scenario, workflow managements, and data postprocessing
- NICS helping validate models



Simulating the Big One on Kraken
T. Jordan, Southern California Earthquake Center

- Biggest Earthquake Simulation on San Andreas Fault, the Big One
- Simulated in a 32 billion grid point subset of the SCEC Community Velocity Model (CVM) V4 with a minimum shear-wave velocity of 500 m/s up to a maximum frequency of 1 Hz.
- 96,000 processor cores used for production runs on Kraken, 2.6 hrs WCT, 53 sustained TeraFlop/s

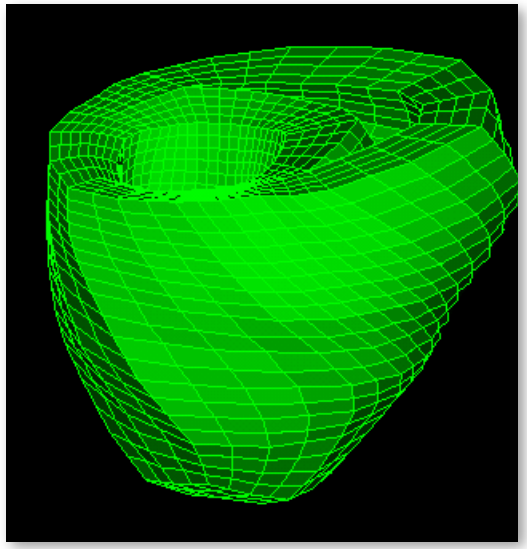


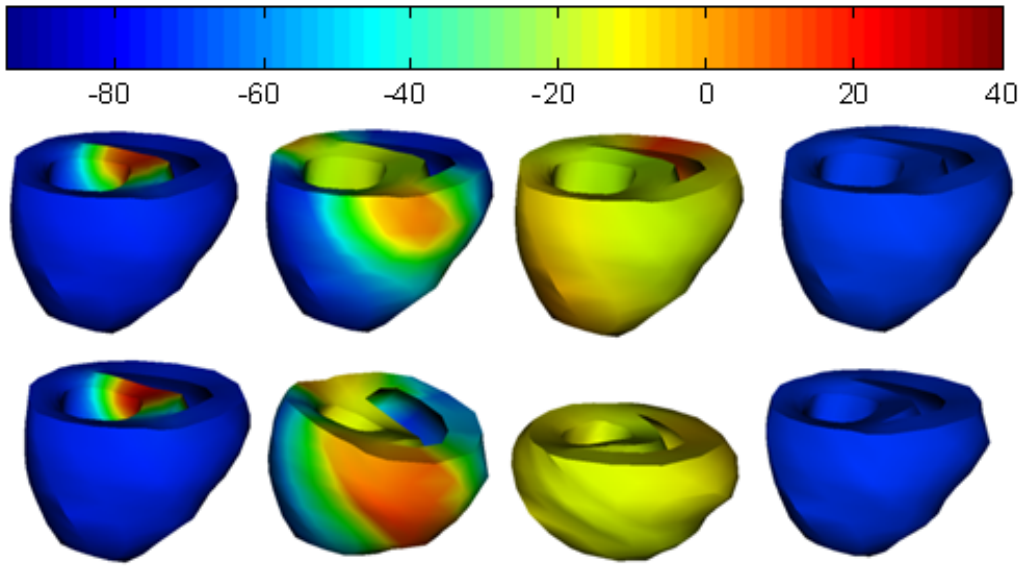


Mathematical Modeling of Heart Rhythm Disorders,

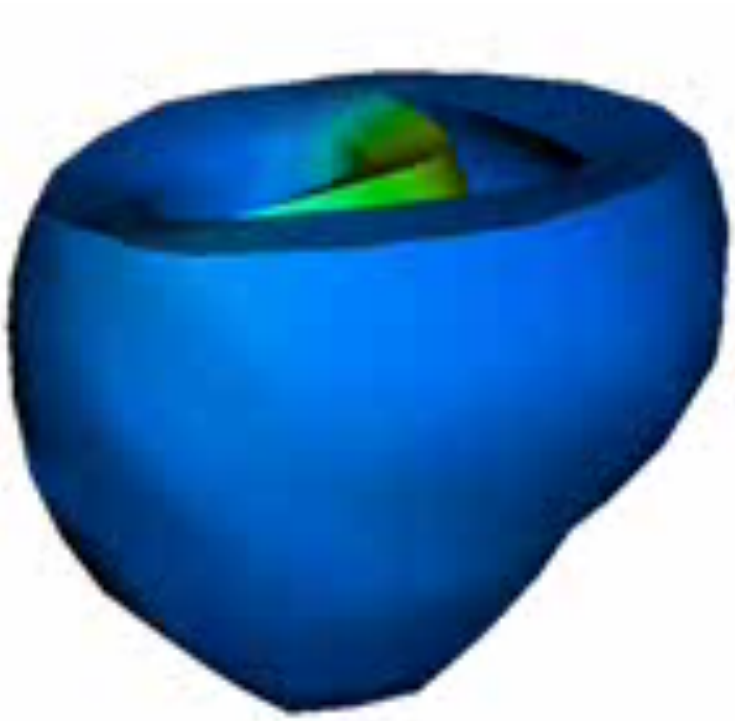
Xiaopeng Zhao, MABE, UTK

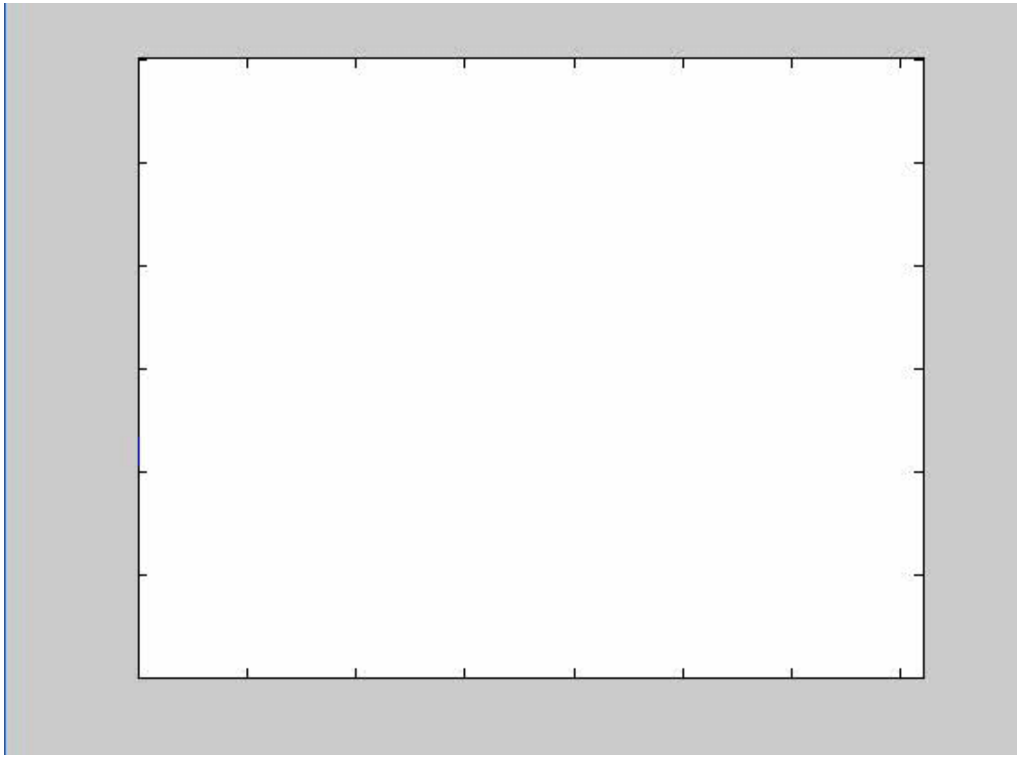
- A complete understanding of heart rhythm disorders requires a system-levels investigation on the interaction between electrical, chemical, and mechanical activities on biological scales ranging from ion channels to single cells to multi-cellular tissue and organ.
- The goal is to develop a viable computing framework to model the cardiac electrical wave propagation of the human heart. The work will integrate models from multiple physics fields including electrophysiology, electro-mechanics, and mechanical deformations.





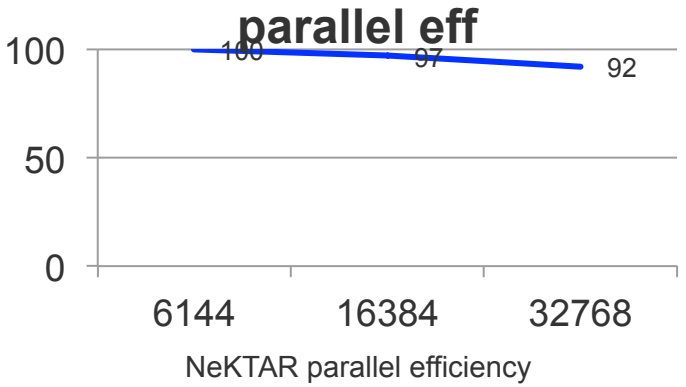
3D Ventricle with Realistic Geometry





Parallel Simulations of Blood Flow

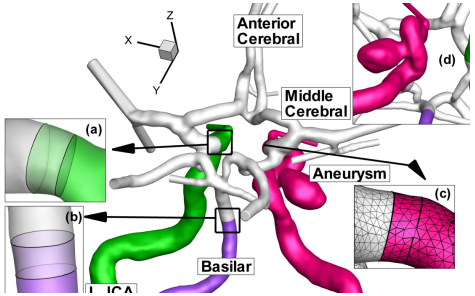
George Karniadakis, Brown U.



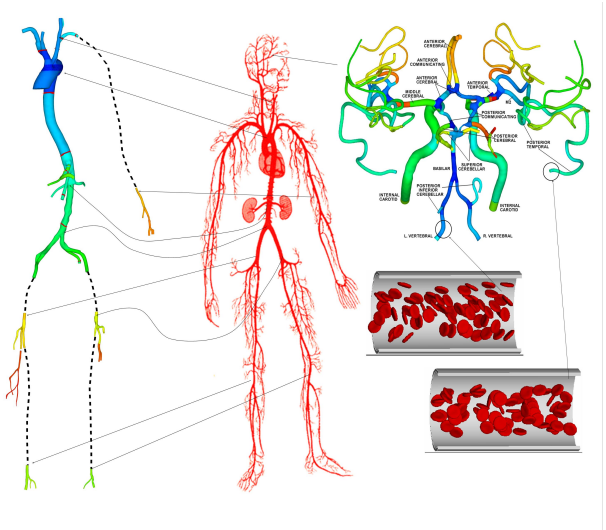
NeKTAR parallel efficiency

Numerical simulations to study fundamental aspects of blood flow.

- **Simulations span multiple scales starting from the individual platelets and red blood cells (*μm* level) to complex arterial networks (*cm* level).**
- **Objectives of the study include modeling platelets aggregation, malaria-infected red blood cells, and intracranial flow in large arterial networks of patients with hydrocephalus and aneurysms (below).**
- **Connect NektarG with LAMMPS to perform coupled continuum – atomistic flow simulations**
  - **NektarG is used for the large arteries (continuum based simulations).**
  - **For red blood cells and platelets, a modified LAMMPS is used and the dissipative particle dynamics (DPD) method we have developed -- a sort of coarse-grained MD**
- **Largest simulation to date was solution of 4 billion degrees of freedom (per time step) problem on 65,536 cores on the Cray XT5**



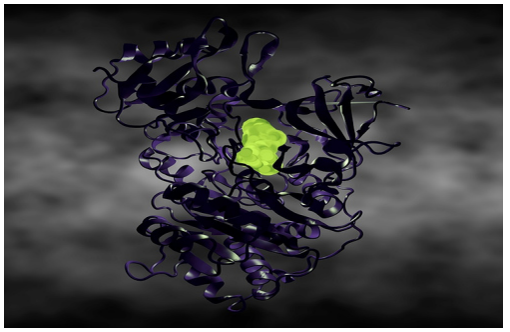
Aneurysm is a pathological disease on the blood vessel wall bulging outward.



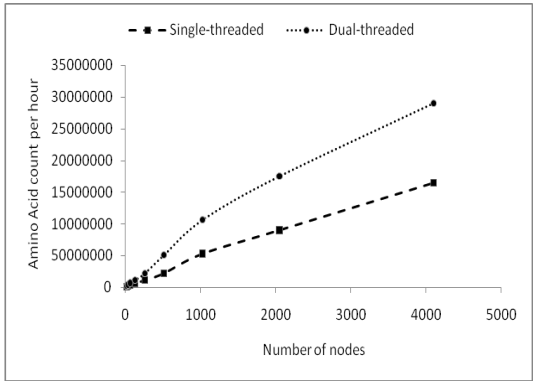
Model of major vessels of the human arterial tree reconstructed from CT images. Colors represent different parts of the model. Left: aorta and adjacent arteries. Right: (top) cranial arterial network and (bottom) modeling red blood cells dynamics.

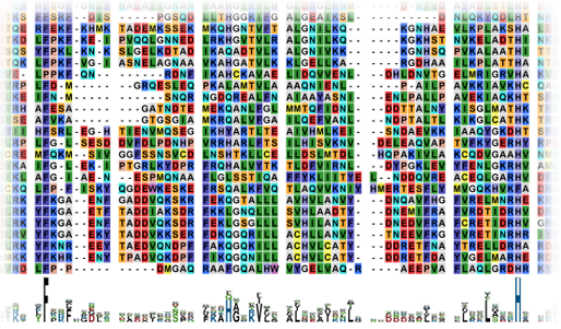
Highly Scalable Parallel HMMER and BLAST

C. Halloy, B. Rekepalli, and I. Jouline*, UTK



- **HMMER – Protein Domain Identification tool**
  - **MPI-HMMER limited performance**
  - **HMMER compares sequences to a database of hidden Markov models to identify known domains within the sequences**
- **New HSP-HMMER code - Excellent performance**
  - **Currently ~10000x faster than MPI-HMMER for 1K processes**
  - **Scales up to 98,000 cores very well**
- **HSP-HMMER reduces time to identify the Pfam functional domains in 6.5 millions proteins of the "nr" (non redundant) database from 2 months on clusters down to less than 10 minutes! using 98000 processing cores..**
  - **B Rekepalli, C Halloy, IB Jouline. "HSP-HMMER: a Tool for Protein Domain Identification on a Large Scale," ACM SAC 2009, 766-770.**
- ***This is critical, considering that the protein database continues doubling in size every 6 months!***
- ***HSParallel BLAST now scales to 50,000 cores on Kraken. Tests are still under way.***



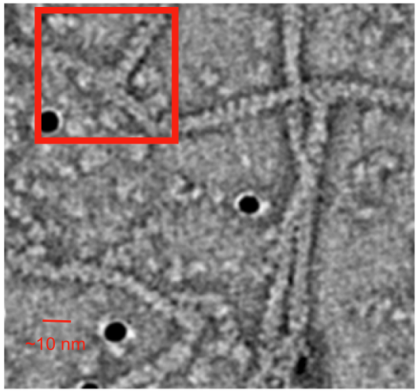


A part of an alignment for the Globin family from Pfam

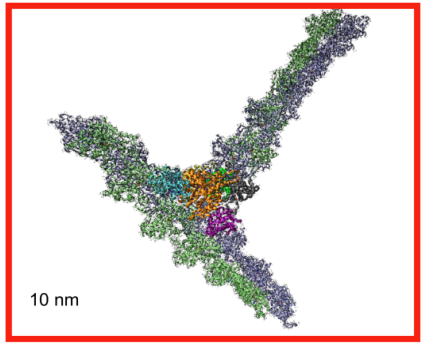
Multiscale Simulation of Biological Assemblies
Greg Voth, U. Utah

The actin-Arp2/3 branch junction

- Confers shape and structure to most types of cells
- Among the largest biomolecular MD simulations performed to date (NAMD, 4,000 cores).

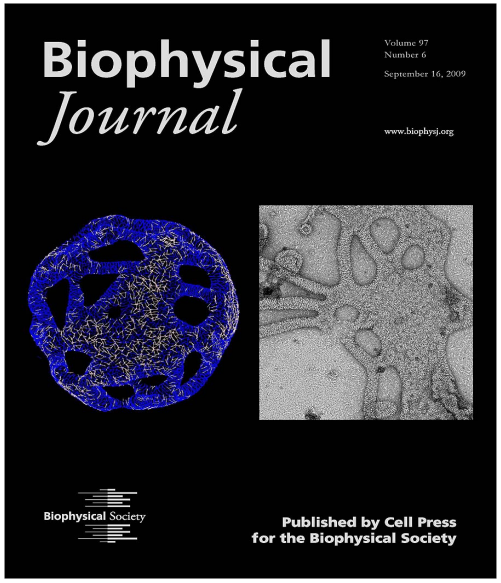


Cryo-EM image of actin cytoskeleton[1]



Atomic model of the actin-Arp2/3 branch junction

[1] I. Rouiller et al. 2008. *J. Cell Biol.* 180:887-895.

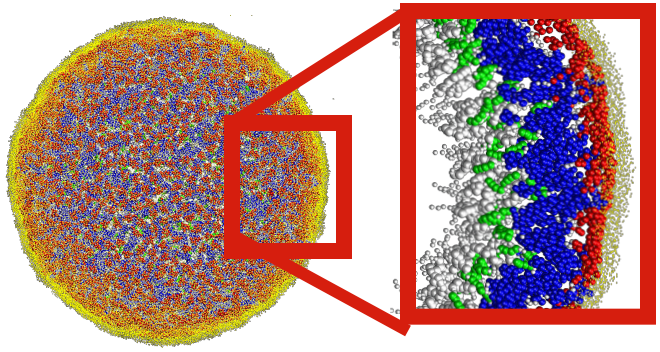


Blue figure is simulation. Grey figure is experimental collaborator results. Blue figure matches grey figure -- theory meets experiment in biology.

Multiscale simulations of membrane remodeling. The *first* direct comparison of mesoscale simulation with electron microscopy imaging. (0.75 million CG sites, equivalent to 10^{11} atoms, using TANTALUS over 2,000 cores).

The *first* CG-MD simulations of the entire immature HIV-1 virion

- 0.75 million CG sites, equivalent to 10^8 atoms, using TANTALUS over 2,000 cores



Science enabled by NICS

- Petascale supercomputing resources allow for long timescale simulation.
- Thus able to provide meaningful feedback to experimental research in structural biology
- "Kraken is fundamentally changing how we think about molecular simulation: things we used to dream about doing are now possible"

Atomistic Simulations of Future Nanoelectronics Transistors
Mathieu Luisier, Purdue

Objectives:

- accelerate nanoscale transistor innovation with petascale simulation
- help experimentalists design low power nanodevices

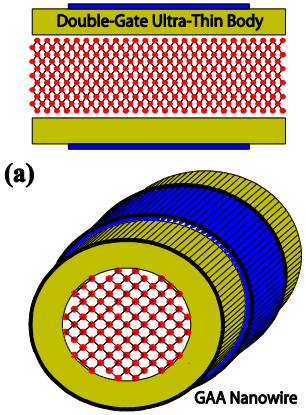
Approach: OMEN - a massively parallel, atomistic, full-band quantum transport simulator for 1-D, 2-D, and 3-D nanodevices based on the Non-equilibrium Green's Function Formalism

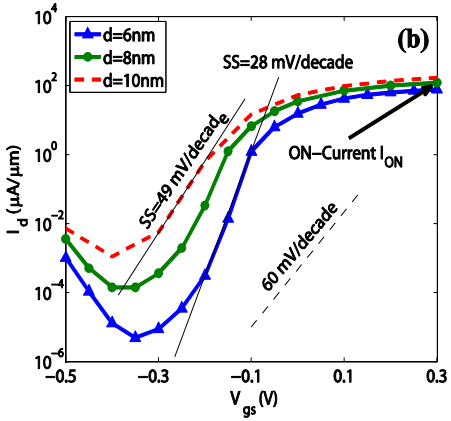
Results:

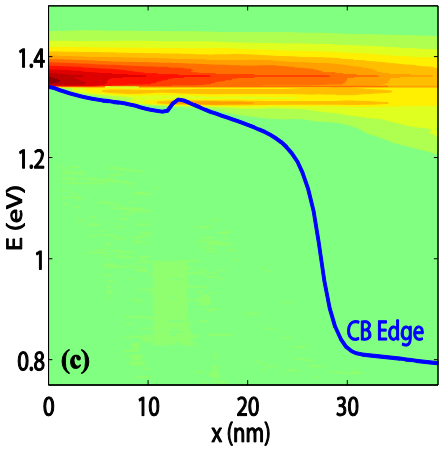
- Reproduced InAs HEMT experimental data in an hour with 96,768 cores rather than weeks on cluster; 368 Tflops/s
  - Will be part of a paper on electron-phonon scattering in nanowire TFETs and opens a door towards larger device structures
- Si nanowire with 4nm diameter simulated on 3,000 nodes with 8GB of memory per core

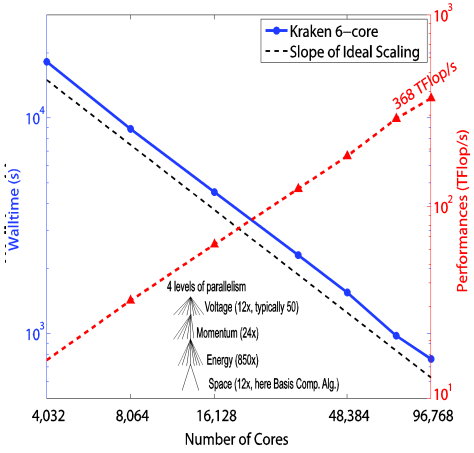
Impacts:

- first demonstration of electron-phonon scattering in a 3-D, atomistic, and full-band basis on Kraken.
  - proved that electron-phonon scattering plays a more important role when the diameter of the nanowire increases, as expected
- Full machine runs are the key to simulating large device structures and to reducing the computational time down to the minute scale instead of months on a single core.



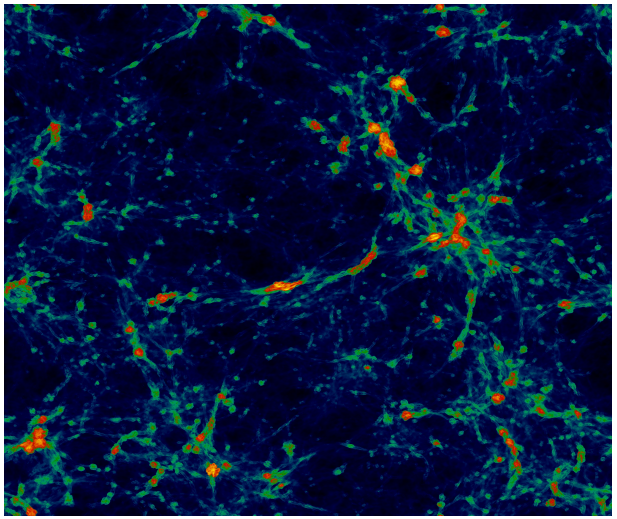






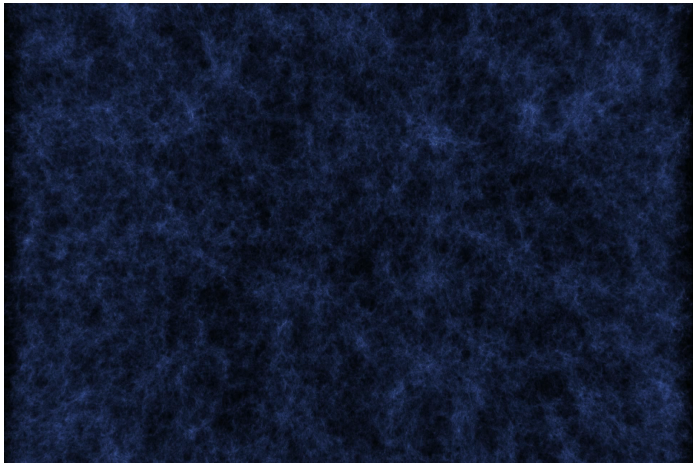
Overview of OMEN capabilities. (a) Examples of nanoelectronics devices that OMEN can handle (double-gate ultra-thin-body and gate-all-around nanowire FET). (b) Transfer characteristics Id-Vgs of different types of TFETs. (c) Spectral current of a GAA NW FET with electron-phonon scattering. (d) OMEN scaling performances and sustained performance on Kraken up to 96,768 cores.

Simulating the formation of early galaxies
Robert Harkness, UC San Diego



ENZO - Hybrid MPI/OpenMP code

- Current model is $6{,}400^3$ = 268 billion cells and dark matter particles!
  - **Definitely the World's Largest!**
  - **Star formation and feedback (energy & momentum)**
  - Running on 93,750 cores, 125 TB of Kraken
    - "A Blue Waters scale" simulation
  - Largest hydrodynamic cosmology simulation ever done
  - First to simulate large enough volume of the universe to resolve galaxies across a sufficiently wide range of masses and luminosities
  - Last checkpoint at redshift 15.5; need to get to 6
    - Requires about 10 more 24-hour 94,000 core runs
- *"The most productive platform in NSF portfolio for ENZO simulations, bar none," Harkness*



HELP!
How do I get started?

accounts
programs
compiling
jobs scripts
running jobs
modules
software
file storage
etc.

www.nics.utk.edu

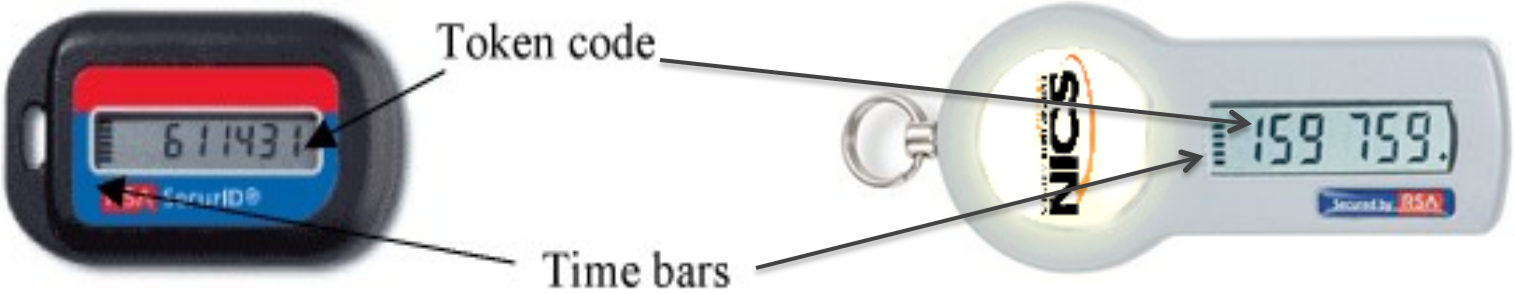
Getting started: Accounts + Connecting

- Request an account

  Kraken: http://www.nics.utk.edu/user-support/accounts
  [Jaguar: http://www.nccs.gov/user-support/access/]

- Use Secure Shell (with passcode or password)
  - ssh username@kraken.nics.utk.edu

  One time password (OTP)
  - RSA SecurID (aka “fob” or “dongle”, or “thingamajig”)
  - PASSCODE = PIN + Token Code
  - SSH client need keyboard-interactive



What do you get with your account

- A Unix account (userid and Project account)
- A One Time Password generator (token) to login via ssh
- Access through Globus Grid tools (gssissh, GridFTP)
- A NFS home area (default 2GB quota)
- Lustre scratch space (<2.4PT)
- HPSS mass storage archival via `hsi/htar` (OTP only)
- >100 applications ready to run on Kraken
- Up to 112,896 cores
- User Assistance
- Bash as default Unix shell

HowTo compile

- Available C, C++ and Fortran compilers: PGI, GNU, (Pathscale), Intel, and Cray. Select compiler with ‘module’ command. Default is PGI.
- Use the compiler wrappers `cc`, `CC` and `ftn`, to compile programs for the compute nodes.
- The compiler wrappers know where most of the correct Cray provided libraries and include files are, if the corresponding module is loaded.
- You do not need to know where the MPI libraries are.
- The wrappers automatically add the correct tuning parameters for the Istanbul Processor.

