1: Parallel Computations

Course in *Scalable Computing*

Vittorio Scarano

Università di Salerno

Dottorato di Ricerca in Informatica

Plan

Plan

Aims of the course

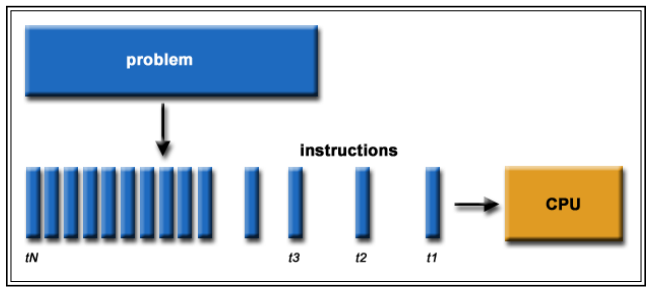
- Introduce the problems and the solutions to scalability issues in parallel programming
- Structure of the course:
  - Design Principles of Multicore Processors: to ensure scalability, how should the next "desktop" processor be designed?
  - Scalable Programming Techniques: what language/model should be used to ensure scalability of parallel programs?
- Target: PhD students (I mean you!) and long-term "*aficionados*"

Logistics

- 10 2-hour lessons, held every two weeks (approx.)
  - usually into the Meeting Room of the DI
- Exams: some possibilities
  - Note-taking of a couple of (connected) lessons: with material, slides, references, etc. provide the reading material for that part of the course (in english, latex)
  - Essay on a specific subject, partly developed into the course (or proposed by the student/teacher)
- Material of the course accessible from my home page `http://www.dia.unisa.it/professori/vitsca`
  - No rush.. wait few days!
- Next class: approx. after Easter, but provide me with your mail and I'll write you

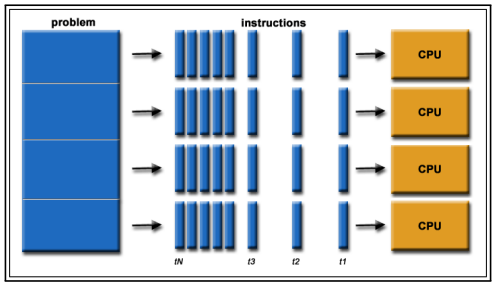
Plan

What is Parallel Computing - 1



- Traditionally, software has been written for serial computation
- A problem is broken into a discrete series of instructions
- . . . instructions are executed one after another.

What is Parallel Computing - 2

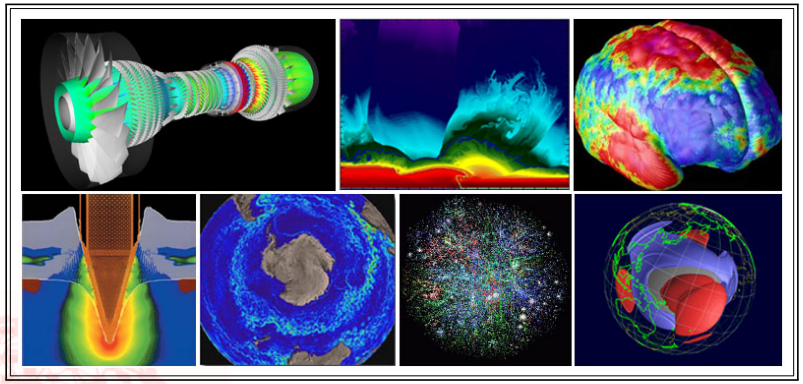


- Simultaneous use of multiple compute resources
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- . . . instructions from each part execute simultaneously on different CPUs

