Programming of Parallel Computers, 2016

Course introduction

Jarmo Rantakokko
Karl Ljungkvist
Liu Jing
Anton Artemov

Parallel Computers

Range of parallel computers:

- Multi-core processor (2-16 cores/processor)
- Multi-processor PC (2-8 processors)
- Servers (up to 64 processors)
- PC-Cluster (multiple PCs, 100's-1000's procs)
- Integrated parallel computers (1000's procs)
- Supercomputers (100,000's procs, top10)
- GRIDs (Networks, e.g., LHC Grid 140 sites)
- Internet (1,000,000's procs)

SETI@home
The Search for Extraterrestrial Intelligence

- SETI@home (setiathome.berkeley.edu)
Distributed computing software runs as a screensaver, making use of processor time that would otherwise be unused. Has 1,300,000 users running over 3,300,000 processors for a compute power over 1 PetaFlops (10^{15} arithmetic ops per sec)

- Folding@home
- FightMalaria@home
- DrugDiscovery@home
- Neurona@home
- etc

Parallel Computers at UU

Intel iPSC/2 (Installed 1987)
 - 32 processors, Intel i386, 16 MHz
 - 32 x 4 Mbyte RAM
=> 32 x 16 MFlop/s = 0.512 GigaFlop/s

Today:

Quad-Core Laptop
384 core GPU

4 core mobile

2 processor x 6-Core
1024 core GPU

Uppsala University: Tintin (2012)

- 160 nodes x 2 proc x 8 cores = 2560 cores
- QDR Infiniband interconnect (integrated network)
- AMD Opteron 6220, 3GHz
- Peak performance 30.7 TeraFlop/s
- Total memory 10.2 TeraByte RAM

Tianhe-2 (1:st in Top 500, Nov 2014)

- National Super Computer Centre in Guangzhou, China
- 3.12 million cores, Intel Xeon processors
- Power consumption 17.8 MW (eq 6000 houses)
- 1024 TeraByte RAM
- 55 PetaFlop/s peak performance
- Cost: $390.000.000

---

www.top500.org

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3120000	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
3	DOE/NNSA/LLNL United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8586.6	10066.3	3945

Sweden on Top500

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
52	KTH - Royal Institute of Technology Sweden	**Beskow** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	53,632	1,397.0	1,973.7	
238	National Supercomputer Centre (NSC) Sweden	**Triolith** - Cluster Platform SL230s Gen8, Xeon E5-2660 8C 2.200GHz, Infiniband FDR Hewlett-Packard	25,376	407.2	446.6	519
305	National Supercomputer Centre (NSC) Sweden	**Bifrost** - ClusterVision RS720Q/RS700, Xeon E5-2640v3 8C 2.6GHz, Infiniband QDR ClusterVision	10,256	326.5	426.6	205

Systems per segment (top500 list)

Systems per location

Systems per area
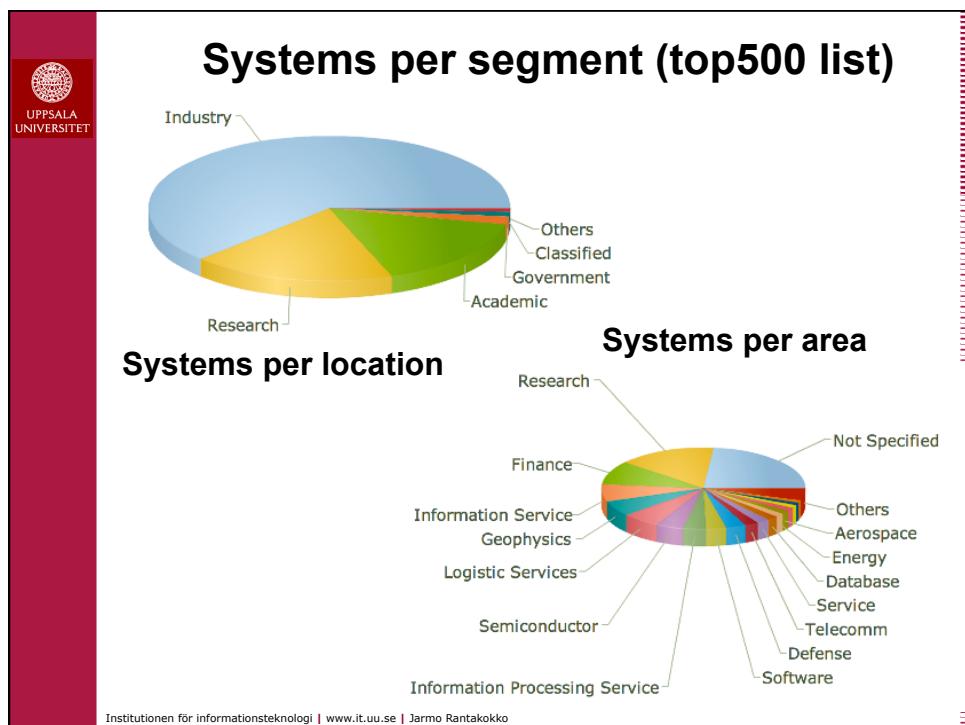
Note:

Multi-Processor, Multi-Core PCs with powerful GPU cards are here! Number of cores are increasing in each generation, our computers are becoming more and more parallel!

$\Rightarrow$ Need parallel programming on all levels and applications (OS, Games, Internet servers, Data bases, Scientific applications, etc).
Impossible to do automatically, compilers can not analyze all dependencies.

To exploit the full potential of our computers we need to explicitly parallelize our codes!