Simple Hello World C Program

```c
/* Simple serial Hello World C Example : hello.c */
#include <stdio.h>

int main (int argc, char *argv[])
{
     int rank, size;
     printf( "Hello World from process %d of %d\n", rank, size );
return 0;
}
```

```c
/* Parallel Hello world C Example : hello.c */
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[])
{
     int rank, size;
     MPI_Init (&argc, &argv);    /* starts MPI */
     MPI_Comm_rank (MPI_COMM_WORLD, &rank);    /* get current process id */
     MPI_Comm_size (MPI_COMM_WORLD, &size);    /* get number of processes */
     printf( "Hello World from process %d of %d\n", rank, size );
     MPI_Finalize();
return 0;
}
```

To compile : > cc –o hexe ./hello.c
To run serial code : > ./hexe

To compile : > cc –o hexe ./hello.c
To run in parallel : > aprun –n 4 ./hexe

Compile and run on a CRAY XT

---compile your code---

```
kraken> cc hello.c -o hello
kraken> ftn hello.f -o hello
```

---edit a job script---

```
## myjob.pbs   helloworld job script
#!/bin/bash
#PBS -A UT-TNEDU002
#PBS -N test
#PBS -j oe
#PBS -l walltime=5:00, size=24
cd $PBS_O_WORKDIR
date
aprun -n 4 –N2 –S1  ./hello
```

---submit your job---

```
kraken>  qsub myjob.pbs
84628.nid00016
```

---check status of your job---

```
kraken>  showq --noblock | grep halloy
```

---view all cabinets and jobs---

```
kraken>  xtshowcabs
```

output:

```
Sun Aug 07 20:12:06 EDT 2011
Hello World from process 2 of 4
Hello World from process 0 of 4
Hello World from process 3 of 4
Hello World from process 1 of 4
```

OPENMP Hello World – Fortran Code

```
PROGRAM HELLO
INTEGER   NTHREADS, TID, OMP_GET_NUM_THREADS,
INTEGER   OMP_GET_THREAD_NUM
C     Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL PRIVATE(NTHREADS, TID)
C     Obtain thread number
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID
C     Only master thread does this
      IF (TID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF
C     All threads join master thread and disband
!$OMP END PARALLEL
      END
```

OPENMP Hello World – C code

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  int nthreads, tid;
  /* Fork a team of threads giving them their own copies of variables */
  #pragma omp parallel private(nthreads, tid) {
     /* Obtain thread number */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
       nthreads = omp_get_num_threads();
       printf("Number of threads = %d\n", nthreads); }
  }
  /* All threads join master thread and disband */ }
```

Compile and run OpenMP on Kraken

```
%cc -mp  -fast omp_hello.c -o hello
%ftn –mp -fast omp_hello.f -o hello
%qsub submit.pbs
```

```
#!/bin/bash
#PBS -A UT-TNEDU002
#PBS -N test
#PBS -j oe
#PBS -l walltime=1:00:00,size=12
cd $PBS_O_WORKDIR
Date
export OMP_NUM_THREADS=4
aprun -n 1 –S1 –d4 ./hello
```

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 2
```

HowTo use modules

- All software/packages are managed via modules
- This allows environment variables, libraries, include paths to be cleanly entered and/or removed from your software environment.
- Conflicts are detected and loads that would cause conflicts are not allowed
- There are a number of basic modules loaded by default

```
  1) modules/3.1.6.5
  2) torque/2.4.1b1
  3) moab/5.2.5.s12399
  4) /opt/cray/xt-asyncpe/default/
modulefiles/xtpe-istanbul
  5) tgusage/3.0-r2
  6) DefApps
  7) cray/MySQL/5.0.64-1.0000.2342.16.1
  8) xtpe-target-cnl
  9) xt-service/2.2.41A
 10) xt-os/2.2.41A
 11) xt-boot/2.2.41A
 12) xt-lustre-ss/2.2.41A_1.6.5
 13) cray/job/1.5.5-0.1_2.0202.18632.46.1
 14) cray/csa/3.0.0-1_2.0202.18623.63.1
 15) cray/account/1.0.0-2.0202.18612.42.3
 16) cray/projdb/1.0.0-1.0202.18638.45.1
 17) Base-opts/2.2.41A
 18) pgi/9.0.3
 19) totalview-support/1.0.5
 20) xt-totalview/8.4.1b
 21) xt-libsci/10.3.9
 22) xt-mpt/3.5.0
 23) xt-pe/2.2.41A
 24) xt-asyncpe/3.3
 25) PrgEnv-pgi/2.2.41A
 26) /sw/altd/modulefiles/altd

module list
```

HowTo use modules

The complete list of all available modules can be viewed with the command `module avail`. The 3rd party list of software can also be viewed from our website at:
http://www.nics.tennessee.edu/user-support/software/Kraken

Loading commands

`module [load|unload] <my_module>`
Loads/unloads module

`module swap <module1><module2>`
Replaces <module1> with <module2>

```
> module swap PrgEnv-pgi PrgEnv-gnu
```

Informational commands

`module help [my_module]`
Lists available commands and usage

`module show <my_module>`
Displays the actions upon loading the module <my_module>

`module list`
Displays all currently loaded modules

`module avail <name>`
Lists all modules (beginning with name)

HowTo debug and profile

The following tools are available on Kraken for debugging, profiling and analysis parallel programs

Debugging	Profiling and Analysis
Totalview, lgdb, atp, gdb, pgdbg	Cray PAT, pgprof, TAU, FPMPI, PAPI, Scalasca

Always check the compatibility of the compiler options you want to use. For example, the following PGI compiler options are not supported:

```
-Mprof=mpi, -Mmpi, and -Mscalapack
```

NICS survival kit

This is a list of Unix commands available on Kraken that all users should be aware of:

`module <command>`	All software packages like compilers, libraries and applications, are handled via modules
`qsub<jobscript>`	All jobs are submitted with the `qsub` command
`aprun -{n|N|S|d|cc}`	Programs get executed on the compute nodes with this command
`showq --noblock [-r]`	Shows the current state of the queue

NICS survival kit

`qstat<jid>`	Shows the status of a job with job id `<jid>`
`glsjob<jid>`	It can be used to query information about a previous job or allprevious jobs with `-u<uid>`
`showstart<jid>`	Shows *approximate* start time for job with job id `<jid>`
`showusage`	Displays the current balance of all the project accounts on Kraken a user has access to
`showbf`	Shows what resources are available for “immediate” use. (bf = back fill)

Other commands include: `checkjob, apstat, glsuser, glsproject`

NICS survival kit

A better/faster way to work with files in Lustre, can be done with the help of the `lfs` instead of the standard Unix commands: `ls`, `find`, `df`. (see man-pages)

```
lfs<command> [options]
```

`setstripe/getstripe`	Used to manipulate the striping of files and directories in Lustre
`find`	A much faster way to find files in Lustre. Example: `lfs find /lustre/scratch/challoy --name *.c`
`df /lustre`	Shows how much space is left in Lustre
`quota`	Shows how much space I am using in Lustre. Example: `lfs quota -u challoy/lustre/scratch | sed -n 3p`

How to get help

Send your questions via email to

help@xsede.org
(was help@teragrid.org)

Or contact us by phone (office hours only)

1.865.241.1504

or
to the TG / Xsede helpdesk
1.866.907.2383 (off hours)

Science Movies

Message Passing Interface

Topics:

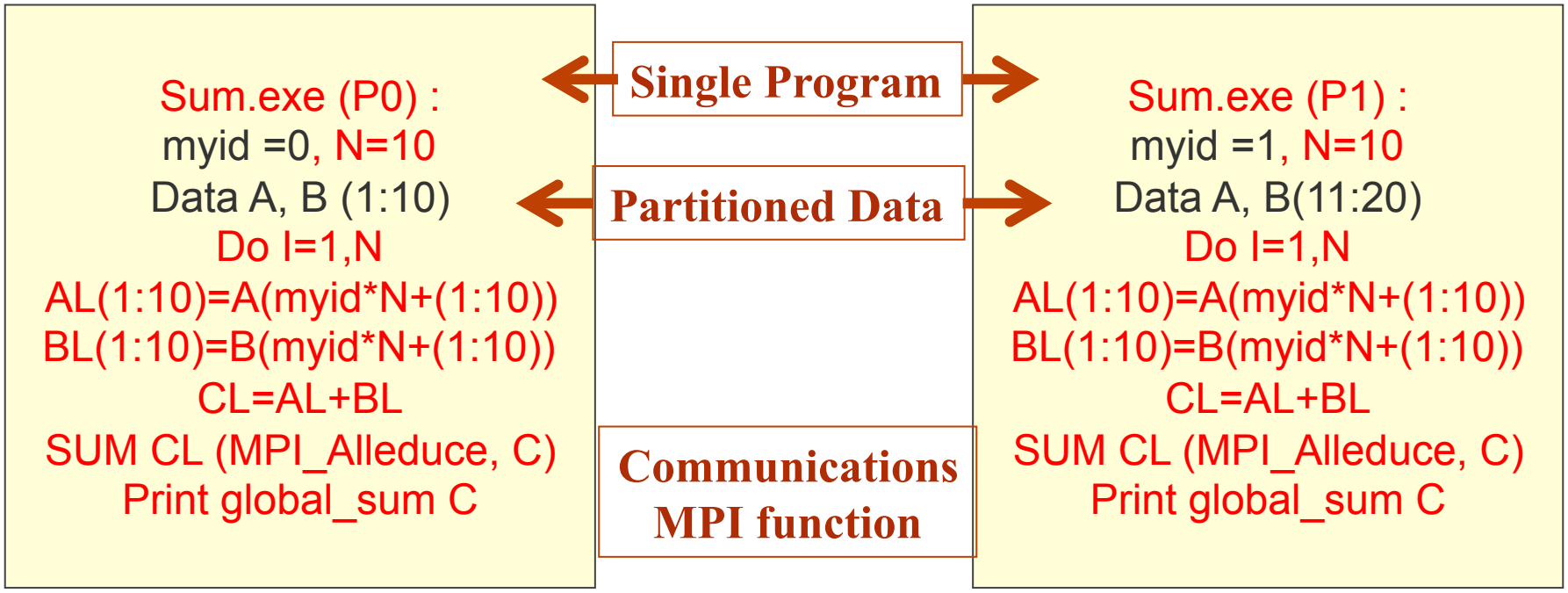
- Message Passing Interface (MPI)
- MPI Point to Point Communication
- Collective Communication
- Derived Datatype
- Parallel Code Development

Message Passing Interface (MPI)

- A first portable message passing communications standard defined by the MPI Forum which consists of hardware vendors, researchers, academics, software developers, and users, representing over forty different organizations
- The syntax of MPI is standardized
- The functional behavior of MPI calls is standardized
- Vendors have tuned MPI calls to suit the underlying hardware and software, hence efficiency of programs are generally preserved.
- IBM MPI : IBM implementation for the SP
- MPICH, LAM .. Popular implementations
- Cray MPT, OpenMPI …..

Message Passing Programming

- Used primarily on distributed memory computing environment
- Since memory are local, any data stored in remote processor's memory must be explicitly requested by programmer.
- Each processor runs the SAME program using partitioned data set
- Written in sequential language (FORTRAN, C, C++) + MPI functions



MPI Message Components

- Envelope:
  - sending processor (processor_id)
  - source location (group_id, tag)
  - receiving processor (processor_id)
  - destination location (group_id, tag)
- Data (letter) :
  - data type (integer, float, complex, char….)
  - data length (buffer size, count, strides)

Typical Message Passing Subroutines

- Environment Identifiers
  - processor_id, group_id, initialization
- Point to Point Communications
  - blocking operations
  - non-blocking operations
- Collective Communications
  - barriers
  - broadcast
  - reduction operations

Essentials of MPI programs

- Header file:
  - C : #include<mpi.h>
  - Fortran : include ‘mpif.h’
- Initializing MPI :
  - C : int MPI_Init(int argc, char**argv)
  - Fortran : call MPI_INIT(IERROR)
- Exiting MPI :
  - C : int MPI_Finalize()
  - Fortran :call MPI_FINALIZE(IERROR)

MPI Function Format

- MPI
  - MPI_Xxxx(parameter, …….)
- C :
  - error = MPI_Xxxxx(parameter, …….)
  - Case is IMPORTANT
- Fortran:
  - call MPI_XXXX(parameter,….., IERROR)
  - Case is NOT important

MPI Process Identifiers

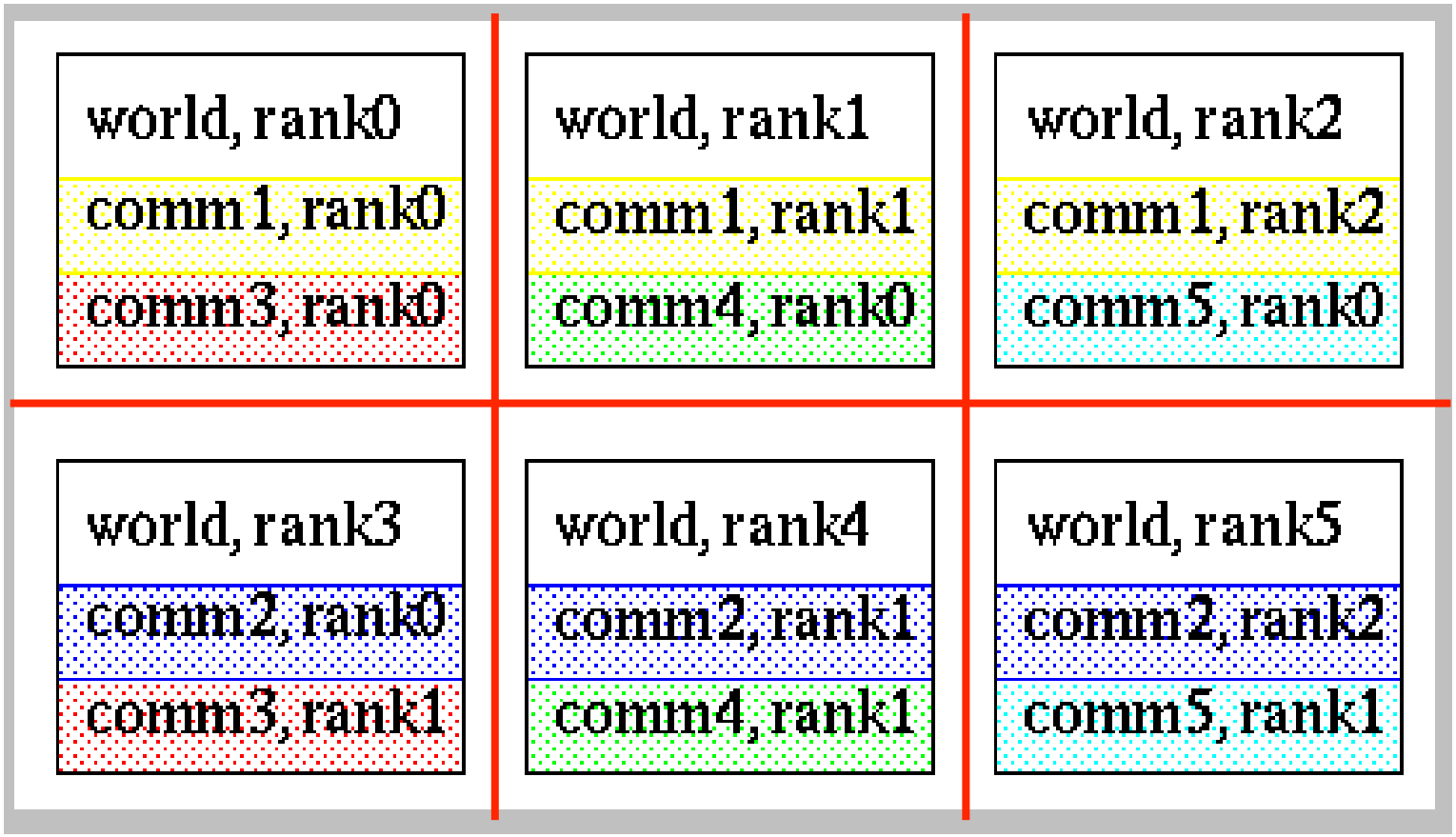
- MPI_COMM_RANK:
  - Gets a process's rank (ID) within a process group
  - C : int MPI_Comm_rank(MPI_Comm comm, int *rank)
  - F : call MPI_COMM_RANK(mpi_comm,rank,ierror)
- MPI_COMM_SIZE:
  - Gets the number of processes within a process group
  - C : int MPI_Comm_size(MPI_Comm comm, int *size)
  - F : call MPI_COMM_SIZE(mpi_comm,size,ierror)
- MPI_Comm : Communicator

MPI Communicator

- A communicator is the communicating space among processes
- All MPI communication calls require a communicator argument and MPI processes can only communicate if they share a communicator.
- Every communicator contains a group of tasks with a *system supplied identifier* (for message context) as an extra match field.
- MPI _Init initializes all tasks with MPI_COMM_WORLD
- So, the base group is the group that contains all processes, which is associated with the MPI_COMM_WORLD communicator.
- Communicators are particularly important for user supplied libraries
- Communicators are used to create independent “message universe”

Communicating Group

ALL : MPI_COMM_WORKD (world)
ROW : comm1, comm2 ; COLUMN : comm3, comm4, comm5
Every process has three communicating groups and a distinct rank associated to it



Sample Program : Hello World

```
#include <stdio.h>
#include “mpi.h”
main(int argc, char** argv) {
  int my_PE_num;
  MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
  printf(“hello from %d. \n”, my_PE_num);
  MPI_Finalize();
}
```

```
  program Hello_World
  include ‘mpif.h’
  integer my_PE_num, ierror
  call MPI_INIT(ierror)
  call MPI_COMM_RANK(MPI_COMM_WORLD,my_PE_num, ierror)
  print *, ‘Hello from’, my_PE_num
  call MPI_FINALIZE(ierror)
  end
```

Compiling and Running MPI (MPICH PC Cluster)

- **Compile :**
  - **mpicc -o fileexe filename.c**
  - **mpif77 -o fileexe filename.f**
- **Running :**
  - **mpirun -np <num_of_processes> fileexe**
- **Options :**
  - **-np : number of processes**
  - **-machinefile : list of machines to be used**
  - **-t : testing option**
  - **use "man mpirun" to see more details**

```
%mpirun –np 3 –machinefile hostfile hello.exe
hello from 0
hello from 2
hello from 1
```

Compiling and Running (Cray XT)

- Compile : cc -o hello hello.c or ftn -o hello hello.f
- On kraken.nics.utk.edu, jaguar.ccs.ornl.gov
- Use PBS Batch Script, e.g. submit.pbs
- run : qsub submit.pbs

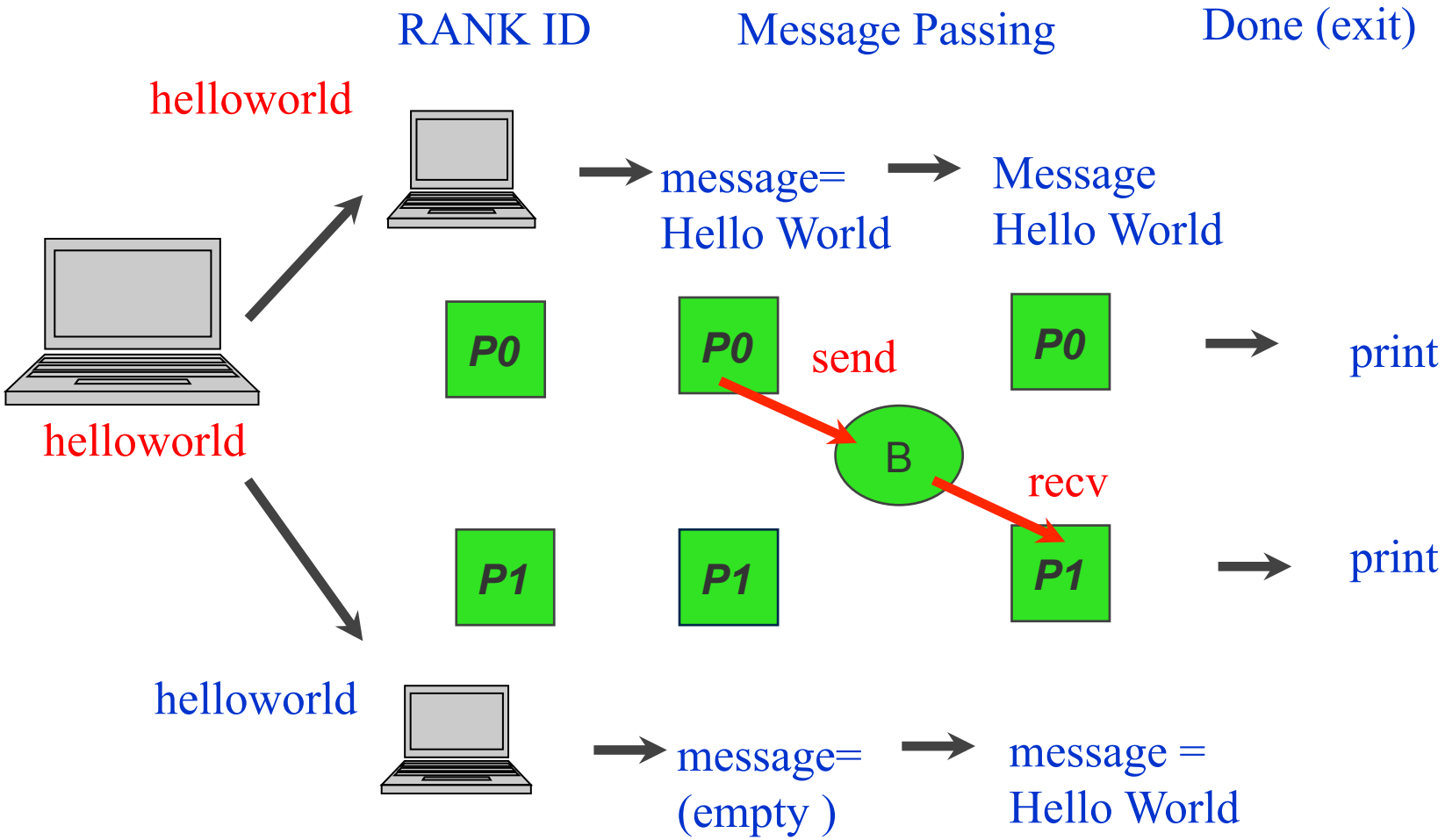
```
#!/bin/bash
#PBS -A your-project-id
#PBS -N test
#PBS -j oe
#PBS -l walltime=1:00:00,size=8
cd $PBS_O_WORKDIR
date
aprun -n 4 ./hello
```

SPMD

- Single Program, Multiple Data Programming Paradigm
- Same program runs on all processors, however, each processor operates on different set of data
- The simplest parallel parallel paradigm
- Generally contains :

```
if ( my_processor_id  .eq. designated_id ) then
        -------
end if
```

Parallel Processing



Hello World Again

```
program Hello_World
include ‘mpif.h’
integer me, ierror, ntag, status(MPI_STATUS_SIZE)
character(12) message
call MPI_INIT(ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierror)
ntag = 100
if ( me .eq . 0) then
    message = ‘Hello, World’
    call MPI_Send(message, 12, MPI_CHARACTER, 1, ntag,
     MPI_COMM_WORLD, ierror)
else
    call MPI_Recv(message, 12, MPI_CHARACTER, 0, ntag,
     MPI_COMM_WORLD, status, ierror)
    print *, ‘node’,me, ‘:’, message
endif
call MPI_FINALIZE(ierror)
end
```

Example: Passing a Message – Hello World Again!

```
program Hello_World
include 'mpif.h'
integer me, ierror, ntag, status(MPI_STATUS_SIZE)
character(12) message
call MPI_INIT(ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierror)
ntag = 100
if ( me .eq . 0) then
     message = 'Hello, World'
     call MPI_Send(message, 12, MPI_CHARACTER, 1, ntag,
   MPI_COMM_WORLD, ierror)
else if ( me .eq . 1) then
     call MPI_Recv(message, 12, MPI_CHARACTER, 0, ntag,
   MPI_COMM_WORLD, status, ierror)
     print *, 'Node',me, ':', message
endif
call MPI_FINALIZE(ierror)
end
```

Example: Passing a Message – Hello World Again!

```
#include "mpi.h"
#include <iostream>
#include <string>
int main(int argc, char ** argv)
  {
        int my_PE_num, ntag = 100;
        char message[13] = "Hello, world";
        MPI::Status status;
        MPI::Init(argc, argv);
        my_PE_num = MPI::COMM_WORLD.Get_rank();

        if ( my_PE_num == 0 )
          MPI::COMM_WORLD.Send(message,12,MPI::CHAR,1,ntag);
        else if ( my_PE_num == 1 ) {
          MPI::COMM_WORLD.Recv(message,12,MPI::CHAR,0,ntag, status);
          cout << "Node " << my_PE_num <<" : " << message << endl; }
        MPI::Finalize();
  }
```

MPI Point to Point Communications

- Communication between two processes
- Source process sends message to destination process
- Communication takes place within a communicator
- Blocking v.s non-blocking calls
- MPI defines four communication modes for blocking and non-blocking send : synchronous, buffered, ready, and standard
- The receive call does not specify communication mode-- it is simply blocking and non-blocking.
- Two messages sent from one process to another will arrive in that relative order.

Sending a Message

- C :: Standard send

  int MPI_Send(&buf, count, datatype, dest, tag, comm)
  - &buf : pointer of object to be sent
  - count : the number of items to be sent, e.g. 10
  - datatype : the type of object to be sent, e.g. MPI_INT
  - dest : destination of message (rank of receiver), e.g. 6
  - tag : message tag, e.g. 78
  - comm : communicator, e.g.(MPI_COMM_WORLD)
- Fortran ::

  call MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)

Receiving a Message

- C :: Blocking receive

  int MPI_Recv(&buf, count, datatype, source, tag, comm, &status)

  – source : the node to receive from, e.g. 0

  – &status : a structure which contains three fields, the source, tag, and error code of the incoming message.

- Fortran ::

  call MPI_RECV(buf, count, datatype, source, tag, comm, status(MPI_STATUS_SIZE), ierror)

  – status : an array of integers of size MPI_STATUS_SIZE

In MPI

- Order:
  - MPI guarantees that messages from same process will not overtake each other.
    - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
    - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
    - Order rules do not apply if there are multiple threads participating in the communication operations.

- Fairness:
  - A parallel algorithm is *fair* if no process is effectively ignored
  - MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.

For a Communication to Succeed…

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
  - may use wildcard : `MPI_ANY_SOURCE`
- The communicator must be the same
- Tags must match
  - may use wildcard : `MPI_ANY_TAG`
- Message types must match
- Receiver's buffer must be large enough

MPI Basic Datatypes - C

MPI Datatypes	C Datatypes	MPI Datatypes	C Datatypes
MPI_CHAR	signed char	MPI_SHORT	signed short int
MPI_INT	signed int	MPI_UNSIGNED_CHAR	unsigned char
MPI_LONG	signed long int	MPI_UNSIGNED_SHORT	unsigned short int
MPI_FLOAT	float	MPI_UNSIGNED_LONG	unsigned long int
MPI_LONG_ DOUBLE	long double	MPI_UNSIGNED	unsigned int
MPI_BYTE	--------	MPI_PACKED	---------

MPI Basic Datatypes - Fortran

MPI Datatypes	Fortran Datatypes
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	----------
MPI_PACKED	----------

Sample Program : Send and Receive

```
#include “mpi.h”
main(int argc, char ** argv) {
      int my_PE_num, numtorecv,  numtosend=42;
     MPI_Status status;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

     if ( my_PE_num == 0) {
       MPI_Recv(&numtorecv, 1, MPI_INT, MPI_ANY_SOURCE,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &status);
       printf(“Number received is : %d\n”, numtorecv); }
     else MPI_Send ( &numtosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD) ;
     MPI_Finalize() ;
}
```

MPI Send

- **MPI has 8 different types of Send**
- **The nonblocking send has an extra argument of request handle**

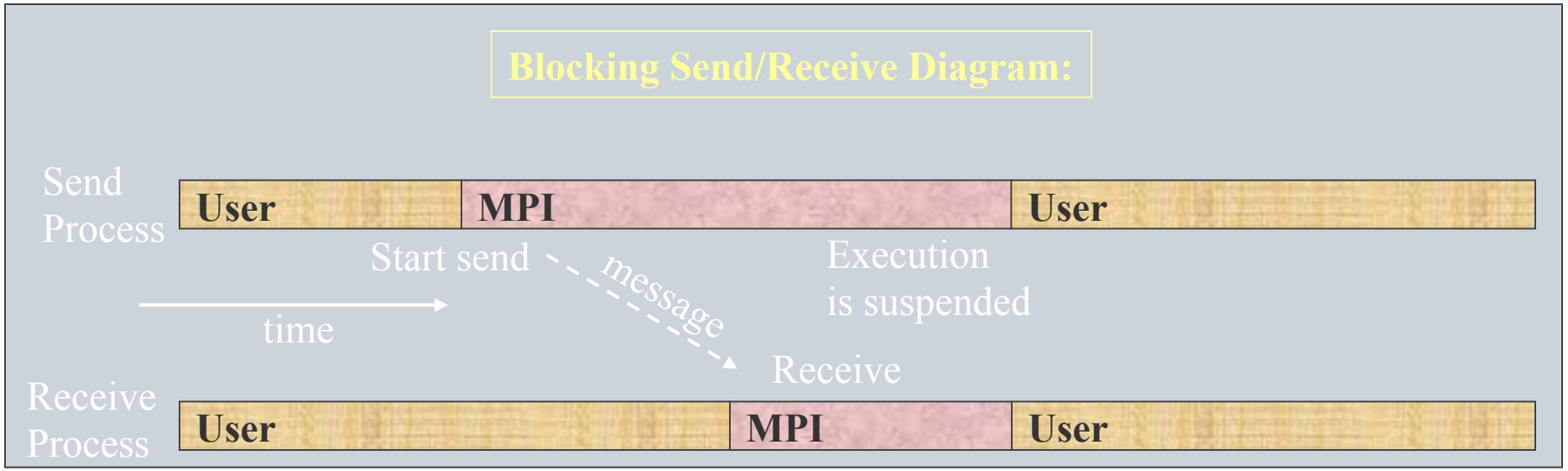
	blocking	nonblocking
standard	MPI_Send	MPI_Isend
synchronous	MPI_Ssend	MPI_ISsend
buffer	MPI_Bsend	MPI_Ibsend
ready	MPI_Rsend	MPI_Iresend

Blocking Calls

- A blocking send or receive call suspends execution of user's program until the message buffer being sent/ received is safe to use.
- In case of a blocking send, this means the data to be sent have been copied out of the send buffer, but they have not necessarily been received in the receiving task. The contents of the send buffer can be modified without affecting the message that was sent
- The blocking receive implies that the data in the receive buffer are valid.