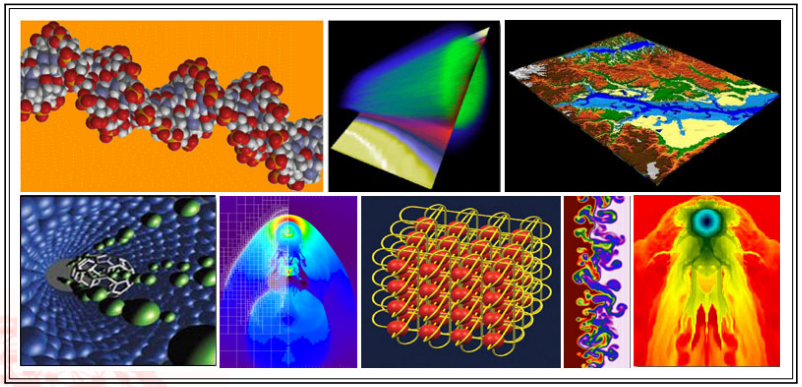
The Universe is Parallel

"Massively" parallel

Uses for Parallel Computing:o - 1



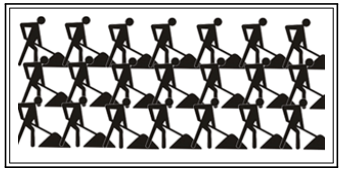
Uses for Parallel Computing: - 2

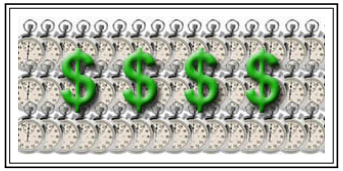


Plan

Why Use Parallel Computing? - 1

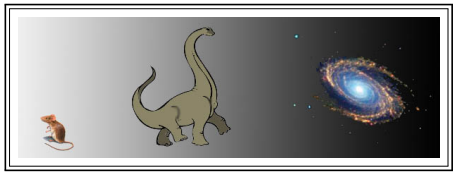
- Save time and/or money
- More resources at a task will shorten its time to completion
- Parallel clusters can be built from cheap, commodity components





Why Use Parallel Computing? - 2

- Solve larger problems
- Many problems are so large that it is impractical to solve them on a single computer
- Problems requiring PetaFLOPS and PetaBytes of computing
- Web search engines/DB processing millions of transactions per second
- Provide concurrency (the "Grid")



Why Use Parallel Computing? - 3

- Use of non-local resources
- Available for free
- Seti@Home (330,000 PC with 528 TeraFlops)
- Folding@Home (340,000 PC with 4.2 PetaFlops)

Why Use Parallel Computing? - 4

- Limits to serial computing
- Speed of light (light: 30cm/nanosecond, transmission on copper wire 9 cm/nanosecond)
- Limits to miniaturization (molecules!)
- Economic limitations: better commodity PCs than costly single, faster processors

Some limitations: area and speed of light - 1

- Requirement: this loop must take one second

```
/* x, y, and z are arrays of floats, each containing
 * one trillion of entries
 */
for (i = 0; i < ONE_TRILLION ; i++)
    z[i]= x[i] + y[i]
```

- $\Rightarrow 3 \times 10^{12}$ copies between memory and register per second
- Since data travels at speed of light (at most): 3×10^8 m/s
- Avg distance between word of memory to CPU must satisfy:

$$(3 \times 10^{12} r)m \propto 3 \times 10^8 m/s \times 1s \Rightarrow r \propto 10^{-4} m$$

Some limitations: area and speed of light - 2

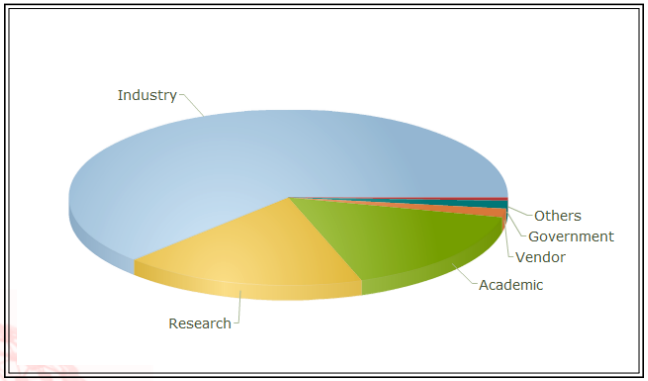
- Now, 3×10^{12} word memory must be stored somewhere
  - (and, also, better be quickly accessible!)
- If a square grid of size s is to be used, the average distance from CPU to a memory location will be $s/2$
- If we ask that $s/2 = r \propto 10^{-4} m$, it means that the $s \propto 10^{-4} m$
- In a "row" of this square grid, there must be $\sqrt{3 \times 10^{12}} = \sqrt{3} \times 10^6$ words
- So a single word must fit into a "cell" with size proportional to:

$$\frac{10^{-4} m}{\sqrt{3} \times 10^6} \propto 10^{-10} m$$

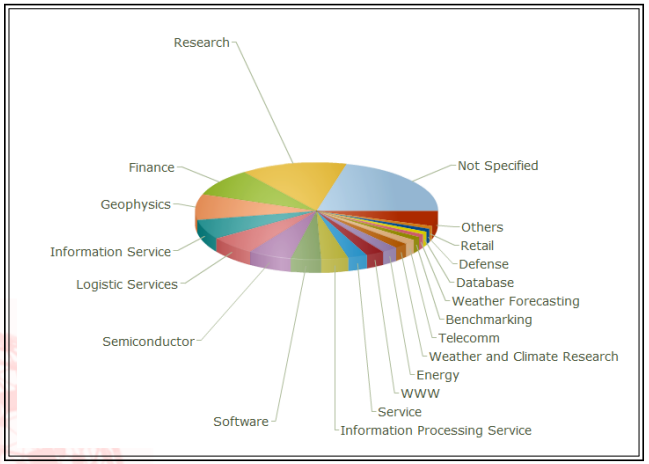
that is the size of a (small) atom

- Unless we are confident in embedding a 32/64/128 bit word into an atom, we are hitting a wall!