Throughput computing:
(Run P instances of the same program in parallel)

• Interfere in cache utilization (bad performance)
• Requires P times more memory
• Problems get worse with increased P

If limited by mem capacity/bandwidth go parallel
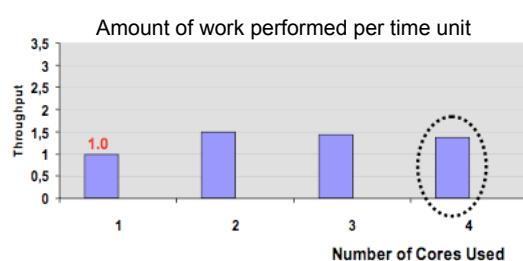
Example: "Lattice Boltzmann Method" to simulate incompressible fluids in 3D on a quad-core processor

What is Parallel Computing

Traditionally, software has been written for *serial* computation:
• To be run on a single computer having a single CPU.
• A problem is broken into a discrete series of instructions.
• Instructions are executed one after another.
• Only one instruction may execute at any moment in time.
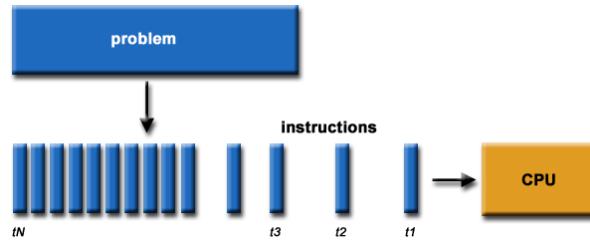
In the simplest sense, *parallel computing* is the simultaneous use of multiple compute resources to solve a computational problem:
• A problem is broken into discrete parts that can be solved concurrently
• Each part is further broken down to a series of instructions
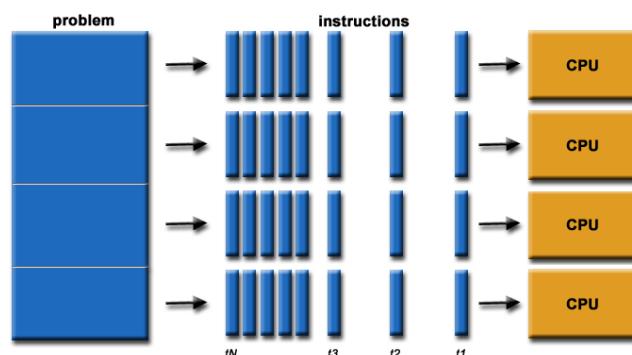
Parallel Programming models

Two main models (scientific computing)
1. Local name space or private memory
   C/C++/(Fortran) and **MPI**
2. Global name space or shared memory
   C/C++/(Fortran) and **OpenMP/Pthreads**

SPMD - Single Program Multiple Data
(All processors run the same program)

Parallelism through data ownership and/or
branches (if MyPid==1 then)
Need to communicate and synchronize!

MPI: Message Passing Interface (1994)

Used on PC-Clusters and other large parallel
computers. Each process has its own private
address space => Data is shared through
explicit communication calls (library).

Point-to-point : `MPI_Send` – `MPI_Recv`
Collective calls: `MPI_Bcast`, `MPI_Reduce`, etc
[Over 100 MPI function]

Need to specify exactly how to divide data
and what each processor should do and who
to communicate with => low-level model
But a scalable model with high performance!

Pthreads: POSIX threads

Used on multi-core machines and other shared memory computers. Shared address space model, based on threads ("*light weight process*"). All threads have access to global data.

Memory coherence handled by hardware but requires explicit synchronization and protection of shared variables from multiple updates.

Low-level model but easier to program as data is global to all threads (no need to explicitly create ownership and communicate data between processors). Still need to divide work manually and synchronize threads.

OpenMP: Open specification for Multi Processing

Used on multi-core machines and other shared memory computers. Shared address space model, based on threads. All threads have access to global data. Memory coherence handled implicitly by compiler (and hardware).

Insert compiler directives for parallelization of computations => high-level model

```
#pragma omp parallel for
 for (i=1;i<N-1;i++)
   A[i]=F(B[i-1]+B[i]+B[i+1]);
```

Loop is automatically parallelized over all threads, arrays A and B are global data.

OpenMP directives:

Parallel   (main, fork threads)

Data sharing	**Work sharing**	**Serial sections**	**Synchronization**
- shared	- do/for	- single	- barrier
- private	reduction	- master	- flush
- firstprivate	schedule	- critical	- nowait
- lastprivate	ordered	- atomic	
- threadprivate	- sections	- ordered	
	- tasks		

Master thread   Master   Master

Parallel region   Parallel region with single section
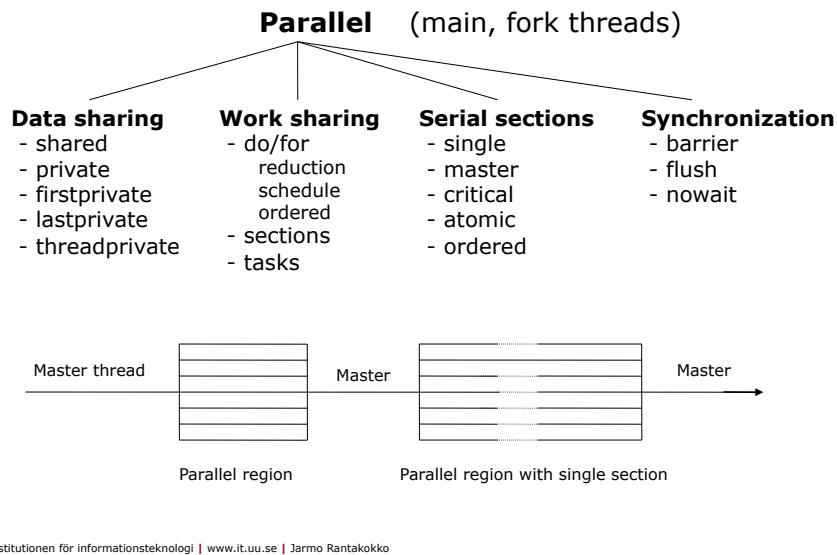
Example: Enumeration Sort

```
for (j=0;j<len;j++)
{
  rank=0;
  for (i=0;i<len;i++)
    if (indata[i]<indata[j]) rank++;
  outdata[rank]=indata[j];
}
```

Where is the parallelism? Identify parallel tasks!