Blocking Send and Receive

- **A blocking MPI call means that the program execution will be suspended until the message buffer is safe to use. The MPI standards specify that a blocking SEND or RECV does not return until the send buffer is safe to reuse (for MPI_SEND), or the receive buffer is ready to use (for MPI_RECV).**

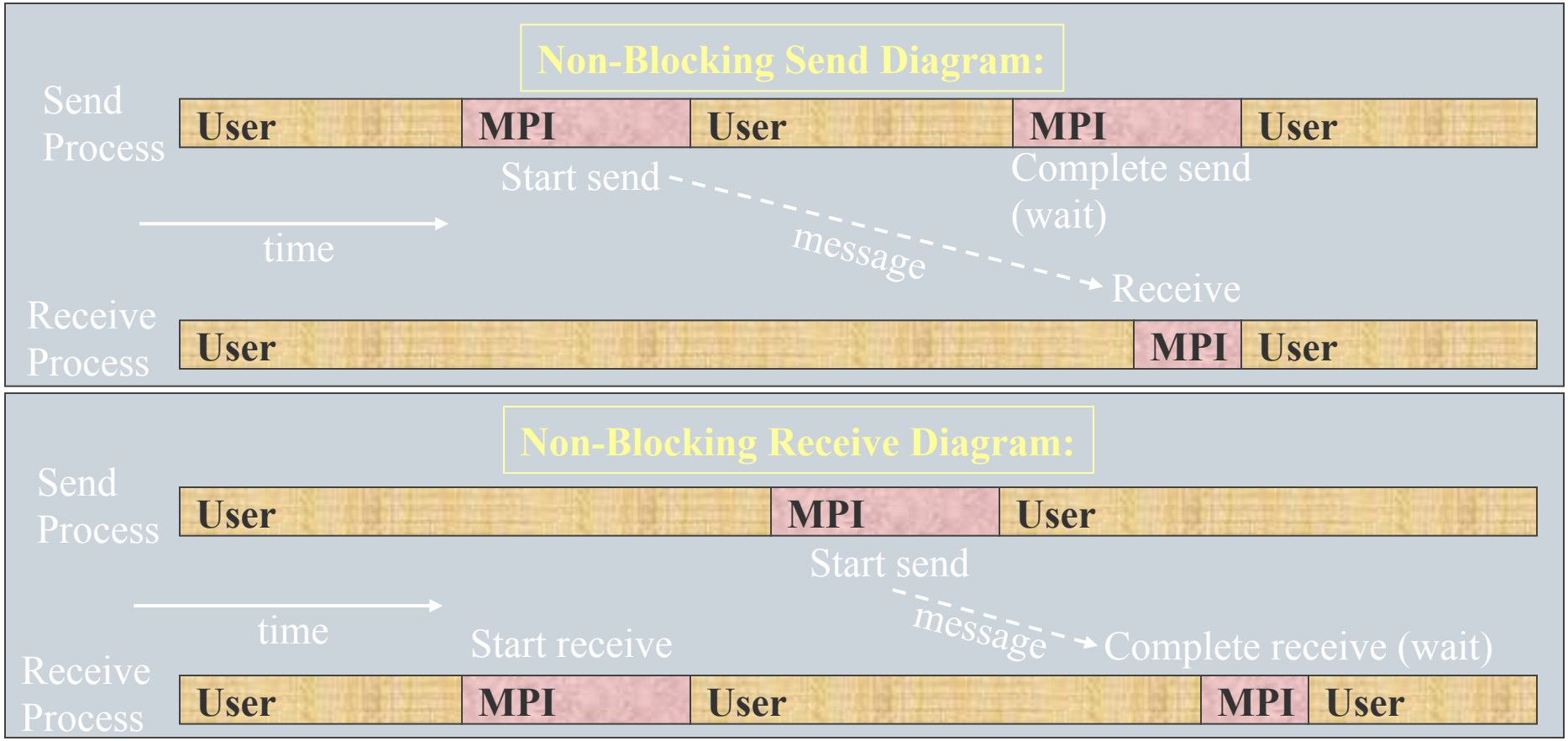


Non-Blocking Calls

- Non-blocking calls return immediately after initiating the communication.
- In order to reuse the send message buffer, the programmer must check for its status.
- The programmer can choose to block before the message buffer is used or test for the status of the message buffer.
- A blocking or non-blocking send can be paired to a blocking or non-blocking receive

Non-Blocking Send and Receive

- **Separate Non-Blocking communication into three phases:**
  - **Initiate non-blocking communication.**
  - **Do some work (perhaps involving other communications?)**
  - **Wait for non-blocking communication to complete.**

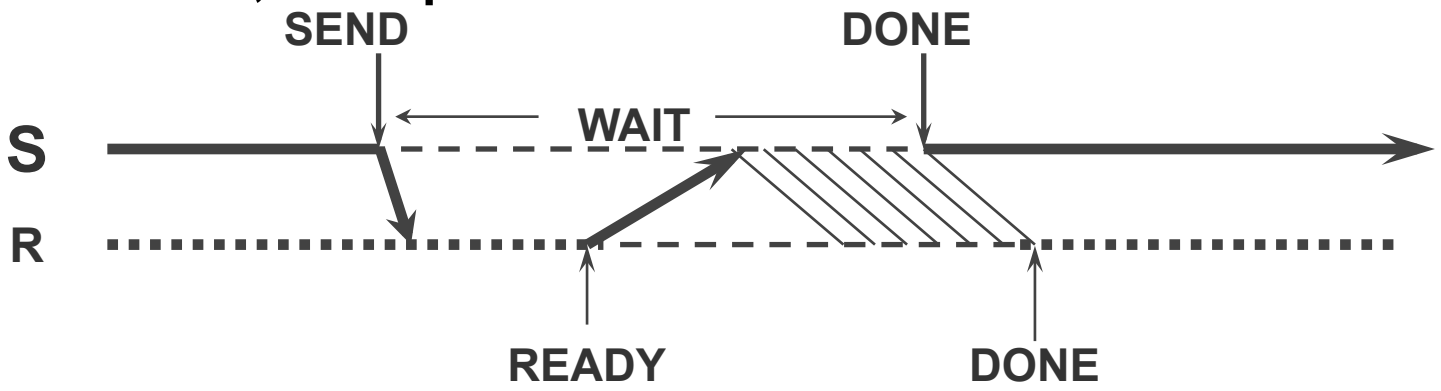


Communication Modes (Blocking)

Synchronous Send MPI_SSEND	Return when the message buffer can be safely reused. Send is locally complete
Buffered Send MPI_BSEND	Return when message is copied to the system buffer
Ready Send MPI_RSEND	Complete if matching receive is already waiting
Standard Send MPI_SEND	Either synchronous or buffered, implemented by vendor to give good performance for most programs

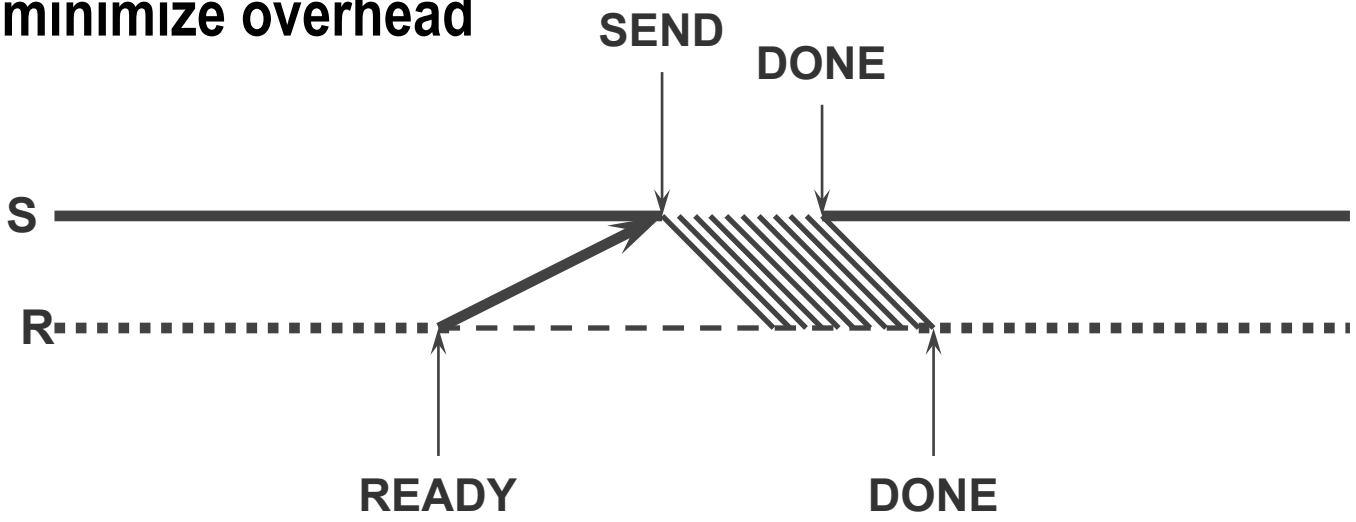
MPI_SSEND

- blocking synchronous send
- the sending task tells the receiver that a message is ready for it and waits for the receiver to acknowledge
- system overhead : buffer to network and vice versa
- synchronization overhead : handshake + waiting
- safest , most portable



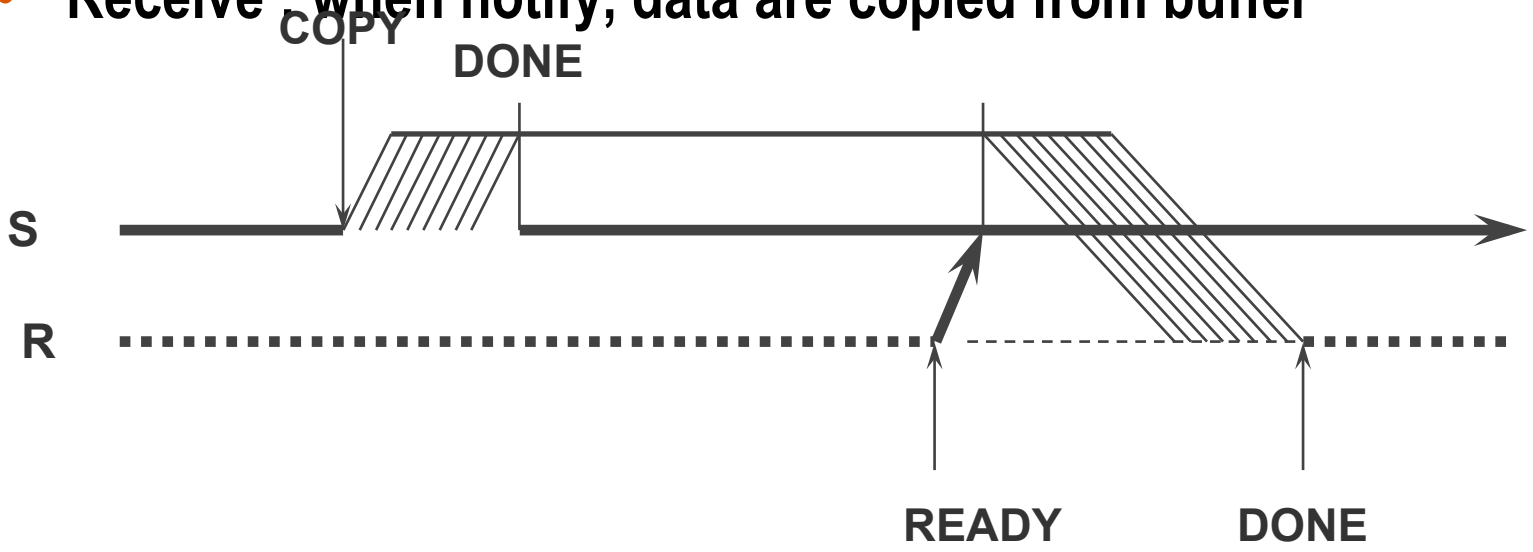
MPI_RSEND

- blocking ready send
- requires a “ready to receive” notification, if not => error, exit
- sends message out over network
- minimize overhead



MPI_BSEND

- Blocking buffer send (user-supply buffer on send node)
- Buffer can be statically or dynamically allocated
- Send : data copy to buffer and return
- Receive : when notify, data are copied from buffer



MPI_Send (Standard Send)

- Implemented by vendor to give good performance for most programs.
- Simple and easy to use
- Either synchronous or buffered
- MPICH : buffered send
- CRAY XT :
  - Based on MPICH2
  - Use a portals device for MPICH2
  - Support MPI2-RMA (one-sided)
  - Full MPI-IO support
  - No dynamic process management (NO Spawn process!!)
  - Man intro_mpi

Deadlock

- all tasks are waiting for events that haven't been initiated
- common to SPMD program with blocking communication, e.g every task sends, but none receives
- insufficient system buffer space is available
- remedies :
  - arrange one task to receive
  - use MPI_Ssendrecv
  - use non-blocking communication

Example: Deadlock

```
c     Improper use of blocking calls results in deadlock., run on two nodes
c     author : Roslyn Leibensperger, (CTC)
      program deadlock
      implicit none
      include 'mpif.h'
      integer MSGLEN, ITAG_A, ITAG_B
      parameter (MSGLEN = 2048, ITAG_A = 100, ITAG_B = 200)
      real rmsg1(MSGLEN) , rmsg2(MSGLEN)
      integer irank, idest, isrc, istag, iretag, istatus(MPI_STATUS_SIZE), ierr, I


      call MPI_Init (ierr)
      call MPI_Comm_rank( MPI_COMM_WORLD, irank, ierr)
      do I = 1, MSGLEN
          rmsg1(I) = 100
          rmsg2(I) = -100
      end do
```

Example : Deadlock (Cont'd)

```
if ( irank .eq. 0 ) then
   idest = 1
   isrc = 1
   istag = ITAG_A
   iretag = ITAG_B
else if ( irank .eq. 1 ) then
   idest = 0
   isrc = 0
   istag = ITAG_B
   iretag = ITAG_A
end if
print *, "Task ", irank, " has sent the message "
call MPI_Ssend (rmsg1,MSGLEN, MPI_REAL, isrc, iretag, MPI_COMM_WORLD,
                ierr)
call MPI_Recv (rmsg2, MSGLEN, MPI_REAL, isrc, iretag,
               MPI_COMM_WORLD,istatus, ierr)
print*, "Task", irank, " has received the message "
call MPI_Finalize (ierr)
end
```

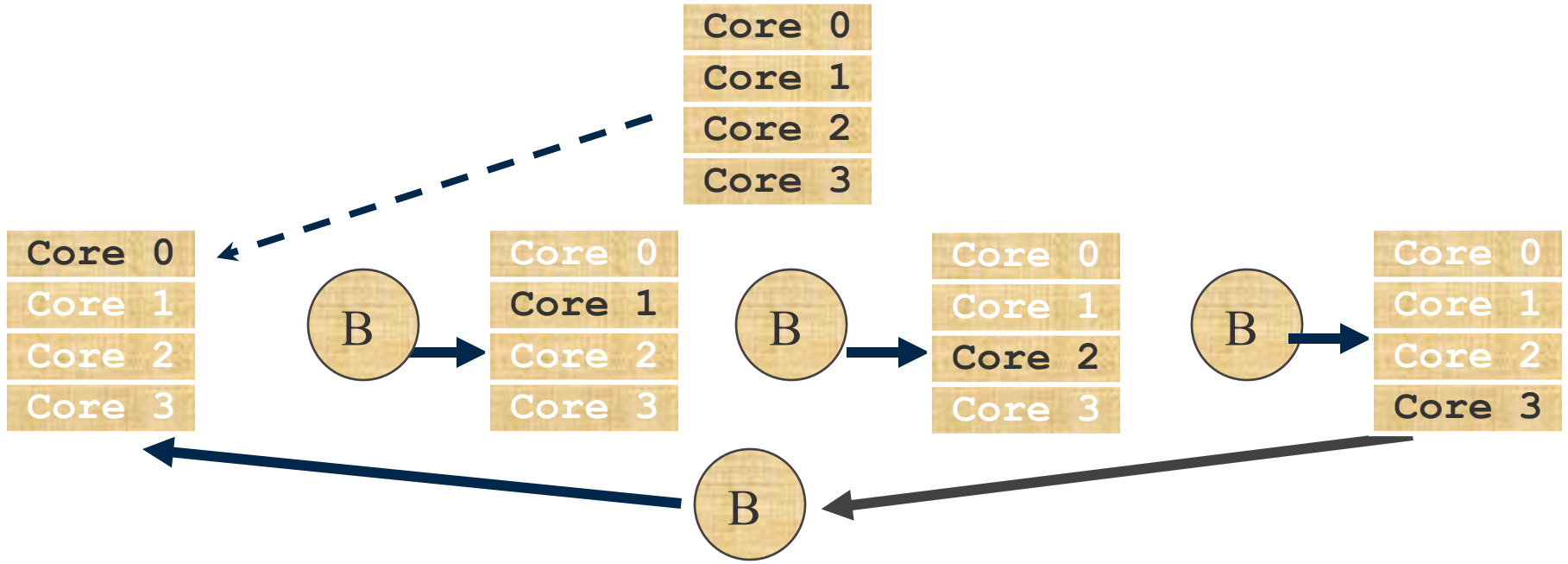
Example : Deadlock (fixed)

```
c     Solution program showing the use of a non-blocking send to eliminate deadlock
c     author : Roslyn Leibensperger (CTC)
      program fixed
      implicit none
      include 'mpif.h'
      ----------------------------------------
      ----------------------------------------
      print *, " Task", irank, " has started the send"
      call MPI_Isend ( rmsg1, MSGLEN, MPI_REAL, idest, istag,
     +        MPI_COMM_WORLD,  request, ierr)
      call MPI_Recv (rmsg2, MSGLEN, MPI_REAL, isrc, iretag,
     +                                   MPI_COMM_WORLD, irstatus, ierr)
      call MPI_Wait (irequest, isstatus, ierr)
      print *, "Task ", irank, "has completed the send"
      call MPI_Finalize (ierr)
      end
```

Example - Ring

```
#include <stdio.h>
#include “mpi.h”
main(int argc, char *argv[])
{
  int myrank, nprocs, leftid, rightid, val, sum, tmp;
  MPI_Status recv_status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  if((leftid=(myrank-1)) < 0) leftid = nprocs -1;
  if((rightid=(myrank+1) == nprocs) rightid = 0;
  val = myrank
  sum = 0;
 do {
   MPI_Send(&val,1,MPI_INT,rightid,99, MPI_COMM_WORLD);
   MPI_Recv(&tmp,1, MPI_INT, leftid, 99, MPI_COMM_WORLD, &recv_status);
   val = tmp;
   sum += val;
  } while (val != myrank);
  printf(“proc %d sum = %d \n”, myrank, sum);
  MPI_Finalize();
}
```

Example: Ring (Blocking Communication) – Schematic

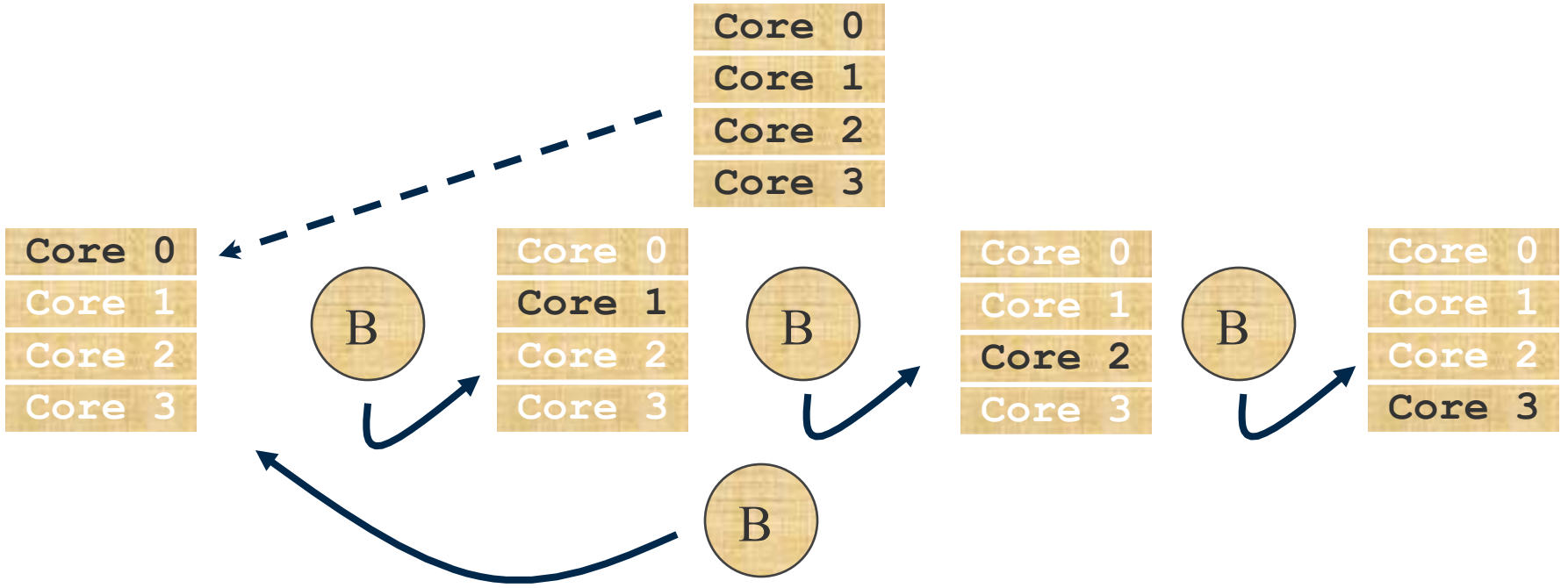


myrank=0 size=4 leftid=3 rightid=1 Val=3,2,1,0	myrank=1 size=4 leftid=0 rightid=2 Val=0,3,2,1	myrank=2 size=4 leftid=1 rightid=3 Val=1,0,3,2	myrank=3 size=4 leftid=2 rightid=0 Val=2,1,0,3

Example: Ring (Blocking Communication) – Fortran

```
      PROGRAM ring
      IMPLICIT NONE
      include "mpif.h"
      INTEGER ierror, val, my_rank, nprocs, rightid, leftid, tmp, sum, request
      INTEGER send_status(MPI_STATUS_SIZE), recv_status(MPI_STATUS_SIZE)
      CALL MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierror)
      rightid = my_rank + 1
      IF (rightid .EQ. nprocs) rightid = 0
      leftid = my_rank - 1
      IF (leftid .EQ. -1) leftid = nprocs-1
      sum = 0
      val = my_rank
  100 CONTINUE
      CALL MPI_SEND(val, 1, MPI_INTEGER, rightid, 99,
     $     MPI_COMM_WORLD, request, ierror)
      CALL MPI_RECV(tmp, 1, MPI_INTEGER, leftid, 99,
     $     MPI_COMM_WORLD, recv_status, ierror)
      sum = sum + tmp
      val = tmp
      IF(tmp  .NE. my_rank) GOTO 100
      PRINT *, 'Proc ', my_rank, ' Sum = ',sum
      CALL MPI_FINALIZE(ierror)
      STOP
      END
```

Example: Ring (Non-blocking Communication) – Schematic




```
myrank=0
size=4
leftid=3
rightid=1
```

```
myrank=1
size=4
leftid=0
rightid=2
```

```
myrank=2
size=4
leftid=1
rightid=3
```

```
myrank=3
size=4
leftid=2
rightid=0
```

Example: Ring (Non-blocking Communication) – C

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char *argv[]) {
  int myrank, nprocs, leftid, rightid, val, sum, tmp;
  MPI_Status recv_status, send_status;
  MPI_request send_request;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  if((leftid=(myrank-1)) < 0) leftid = nprocs -1;
  if((rightid=(myrank+1) == nprocs) rightid = 0;
  val = myrank
  sum = 0;
 do {
   MPI_Issend(&val,1,MPI_INT,right,99, MPI_COMM_WORLD,&send_request);
   MPI_Recv(&tmp,1, MPI_INT, left, 99, MPI_COMM_WORLD, &recv_status);
   MPI_Wait(&send_request,&send_status);
   val = tmp;
   sum += val;
  } while (val != myrank);
  printf("proc %d sum = %d \n", myrank, sum);
  MPI_Finalize();              }
```

Example: Ring (Non-blocking Communication) – Fortran