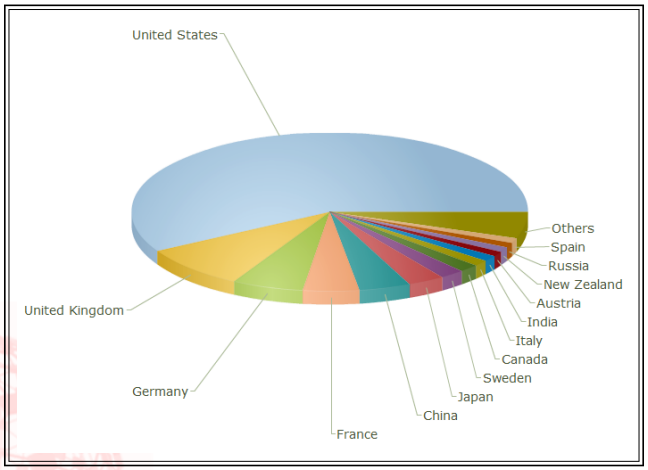
Who is doing "parallel"?



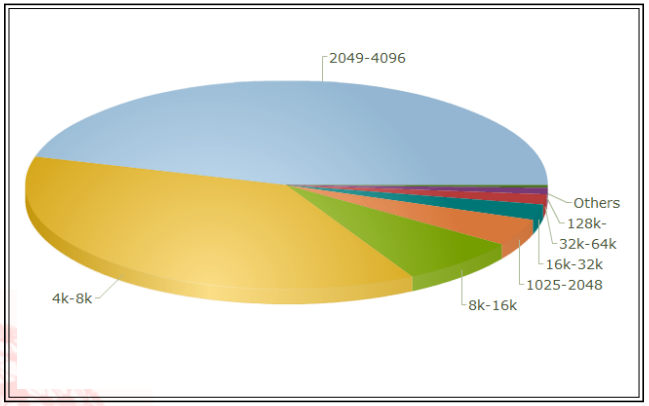
What are those-who-do-"parallel" doing?



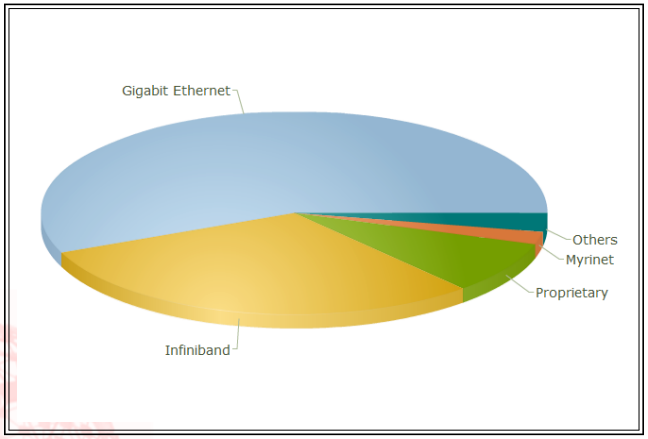
Where is "parallel" done?



How many processors do "parallel"...

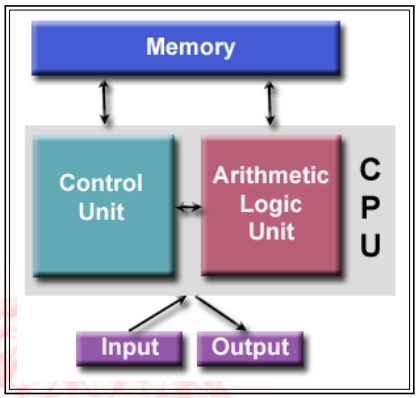


Which networks are used to do "parallel"



Plan

Von Neumann Architecture



- 1945 paper
- 4 components: memory, Control Unit, ALU, I/O
- Read/write in memory
- Instructions as stored data
- UC fetches instructions/data and **sequentially** coordinates the execution