OpenMP Solution:

```
#pragma omp parallel for private(rank,i)
for (j=0;j<len;j++)
{
  rank=0;
  for (i=0;i<len;i++)
    if (indata[i]<indata[j]) rank++;
  outdata[rank]=indata[j];
}
```

The j-loop perfectly is parallel, each iteration can be done in parallel, i.e., independently. (Impossible for compiler to analyze, you guarantee the correctness.)

Example: Quick Sort

Algorithm:
1. *Select pivot element*
2. *Divide data into two sets (smaller or larger)*
3. *Sort each set with Quick Sort*

Parallelism:

In each split acquire a new processor/thread and proceed with the two lists in parallel.

=> Limited parallelism (in the first step only one processor have something to do, waste if resources) and poor load balance due to pivot selection.

Reformulate the algorithm!

Parallel Quick Sort

Algorithm:
1. Divide the data into p equal parts
2. Sort the data locally in each processor
3. Perform global sort
    3.1 Select pivot in each processor set
    3.2 In each processor, divide the data
       into two sets (smaller or larger)
    3.3 Split the processors into two groups
       and exchange data pair-wise
    3.4 Merge data into a sorted list in each
       processor
4. Repeat 3.1-3.4 recursively for each
   processor group

Unsorted data
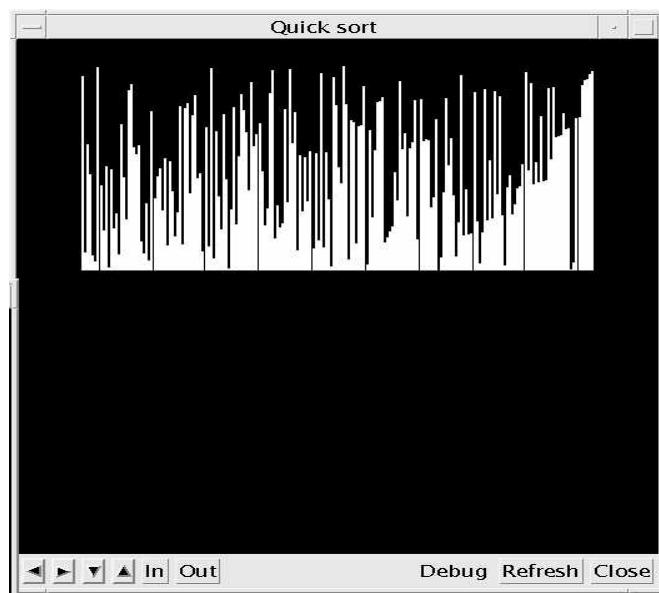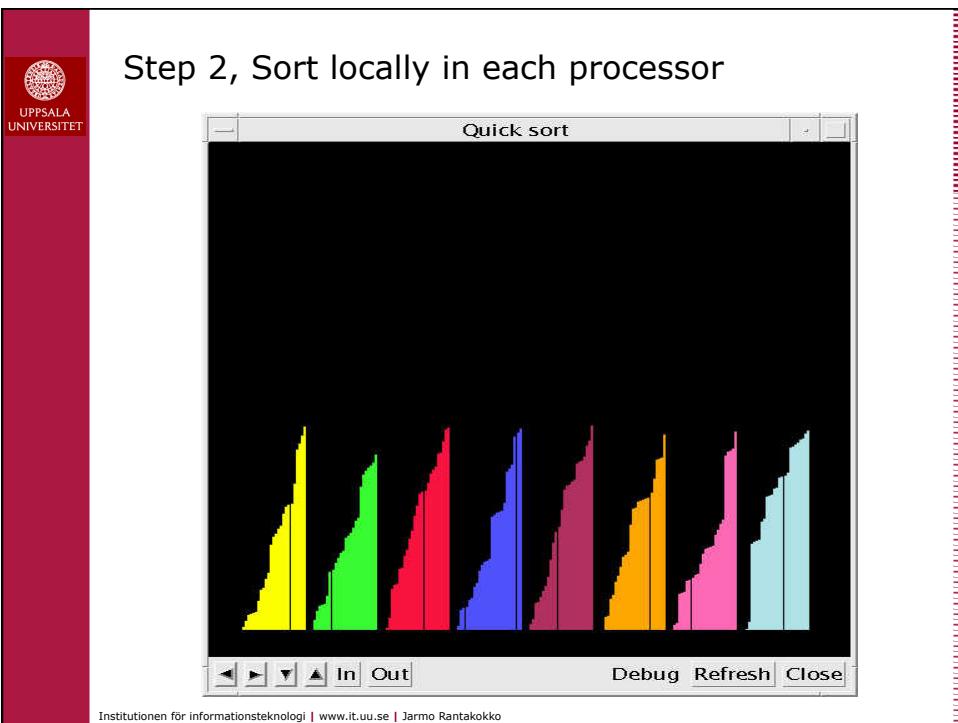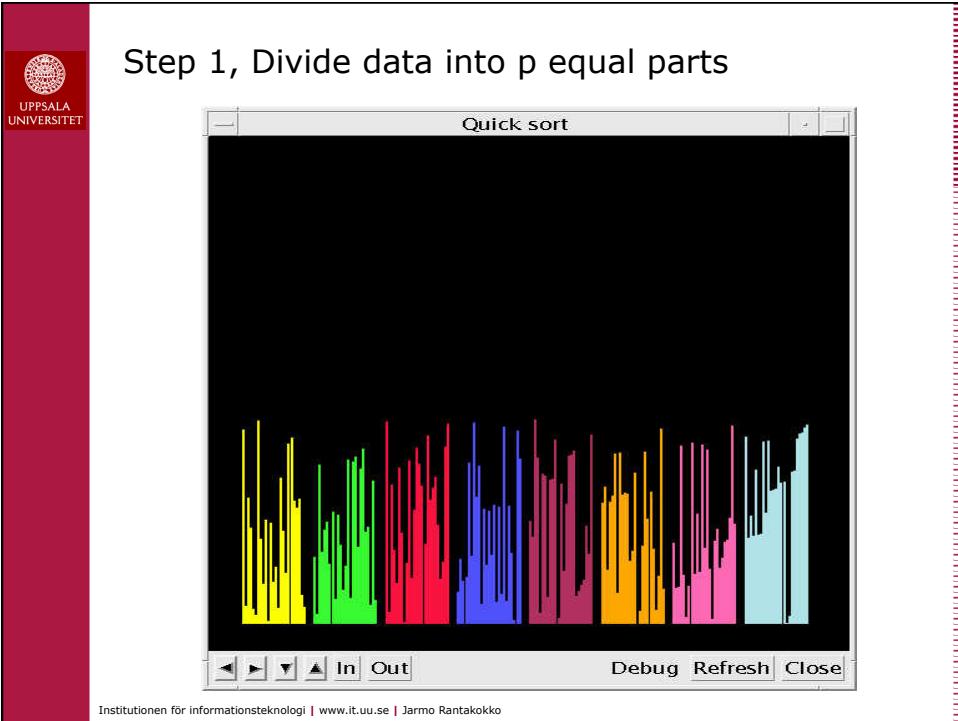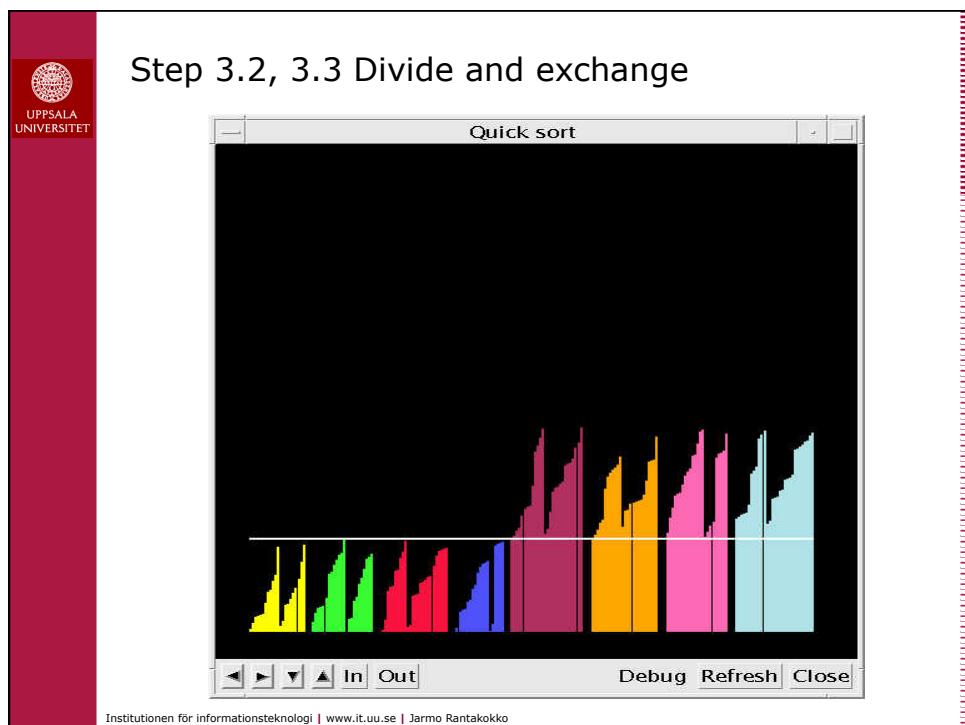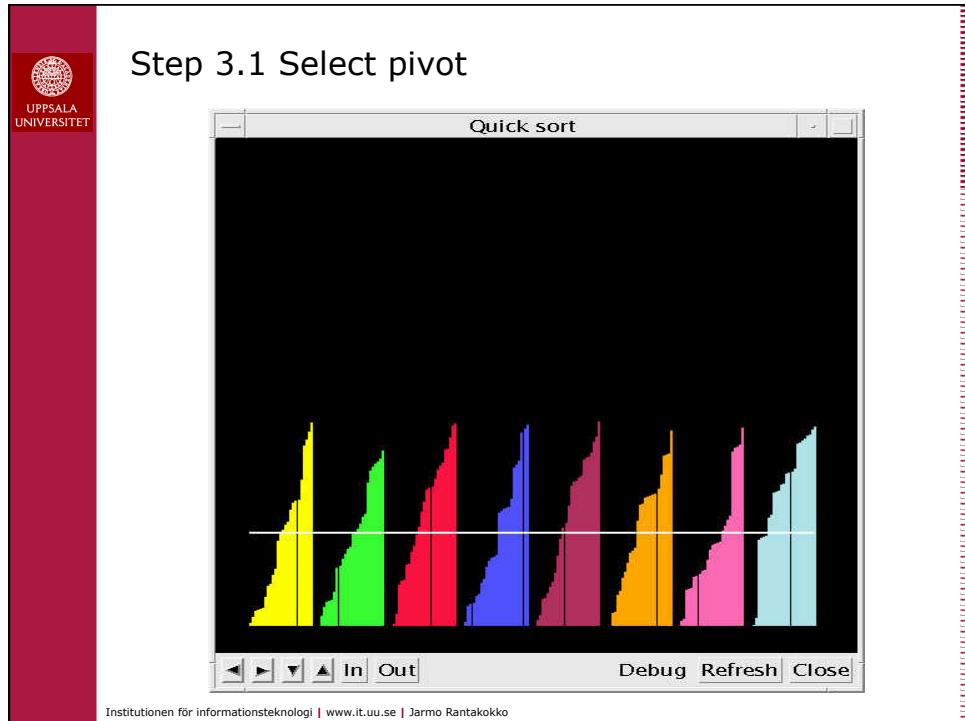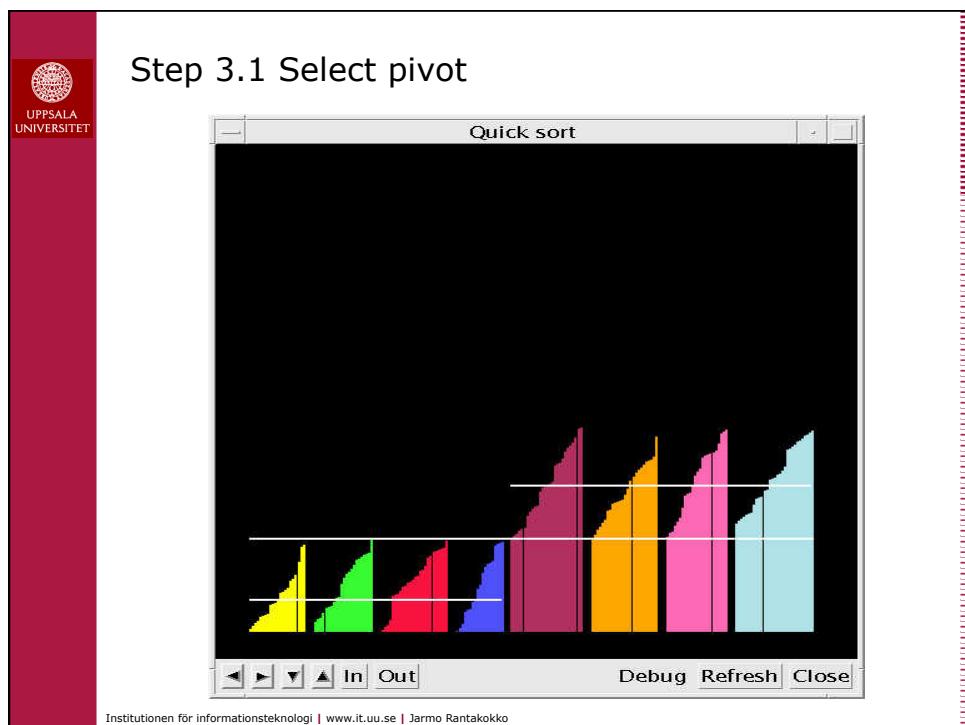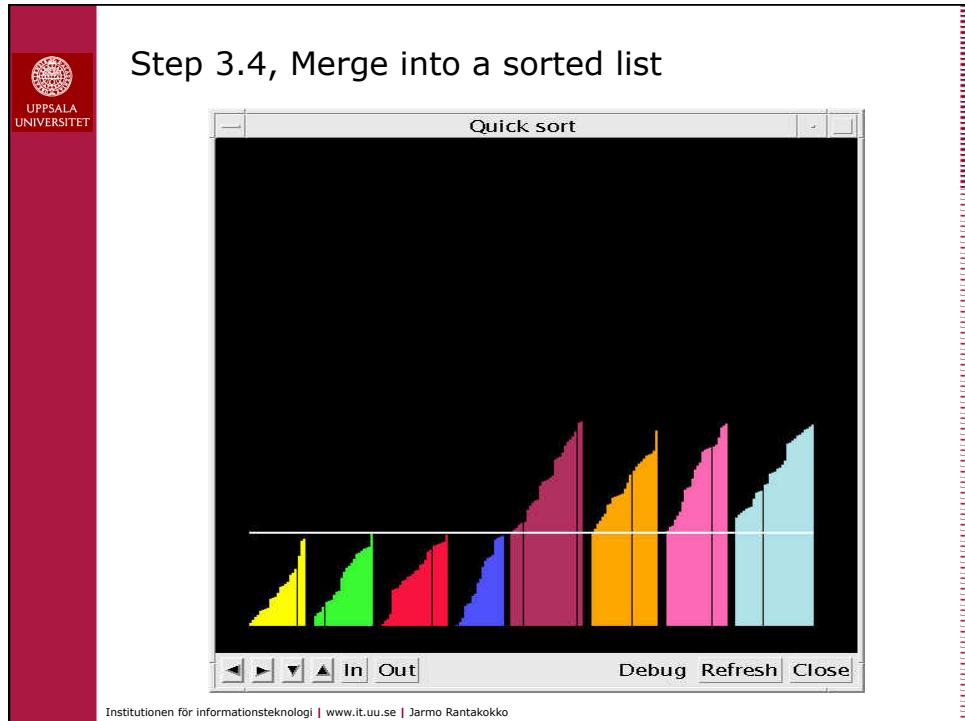
Step 1, Divide data into p equal parts
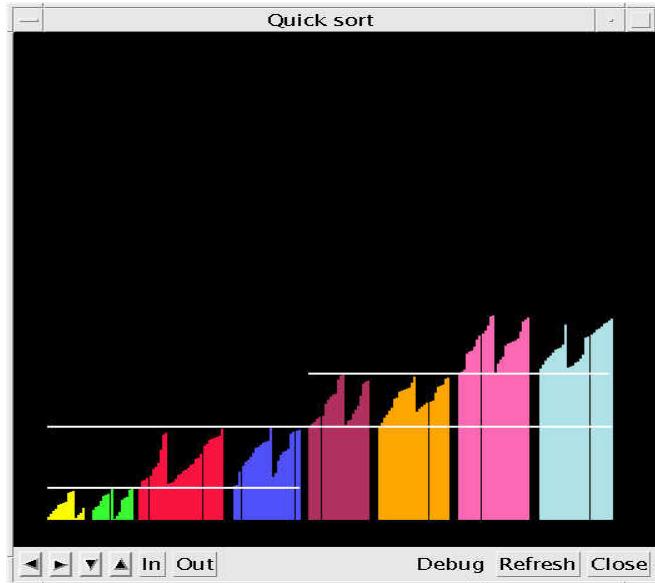
Institutionen för informationsteknologi | www.it.uu.se | Jarmo Rantakokko