```
      PROGRAM ring
      IMPLICIT NONE
      include "mpif.h"
      INTEGER ierror, val, my_rank, nprocs, rightid, leftid, tmp, sum
      INTEGER send_status(MPI_STATUS_SIZE), recv_status(MPI_STATUS_SIZE)
      INTEGER request
      CALL MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierror)
      rightid = my_rank + 1
      IF (rightid .EQ. nprocs) rightid = 0
      leftid = my_rank - 1
      IF (leftid .EQ. -1) leftid = nprocs-1
      sum = 0
      val = my_rank
100   CONTINUE
      CALL MPI_ISSEND(val, 1, MPI_INTEGER, rightid, 99,
     $     MPI_COMM_WORLD, request, ierror)
      CALL MPI_RECV(tmp, 1, MPI_INTEGER, leftid, 99,
     $     MPI_COMM_WORLD, recv_status, ierror)
      CALL MPI_WAIT(request, send_status, ierror)
      sum = sum + tmp
      val = tmp
      IF(tmp .NE. my_rank) GOTO 100
      PRINT *, 'Proc ', my_rank, ' Sum = ',sum
      CALL MPI_FINALIZE(ierror)
      STOP
      END
```

Example: MPI Communication Timing Test

- The objective of this exercise is to investigate the amount of time required for message passing between two processes, i.e. an MPI communication timing test is performed.
- In this exercise different size messages are sent back and forth between two processes a number of times. Timings are made for each message before it is sent and after it has been received. The difference is computed to obtain the actual communication time. Finally, the average communication time and the bandwidth are calculated and output to the screen.
- For example, one can run this code on two nodes (one process on each node) passing messages of length 1, 100, 10,000, and 1,000,000 and record the results in a table.

Sendrecv

- useful for executing a shift operation across a chain of processes
- system take care of possible deadlock due to blocking calls

MPI_Sendrecv(sbuf, scount, stype, dest, stag, rbuf, rcount, rtype, source, rtag, comm, status)

- sbuf (rbuf) : initial address of send (receive)buffer
- scount (rcount) : number of elements in send (receive) buffer
- stype (rtype) : type of elements in send (receive) buffer
- stag (rtag) : send (receive) tag
- dest : rank of destination
- source : rank of source
- comm : communicator
- status : status object

Non-blocking Communications

- The non-blocking calls have the same syntax as the blocking calls, with two exceptions:
  - Each call has an “I” immediately following the “_”
  - The last argument is a handle to an opaque request object that contains information about the message
- Non-blocking call returns immediately after initiating the communication
- The programmer can block or check for the status of the message buffer : MPI_Wait or MPI_Test.

Non-blocking Send and Receive

- **Fortran :**

  call MPI_Isend(buf,count,datatype,dest,tag,comm,handle,ierr)

  call MPI_Irecv(buff,count,datatype,src,tag,comm,handle,ierr)

  call MPI_Test(handle, flag, status, ierr)

  call MPI_Wait (handle, status, ierr)

- **C :**

  MPI_Isend(&buf, count, datatype, dest, tag, comm, &handle)

  MPI_Irecv(&buff, count, datatype, src, tag, comm, &handle)

  MPI_Wait (&handle, &status)

  MPI_Test(&handle, &flag, &status)

Testing Communications for Completion

- MPI_Wait (request, status)

  These routines block until the communication has completed. They are useful when the data from the communication buffer is about to be re-used

- MPI_Test (request, flag, status)

  This routine blocks until the communication specified by the handle *request* has completed. The *request* handle will have been returned by an earlier call to a non-blocking communication routine. The routine queries completion of the communication and the result (TRUE of FALSE) is returned in *flag*

Timer

- C

  double MPI_Wtime(void)

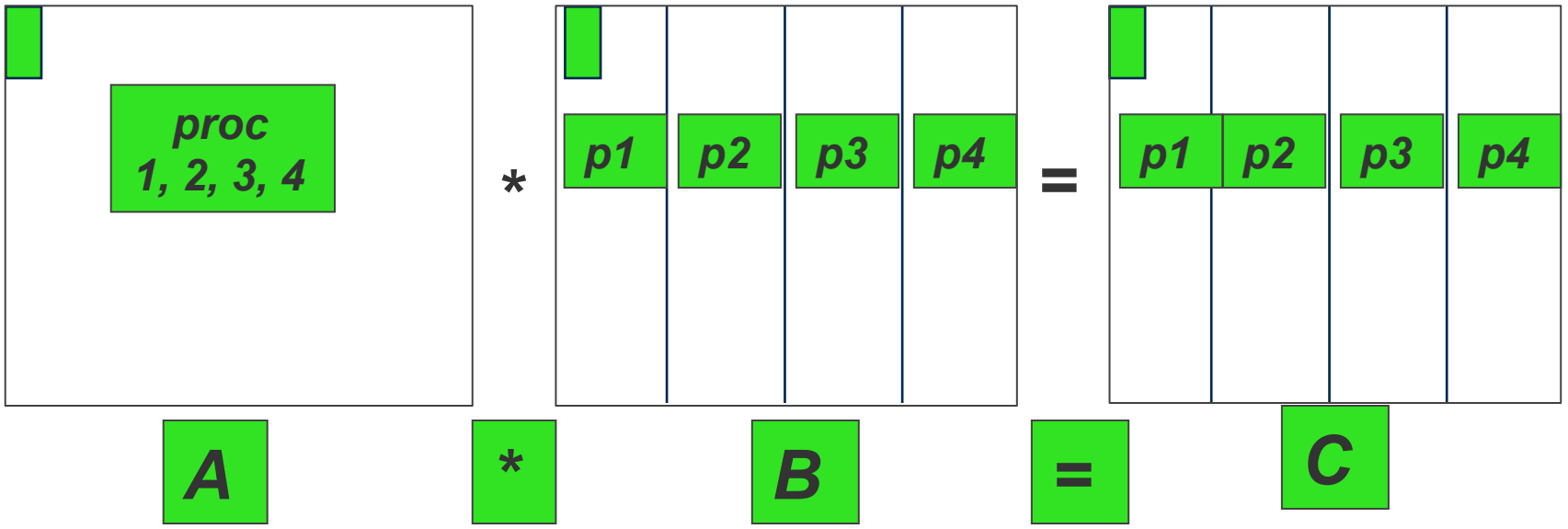
- Fortran :

  double precision MPI_Wtime()

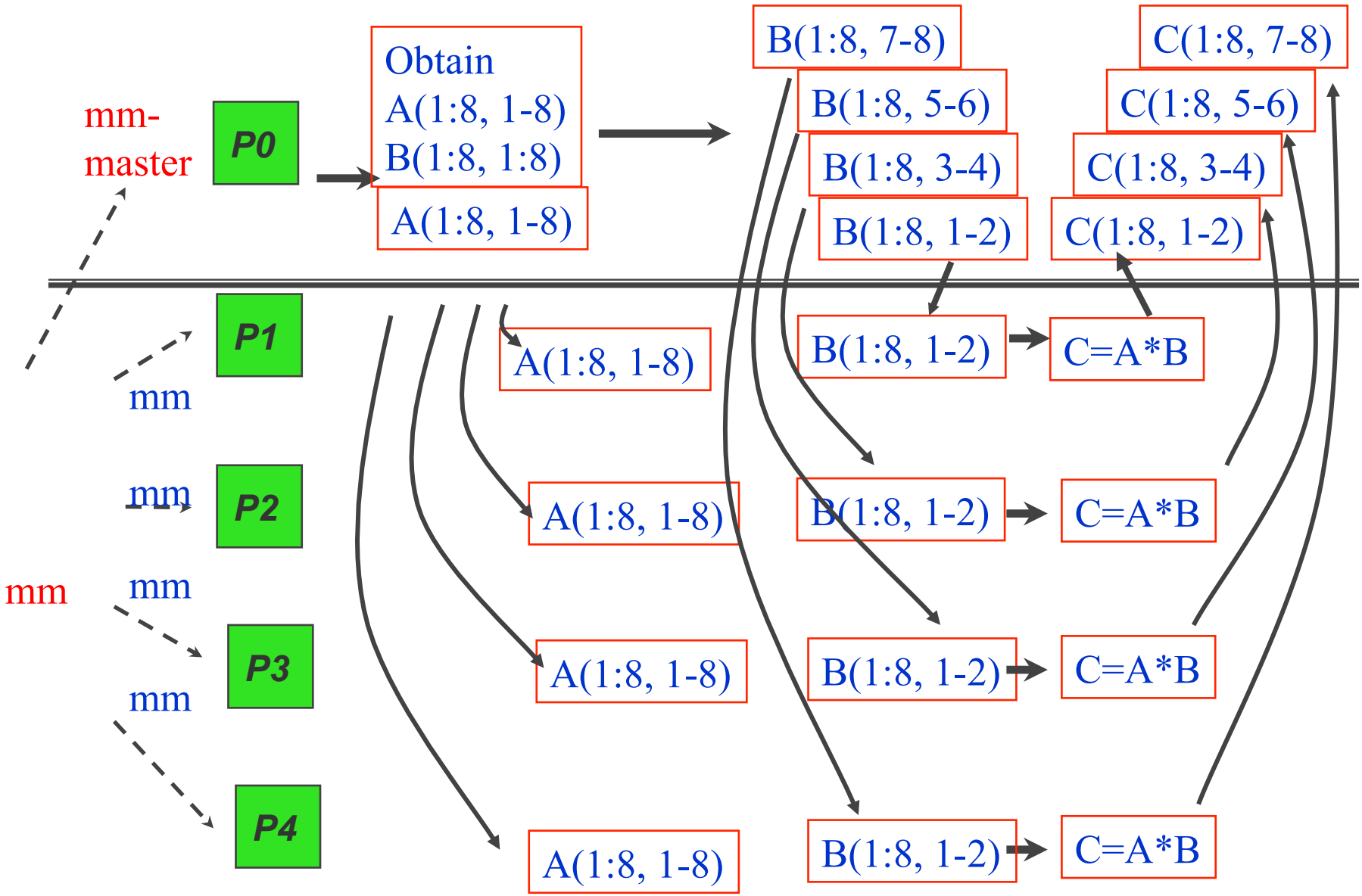
- Time is measured in seconds.
- Time to perform a task is measured by consulting the timer before and after
- Modify your program to measure its execution time and print it out

Simple Matrix Multiplication Algorithm

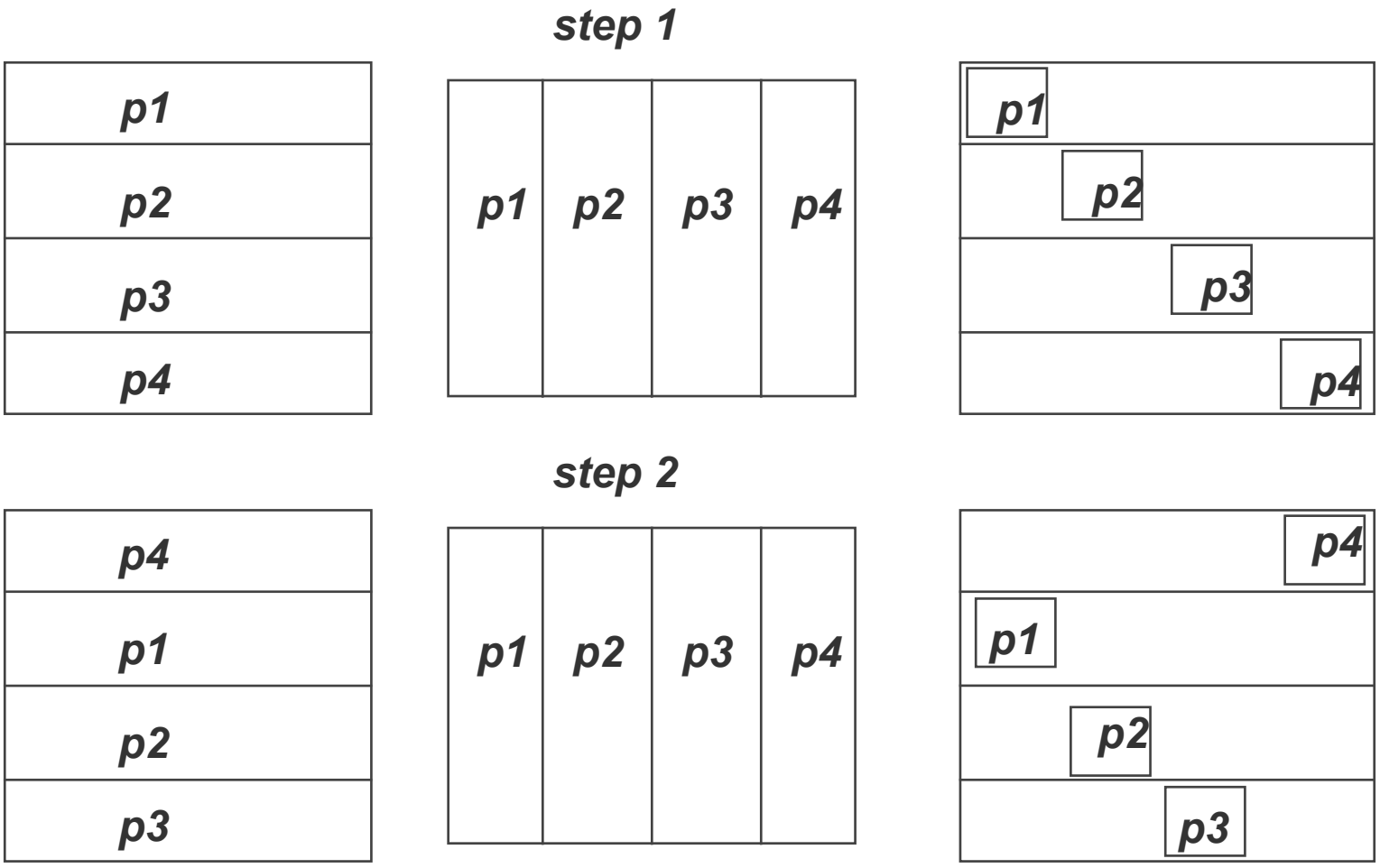
- Matrix A is copied to every processors (FORTRAN)
- Matrix B is divided into blocks and distributed to processors
- Perform matrix multiplication simultaneously
- Output solutions



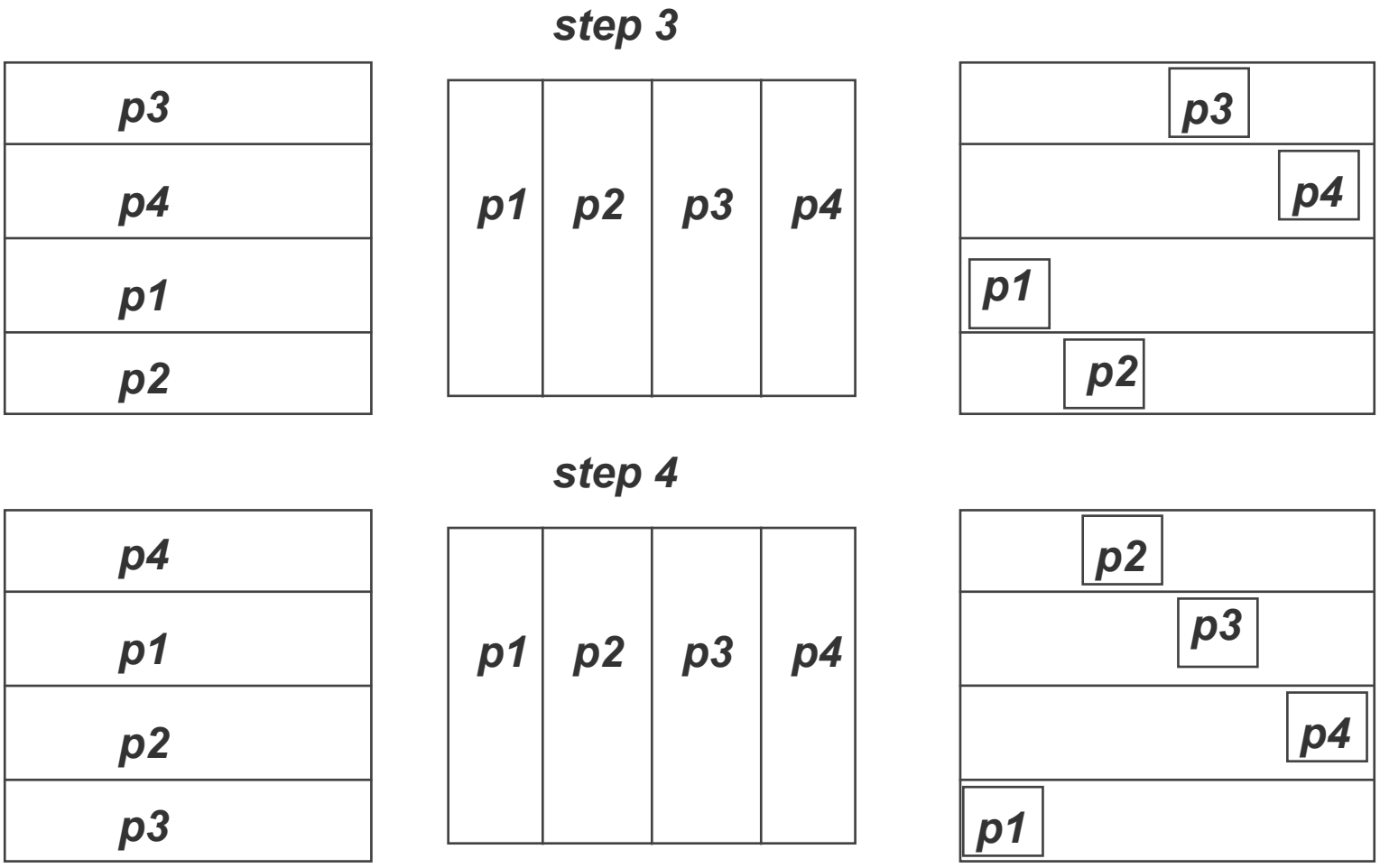
Parallel Processing



Matrix Multiplication

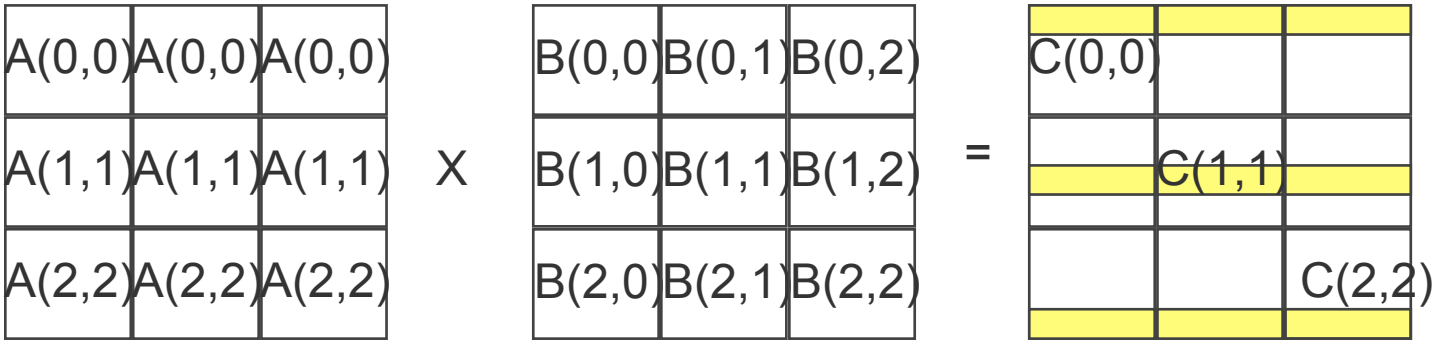


Matrix Multiplication (continued)



Fox's Algorithm (1)

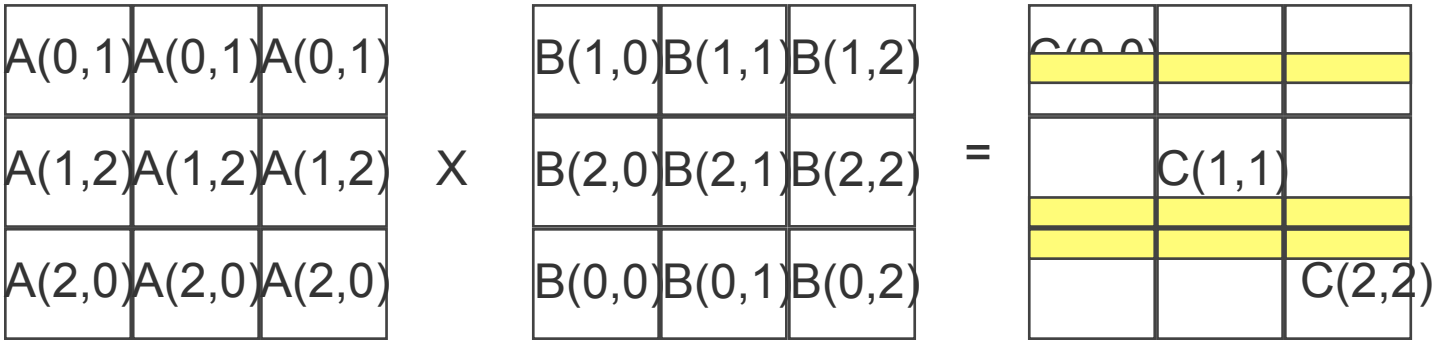
- **Broadcast the diagonal element of block A in rows, perform multiplication.**



- C(0,0) = A(0,0)*B(0,0) + A(0,1)*B(1,0) + A(0,2)*B(2,0)
- C(0,1) = A(0,0)*B(0,1) + A(0,1)*B(1,1) + A(0,2)*B(2,1)
- C(0,2) = A(0,0)*B(0,2) + A(0,1)*B(1,2) + A(0,2)*B(2,2)
- C(1,0) = A(1,0)*B(0,0) + A(1,1)*B(1,0) + A(1,2)*B(2,0)
- C(1,1) = A(1,0)*B(0,1) + A(1,1)*B(1,1) + A(1,2)*B(2,1)
- C(1,2) = A(1,0)*B(0,2) + A(1,1)*B(1,2) + A(1,2)*B(2,2)
- C(2,0) = A(2,0)*B(0,0) + A(2,1)*B(1,0) + A(2,2)*B(2,0)
- C(2,1) = A(2,0)*B(0,1) + A(2,1)*B(1,1) + A(2,2)*B(2,1)
- C(2,2) = A(2,0)*B(0,2) + A(2,1)*B(1,2) + A(2,2)*B(2,2)

Fox's Algorithm (2)

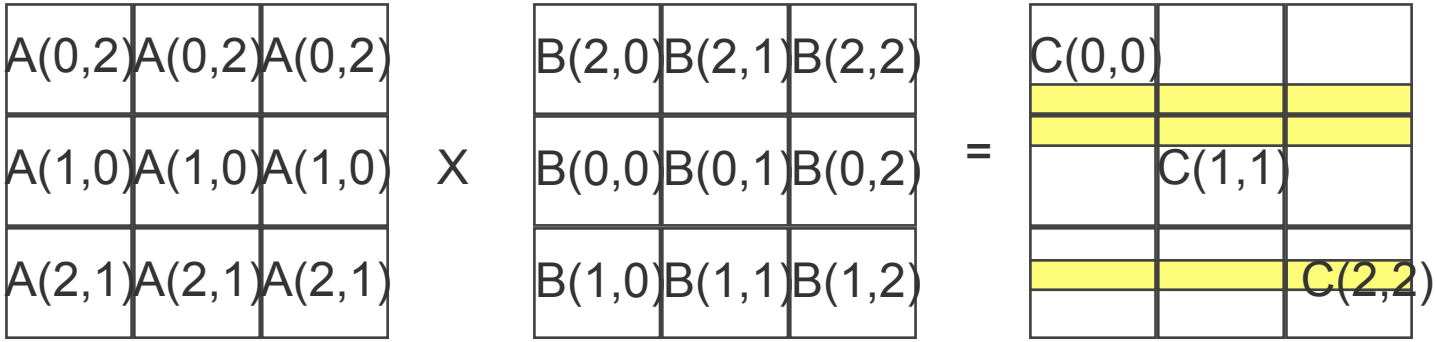
- **Broadcast next element of block A in rows, shift Bij in column, perform multiplication.**



- C(0,0) = A(0,0)*B(0,0) + A(0,1)*B(1,0) + A(0,2)*B(2,0)
- C(0,1) = A(0,0)*B(0,1) + A(0,1)*B(1,1) + A(0,2)*B(2,1)
- C(0,2) = A(0,0)*B(0,2) + A(0,1)*B(1,2) + A(0,2)*B(2,2)
- C(1,0) = A(1,0)*B(0,0) + A(1,1)*B(1,0) + A(1,2)*B(2,0)
- C(1,1) = A(1,0)*B(0,1) + A(1,1)*B(1,1) + A(1,2)*B(2,1)
- C(1,2) = A(1,0)*B(0,2) + A(1,1)*B(1,2) + A(1,2)*B(2,2)
- C(2,0) = A(2,0)*B(0,0) + A(2,1)*B(1,0) + A(2,2)*B(2,0)
- C(2,1) = A(2,0)*B(0,1) + A(2,1)*B(1,1) + A(2,2)*B(2,1)
- C(2,2) = A(2,0)*B(0,2) + A(2,1)*B(1,2) + A(2,2)*B(2,2)

Fox's Algorithm (3)

- Broadcast next element of block A in rows, shift Bij in column, perform multiplication.



- C(0,0) = A(0,0)*B(0,0) + A(0,1)*B(1,0) + A(0,2)*B(2,0)
- C(0,1) = A(0,0)*B(0,1) + A(0,1)*B(1,1) + A(0,2)*B(2,1)
- C(0,2) = A(0,0)*B(0,2) + A(0,1)*B(1,2) + A(0,2)*B(2,2)
- C(1,0) = A(1,0)*B(0,0) + A(1,1)*B(1,0) + A(1,2)*B(2,0)
- C(1,1) = A(1,0)*B(0,1) + A(1,1)*B(1,1) + A(1,2)*B(2,1)
- C(1,2) = A(1,0)*B(0,2) + A(1,1)*B(1,2) + A(1,2)*B(2,2)
- C(2,0) = A(2,0)*B(0,0) + A(2,1)*B(1,0) + A(2,2)*B(2,0)
- C(2,1) = A(2,0)*B(0,1) + A(2,1)*B(1,1) + A(2,2)*B(2,1)
- C(2,2) = A(2,0)*B(0,2) + A(2,1)*B(1,2) + A(2,2)*B(2,2)

Collective Communications

- Substitutes for a more complex sequence of p-p calls
- Involve all the processes in a process group
- Called by all processes in a communicator
- All routines block until they are locally complete
- Receive buffers must be exactly the right size
- No message tags are needed
- Divided into three subsets :
  - synchronization, data movement, and global computation

Barrier Synchronization Routines

- To synchronize all processes within a communicator
- A node calling it will be blocked until all nodes within the group have called it.
- C:

  ierr = MPI_Barrier(comm)

- Fortran:

  call MPI_Barrier(comm, ierr)

Broadcast

- one processor sends some data to all processors in a group
- C

  ierr = MPI_Bcast(buffer, count, datatype, root, comm)

- Fortran

  call MPI_Bcast(buffer,count,datatype, root, comm, ierr)

- The MPI_Bcast must be called by each node in a group, specifying the same communicator and root. The message is sent from the root process to all processes in the group, including the root process.

Scatter

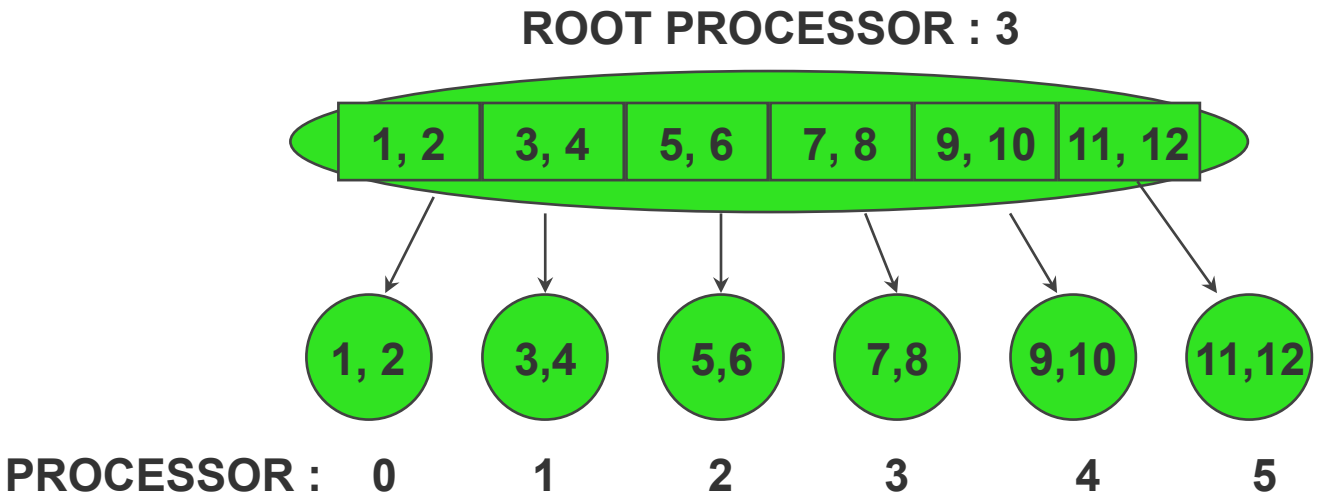
- Data are distributed into n equal segments, where the ith segment is sent to the ith process in the group which has n processes.
- C :

```
ierr = MPI_Scatter(&sbuff, scount, sdatatype, &rbuf, rcount,
    rdatatype, root, comm)
```

- Fortran :

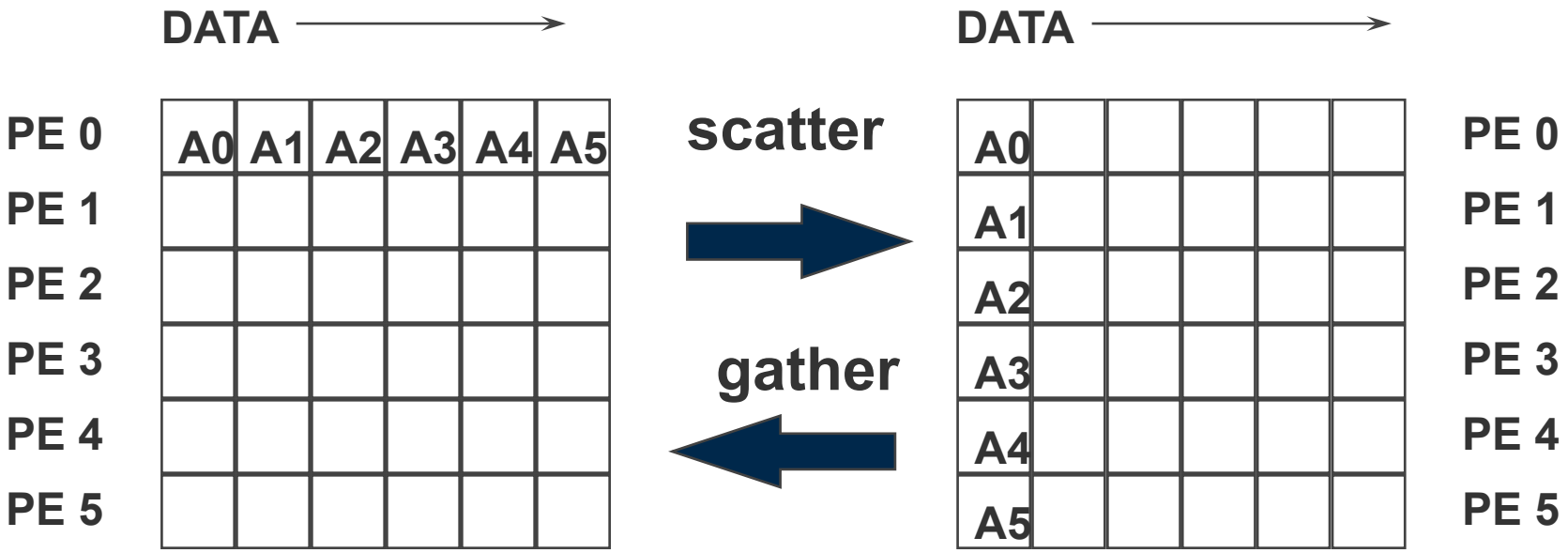
```
call MPI_Scatter(sbuff, scount, sdatatype, rbuf, rcount,
    rdatatype, root , comm, ierr)
```

Example : Scatter