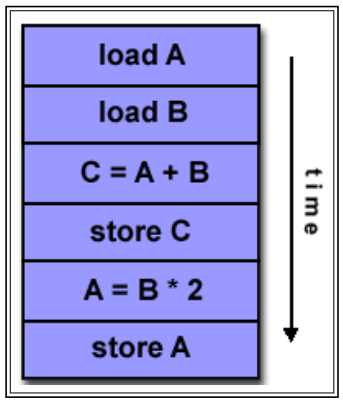
Flynn Taxonomy

- Along two axes:
  - Instruction/Data
  - Execution Single/Multiple

SISD Single Instruction, Single Data	**SIMD** Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	**MIMD** Multiple Instruction, Multiple Data

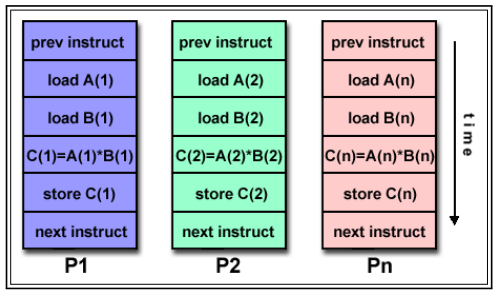
Single Instruction, Single Data (SISD)

- Serial (non parallel) computer
- Single Instruction: only one instruction stream is being acted on by the CPU
- Single Data: only one data stream is being used as input



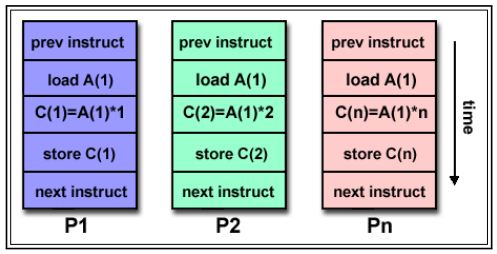
Single Instruction, Multiple Data (SIMD)

- Single Instruction: one instruction per cycle (on all Processing Units)
- Multiple Data: different data elements are processed by each PU
- Synchronous
- Processor Arrays and Vector Pipelines
- Nowadays: GPUs



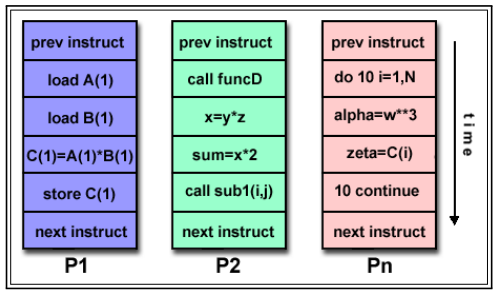
Multiple Instruction, Single Data (MISD)

- Multiple Instruction: several instructions executed per cycle
- Single Data: one single data used
- Few actual examples (CM University)
- Possible applications: multiple, simultaneous crack attempts for a single coded message



Multiple Instruction, Multiple Data (MIMD)

- Multiple Instruction: several instructions executed per cycle
- Multiple Data: different data elements are processed by each PU
- Synch/asynch (same clock)
- Cluster, Grid, Multiprocessors, Multicore



Terminology - 1

- Symmetric Multi-Processor (SMP): multiple processors share a single address space and access to all resources
- Distributed Memory
- Granularity:
  - coarse grain: relatively large amounts of computational work are done between communication events
  - fine grain: relatively small amounts of computational work are done between communication events

Terminology - 2

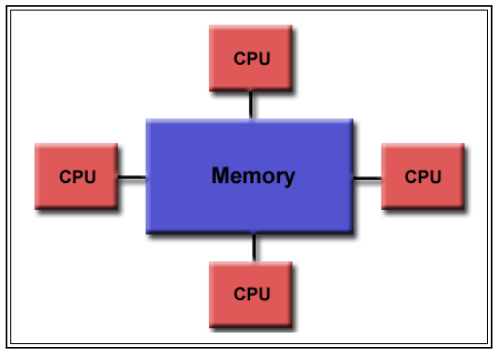
- Observed Speedup : *wall-clock* of serial execution / *wall-clock* of parallel execution
- Parallel Overhead : time required to coordinate parallel task (start-up, synchronization, data communication, sw (libraries, tools, OS, etc.), termination)
- Scalability: ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors

Plan

Plan

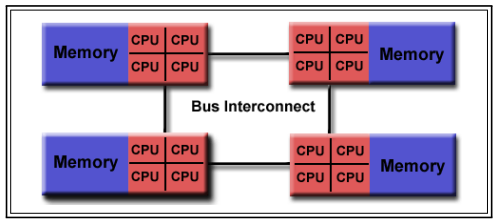
Shared Memory - 1 (UMA)

- All processors to access all memory as global address space.
- Processors can operate independently
- Uniform Memory Access: symmetric multiprocessors
- Equal access and access times
- *Cache-coherent UMA*