Step 2, Sort locally in each processor
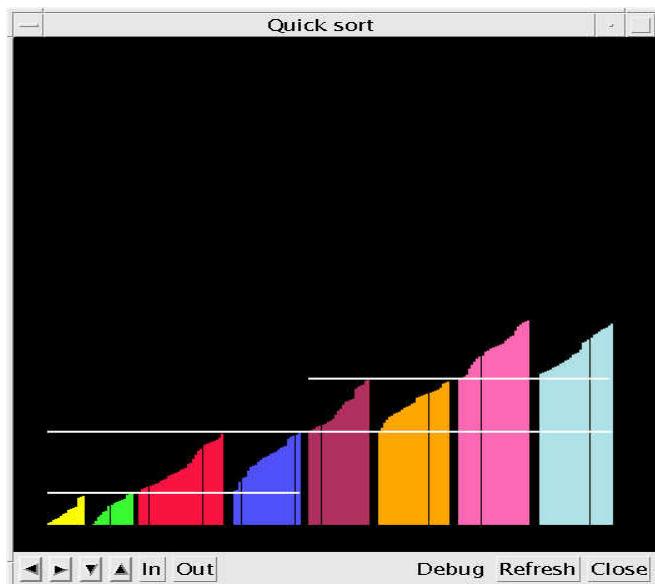
Institutionen för informationsteknologi | www.it.uu.se | Jarmo Rantakokko