```
real sbuf(12), rbuf(2)

call MPI_Scatter(sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 3,
    MPI_COMM_WORLD, ierr)
```

Scatter and Gather



Gather

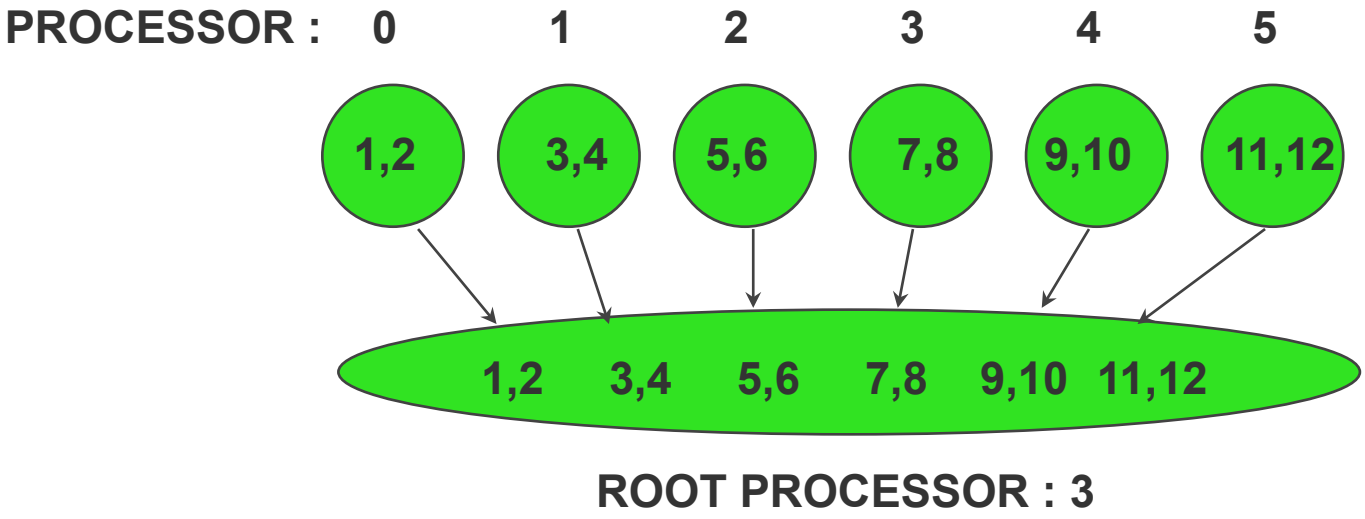
- Data are collected into a specified process in the order of process rank, reverse process of scatter.
- C :

```
ierr = MPI_Gather(&sbuf, scount, sdtatatype, &rbuf, rcount,
    rdatatype, root, comm)
```

- Fortran :

```
call MPI_Gather(sbuff, scount, sdatatype, rbuff, rcount,
    rdtatatype, root, comm, ierr)
```

Example : Gather




```
real sbuf(2),rbuf(12)

call MPI_Gather(sbuf,2,MPI_INT, rbuf, 2, MPI_INT, 3,
     MPI_COMM_WORLD, ierr)
```

Scatterv and Gatherv

- allow varying count of data and flexibility for data placement
- C :

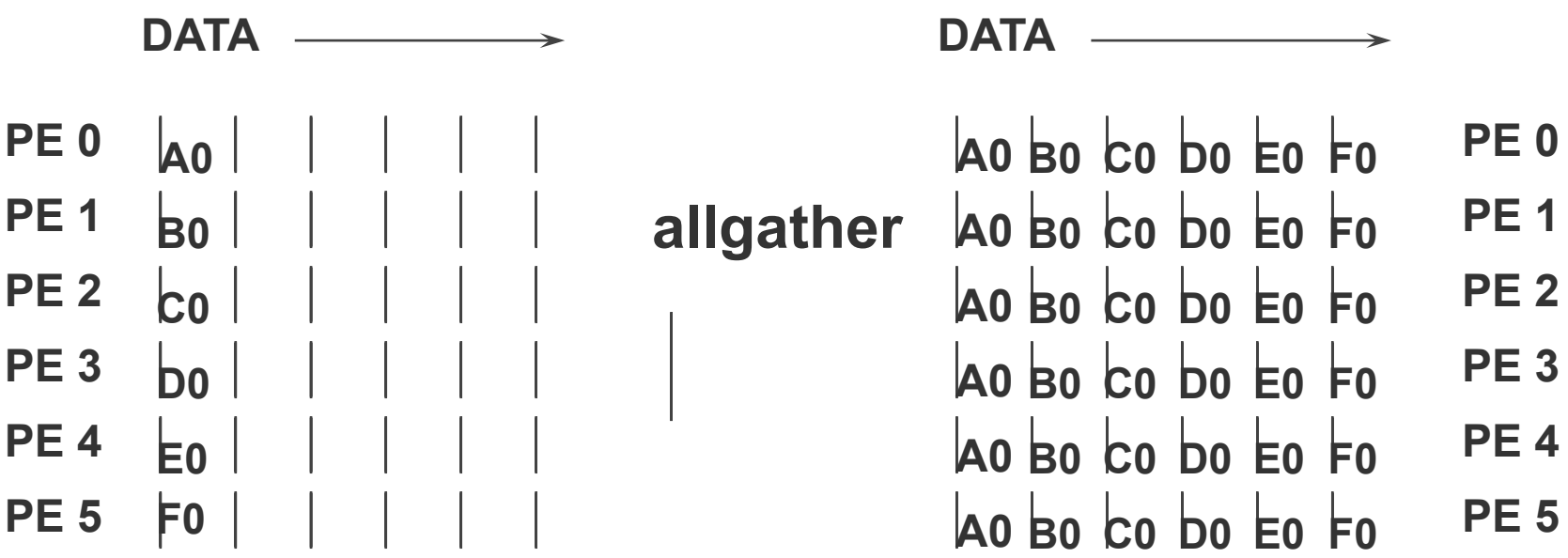
```
ierr = MPI_Scatterv( &sbuf, &scount, &displace, sdatatype,
    &rbuf, rcount, rdatatype, root, comm)

ierr = MPI_Gatherv(&sbuf, scount, sdatatype, &rbuf, &rcount,
    &displace, rdatatype, root, comm)
```

- Fortran :

```
call MPI_Scatterv(sbuf,scount,displace,sdatatype, rbuf, rcount,
    rdatatype, root, comm, ierr)
```

Allgather

```
ierr = MPI_Allgather(&sbuf, scount, stype,&rbuf, rcount, rtype,
                     comm)
```

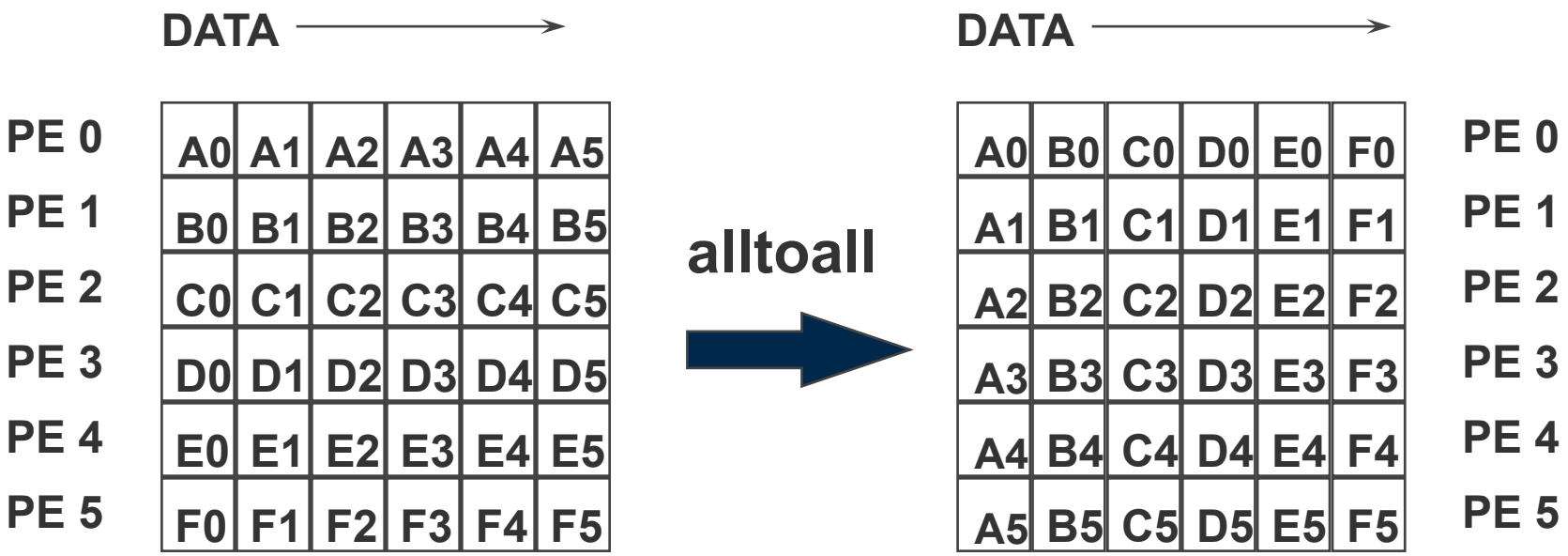


All to All

MPI_Alltoall(sbuf, scount, stype, rbuf, rcount, rtype, comm)

sbuf :	**starting address of send buffer (*)**
scount :	**number of elements sent to each process**
stype :	**data type of send buffer**
rbuff :	**address of receive buffer (*)**
rcount :	**number of elements received from any process**
rtype :	**data type of receive buffer elements**
comm :	**communicator**

All to All


DATA
PE 0
PE 1
PE 2
PE 3
PE 4
PE 5
A0 A1 A2 A3 A4 A5
B0 B1 B2 B3 B4 B5
C0 C1 C2 C3 C4 C5
D0 D1 D2 D3 D4 D5
E0 E1 E2 E3 E4 E5
F0 F1 F2 F3 F4 F5
alltoall
DATA
A0 B0 C0 D0 E0 F0
A1 B1 C1 D1 E1 F1
A2 B2 C2 D2 E2 F2
A3 B3 C3 D3 E3 F3
A4 B4 C4 D4 E4 F4
A5 B5 C5 D5 E5 F5
PE 0
PE 1
PE 2
PE 3
PE 4
PE 5

Global Reduction Routines

- The partial result in each process in the group is combined together using some desired function.
- The operation function passed to a global computation routine is either a predefined MPI function or a user supplied function
- Examples :
  - global sum or product
  - global maximum or minimum
  - global user-defined operation

Reduce and Allreduce

- MPI_Reduce(sbuf, rbuf, count, stype, op, root, comm)
- MPI_Allreduce(sbuf, rbuf, count, stype, op, comm)

  sbuf : address of send buffer

  rbuf : address of receive buffer

  count : the number of elements in the send buffer

  stype : the datatype of elements of send buffer

  op : the reduce operation function, predefined or user-defined

  root : the rank of the root process

  comm : communicator

  -- mpi_reduce returns results to single process

  -- mpi_allreduce returns results to all processes in the group

Predefined Reduce Operations

MPI NAME	FUNCTION	MPI NAME	FUNCTION
MPI_MAX	Maximum	MPI_LOR	Logical OR
MPI_MIN	Minimum	MPI_BOR	Bitwise OR
MPI_SUM	Sum	MPI_LXOR	Logical exclusive OR
MPI_PROD	Product	MPI_BXOR	Bitwise exclusive OR
MPI_LAND	Logical AND	MPI_MAXLOC	Maximum and location
MPI_LOR	Bitwise AND	MPI_MINLOC	Minimum and location

Example of Reduce w/Sum

```
c     A routines that computes the dot product of two vectors that are distributed
c   across a group of processses and returns the answer to node zero
      subroutine PAR_BLAS1( m , a , b , c , comm)
      real a(m) , b(m) , sum , c
      integer m , comm , I , ierr

      sum = 0.0
      do I = 1 , m
         sum = sum + a( I ) * b ( I )
      end do

      call MPI_Reduce( sum , c , 1 , MPI_REAL , MPI_SUM,0, comm , ierr )
      return
```

Derived Datatypes

- To provide a portable and efficient way of communicating non-contiguous or mixed types in a message
- datatypes that are built from the basic MPI datatypes.
- MPI datatypes are created at run-time through calls to MPI library
- Steps required
  - construct the datatype : define shapes and handle
  - allocate the datatype : commit types
  - use the datatype : use constructors
  - deallocate the datatype : free types

Datatypes

- **Basic datatypes :**
  - **MPI_INT, MPI_REAL, MPI_DOUBLE, MPI_COMPLEX,**
  - **MPI_LOGICAL, MPI_CHARACTER, MPI_BYTE ….**

- **MPI also supports array sections and structures through general datatypes. A general datatypes is a sequence of basic datatypes and integer byte displacements. These displacements are taken to be relative to the buffer that the basic datatype is describing. ==> *typemap***
  - **Datatype = { (type0, disp0) , (type1, disp1) , ….., (typeN, dispN)}**

Defining Datatypes

MPI_Type_contiguous(count,oldtype,newtype,ierr)
- ‘count’ copies of ‘oldtype’ are concatenated

MPI_Type_vector(count, buffer, strides, oldtype,newtype,ierr)
- ‘count’ blocks with ‘blen’ elements of ‘oldtype’ spaced by ‘stride’

MPI_Type_indexed(count, buffer, strides, oldtype,newtype,ierr)
- Extension of vector: varying ‘blens’ and ‘strides’

MPI_Type_struct(count, buffer, strides, oldtype,newtype,ierr)
-extension of indexed: varying data types allowed

MPI_Type_vector

- It replicates a datatype, taking blocks at fixed offsets.

  MPI_Type_vector(count, blocklen, stride, oldtype, newtype)

- The new datatype consists of :
  - *count : number of* blocks
  - each block is a repetition of *blocklen* items of *oldtype*
  - the start of successive blocks is offset by *stride* items of o*ldtype*

  If *count* = 2, *stride* = 4, *blocklen* = 3, *oldtype* = {(double, 0), (char, 8)}

  newtype = { (double, 0) , (char, 8) , (double, 16), (char, 24) ,
  (double, 32) , (char, 40) , (double, 64), (char, 72),
  (double, 80) , (char, 88) , (double, 96), (char, 104) }

Example

```
#include <mpi.h>
{
    float mesh[10][20];
    int dest, tag;
    MPI_Datatype newtype;
    /* * Do this once. */
    MPI_Type_vector(10,                 /* # column elements */
                    1,                  /* 1 column only */
                    20,                 /* skip 20 elements */
                     MPI_FLOAT,          /* elements are float */
                    &newtype);          /* MPI derived datatype */
MPI_Type_commit(&newtype); /* * Do this for every new message. */
MPI_Send(&mesh[0][19], 1, newtype, dest, tag, MPI_COMM_WORLD);
}
```

MPI_Type_struct

- to gather a mix of different datatypes scattered at many locations in space into one datatype

  MPI_Type_struct (count , array_of_blocklength , array_of_ displacements, array_of_types , newtype , ierr)

  - count : number of blocks
  - array_of_blocklength (B) : number of elements in each block
  - array_of_displacements (I) : byte of displacement of each block
  - array_of_type (T) : type of elements in each block

  if count = 3 , T = {MPI_FLOAT, type1, MPI_CHAR} , I = {0 , 16, 26}

  B = {2 , 1 , 3} , type1 = { (double, 0), (char, 8)} ,

  - newtype = { (float, 0) , (float, 4) , (double, 16), (char, 24), (char, 26) , (char, 27), (char, 28) }

example

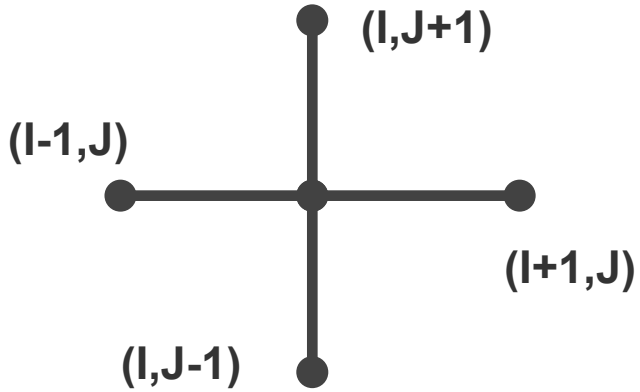
```
Struct{     char        display[50];
            int         maxiter;
            double      xmin,ymin,  xmax, ymax;
            int         width,  height;          } cmdline;


/* set up 4 blocks */
Int                     blockcounts[4] = {50, 1, 4, 2};
MPI_Datatype            types[4];
MPI_Aint                displs[4];
MPI_Datatype            cmdtype;


/* initialize types and displacements with addresses of items */
MPI_Address(&cmdline.display, &displs[0];
MPI_Address(&cmdline.maxiter, &displs[1];
MPI_Address(&cmdline.xmin, &displs[2];
MPI_Address(&cmdline.width, &displs[3];
types[0] = MPI_CHAR; types[1]=MPI_INT; types[2]=MPI_DOUBLE; types[3]=MPI_INT;
```

Parallel Code Development

Example : Laplace Equation

$$\nabla^2 T = \frac{\partial^2 T}{\partial X^2} + \frac{\partial^2 T}{\partial Y^2} = 0$$

5- POINT FD STENCIL



POINT JACOBI ITERATION

```
T( I , J ) =  0.25 * { Told( I -1 , J  )
                     +( Told( I+1 , J )
                     +( Told( I , J  -1 )
                     +( Told( I , J + 1) }
```

Codes

Initial guess : T = 0 ,

Boundaries : T = 100 ,

Grid : 1000x1000

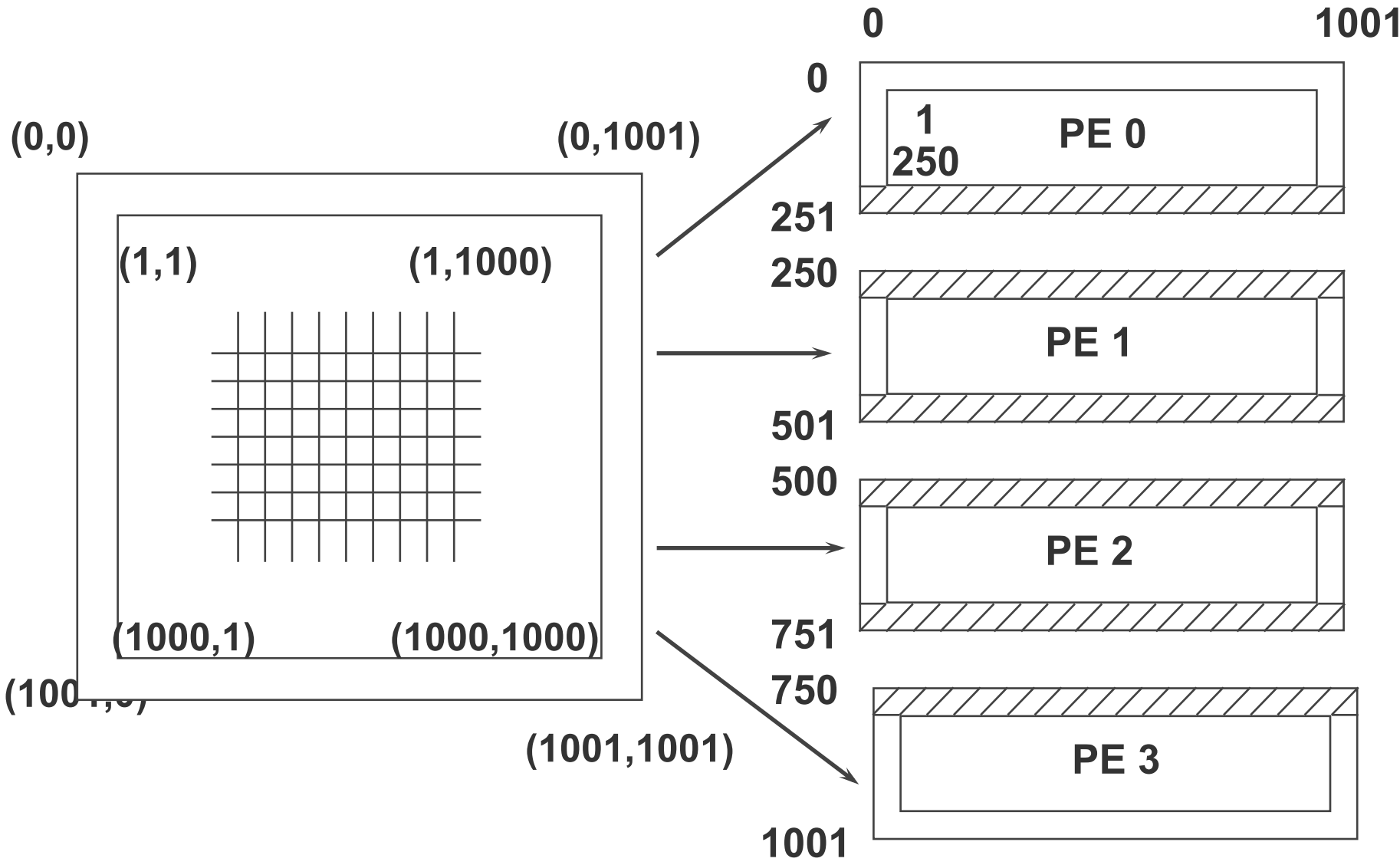
Fortran :

```
Do J = 1 , NC
  Do I = 1 , NR
  T ( I , J ) = 0.25 * (Told ( I -1 ,J ) + Told ( I + 1 ,J) + Told( I , J-1 ) + Told(I, J+1) )
  end do
end do
```

C :

```
for  ( I=1 ;  I <=NR ;  I++)
  for (j=1 ; j <= NC ; j++ )
  T[I][j] = 0.25*( Told[I+1][j]  + Told[I-1][j] + Told[I][j+1] + Told[I][j-1] );
```

Domain Decomposition



Program formulation

- Include file
- Initialization
- processor information
- Iteration : {
  - do averaging
  - copy T into Told
  - send values down
  - send values up
  - receive values from above
  - receive values from below
  - find the max change
  - synchronize }

Processor Information

- **Header:**

  #include <mpi.h> or include 'mpif.h'

- **Initialization :**

  call MPI_Init(ierr)

  ierr = MPI_Init(& argc, & argv)

- **Number of processors:**

  call MPI_Comm_size(MPI_COMM_WORLD, npes, ierr)

  ierr = MPI_Comm_size(MPI_COMM_WORLD, &npes)

- **Processor number :**

  call MPI_Comm_rank(MPI_COMM_WORLD, mype, ierr)

  ierr= MPI_Comm_rank(MPI_COMM_WORLD, &mype)

Initialization

Serial (interior)

```
for (I = 0 ; I <= NR + 1 ; I ++)
      for ( J = 0 ; <= NC +1 ; J++)
            T[I][J] = 0. 0;


do I = 0 , NR +1
      do J =0 , NC + 1
            T( I, J ) = 0.0
      enddo
enddo
```

Parallel (boundaries)

```
/* Left and right boundaries */
for (I = 0; I <= NRL + 1; I++) {
      T[I][0] = 100.0;
      T[I][NCL+1] =100.0; }


/*Top and Bottom Boundaries*/
if (MYPE == 0)
      for (J = 0; J <=NCL+1; j++)
            T[0][J] = 100.0;
if (MYPE == NPES - 1)
      for (J = 0 ; J <= NCL +1 ; J++)
            T[NRL+1][J] = 100.0
```

Do Averaging

NRL : NR / NPES

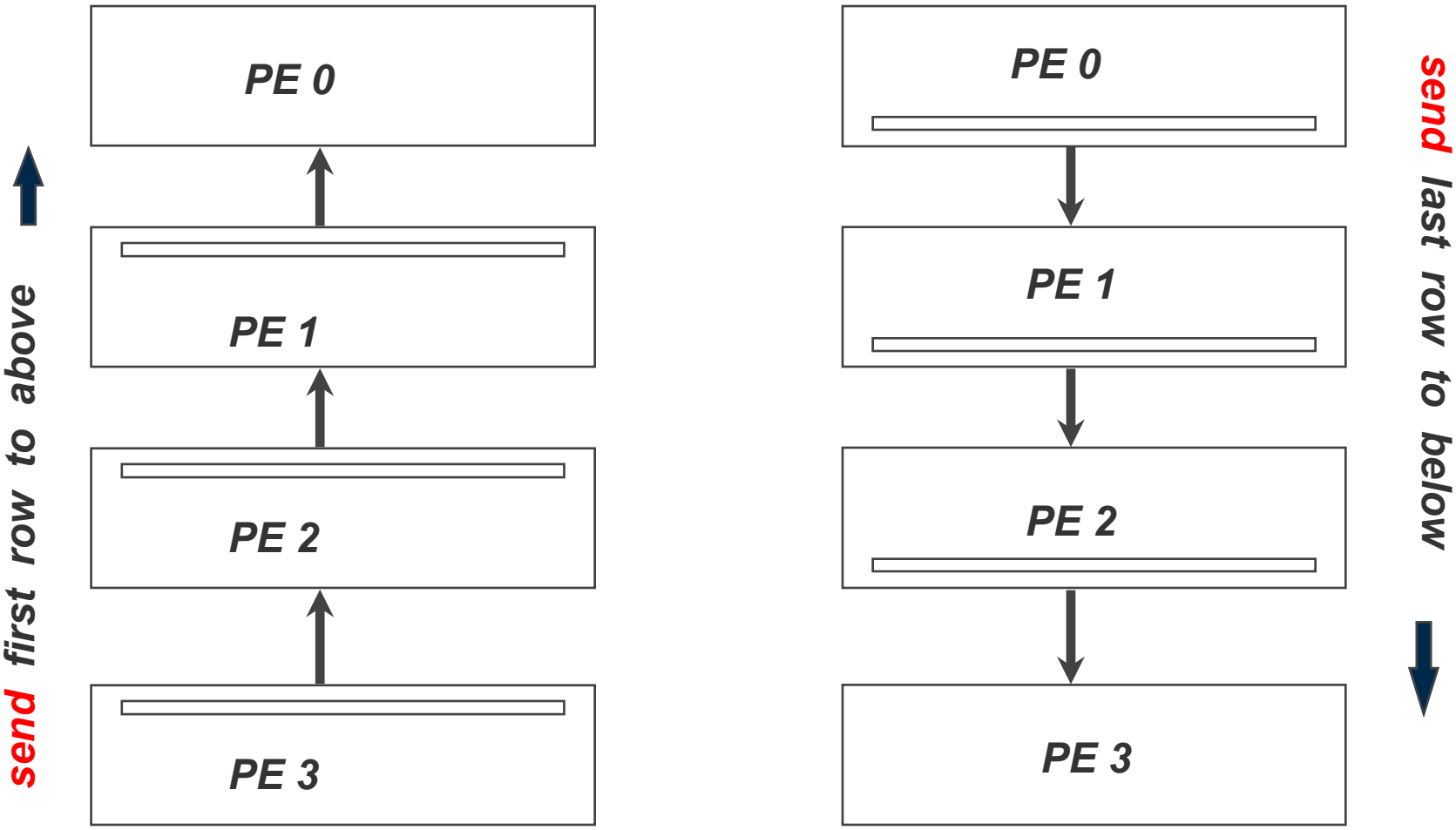
Fortran :

```
Do J = 1 , NC
  Do I = 1 , NRL
  T ( I , J ) = 0.25 * (Told ( I -1 ,J ) + Told ( I + 1 ,J) + Told( I , J-1 )
          + Told(I, J+1) )
  end do
end do
```

C :