Shared Memory - 2 (NUMA)

- Different processors (or SMPs) interconnected
- Not all processors have equal access time to all memories
- *Cache-coherent* NUMA

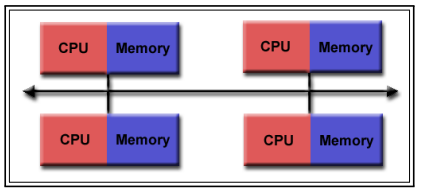


Comments

- user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform
- Lack of scalability between memory and CPUs (geometrical increase); even worst with cache-coherence
- Programmer responsibility for "safe" memory access
- Cost

Distributed memory

- Require a communication network to connect inter-processor memory (even just Ethernet)
- Memory addresses in one processor do not map to another processor (no global address space)
- No cache is needed
- Explicit request for data from one CPU to another (programmer driven)

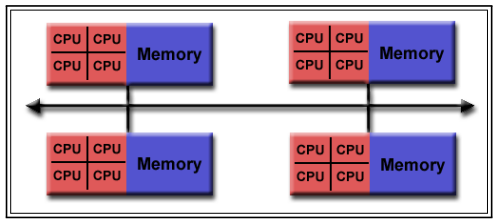


Comments

- Scalable Memory
- Quick access to its own memory without interference
- Cost-effective
- Programmer is responsible for the details of data communication
- Difficult to map existing data structures, based on global memory, to this memory organization
- Non-uniform access

Hybrid Distributed-Shared Memory

- Merges advantages of shared and distributed memory (if you are lucky ☺)
- The shared memory component is usually a cache coherent SMP machine
- The distributed memory component is the networking of multiple SMPs



Plan

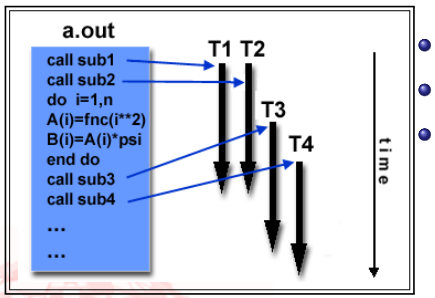
The models

- Abstraction between sw and architecture(hw, memory)
- Not strictly tied to architecture:
  - Kendall Square Research: shared memory on an architecture with distributed memory
- No "best" model: depends (at least) by the nature of the problem and availability of resources /hw/sw

Shared memory

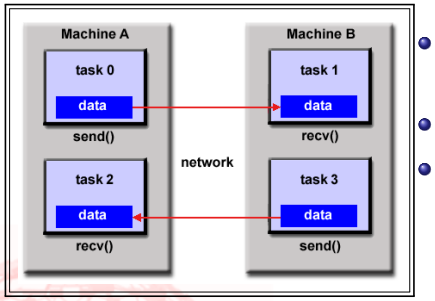
- Tasks share a common address space
- Locks / semaphores may be used to control access to the shared memory
- ⊕ No data "ownership" ⇒ sw development simplified
- ⊖ Difficult to understand and manage data locality (effects on performances (cache, bus traffic, etc))
- Implementations: KSR ALLCACHE

Multithreading



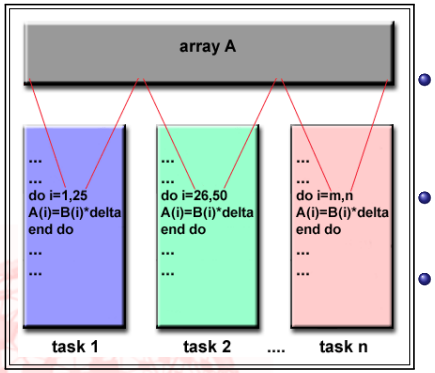
- Threads part of a process (they share memory)
- Subroutines
- Available on OSs, since long ago
- Implementations: Posix Thread, OpenMP

Message Passing - 1



- Task exchange data with `send()` and `recv()`
- Collaboration (matching recv per send)
- Implementations: via libraries
- MPI (since 1992) standard "de facto", also available on shared memory architectures

Data Parallel



- Parallel work focuses on performing operations on a data set
- Each task works on a different partition of the same data structure.
- Coordination realized by modified compilers
- Implementations: Fortran 90 and 95, High Performance Fortran