Step 3.1 Select pivot

Step 3.2, 3.3 Divide and exchange

Step 3.4, Merge into a sorted list

Step 3.1 Select pivot

Step 3.2, 3.3 Divide and exchange

Step 3.4, Merge into a sorted list

Step 3.1 Select pivot

Step 3.2, 3.3 Divide and exchange

Step 3.4, Merge into a sorted list

Final state

Execution profile, second view
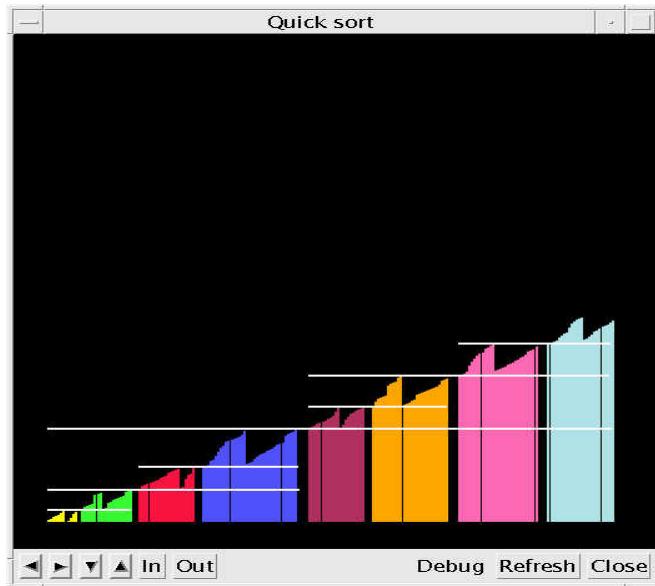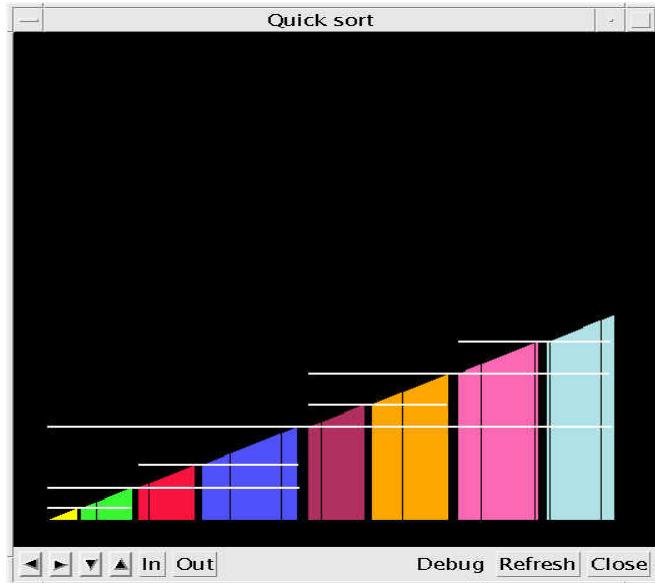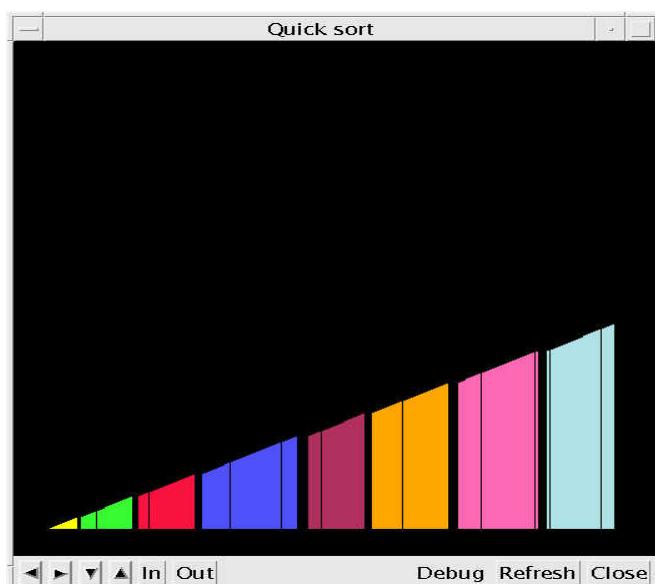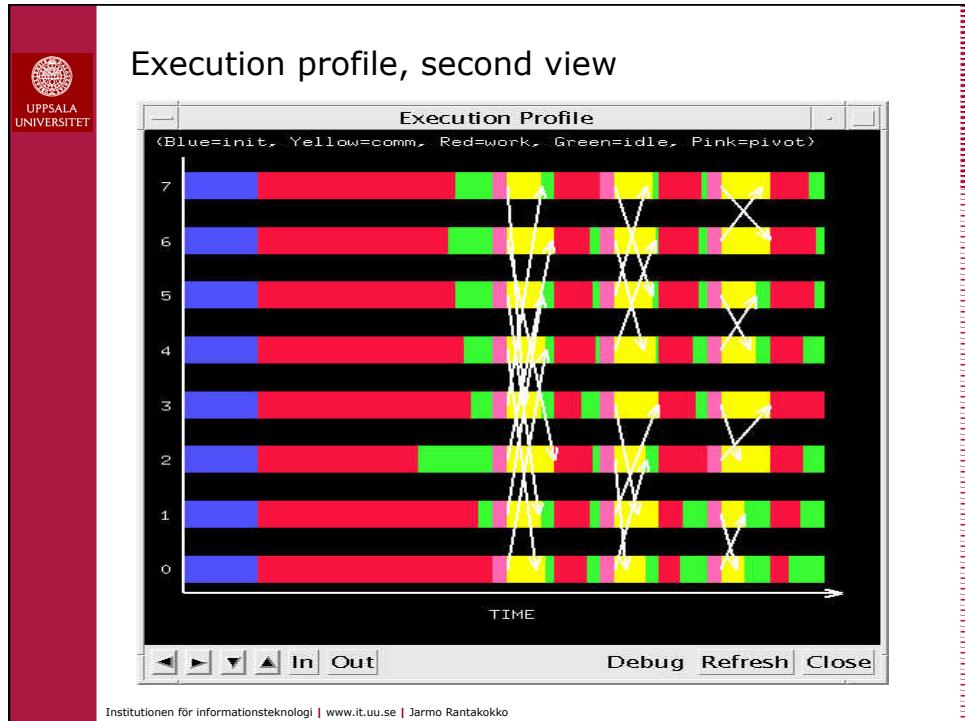
Performance obstacles:

- **Communication**
  -Use asynchronous comm and message probing

- **Load balance**
  - Choose pivot carefully, different strategies

- **Synchronization**
  - Split communicator (topology) in each step
    to avoid global synchronization

Example: Numerical PDE Solver

Consider the Hyperbolic PDE:

$$u_t + u_x + u_y = F(t, x, y) \quad 0 \le x \le 1, 0 \le y \le 1$$

$$\begin{cases} u(t, 0, y) = h_1(t, y) & 0 \le y \le 1 \\ u(t, x, 0) = h_2(t, x) & 0 \le x \le 1 \end{cases} \quad \textit{Boundary Conditions}$$

$$u(0, x, y) = g(x, y) \quad \textit{Initial Conditions}$$

Solve with explicit finite difference method,
for example Leap-Frog

Core of the computations:

```
do k=2,Nt
   t=k*dt; Uold=U; U=Unew;
   do j=1,Ny-1
      do i=1,Nx-1
        x=i/Nx; y=j/Ny
        Unew(i,j)=Uold(i,j)+2*dt*(F(t,x,y)-
                (U(i+1,j)-u(i-1,j))/(2*dx)-
                (U(i,j+1)-U(i,j-1))/(2*dy))
      end do
   end do
end do
```

Computational stencil:
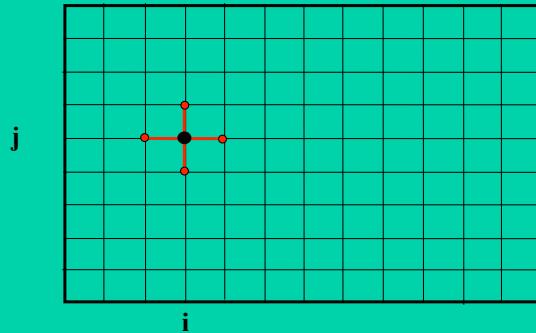
Parallelization, partition grid:
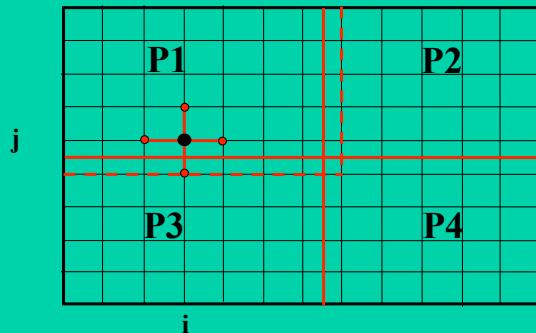
Message Passing Version:

```
do k=2,Nt
    t=k*dt; Uold=U; U=Unew;
    update partition boundary - communicate
    do j=j1,j2
        do i=i1,i2
            x=i/Nx; y=j/Ny
            Unew(i,j)=Uold(i,j)+2*dt*(F(t,x,y)-
                     (U(i+1,j)-u(i-1,j))/(2*dx)-
                     (U(i,j+1)-U(i,j-1))/(2*dy))
        end do
    end do
end do
```

```
Compute 'left, right, up, down' node

! Send and receive left-right
if ('not at left boundary')
    call mpi_send(left, ... )
end if

if ('not at right boundary')
    call mpi_recv(right, ... )
    call mpi_send(right, ... )
end if

if ('not at left boundary')
    call mpi_recv(left, ... )
end if

! Send and receive up-down
...
```