```
for  ( I=1 ;  I <=NRL ;  I++)
  for (j=1 ; j <= NC ; j++ )
  T[I][j] = 0.25*( Told[I+1][j]  + Told[I-1][j] + Told[I][j+1] +
          Told[I][j-1] );
```

Parallel Send



Updating T Values at Domain Boundaries (Send)

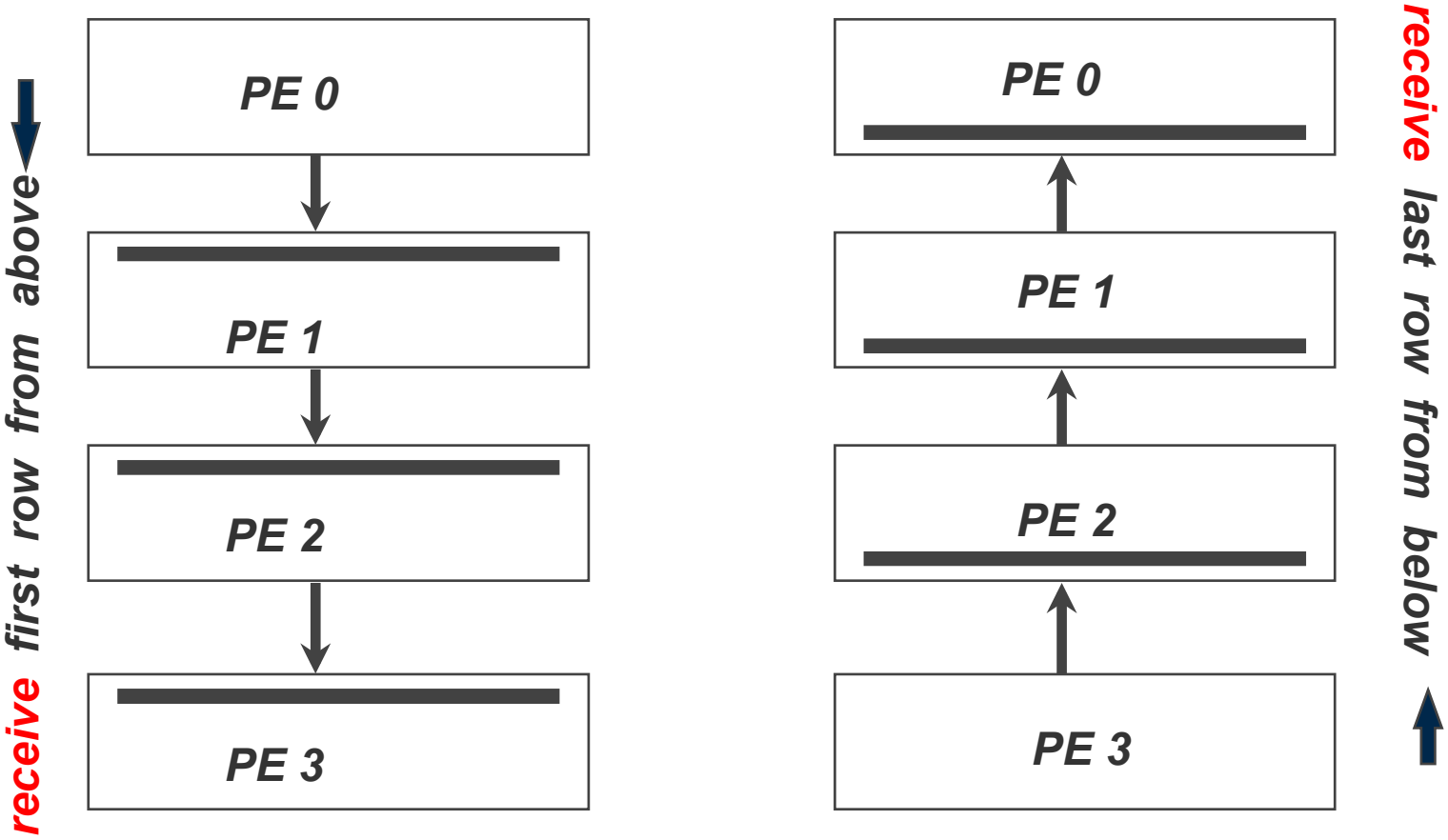
Sending up :

```
If (mype != 0 )    MPI_Send( &t[1][1], NC, MPI_FLOAT, mype-1, UP,
                   MPI_COMM_WORLD )
```

Sending down :

```
If (mype < npes-1 )    MPI_Send( &t[NRL][1], NC, MPI_FLOAT, mype+1, UP,
     MPI_COMM_WORLD)
```

Parallel Template : Receive



Updating Values at Domain Boundaries (Receive)

Receiving from up :

```
If (mype != 0 )    MPI_Recv( &t[0][1], NC, MPI_FLOAT,
    MPI_ANY_SOURCE, DOWN, &status, MPI_COMM_WORLD )
```

Receiving from down :

```
If (mype  != npes-1 )    MPI_Recv( &t[NRL+1][1], NC, MPI_FLOAT,
    MPI_ANY_SOURCE, UP, &status, MPI_COMM_WORLD)
```

Find Maximum Change

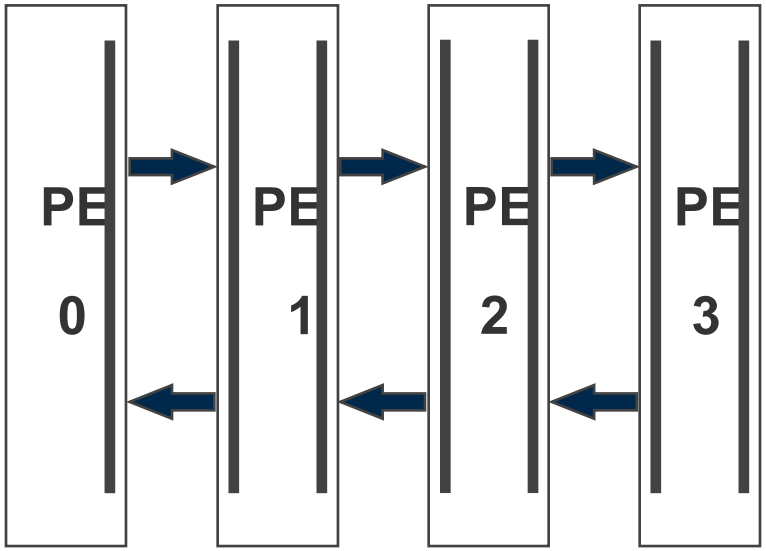
- each processor finds its own maximum change, *dt*
- find the global change using reduce, *dtg*

```
MPI_Reduce (&dt, &dtg, 1, MPI_FLOAT, MPI_MAX, PE0, comm)
call MPI_Reduce( dt, dtg, 1, MPI_FlOAT, MPI_MAX, PE0, comm, ierr)
```

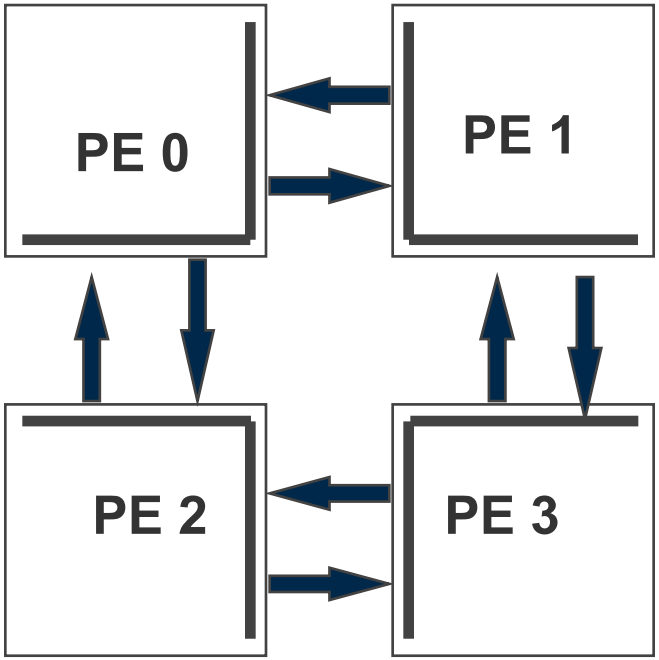
- synchronization :

```
MPI_Barrier(MPI_COMM_WORLD)
call MPI_Barrier(MPI_COMM_WORLD, ierr)
```

Data Distribution



Columnwise domain decomposition with grid overlap (FORTRAN)



X-Y 2Ddomain decomposition with grid overlap

Introduction to OpenMP

I. About OpenMP

II. OpenMP Directives

III. Data Scope

IV. Runtime Library Routines
and Environment Variables

V. Using OpenMP

Outline

I. About OpenMP

II. OpenMP Directives

III. Data Scope

IV. Runtime Library Routines and Environment Variables

V. Using OpenMP

I. About OpenMP

- Industry-standard *shared memory programming model*
- Developed in 1997
- OpenMP Architecture Review Board (ARB) determines additions and updates to standard

Advantages to OpenMP

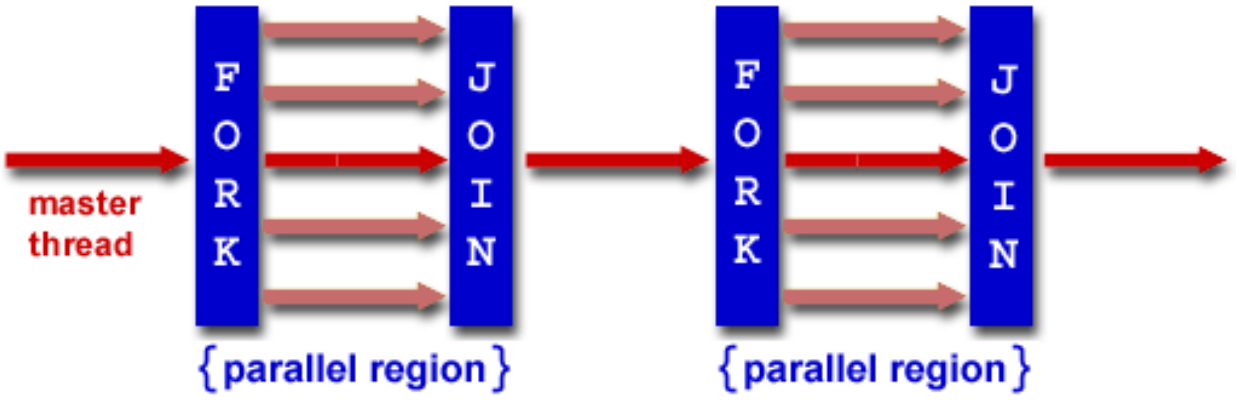
- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can express simple or complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, so code is easy to read
- Single source code for OpenMP and non-OpenMP
  – non-OpenMP compilers simply ignore OMP directives

OpenMP Programming Model

- Application Programmer Interface (API) is combination of
  - Directives
  - Runtime library routines
  - Environment variables
- API falls into three categories
  - Expression of parallelism (flow control)
  - Data sharing among threads (communication)
  - Synchronization (coordination or interaction)

Parallelism

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model



Source: https://computing.llnl.gov/tutorials/openMP/

II. OpenMP Directives

- Syntax overview
- Parallel
- Loop
- Sections
- Synchronization
- Reduction

Syntax Overview: C/C++

- Basic format

```
#pragma omp directive-name [clause] newline
```

- All directives followed by newline
- Uses pragma construct (pragma = Greek for “thing”)
- Case sensitive
- Directives follow standard rules for C/C++ compiler directives
- Long directive lines can be continued by escaping newline character with “\”

Syntax Overview: Fortran

- Basic format:

  sentinel directive-name [clause]

- Three accepted sentinels: `!$omp` `*$omp` `c$omp`
- Some directives paired with `end` clause
- Fixed-form code:
  - Any of three sentinels beginning at column 1
  - Initial directive line has space/zero in column 6
  - Continuation directive line has non-space/zero in column 6
  - Standard rules for fixed-form line length, spaces, etc. apply
- Free-form code:
  - `!$omp` only accepted sentinel
  - Sentinel can be in any column, but must be preceded by only white space and followed by a space
  - Line to be continued must end in "`&`" and following line begins with sentinel
  - Standard rules for free-form line length, spaces, etc. apply

OpenMP Directives: Parallel

- A block of code executed by multiple threads
- Syntax: C

```
#pragma omp parallel private(list)\
  shared(list)
{
    /* parallel section goes here */
}
```

- Syntax: Fortran

```
!$omp parallel private(list) &
!$omp shared(list)
!     Parallel section goes here
!$omp end parallel
```

Simple Example (C/C++)

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[]) {
   int tid;
   printf("Hello world from C threads:\n");
   #pragma omp parallel private(tid)
   {
       tid = omp_get_thread_num();
       printf("<%d>\n", tid);
   }
   printf("I am sequential now\n");
   return 0;
}
```

Simple Example (Fortran)

```
program hello
integer tid, omp_get_thread_num
write(*,*) 'Hello world from Fortran threads:'
!$OMP parallel private(tid)
tid = omp_get_thread_num()
write(*,*) '<', tid, '>'
!$omp end parallel
write(*,*) 'I am sequential now'
end
```

Output (Simple Example)

Output 1

```
Hello world from C
  threads:
<0>
<3>
<2>
<4>
<1>
I am sequential now
```

Output 2

```
Hello world from
  Fortran threads:
<1>
<2>
<0>
<4>
<3>
I am sequential now
```

Order of execution is scheduled by OS!!!

OpenMP Directives: Loop

- Iterations of the "for" or "do" loop following the directive are executed in parallel
- Syntax: C

```
#pragma omp for schedule(type [,chunk]) \
private(list) shared(list) nowait
  {
    /* for loop goes here */
  }
```

- Syntax: Fortran

```
!$OMP do schedule(type [,chunk]) &
!$OMP private(list) shared(list)
! do loop goes here
!$OMP end do nowait
```

  - *type* = {static, dynamic, guided, runtime}
  - If *nowait* specified, threads do not synchronize at end of loop

Which Loops Are Parallelizable?

Parallelizable

- Number of iterations known upon entry, and does not change
- Each iteration independent of all others
- No data dependence

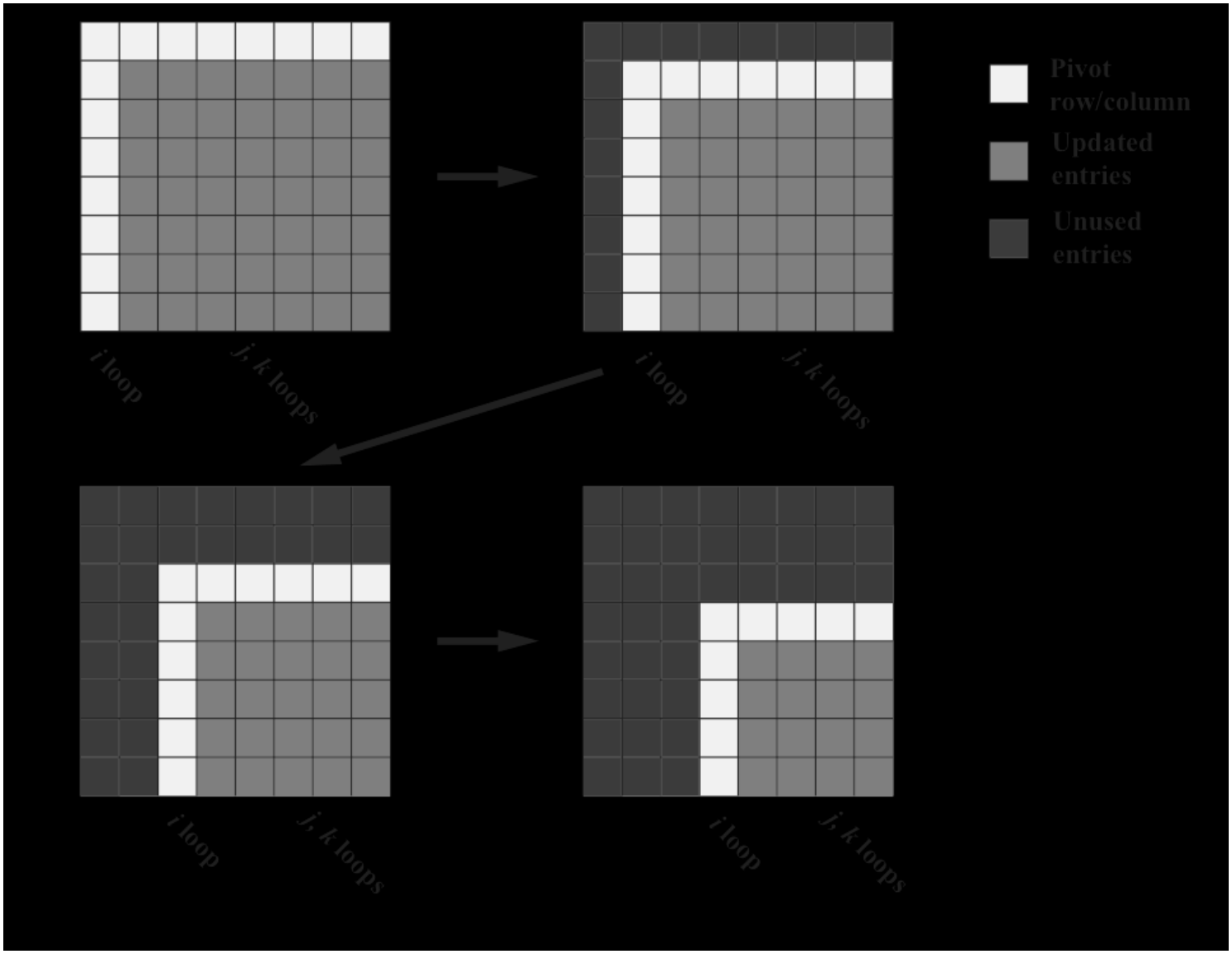
Not Parallelizable

- Conditional loops (many while loops)
- Iterator loops (e.g., iterating over a `std::list<...>` in C++)
- Iterations dependent upon each other
- Data dependence

Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):
      x = A\b                          */

for (int i = 0; i < N-1; i++) {
  for (int j = i; j < N; j++) {
    double ratio = A[j][i]/A[i][i];
    for (int k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

Example: Parallelizable?



Example: Parallelizable?

- Outermost Loop (`i`):
  - `N-1` iterations
  - Iterations depend upon each other (values computed at step `i-1` used in step `i`)
- Inner loop (`j`):
  - `N-i` iterations (constant for given `i`)
  - Iterations can be performed in any order
- Innermost loop (`k`):
  - `N-i` iterations (constant for given `i`)
  - Iterations can be performed in any order

Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):
   x = A\b                                  */

for (int i = 0; i < N-1; i++) {
#pragma omp parallel for
  for (int j = i; j < N; j++) {
    double ratio = A[j][i]/A[i][i];
    for (int k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

Note: can combine `parallel` and `for` into single `pragma` line

OpenMP Directives: Loop Scheduling

- Default scheduling determined by implementation
- Static
  - ID of thread performing particular iteration is function of iteration number and number of threads
  - Statically assigned at beginning of loop
  - Load imbalance may be issue if iterations have different amounts of work
- Dynamic
  - Assignment of threads determined at runtime (round robin)
  - Each thread gets more work after completing current work
  - Load balance is possible

Loop: Simple Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000
int main ()  {
  int i, chunk;
  float a[N], b[N], c[N];
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
  #pragma omp parallel shared(a,b,c,chunk) private(i)
  {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }  /* end of parallel section */
  return 0;
}
```

OpenMP Directives: Sections

- Non-iterative work-sharing construct
- Divide enclosed sections of code among threads
- Section directives nested within sections directive
- Syntax: C/C++ Fortran

```
#pragma omp sections
{
  #pragma omp section
  /* first section */
  #pragma omp section
  /* next section */
}
```

```
!$OMP sections

!$OMP section
C First section
!$OMP section
C Second section
!$OMP end sections
```

Sections: Simple Example

```
#include <omp.h>
#define N     1000
int main () {
  int i;
  double a[N], b[N],
   c[N], d[N];
  /* Some initializations
   */
  for (i=0; i < N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
  }
```

```
  #pragma omp parallel \
   shared(a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i < N; i++)
          c[i] = a[i] + b[i];
      #pragma omp section
        for (i=0; i < N; i++)
         d[i] = (a[i] * b[i]) + 0.2;
     }  /* end of sections */
   }  /* end of parallel section */
return 0;
}
```

OpenMP Directives: Synchronization

- Sometimes, need to make sure threads execute regions of code in proper order
  - Maybe one part depends on another part being completed
  - Maybe only one thread need execute a section of code
- Synchronization directives
  - Critical
  - Barrier
  - Single

OpenMP Directives: Synchronization

- **Critical**
  - Specifies section of code that must be executed by only one thread at a time
  - Syntax: C/C++

```
#pragma omp critical [name]
```

  Fortran

```
!$OMP critical [name]
!$OMP end critical
```

  - Names are global identifiers – critical regions with same name are treated as same region
- **Single**
  - Enclosed code is to be executed by only one thread
  - Useful for thread-unsafe sections of code (e.g., I/O)
  - Syntax: C/C++

```
#pragma omp single
```

  Fortran

```
!$OMP single
!$OMP end single
```

OpenMP Directives: Synchronization

- **Barrier**
  - Synchronizes all threads: thread reaches barrier and waits until all other threads have reached barrier, then resumes executing code following barrier
  - Syntax: C/C++ Fortran

```
#pragma omp barrier      !$OMP barrier
```

  - Sequence of work-sharing and barrier regions encountered must be the same for every thread

OpenMP Directives: Reduction

- **Reduces list of variables into one, using operator (e.g., max, sum, product, etc.)**
- **Syntax**

  C `#pragma omp reduction(`*`op`*` : `*`list`*`)`

  Fortran `!$OMP reduction(`*`op`*` : `*`list`*`)`

  where ***`list`*** is list of variables and ***`op`*** is one of following:

  - C/C++: **`+`**, **`-`**, **`*`**, **`&`**, **`^`**, **`|`**, **`&&`**, or **`||`**
  - Fortran: **`+`**, **`-`**, **`*`**, **`.and.`**, **`.or.`**, **`.eqv.`**, **`.neqv.`**, or **`max`**, **`min`**, **`iand`**, **`ior`**, **`ieor`**

III. Data (variable) Scope

- By default, all variables are shared except
  - Certain loop index values – private by default
  - Local variables and value parameters within subroutines called within parallel region – private
  - Variables declared within lexical extent of parallel region – private

Default Scope Example

```
void caller(int *a, int n) {
int i,j,m=3;
#pragma omp parallel for
for (i=0; i<n; i++) {
  int k=m;
  for (j=1; j<=5; j++) {
    callee(&a[i], &k, j);
  }
}
void callee(int *x, int *y, int z)
   {
  int ii;
  static int cnt;
  cnt++;
  for (ii=1; ii<z; ii++) {
    *x = *y + z;
  }
}
```

Var	Scope	Comment
a	shared	Declared outside parallel construct
n	shared	same
i	private	Parallel loop index
j	shared	Sequential loop index
m	shared	Declared outside parallel construct
k	private	Automatic variable/parallel region
x	private	Passed by value
*x	shared	(actually a)
y	private	Passed by value
*y	private	(actually k)
z	private	(actually j)
ii	private	Local stack variable in called function
cnt	shared	Declared static (like global)

Variable Scope

- Good programming practice: explicitly declare scope of all variables
- This helps you as programmer understand how variables are used in program
- Reduces chances of data race conditions or unexplained behavior

Variable Scope: Shared

- Syntax
  - `shared(`*`list`*`)`
- One instance of shared variable, and each thread can read or modify it
- WARNING: watch out for multiple threads simultaneously updating same variable, or one reading while another writes
- Example

```
#pragma omp parallel for shared(a)
for (i = 0; i < N; i++) {
  a[i] += i;
}
```

Variable Scope: Shared – Bad Example

```
#pragma omp parallel for shared(n_eq)
for (i = 0; i < N; i++) {
  if (a[i] == b[i]) {
    n_eq++;
  }
}
```

- `n_eq` will not be correctly updated
- Instead, put `n_eq++;` in critical block (slow);
  or
  introduce private variable `my_n_eq`, then update `n_eq` in critical block after loop (faster);
  or
  use `reduction` pragma (best)

Variable Scope: Private

- Syntax
  - `private(`*`list`*`)`
- Gives each thread its own copy of variable
- Example

```
#pragma omp parallel private(i, my_n_eq)
{
  #pragma omp for
  for (i = 0; i < N; i++) {
    if (a[i] == b[i])  my_n_eq++;
  }
  #pragma omp critical (update_sum)
  {
    n_eq+=my_n_eq;
  }
}
```

Best Solution for Sum

```
#pragma parallel for reduction(+:n_eq)
for (i = 0; i < N; i++) {
  if (a[i] == b[i]) {n_eq = n_eq+1;}
  }
```

IV. OpenMP Runtime Library Routines and Environment Variables

OpenMP Runtime Library Routines

```
C: void omp_set_num_threads(int num_threads)

Fortran:  subroutine omp_set_num_threads
  (  scalar_integer_expression )
```

- Sets number of threads used in next parallel region
- Must be called from serial portion of code

OpenMP Runtime Library Routines

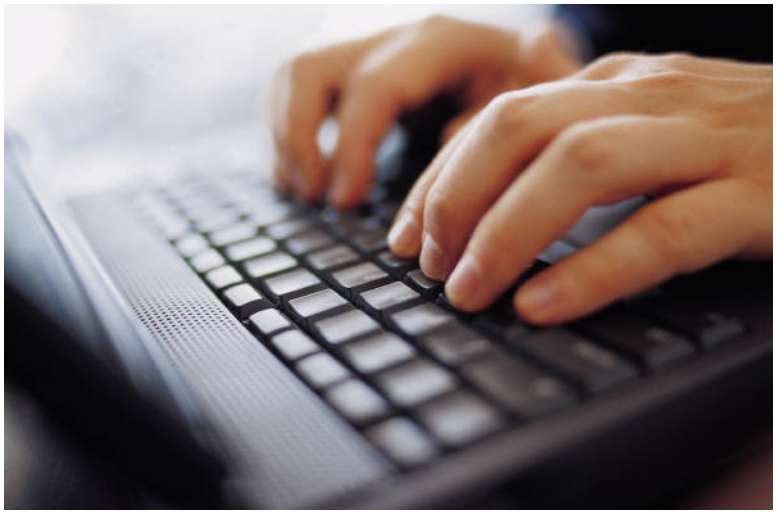
- `int omp_get_num_threads()`

  `integer function omp_get_num_threads()`
  - Returns number of threads currently in team executing parallel region from which it is called
- `int omp_get_thread_num()`

  `integer function omp_get_thread_num()`
  - Returns rank of thread
  - `0 ≤ omp_get_thread_num() < omp_get_num_threads()`

OpenMP Environment Variables

- Set environment variables to control execution of parallel code
- `OMP_SCHEDULE`
  - Determines how iterations of loops are scheduled
  - E.g., `setenv OMP_SCHEDULE "guided, 4"`
- `OMP_NUM_THREADS`
  - Sets maximum number of threads
  - E.g., `setenv OMP_NUM_THREADS 4`



V. USING OPENMP

Conditional Compilation

- Can write single source code for use with or without OpenMP
- Pragmas/sentinels are ignored
- What about OpenMP runtime library routines?
  - `_OPENMP` macro is defined if OpenMP available: can use this to conditionally include `omp.h` header file, else redefine runtime library routines

Conditional Compilation

```
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif
…
int me = omp_get_thread_num();
…
```

Compiling Programs with OpenMP Directives on Jaguar and Kraken

- Compiler flags:
  - `-mp=nonuma` (PGI)
  - `-fopenmp` (GNU)
  - `-mp` (Pathscale)
  - (default w/Cray compiler) `-xomp or -hnoomp` to deactivate
- Many libraries already compiled with OpenMP directives
- Libsci (default version 11.0.04, module xt-libsci)
  - link with `-lsci_istanbul_mp`

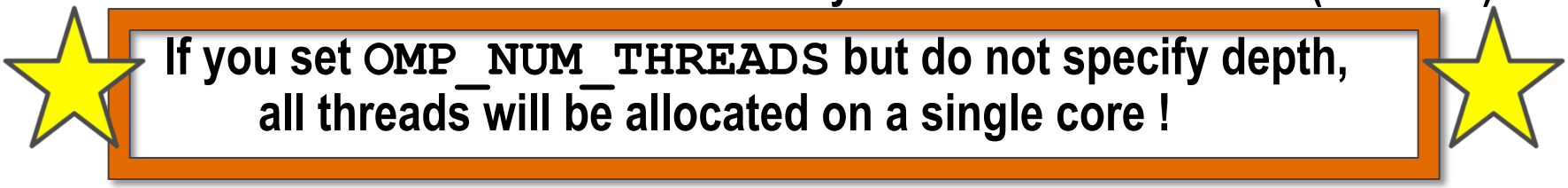
Running Programs with OpenMP Directives on Jaguar and Kraken

- Set environment variable `OMP_NUM_THREADS` in batch script
- Use the depth (`-d`) in aprun command to represent number of threads per MPI process, and `-N` for number of MPI processes per node, and –S to distribute MPI procs per Socket.
- Example: to run 8 MPI tasks, each with 6 threads on the hex-core nodes on Jaguar or Kraken, add the following to your script requesting 48 procs (and 4 compute nodes):

```
export OMP_NUM_THREADS=6

aprun -n 8 -S 1 -d 6 myprog.exe  (-N 2 not needed)

NOTE: size has to be multiple of 12 !!!
  #PBS -l size=48
```

More about aprun

- `-n pes`
  - Allocates *pes* processing elements (PEs, think MPI tasks)
- `-N pes_per_node / -S pes_per_socket`
  - Specifies number of processing elements to place per node / per socket
  - Reducing number of PEs per node makes more resources (i.e. memory!) available per PE
- `-d depth`
  - Allocates number of CPUs to be used by each PE and its threads (default 1)



- *pes * pes_per_socket * depth* ≤ "size" in PBS header!

Simple Example – performance issues !