Plan

How to parallelize a program

- Usually a manual process: complex task, *error-prone*
- Tools for automatic parallelization
  - assisting the programmer
- *Fully automatic* compiler
  - analyzes the source code and identifies opportunities for parallelism (e.g. loops)

  errors, no flexibility, only working on part of the code
- *Programmer directed* compiler
  - compiler directives to help the parallelization

  difficult to use

Understand the problem and the program

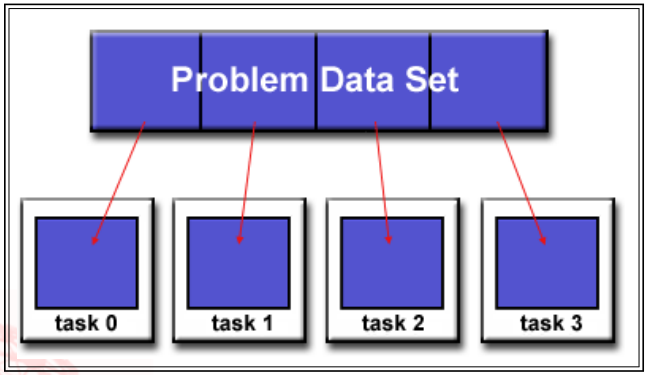
- Problems that look more (or less) parallelizable
  - Fibonacci: calculating F_{k+2} uses F_{k+1} and F_k. Non trivial solutions
- Identify *hotspots* of the program
  - "profilers" identify where the most part of the work is done
- Identify *bottlenecks*
  - Synchronization, I/O
- Identify inhibitors to parallelism
  - data dependancy

Plan

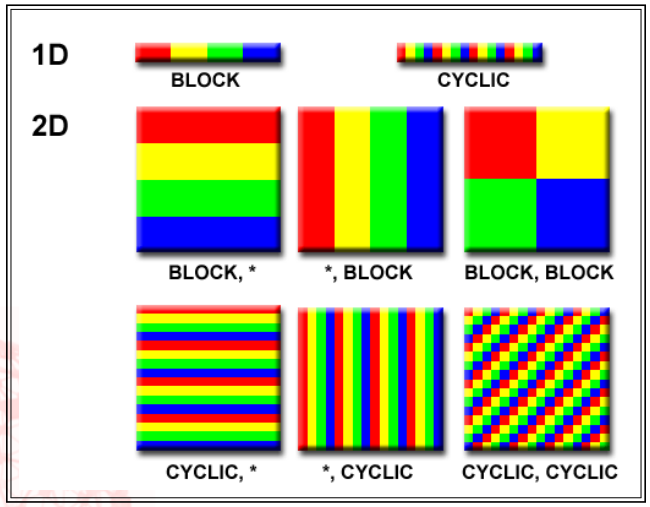
Partitioning

- Break the problem into discrete chunks of work
- ... that can be distributed
- Two basic ways to partition computational work:
  - Domain decomposition: data associated with a problem is decomposed and each task works on a portion of the data
  - Functional decomposition: problem is decomposed according to the work that must be done