Performance, 256x256 grid:
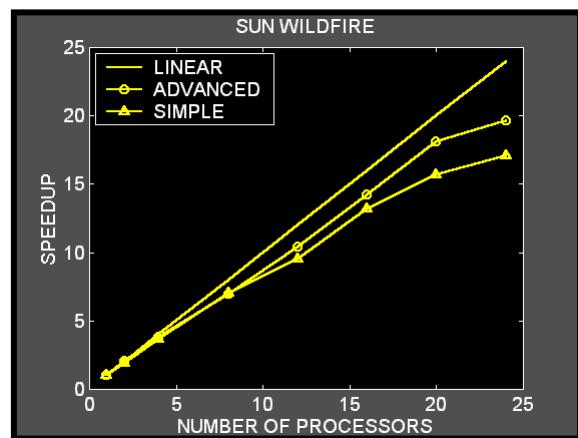
Hands-on session: Study implementation details and measure parallel performance

Real world example:
• Irregular shape
• Irregular work load (due to sea depth)
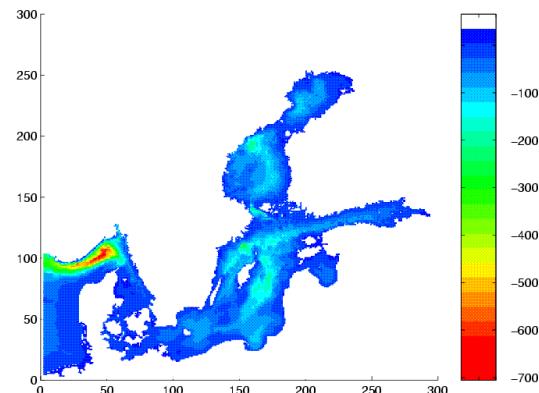
Oceanographic model from SMHI

Need to use *data partitioning algorithms* to:
• optimize load balance
• minimize communication
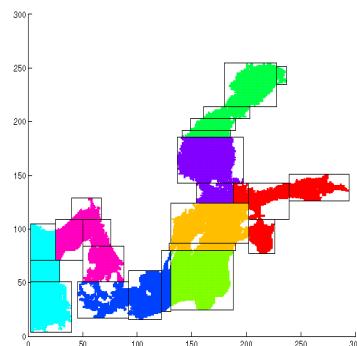
Block-structured partitioning   Unstructured graph partitioning
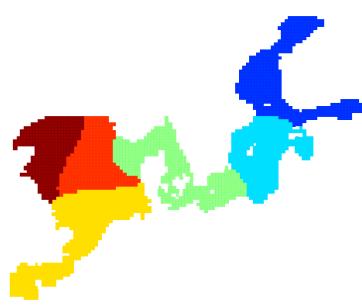
---

Summary:

Parallel computers are here, ranging from laptops to supercomputers. Need parallel programming in all programs and applications.

When writing parallel programs need to:
    • think about algorithm (parallelism)
    • optimize load balance (equal work load)
    • minimize communication (parallel OH)

The topics are interdependent which makes it very hard (impossible) for a compiler to create efficient code.

Course content:

- Parallel computer architecture
- Programming with message passing, **MPI**
- Programming with threads, **Pthreads**
- Programming with compiler directives, **OpenMP**
- Algorithms in linear algebra
- Algorithms in sorting
- Performance analysis
- Load balancing and data partitioning
- Programming of GPUs, **CUDA**
- Other programming models, e.g., **UPC, Matlab**

Aim of the course: To give skills in parallel programming with MPI, Pthreads and OpenMP! Examination is to a large extent through programming assignments.

Schedule:

Lecture	Theme
20/1	Introduction (JR)
22/1	Lab C-Programming (optional) (LJ)
26/1	Parallel Computer Architecture (JR)
28/1	MPI, 1 (LJ)
29/1	MPI, 2 (LJ)
2/2	Lab MPI (AA, LJ)
3/2	MPI, 3 (LJ)
4/2	Parallel Algorithms, Algebra (LJ)
5/2	Parallel Algorithms, Algebra (LJ)
9/2	Parallel Algorithms, Sorting (LJ)

Lecture	Theme
12/2	Performance analysis (LJ)
15/2	Data partitioning,load balancing (LJ)
16/2	Pthreads 1 (KL)
17/2	Pthreads 2 (KL)
18/2	Lab Pthreads (AA, KL, LJ)
19/2	GPU programming 1 (KL)
25/2	GPU programming 2 (KL)
26/2	Lab GPU (KL,LJ)
29/2	OpenMP 1 (JR)
1/3	OpenMP 2 (JR)
4/3	Lab OpenMP (AA, LJ)
7/3	OpenMP 3 (JR)
10/3	UPC, Parallel Matlab (JR)

Examination:

- Four labs (MPI, Pthreads, OpenMP, GPU)
- Three programming assignments
  Can be done in groups of 2-3 students

Then chose:

- Project work (grade 3)
- Written exam (grade 3, 4 or 5)

- Project work + written exam (3, 4 or 5)
  Project work can raise your exam grade

Project work:

- Individual assignment

- *You* formulate a problem that you want to parallelize, write a project proposal

- Hand in project proposal to us

- If approved you can get an account on Uppmax systems, e.g., Tintin.

- Perform the project and write a full report