```
#include <omp.h>
#define N 100000    /* or 1000000 */
int main () {
  int i, iter;   double a[N],b[N],c[N];
  for (iter=0; iter < 100000; iter++) {    /* or 10000 */
#pragma omp parallel for shared(a,b,c) private(i)
   for (i=0; i < N; i++) {
       a[i] = i * 1.5;  b[i] = i + 22.35;
       c[i] = a[i] * b[i]; }           }  }
```

	iter= 100,000 N=100,000	iter= 10,000 N=1,000,000	
no threads	33.79 secs	59.37 secs	
6 threads	5.35 secs	46.94 secs	(compile with mp=nonuma)
speedup	~6.3	~1.3	
Memory	~400KB/core	~4MB/core	

Where's the catch? *The catch is in the Cache!* *Mem= 3 * N * 8 bytes used*
Performance is limited by size of Cache memory (~1 Mbytes / core)

OPENMP Hello World – Fortran Code

```
PROGRAM HELLO
INTEGER   NTHREADS, TID, OMP_GET_NUM_THREADS,
INTEGER   OMP_GET_THREAD_NUM
C     Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL PRIVATE(NTHREADS, TID)
C     Obtain thread number
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID
C     Only master thread does this
      IF (TID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF
C     All threads join master thread and disband
!$OMP END PARALLEL
      END
```

OPENMP Hello World – C code

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  int nthreads, tid;
  /* Fork a team of threads giving them their own copies of variables */
  #pragma omp parallel private(nthreads, tid) {
     /* Obtain thread number */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
       nthreads = omp_get_num_threads();
       printf("Number of threads = %d\n", nthreads); }
  }
  /* All threads join master thread and disband */
}
```

Compile and run OpenMP on Kraken

```
%cc -mp  -fast omp_hello.c -o hello
%ftn –mp -fast omp_hello.f -o hello
%qsub submit.pbs
```

```
#!/bin/bash
#PBS -A UT-TNEDU002
#PBS -N test
#PBS -j oe
#PBS -l walltime=1:00:00,size=12
cd $PBS_O_WORKDIR
Date
export OMP_NUM_THREADS=4
aprun -n 1 –S1 –d4 ./hello
```

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 2
```

Improving Scientific Computing: the process

- 1.Write the program, or build it from previous codes, etc.
- 2. Debug your code (with optimization switches off)
- 3. Ensure mathematical correctness of the program!
- 4. Profile your code – determine where most of the computing time is spent
- 5. Optimize the algorithm, the data mapping, the communication, the I/O
- 6. Try out different combinations of compiler flags and/or compiler directives
- 7. Profile your code again
- 8. Re-examine blocks of code that consume the most execution time
- 9. Repeatedly apply various optimizations to such blocks
- 10. Rerun optimized code, compare performance, and start again until "satisfied".

Final thoughts: Strategies for Improved Performance

- Improving performance is a complex task, and the amount of time and effort put into it might not always be worth it.
- A certain trade-off must be reached between the developmental time and the "final" production run time.
- If you need to work on a previously existing code, then take the time to learn the details of its logic (if possible). Sometimes you might be better off rewriting the whole code directly in parallel!
- If you write the program from scratch, take some time to think about the different performance issues presented here and/or elsewhere.
- Examine benchmark results and know the limits of the computing platform

Finally: What else can be done?

- Practice, try new approaches, innovate, ask others
- Remember to concentrate only on subroutines worth improving
- Rethink the whole algorithm from scratch !?
- Remember to re-check the results for “correctness” (whenever possible!)
- Change parallel method (?), or change parallel machine (?)
- (ask someone else to do the calculations! :-)

Performance Evaluation of Codes

- Profiling of codes
  - Use profiling tools, e.g. prof, to identify segment of code that requires intensively computing
  - craypat on the XT5
- Timing codes
  - timers depend on machines platform
  - $ time *executable* ---> cpu time, elapsed time
  - etime, dtime, ctime, itime
  - should provide good timing resolution for section of codes
- MPI timer
  - MPI_Wtime() ---> return seconds of elapse wall-clock time
  - MPI_Wtick() ----> return value of seconds between successive clock ticks

Communication Characteristics

- Relatively slow communication vs. computation
  - Peak bandwidths: ~1 MB/sec w/ethernet connections
  - 12.5 MB/sec with a 100 Mbit/sec switch network
  - 150 MB/sec on the SP2
  - 9.6 GB/sec On the Cray XT5 between nodes
  - Implies advantage of using either coarse -or medium- grained parallelism
- The bigger communication cost is in the "startup" or latency
- overhead - 40 usec (software) latency on the SP2
  - sending separate 1-byte messages --> 1s/40us = 25 KBytes/sec !!
  - better sending few large messages rather than many small ones
  - Cray XT5 – latency : a few us
- Bottom line: try to minimize the ratio of
  - (# messages) / (# computations)

Communication Issues

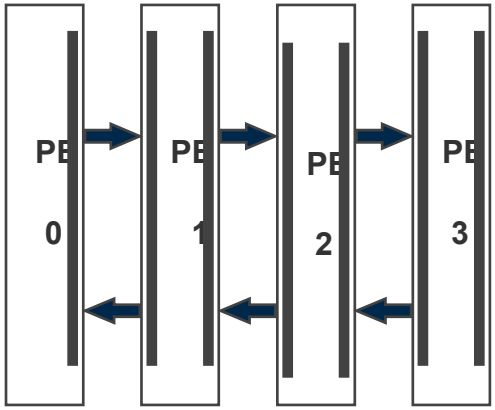
- **Contentions, or traffic jams**
  - **Have good distribution of messages. Circular or round-robin methods in one or two dimensions are fairly efficient for certain problems.**
  - **Avoid as much as possible the use of indirect addressing.**
  - **Use threads on multicore**
- **Ready mode in MPI or post receive before send**
  - **use MPI_Rsend when you are *sure* that a matching receive (MPI_Recv) has been posted appropriately**
  - **this allows faster transfer protocols**
  - **-HOWEVER! behavior is undefined if receive was not posted in time!**
  - **Post receive before send on Cray**
- **Mask communication with computation**
  - **Use asynchronous mode,**
  - **Avoid barrier**

I/O and Parallel I/O

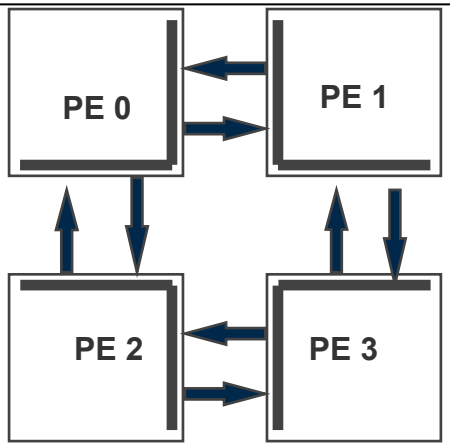
- **I/O can be a serious bottleneck for certain applications. The time to read/write data to disks could be an issue. But sometimes the shear size of the data file is a problem.**
- **Parallel I/O systems allow (in theory) the efficient manipulation of huge files**
- **Unfortunately, parallel I/O is only available on some architectures, and software is not always good. (MPI-2 has parallel MPI-IO on ROMIO implementation)**
- **They are restricted to few (around 4 or so) parallel disk drives, through designated I/O nodes.**
- **On the IBM with GPFS**
- **Lustre on the CRAY XT5**
- **One single files vs file/process**
- **Using local /tmp for input output**
- **Progress is still needed in this area!**

Mapping Problem

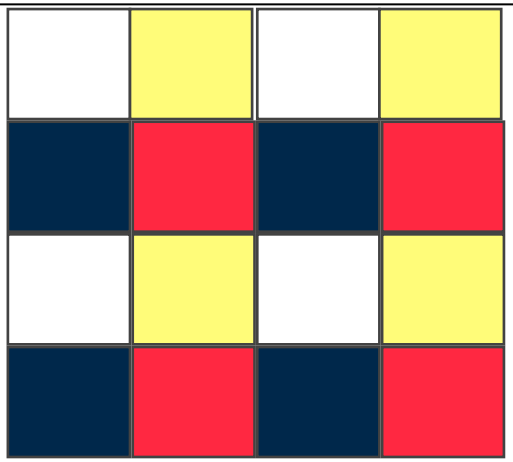
- Each processor should have a similar amount of work
- Expensive communications should be minimized.
- Communications should be:
  - eliminated where feasible
  - localized otherwise (i.e. communicate between close CPU neighbors) (not crucial anymore)
- Concurrency should be maximized
- NOTE: finding the best mapping is an NP-complete problem! :-(



1D Decomposition



2D Decomposition



2D Block Cyclic

Load Balancing

- **Static**
  - **Data or tasks are partitioned initially among the existing node processors**
  - **Problem: finding a good initial mapping of data or tasks to the processors**
- **Dynamic**
  - **Assumes there is a pool of tasks which can be selected and distributed at runtime (e.g. a task queue or bag_of_tasks)**
  - **Next available task is assigned to a free processor**
  - **Or, it implies that the data can be redistributed appropriately during execution of program**
  - **Problem: Synchronization issues**

Strategies for Improved Performance

- **Improving performance is a complex task, and the amount of time and effort put into it might not always be worth it.**
- **A certain trade-off must be reached between the developmental time and the "final" production run time.**
- **If you need to work on a previously existing code, then take the time to learn the details of its logic (if possible). Sometimes you might be better off rewriting the whole code directly in parallel!**
- **If you write the program from scratch, take some time to think about the different performance issues that we have been presenting here.**
- **Examine benchmark results and know the limit of the computing platform**
- **profilers "prof" give information on:**
  - **how much time (seconds) is spent in each subroutine**
  - **what percentage of time each subroutine is consuming**
  - **the cummulative time**
  - **the # of calls to subroutines made**
  - **the time (msecs) per call**
  - **Use available system tools**

Performance Tuning Process

- 1. Debug your code (with optimization switches off)
- 2. Ensure mathematical correctness of the program!
- 3. Profile your code
- 4. Optimize the algorithm
- 5. Compile with optimization switches on
- 6. Profile your code
- 7. Examine blocks of code that consume the most execution time
- 8. Repeatedly apply various optimizations to such blocks
- 9. Ensure again the numerical correctness of the program!

- **Finally: What else can be done?**
  - Practice, try new approaches, innovate, ask others
  - Concentrate only on subroutines worth improving
  - Rethink the whole algorithm from scratch !?
  - Re-check the results for correctness (whenever possible!)
  - Change parallel method (?)
  - Change parallel machine (?)
  - (ask someone else to do it! ;-)

Writing Parallel Programs

- Use prewritten programs
  - There are parallel database codes, genetic algorithms, neural networks, linear algebra, etc available
- Writing code to take advantage of parallel libraries
  - Use libraries like ScaLAPACK (Scalable Linear Algebra Package), and other optimized parallel libraries in your code
  - Usually much faster and more robust than code you could easily write
- Writing your own code from scratch
  - The hardest choice… but used by many because of its flexibility

Acknowledgments

- JICS staff, GRAs, collaborators, and previous workshop participants
- NCCS and NICS staff at UTK/ORNL and their resources
- All contributors to the art and science of parallel computing

Hands-on Exercises information

0.- Activate your guest account by following the included instructions (on paper) CAREFULLY! If you can't login to Kraken, ask for help.

1.- Go to www.jics.utk.edu, select 'JICS Fall Training Workshop" on the left. (This takes you to: http://www.nics.utk.edu/mpi-tutorial; omp-tutorial)

2.- Follow instructions, do as many exercises as you can, and ask for help as needed, or check out the info on the Kraken webpages starting at http://www.nics.utk.edu/computing-resources/kraken

Check also the FAQ at http://www.nics.utk.edu/faq

or http://www.nics.utk.edu/faq-search

3.- If you get stuck later on, consider sending email to help@xsede.org and be sure to include the word "Kraken" in the subject line. A phone call to the NICS User Support team at Tel:**1.865.241.1504** is another option but only during office hours - 9:00 am - 6:00 pm ET

4.- Please return your fob/token by. Thanks!

5.- Fill out survey

Acknowledgements

- NSF, DOE, Cray, NCCS, NICS, XSEDE staff, students, contributors
- many helpful hands and collaborators
- A lot of materials are reproduced from that available on the web

Performance Measurement on kraken using fpmpi and craypat

Kwai Wong

NICS at UTK / ORNL

kwong@utk.edu

fpmpi_papi

- fpmpi_papi are light-weight profiling libraries that use the papi hooks
- fpmpi is a MPI profiling library. It intercepts MPI library calls via the MPI profiling interface and reports CPU times, memory usage, performance counters, and MPI calls statistics.
- All profiling output is written by process rank 0. The default output file name is profile.txt but produce output only when program complete successfully
- Good for performance modeling
    1. module load fpmpi_papi
    2. relink with $FPMPI_LDFLAGS
    3. export MPI_HWPC_COUNTERS=PAPI_FP_OPS
    4. submit job and check for profile.txt

Example – HPL

```
module load fpmpi_papi
cd ~/hpl-2.0/testing/ptest/Cray
cc  -o ~/hpl-2.0/bin/Cray/xhpl  *.o  ~/hpl-2.0/lib/Cray/libhpl.a  $FPMPI_LDFLAGS
```

```
MPI_init     called at: Tue Mar 23 23:20:12 2010
MPI_Finalize called at: Tue Mar 23 23:20:13 2010

Number of processors:   24

---- Process Stats                min         max         avg         imbl      ranks
Wall Clock Time (sec)      1.725063    1.725897    1.725486          0%        21/13
User CPU Time   (sec)      1.680105    1.752109    1.711773          4%         7/5
System CPU Time (sec)      0.112007    0.184011    0.147509          39%        5/7
I/O Read Time              0.000000    0.022161    0.000923          100%       1/0
I/O Write Time             0.000000    0.000131    0.000005          100%       1/0
MPI Comm Time   (sec)      0.206312    0.279243    0.244430          26%        0/23
MPI Sync Time   (sec)      0.030649    0.052847    0.040223          42%        12/1
MPI Calls                  6794        7387        7070              8%         17/0
Total MPI Bytes            112652484   123302568   117878459         9%         21/14
Memory Usage    (MB)       72.097656   72.339844   72.208659         0%         3/12

---- Performance Counters      min         max         avg imbl ranks
OPS_ADD:OPS_ADD_PIPE_LO1.484772e+09 1.737513e+09 1.605853e+09     15% 21/14
OPS_MULTIPLY:OPS_MULTIP1.522243e+09 1.781054e+09 1.646220e+09     15% 21/14
OPS_STORE:OPS_STORE_PIP1.510631e+09 1.767395e+09 1.633547e+09     15% 21/14
PACKED_SSE_AND_SSE2    4.483819e+09 5.248205e+09 4.849793e+09     15% 21/14
```

Example – HPL – profile.txt

```
---- Barriers and Waits        min         max         avg imbl ranks
MPI_Wait
  Number of Calls             2130        2302        2215   7% 4/2
  Communication Time       0.031781    0.058928    0.042610  46% 16/1

---- Message Routines          min         max         avg imbl ranks
MPI_Send           size 5-8B
  Number of Calls                3          40          11  93% 7/0
  Total Bytes                   24         320          93  93% 7/0
  Communication Time       0.000001    0.000090    0.000017  99% 23/8

 MPI_Irecv          size 65-128KB
   Number of Calls              90          90          90   0% 0/0
   Total Bytes             9216000     9230400     9218400   0% 0/3
   Communication Time       0.004478    0.007619    0.005693  41% 18/15

 ---- Number Of Comm Partners    min         max         avg imbl ranks
 MPI_Send                        4           7         5.8  43% 1/0
 ……..
  ---- Performance Counters       min         max         avg            imbl        ranks
  PAPI_FP_OPS          5.822562e+09 6.817514e+09 6.298254e+09  15%     21/14
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR00L2L2       6000    80     4     6               1.45                 9.927e+01
```

Example – All to All

Number of processors: 16384

---- Process Stats	min	max	avg	imbl	ranks
Wall Clock Time (sec)	140.071693	140.365403	140.184043	0%	11775/12336
User CPU Time (sec)	113.423088	131.980248	122.924940	14%	568/16380
System CPU Time (sec)	4.996312	12.000750	9.787787	58%	16380/12904
I/O Read Time	0.000099	5.188349	1.567495	100%	2791/15420
I/O Write Time	0.000022	0.021117	0.000260	100%	4512/0
MPI Comm Time (sec)	10.663487	70.324075	51.573792	85%	10604/5208
MPI Sync Time (sec)	13.696530	102.137746	38.788597	87%	5208/16380
MPI Calls	360	375	360	4%	1/0
Total MPI Bytes	2623598656	2814971968	2624983552	7%	12/0
Memory Usage (MB)	239.425781	281.371094	261.600875	15%	16380/0

Number of processors: 24576

---- Process Stats	min	max	avg	imbl	ranks
Wall Clock Time (sec)	121.928648	122.535041	122.140123	0%	6655/23052
User CPU Time (sec)	94.485905	111.414963	102.176574	15%	11568/17396
System CPU Time (sec)	3.740233	7.468466	5.936611	50%	18584/1839
I/O Read Time	0.000094	0.283022	0.017945	100%	2345/10472
I/O Write Time	0.000019	0.016692	0.000251	100%	495/0
MPI Comm Time (sec)	22.307244	66.630102	53.792108	67%	23319/15917
MPI Sync Time (sec)	10.785100	79.608090	33.492318	86%	0/23224
MPI Calls	384	449	423	14%	16408/0
Total MPI Bytes	1642116160	2018157952	1752448126	19%	16408/0
Memory Usage (MB)	196.769531	288.710938	236.159241	32%	17304/0

Examples -

Number of processors: 49152

---- Process Stats	min	max	avg	imbl	ranks
Wall Clock Time (sec)	114.937987	117.026509	115.410693	2%	34302/0
User CPU Time (sec)	64.752046	109.842864	98.226364	41%	0/42928
System CPU Time (sec)	1.896118	4.060253	3.221788	53%	42800/995
I/O Read Time	0.000091	0.614506	0.037972	100%	15438/47781
I/O Write Time	0.000018	0.018237	0.000246	100%	43680/46102
MPI Comm Time (sec)	17.175345	46.526769	35.308215	63%	43183/32604
MPI Sync Time (sec)	10.834225	83.356593	54.444489	87%	0/35376
MPI Calls	384	471	422	18%	32784/0
Total MPI Bytes	824226880	1138606464	879396521	28%	32784/0
Memory Usage (MB)	231.203125	277.410156	249.784748	17%	33072/0

Number of processors: 98304

---- Process Stats	min	max	avg	imbl	ranks
Wall Clock Time (sec)	226.579599	233.894804	227.883962	3%	33443/21516
User CPU Time (sec)	59.003687	223.429963	149.935103	74%	45336/45806
System CPU Time (sec)	1.776111	3.612225	2.850374	51%	95904/46854
I/O Read Time	0.000091	152.878905	62.173218	100%	27487/84253
I/O Write Time	0.000018	75.024583	0.000945	100%	2232/0
MPI Comm Time (sec)	14.810977	42.077644	28.898958	65%	77471/36102
MPI Sync Time (sec)	10.483419	164.570942	90.340724	94%	33797/76896
MPI Calls	384	507	422	24%	65567/0
Total MPI Bytes	415282240	698830720	442869914	41%	65567/0
Memory Usage (MB)	352.695312	375.527344	361.504875	6%	65793/0

Examples –

MPI_Alltoallv	size 2-4MB – 16K cores				
Number of Calls	96	96	96	0%	0/0
Total Bytes	402653184	402653184	402653184	0%	0/0
Communication Time	1.262350	3.943996	2.467818	68%	16379/15762
Synchronization Time	0.155048	6.915146	1.380552	98%	16373/15919

MPI_Alltoallv	size 1-2MB - 24K cores				
Number of Calls	120	132	128	9%	16384/0
Total Bytes	251658240	276824064	268435456	9%	16384/0
Communication Time	3.522625	4.787197	4.164852	26%	18575/11194
Synchronization Time	0.320081	2.376628	0.941022	87%	17072/7840

MPI_Alltoallv	size 2-4MB - 48K cores				
Number of Calls	120	132	128	9%	32768/0
Total Bytes	314572800	346030080	335544320	9%	32768/0
Communication Time	6.530760	9.641365	7.591697	32%	37857/32586
Synchronization Time	0.290247	16.706575	5.740633	98%	48447/12688

MPI_Alltoallv	size 1-2MB - 96K cores				
Number of Calls	120	132	128	9%	65536/0
Total Bytes	157286400	173015040	167772160	9%	65536/0
Communication Time	5.828970	11.787860	6.740117	51%	76707/36144
Synchronization Time	0.173055	12.106570	3.967286	99%	96258/19039

Examples – I/O

Number of processors: 9600

---- Process Stats	min	max	avg	imbl	ranks
Wall Clock Time (sec)	13.765523	101.447126	13.784399	86%	4540/0
User CPU Time (sec)	35.598224	101.778360	101.700702	65%	0/2443
System CPU Time (sec)	0.024001	9.900618	0.080618	100%	2443/0
I/O Read Time	0.000000	0.000000	0.000000	0%	0/0
I/O Write Time	0.000000	0.000028	0.000000	100%	1/0
MPI Comm Time (sec)	0.000187	4.446975	0.000907	100%	2/0
MPI Sync Time (sec)	1.259814	13.849340	13.765041	91%	0/8580
MPI Calls	9602	28798	9603	67%	1/0
Total MPI Bytes	480020	4607711980	959939	100%	1/0
Memory Usage (MB)	73.261719	73.726562	73.328657	1%	32/0

Number of processors: 9600

---- Process Stats	min	max	avg	imbl	ranks
Wall Clock Time (sec)	63.538385	63.720123	63.555581	0%	6824/6576
User CPU Time (sec)	33.662103	63.867991	62.392332	47%	0/1537
System CPU Time (sec)	0.028001	0.376023	0.081416	93%	5967/288
I/O Read Time	0.000000	0.000000	0.000000	0%	0/0
I/O Write Time	0.000000	0.005476	0.000001	100%	1/0
MPI Comm Time (sec)	0.000463	0.000954	0.000740	51%	2/7579
MPI Sync Time (sec)	0.008428	1.007395	0.561321	99%	2871/1976
MPI Calls	6	6	6	0%	0/0
Total MPI Bytes	252	252	252	0%	0/0
Memory Usage (MB)	73.183594	74.988281	73.367166	2%	2147/0

CRAYPAT

- pat_build: Utility for application instrumentation
  - No source code modification required
- run-time library for measurements
  - transparent to the user
- pat_report:
  - Performance reports, Performance visualization file

- Important performance statistics:
  - Top time consuming routines
  - Load balance across computing resources
  - Communication overhead
  - Cache utilization
  - FLOPS
  - Vectorization (SSE instructions)
  - Ratio of computation versus communication

Automatic Profiling Analysis (APA)

1. Load CrayPat & Cray Apprentice^2 module files
   - % module load xt-craypat /5.0.1 apprentice2/5.0.1
2. Build application
   - % make clean
   - % make, NEED Object files
3. Instrument application for automatic profiling analysis
   - % pat_build –O apa a.out,
   - % pat_build –Drtenv=PAT_RT_HWPC=1 –g mpi,heap,io,blas a.out
     - You should get an instrumented program a.out+pat
4. Run application to get top time consuming routines
   - Remember to modify <script> to run *a.out+pat*
   - Remember to run on Lustre
   - % aprun … *a.out+pat* *(or* qsub <pat script>)
     - You should get a performance file ("<sdatafile>.xf") or multiple files in a directory <sdatadir>
5. Generate .apa file
   - % pat_report –o *my_sampling_report* [<sdatafile>.xf | <sdatadir>]
     - creates a report file & an automatic profile analysis file <apafile>.apa

APA

6. Look at <apafile>.apa file

   Verify if additional instrumentation is wanted

7. Instrument application for further analysis (a.out+apa)

   % pat_build –O <apafile>.apa

   - You should get an instrumented program a.out+apa

8. Run application

   Remember to modify <script> to run *a.out+apa*

   % aprun … *a.out+apa* *(or* qsub <apa script>)

   - You should get a performance file ("<datafile>.xf") or multiple files in a directory <datadir>

9. Create text report

   % pat_report –o *my_text_report.txt* [<datafile>.xf | <datadir>]

   - Will generate a compressed performance file (<datafile>.ap2)

10. View results in text (my_text_report.txt) and/or with Cray Apprentice^2

    % app2 <*datafile*>.ap2

Example - HPL

```
#  You can edit this file, if desired, and use it
#  to reinstrument the program for tracing like this:
#
#          pat_build -O xhpl+pat+27453-16285sdt.apa

#       HWPC group to collect by default.

  -Drtenv=PAT_RT_HWPC=1  # Summary with TLB metrics.