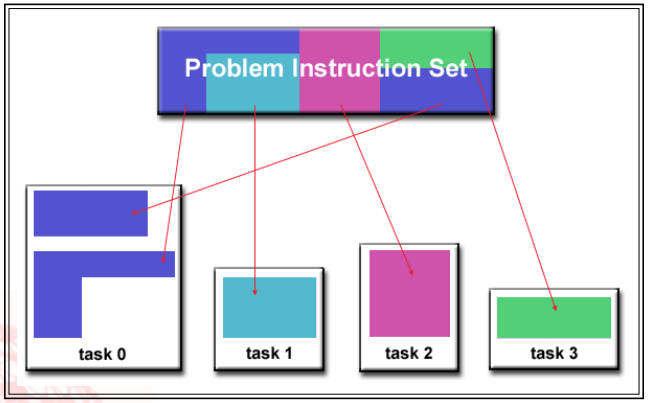
Domain decomposition - 1



Domain decomposition - 2

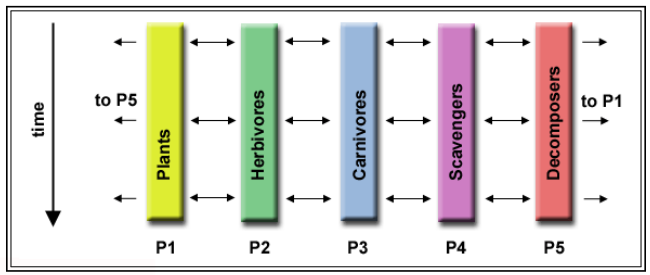


Functional decomposition



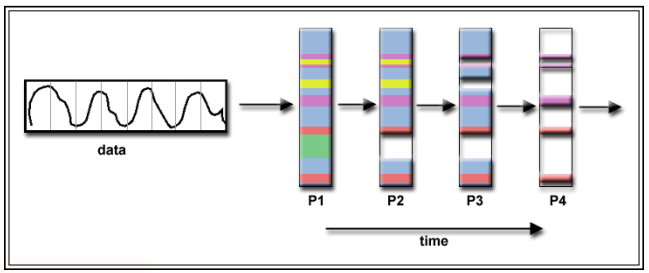
Functional decomposition: example - 1

Modelling an ecosystem



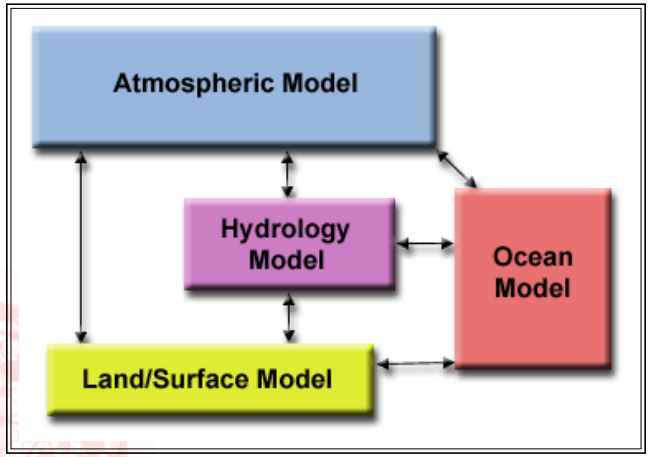
Functional decomposition: example - 2

Signal processing in pipeline



Functional decomposition: example - 3

Climate Modeling



Plan

Communication

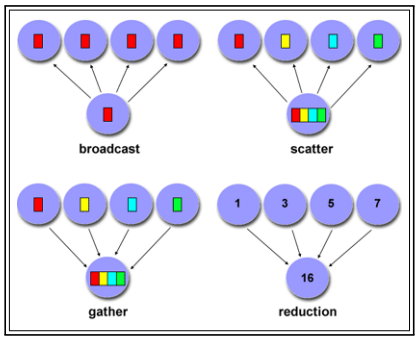
- You DON'T need communications
  - Partitioning an image in regions, with local computations
- . . . or. . . you DO need communication
- Factors to consider
  - will influence the programmer/designer choices

Communication: the factors - 1

- Cost of communications
  - Cycles and resources that could be used for computation are instead used to package and transmit data.
  - May require some synchronization (potential bottleneck)
  - Competition for limited resources (bus)
- Latency vs. Bandwidth
  - packaging many small msgs in a single large msg can be more efficient
- Visibility of communications (explicit or implicit)
- Synchronous vs. asynchronous communications (blocking/non-blocking)

Communication: the factors - 2

- Scope of communications: point-to-point or collective
- Different operations:
  - broadcast
  - scatter
  - gather
  - reduction

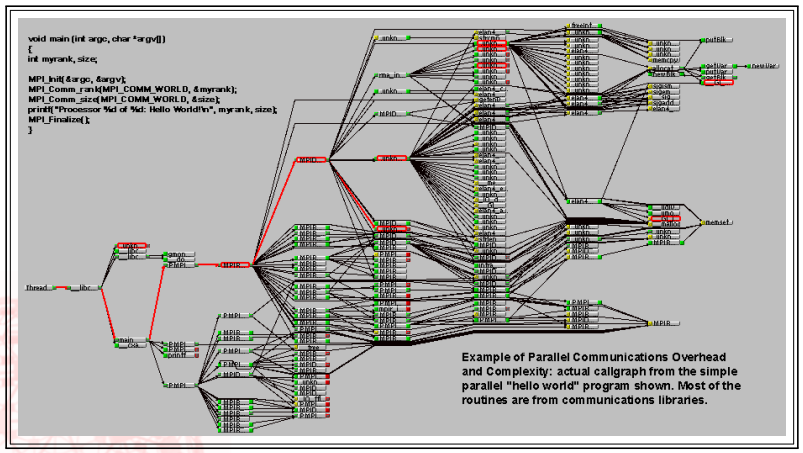


Communication: the factors - 3

- Efficiency of communications
  - Efficient implementations on a given platform
  - Efficiency of asynch. vs synch.
  - Different network media: which is the best?