#       Libraries to trace.

  -g mpi,io,blas,math
```

• blas	Linear Algebra
• heap	dynamic heap
• io	stdio and sysio
• lapack	Linear Algebra
• math	ANSI math
• mpi	MPI
• omp	OpenMP API
• omp-rtl	OpenMP runtime library
• pthreads	POSIX threads
• shmem	SHMEM

```
Instrumented with:
  pat_build  -Drtenv=PAT_RT_HWPC=1 -g mpi,io,blas,math -w -o \
        xhpl+apa /lustre/scratch/kwong/HPL/hpl-2.0/bin/Cray/xhpl
```

```
Runtime environment variables:
  MPICHBASEDIR=/opt/cray/mpt/4.0.1/xt
  PAT_RT_HWPC=1
  MPICH_DIR=/opt/cray/mpt/4.0.1/xt/seastar/mpich2-pgi
Report time environment variables:
  CRAYPAT_ROOT=/opt/xt-tools/craypat/5.0.1/cpatx
Operating system:
  Linux 2.6.16.60-0.39_1.0102.4787.2.2.41-cnl #1 SMP Thu Nov 12 17:58:04 CST 2009
```

Table 1: Profile by Function Group and Function

Time %	Time	Imb. Time	Imb.	Calls	Group Function PE='HIDE'
100.0%	1.889359	--	--	313589.5	Total
39.9%	0.753343	--	--	7830.5	BLAS
36.8%	0.695245	0.054348	7.6%	1047.5	dgemm_
2.4%	0.045313	0.003840	8.2%	1047.5	dtrsm_
0.5%	0.008820	0.000541	6.0%	524.3	dgemv_
0.1%	0.001979	0.000800	30.0%	2714.9	dcopy_
31.4%	0.593267	--	--	305749.0	MPI
18.8%	0.355428	0.108507	24.4%	211.3	MPI_Recv
4.6%	0.087427	0.008492	9.2%	292301.0	MPI_Iprobe
4.6%	0.087252	0.011519	12.2%	2427.2	MPI_Send
28.3%	0.534801	--	--	2.0	USER
28.3%	0.534736	0.018405	3.5%	1.0	main
0.0%	0.000065	0.000001	2.2%	1.0	exit
0.4%	0.007949	--	--	8.0	IO
0.4%	0.007241	0.166543	100.0%	1.3	fgets
0.0%	0.000000	0.000003	100.0%	0.0	MATH

Totals for program

Time%		100.0%
Time		1.889359 secs
Imb.Time		-- secs
Imb.Time%		--
Calls	0.166M/sec	313589.5 calls
PAPI_L1_DCM	16.023M/sec	30281456 misses
PAPI_TLB_DM	0.284M/sec	536336 misses
PAPI_L1_DCA	1688.295M/sec	3190654379 refs
PAPI_FP_OPS	3332.643M/sec	6298255922 ops
User time (approx)	1.890 secs	4913657415 cycles 100.0%Time
Average Time per Call	0.000006 sec	
CrayPat Overhead : Time	11.4%	
HW FP Ops / User time	3332.643M/sec	6298255922 ops 32.0%peak(DP)
HW FP Ops / WCT	3332.643M/sec	
Computational intensity	1.28 ops/cycle	1.97 ops/ref
MFLOPS (aggregate)	79983.43M/sec	
TLB utilization	5948.98 refs/miss	11.619 avg uses
D1 cache hit,miss ratios	99.1% hits	0.9% misses
D1 cache utilization (misses)	105.37 refs/miss	13.171 avg hits

==========

BLAS

Time	0.753343 secs	
HW FP Ops / User time	8309.779M/sec	6265144313 ops 79.9%peak(DP)

==========

BLAS / dgemm_

HW FP Ops / User time	8651.361M/sec	6017797050 ops 83.2%peak(DP)

Hardware counter

```
==========================================================================
USER / mlwxyz_
--------------------------------------------------------------------------
  Time%                                                  43.3%
  Time                                                5.392446
  Imb.Time                                            0.041565
  Imb.Time%                                              0.8%
  Calls                                                    10
  PAPI_L1_DCM                 19.103M/sec           102483839 misses
  PAPI_TOT_INS               275.051M/sec          1475564421 instr
  PAPI_L1_DCA                170.207M/sec           913107237 refs
  PAPI_FP_OPS                393.744M/sec          2112315940 ops
  User time (approx)           5.365 secs          11265843750 cycles
  Average Time per Call                               0.539245 sec/call
  Cycles                       5.365 secs          11265843750 cycles
  User time (approx)           5.365 secs          11265843750 cycles
  Utilization rate                                       99.5%
  Instr per cycle                                         0.13 inst/cycle
  HW FP Ops / Cycles                                      0.19 ops/cycle
  HW FP Ops / User time      393.744M/sec          2112315940 ops   4.7%peak
  HW FP Ops / WCT            391.718M/sec
  HW FP Ops / Inst                                      143.2%
  Computation intensity                                   2.31 ops/ref
  MIPS                      8801.64M/sec
  MFLOPS                   12599.82M/sec
  Instructions per LD ST                                  1.62 inst/ref
  LD & ST per D1 miss                                     8.91 refs/miss
  D1 cache hit ratio                                     88.8%
  LD ST per Instructions                                 61.9%
```

PAT_RT_HWPC=0
Flat profile data Hard counts
Derived metrics

Table 4: MPI Message Stats by Caller

MPI Msg Bytes	MPI Msg Count	MsgSz <16B Count	16B<= MsgSz <256B Count	256B<= MsgSz <4KB Count	4KB<= MsgSz <64KB Count	64KB<= MsgSz <1MB Count	Function Caller PE[mmm]
58939233.0	2427.2	16.5	2.0	2029.0	159.1	220.6	Total
58939233.0	2427.2	16.5	2.0	2029.0	159.1	220.6	MPI_Send
32840500.0	62.5	--	--	--	2.1	60.4	HPL_bcast_1ring
							HPL_bcast
							HPL_pdgesv
							HPL_pdtest
							main
34396024.0	63.0	--	--	--	1.0	62.0	pe.16
32757624.0	63.0	--	--	--	3.0	60.0	pe.4
31169776.0	62.0	--	--	--	3.0	59.0	pe.23

Table 3: Load Balance with MPI Message Stats

Time %	Time	MPI Msg Count	MPI Msg Bytes	Avg MPI Msg Size	Group PE[mmm]
100.0%	2.104323	2427.2	58939230.0	24283.14	Total
38.2%	0.802853	2427.2	58939230.0	24283.14	MPI
1.9%	0.942874	2332.0	56356080.0	24166.42	pe.21
1.6%	0.795338	2523.0	61786964.0	24489.48	pe.14
1.1%	0.547410	2552.0	58963356.0	23104.76	pe.0

Table 8: **Program Wall Clock Time, Memory High Water Mark**

Process Time	Process HiMem (MBytes)	PE[mmm]
2.951542	102	Total
3.061611	103.203	pe.14
2.945643	103.098	pe.6
2.852264	102.188	pe.11

Example – All to All – large size run

- Need to work around to obtain apa report for 96K core job
- Generate apa file using a job of smaller number of cores
- Modify the xxx.apa file to get the right exec+apa
- Not able to complete all .xf files instrumentation for 96K cores because of memory limitation on the login node
- Use only a subset of the .xf files
  - remove old xxx.ap2 file
  - pat_report –o report-apa.txt xxxx+apa+xx.sdt/*1.xf

Example – All to All

```
 100.0% | 131.569738 |       -- |     -- | 2428.1 |Total
|-----------------------------------------------------------
|  43.8% |  57.653286 |       -- |     -- | 1637.3 |MPI
||----------------------------------------------------------
||  22.0% |  28.971955 |  9.770085 |  25.2% |   23.2 | alltoall
||  19.6% |  25.839457 |  4.365176 |  14.5% |  268.0 alltoallv
||   1.8% |   2.324442 |  1.579387 |  40.5% |    1.0 |cart_create
```

```
||===========================================
|  31.5% |  41.506984 |       -- |     -- |  423.2 |MPI_SYNC
||-----------------------------------------------------------
||  10.3% |  13.495955 |  3.961412 |  22.7% |   34.0 bcast
||   7.1% |   9.305306 | 20.992674 |  69.3% |  268.0 alltoallv
||   5.4% |   7.064162 |  7.722861 |  52.2% |   23.2 alltoall
||   5.1% |   6.696518 | 16.221001 |  70.8% |   73.0 reduce
```

```
|  24.6% |  32.409469 |       -- |     -- |  367.6 |USER  - 24K cores
```

```
 HW FP Ops / User time       228.772M/sec   30099476860 ops   2.2%peak(DP)
Computational intensity       0.09 ops/cycle     0.19 ops/ref
```

```
 100.0% | 109.492817 |       -- |     -- | 2425.7 |Total
|-----------------------------------------------------------
|  43.3% |  47.457776 |       -- |     -- | 1635.4 |MPI
||----------------------------------------------------------
||  18.1% |  19.797156 |  5.109597 |  20.5% |  268.0 alltoallv
||  15.4% |  16.822697 |  7.587488 |  31.1% |   22.9  alltoall
||   9.1% |   9.958060 |  8.447915 |  45.9% |    1.0 cart_create
```

```
||===========================================
|  31.5% |  34.473680 |       -- |     -- |  422.9 |MPI_SYNC
||-----------------------------------------------------------
||  12.5% |  13.671121 |  6.116382 |  30.9% |   34.0 bcast
||   5.5% |   6.044712 | 12.697817 |  67.8% |  268.0 alltoallv
||   5.1% |   5.610174 | 10.240988 |  64.6% |   73.0 reduce
||   4.8% |   5.278417 |  4.210131 |  44.4% |   22.9  alltoall
```

```
|  25.2% |  27.561360 |       -- |     -- |  367.4 |USER  - 48K cores
```

```
 HW FP Ops / User time       142.486M/sec   15601173217 ops   1.4%peak(DP)
 Computational intensity       0.05 ops/cycle     0.13 ops/ref
```

```
 100.0% | 104.935954 |       -- |     -- | 2422.7 |Total
|-----------------------------------------------------------
|  41.9% |  43.967002 |       -- |     -- | 1633.2 |MPI
||----------------------------------------------------------
||  16.0% |  16.779321 |  5.607865 |  25.1% |  267.8 |alltoallv
||  13.6% |  14.303518 |  7.110796 |  33.2% |   22.7 |alltoall
||  10.9% |  11.415859 |  6.197070 |  35.2% |    1.0 | cart_create
```

```
||===========================================
|  36.4% |  38.245235 |       -- |     -- |  422.4 |MPI_SYNC
||-----------------------------------------------------------
||  15.8% |  16.597880 |  9.375426 |  36.1% |   34.0 bcast
||   6.5% |   6.782135 |  9.459139 |  58.2% |   73.0  reduce
||   6.0% |   6.321144 |  3.127890 |  33.1% |   22.7  alltoall
||   4.6% |   4.811081 |  9.816945 |  67.1% |  267.8  alltoallv
```

```
|  21.7% |  22.723717 |       -- |     -- |  367.1 |USER – 96K cores
```

```
HW FP Ops / User time       78.995M/sec    8289391712 ops   0.8%peak(DP)
Computational intensity       0.03 ops/cycle     0.07 ops/ref
```

Example – IO
4800 cores

```
 100.0% | 42.480949 |       -- |     -- | 9615.7 |Total
|---------------------------------------------------------------
|  90.0% | 38.225091 |  1.627372 |   4.1% | 4801.0 |
MPI_SYNC
|       |          |           |        |        | mpi_barrier_(sync)
|  10.0% |  4.241510 |       -- |     -- | 4811.0 |MPI
||--------------------------------------------------------------
||  10.0% |  4.238886 |  0.247876 |   5.5% | 4801.0 |barrier
||   0.0% |  0.001046 |  0.000141 |  11.9% |    1.0 |finalize
||   0.0% |  0.000786 |  0.010741 |  93.2% |    2.0 |send
||   0.0% |  0.000784 |  3.764192 | 100.0% |    2.0 |recv
||
==============================================
|   0.0% |  0.013216 |       -- |     -- |    2.0 |USER
||--------------------------------------------------------------
||   0.0% |  0.013142 | 20.825449 | 100.0% |    1.0 |main
||   0.0% |  0.000074 |  0.000028 |  27.4% |    1.0 |exit
||
==============================================
|   0.0% |  0.001132 |       -- |     -- |    1.7 |IO
||--------------------------------------------------------------
||   0.0% |  0.000726 |  3.485694 | 100.0% |    0.3 |write
||   0.0% |  0.000397 |  1.905213 | 100.0% |    0.0 |open
||   0.0% |  0.000006 |  0.030968 | 100.0% |    0.3 |read
||   0.0% |  0.000002 |  0.008761 | 100.0% |    1.1 |lseek
```

```
 100.0% | 34.977625 |       -- |     -- |  198.6 |Total
|---------------------------------------------------------------
|  98.2% | 34.350577 |       -- |     -- |   31.0 |MPI
||--------------------------------------------------------------
||  64.5% | 22.569258 |  0.080680 |   0.4% |    1.0 File_write_all
||  31.3% | 10.931775 |  3.420038 |  23.8% |    1.0 |File_open
||   2.4% |  0.846223 |  0.000072 |   0.0% |    1.0 |File_set_size
||   0.0% |  0.001478 |  0.000163 |   9.9% |    3.0 |Allreduce
||   0.0% |  0.000455 |  0.000112 |  19.8% |    2.0 |File_set_view
||   0.0% |  0.000388 |  0.000093 |  19.4% |    1.0 finalize_
||   0.0% |  0.000335 |  0.000225 |  40.2% |    1.0 File_close
||   0.0% |  0.000262 |  0.000107 |  28.9% |    3.0 |Bcast
||   0.0% |  0.000192 |  0.000044 |  18.7% |    1.0 |barrier_
||   0.0% |  0.000117 |  0.000080 |  40.6% |    1.0 |Barrier
```

```
||==============================================
|   1.0% |  0.344032 |       -- |     -- |    8.0 |MPI_SYNC
||--------------------------------------------------------------
||   0.5% |  0.185885 |  0.006262 |   3.3% |    1.0 barrier
||   0.4% |  0.157168 |  0.152297 |  49.2% |    1.0 Barrier
||   0.0% |  0.000513 |  0.000117 |  18.6% |    3.0 Bcast
||   0.0% |  0.000466 |  0.001049 |  69.3% |    3.0 Allreduce
||==============================================
|   0.8% |  0.279088 |       -- |     -- |  157.6 |IO
||--------------------------------------------------------------
||   0.6% |  0.192985 |  0.157094 |  44.9% |    1.0 |fsync
||   0.2% |  0.080129 | 19.995701 |  99.6% |  150.6 |write
||   0.0% |  0.005964 |  0.186089 |  96.9% |    1.0 |close
|   0.0% |  0.003928 |       -- |     -- |    2.0 |USER
||--------------------------------------------------------------
||   0.0% |  0.003815 |  0.000509 |  11.8% |    1.0 |main
||   0.0% |  0.000113 |  0.000168 |  59.7% |    1.0 |exit
|==============================================
```

Example : I/O code 4800 cores

Table 7: File Output Stats by Filename – Serial IO

```
   Write |    Write MB | Write Rate |      Writes |      Write |Experiment=1
    Time |             |     MB/sec |             |     B/Call |File Name
 |       |             |            |             | PE='HIDE'

 3.485947 | 2747.009743 | 788.023918 | 1377.000000 | 2091828.97 |Total
|-------------------------------------------------------------------------
| 3.485585 | 2747.009727 | 788.105799 | 1375.000000 | 2094871.62 |./TEST2D.netcomref000000
| 0.000362 |    0.000015 |   0.042129 |    2.000000 |       8.00 |stdout
|=========================================================================
```

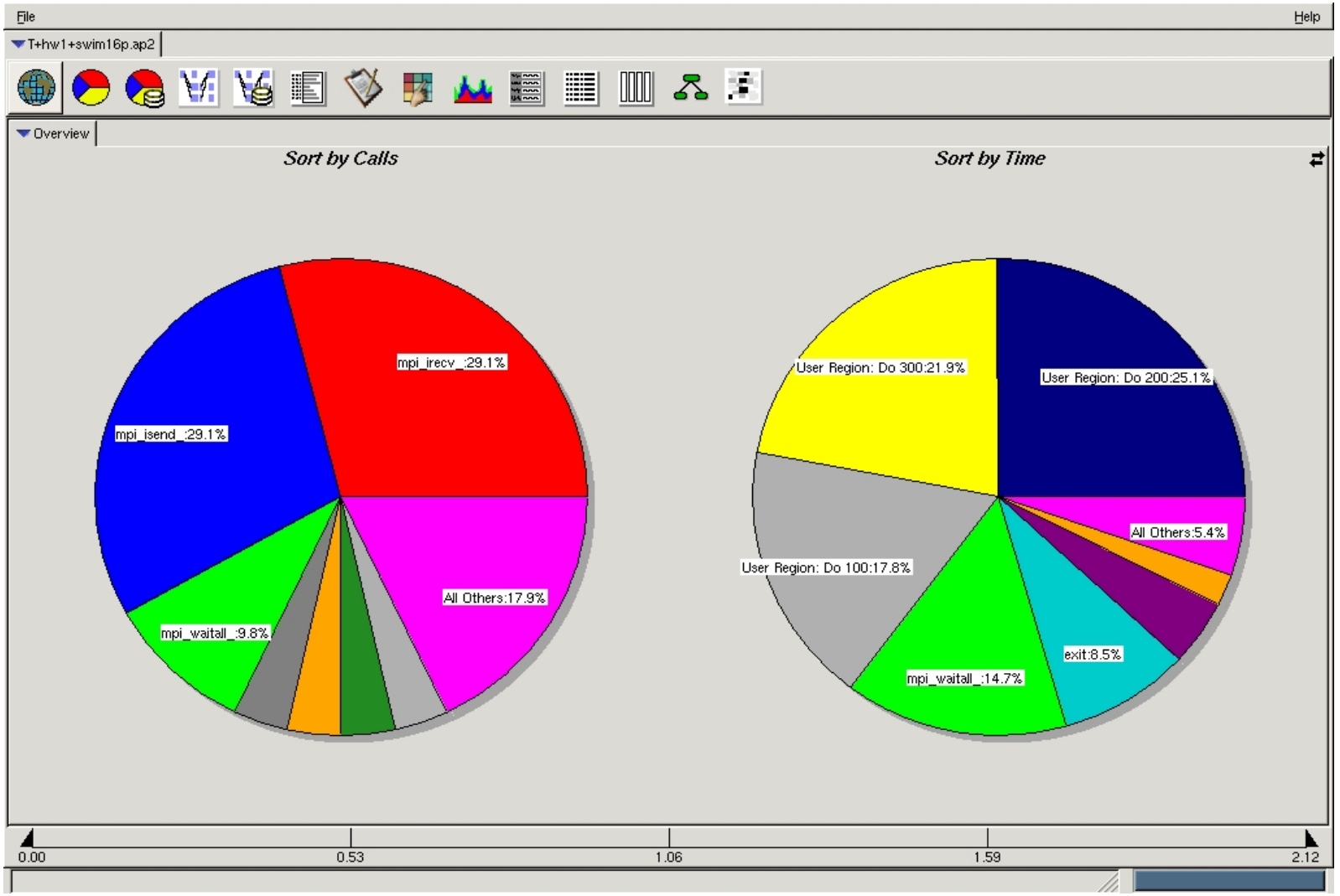
lfs setstripe –c –1 fdir/file – use MPIIO – set stripe count to 160

Table 7: File Output Stats by Filename -- MPIIO

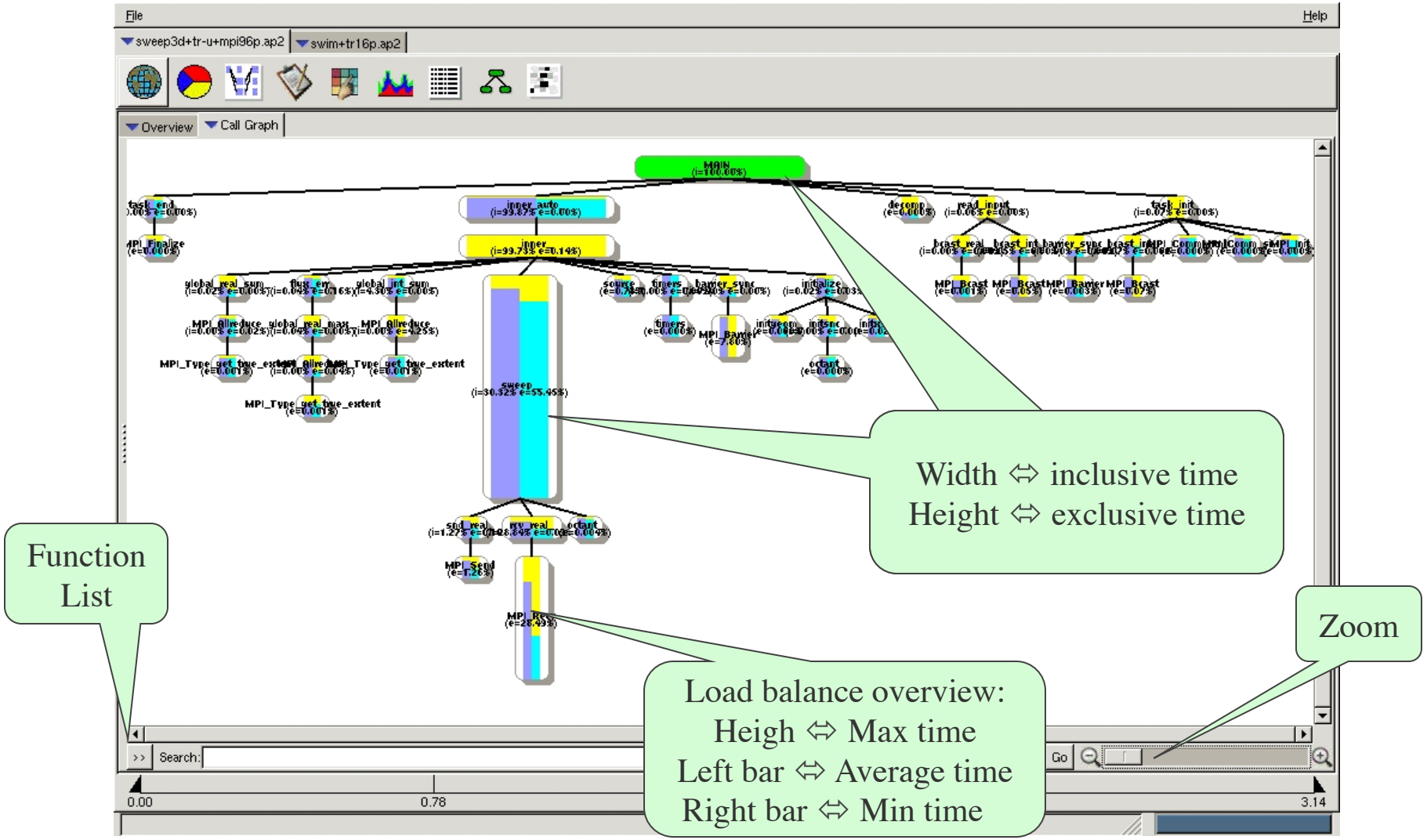
```
 Write Time |    Write MB |    Write |       Writes       |   Write |Experiment=1
|           |     Rate    |          |  B/Call            |       File Name
|           |   MB/sec    |          |                    | PE='HIDE'

 384.952355 | 2746.368641 | 7.134308 | 722729.000000 | 3984.59 |Total
|-----------------------------------------------------------------------
| 384.952075 | 2746.368626 | 7.134313 | 722727.000000 | 3984.60 |NA
|   0.000279 |    0.000015 | 0.054624 |      2.000000 |    8.00 |stdout
|=======================================================================
```

Apprentice2

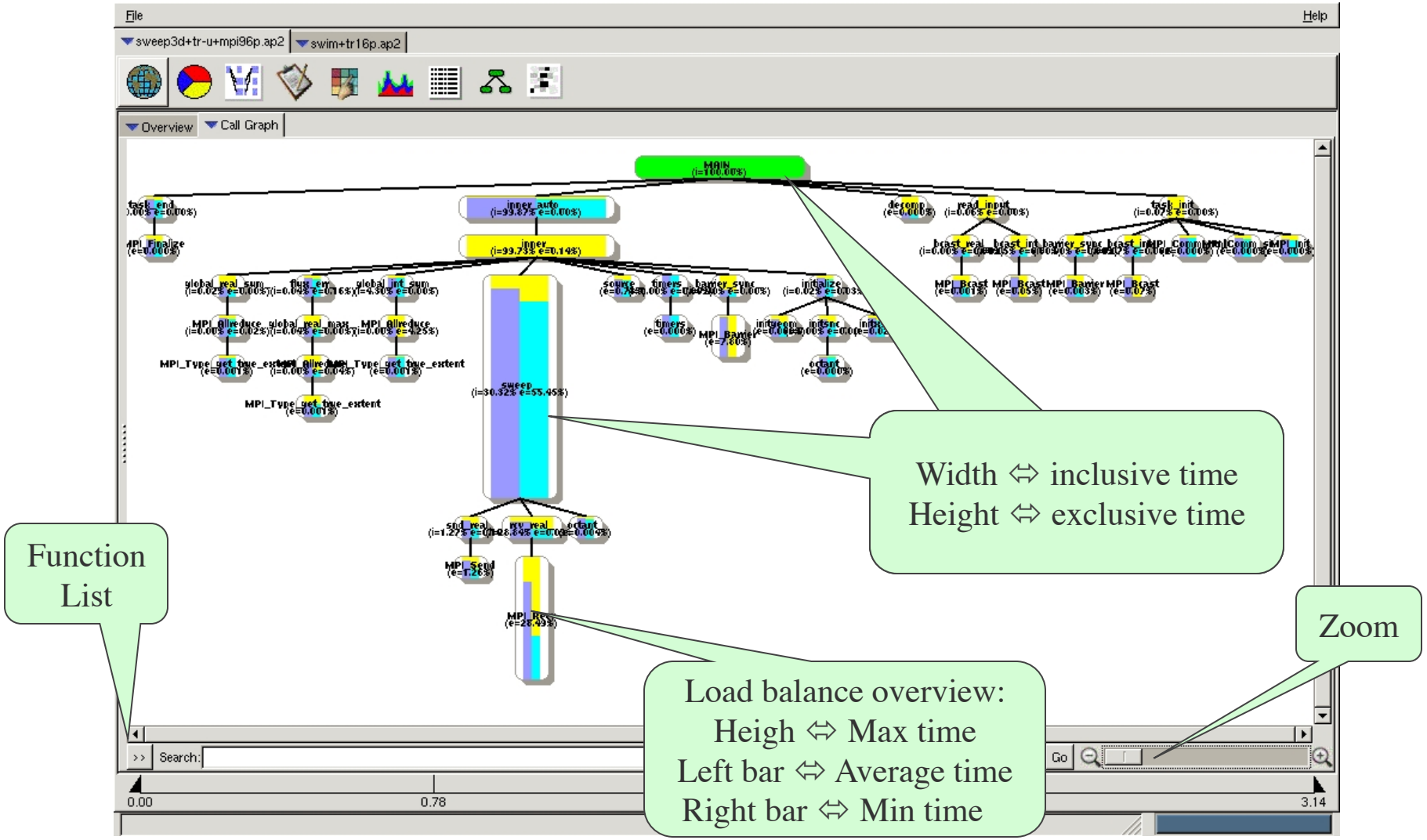
- module load apprentice2/5.0.1 , “app2 xxxx.ap2 “ ; not scalable ; for small scale run only



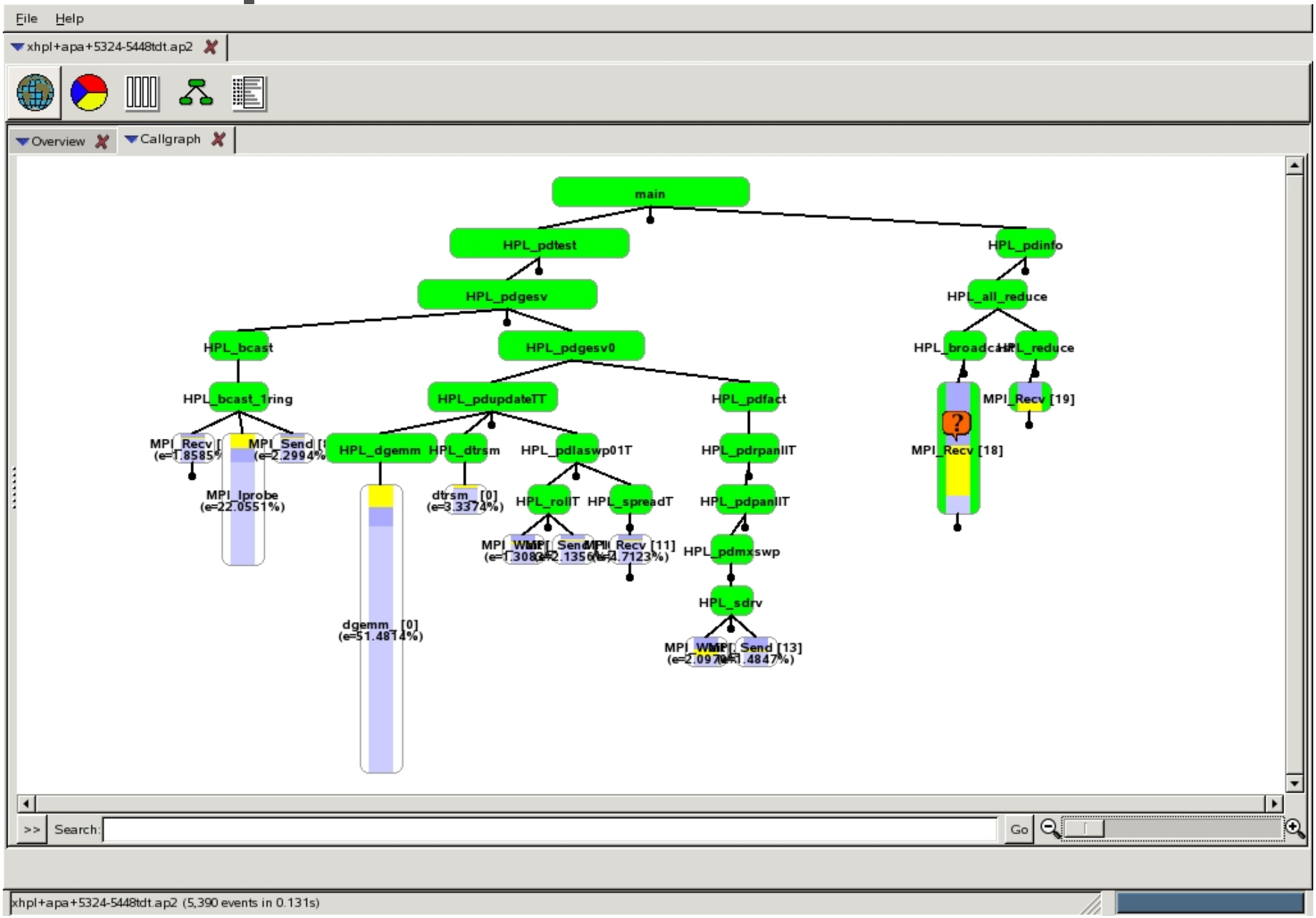
Call graph overview



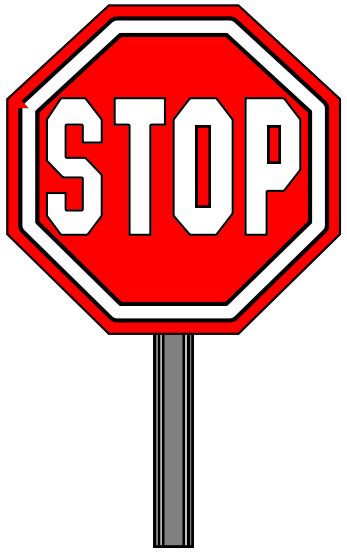
Call graph – functional list



Example - HPL


File Help
xhpl+apa+5324-5448tdt.ap2
Overview
Callgraph
main
HPL_pdtest
HPL_pdinfo
HPL_pdgesv
HPL_all_reduce
HPL_bcast
HPL_pdgesv0
HPL_broadcast
HPL_reduce
HPL_bcast_1ring
HPL_pdupdateTT
HPL_pdfact
MPI_Recv [19]
MPI_Recv [18]
HPL_dgemm
HPL_dtrsm
HPL_pdlaswp01T
HPL_pdrpanllT
MPI_Iprobe
(e=22.0551%)
dtrsm_[0]
(e=3.3374%)
HPL_rollT
HPL_spreadT
HPL_pdpanllT
MPI_Recv [11]
HPL_pdmxswp
HPL_sdrv
dgemm_[0]
(e=51.4814%)
MPI_Send [13]
(e=1.4847%)
>>
Search:
Go
xhpl+apa+5324-5448tdt.ap2 (5,390 events in 0.131s)

The End



- The End!