Communication: the factors - 4

Communication complexity



Synchronization

- Barrier: Each task performs its work until it reaches the barrier.
  - It then stops, or "blocks"
- Lock/semaphore: used to serialize (protect) access to global data or a section of code
- Synchronous communication operations

Data dependencies

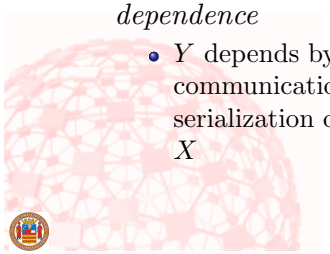
- A dependence exists between program statements when the order of statement execution affects the results of the program
- A data dependence results from multiple use of the same location(s) in storage by different tasks
- One of the primary inhibitors to parallelism

Dependencies: examples

- *Loop carried data dependence*: $A[j-1]$ must be computed before $A[j]$
  - if the two values are on two different tasks, needs synchronization and communication
- *Loop independent data dependence*
  - Y depends by inter-tasks communications or serialization of writes on X

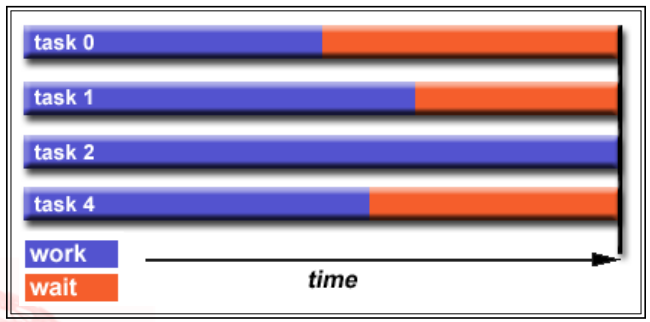
```
DO 500 J = MYSTART,MYEND
   A(J) = A(J-1) * 2.0
500 CONTINUE
```

```
task 1          task 2
------          ------

X = 2           X = 4
  .               .
  .               .
Y = X**2        Y = X**3
```



Load balancing

All tasks are kept busy all of the time

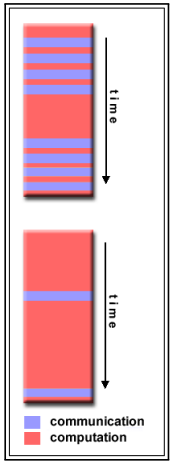


How to load balance

- Equally partition the work each task receives: sometime easy, sometime impossible
  - heterogenous hw, work and impredicibility of (external) load
- Dynamic work assignment: a scheduler where tasks request new work batches

Granularity

- Granularity: Computation / Communication Ratio
- *Fine-grain* parallelism
- *Coarse-grain* parallelism



Fine vs. Coarse

- Fine-grain:
  - Facilitates load balancing
  - High communication overhead
- Coarse-grain:
  - Implies more opportunity for performance increase (lower communication overhead)
  - Harder to load balance efficiently

Input/Output

- I/O operations are generally regarded as inhibitors to parallelism
- Expecially if conducted on the network (NFS)
- Parallel file systems, some optimized for particular applications
  - Example: Google File system
- Guidelines:
  1. (a.k.a. "*The* rule") Reduce overall I/O as much as possible
  2. Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks
  3. Create unique filenames for each task's input/output file(s)

Parallel computation cost

- Complexity (design, coding, debugging, tuning, manteinance)
- Portability: improved but still a problem (OSs, HW)
- Scalability: many factors (memory, bandwidth, latency, processor number/type, libraries, etc.)

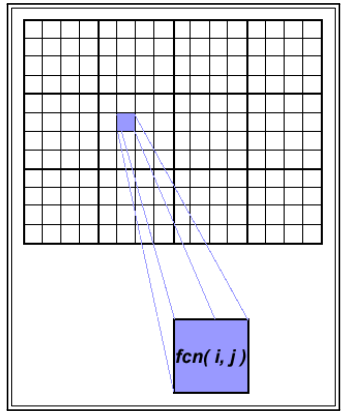
Plan

The problem: array processing

- Computation on each array element being independent from other array elements
- Sequential code:

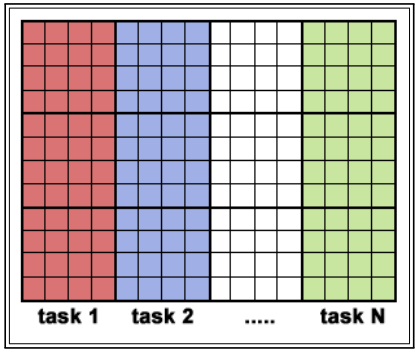
```
do j = 1,n
do i = 1,n
  a(i,j) = fcn(i,j)
end do
end do
```

- Embarassingly parallel



Array processing: solution 1

- Each processor owns a portion of an array (subarray).
- Independent calculation ⇒ no communication
- Partitions suggested by cache efficiency



Solution 1 with SPMD

Single Program Multiple Data model (like SIMD)

```
find out if I am MASTER or WORKER

if I am MASTER

  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array

  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
  end do

  send MASTER results

endif
```

Array processing: solution 2

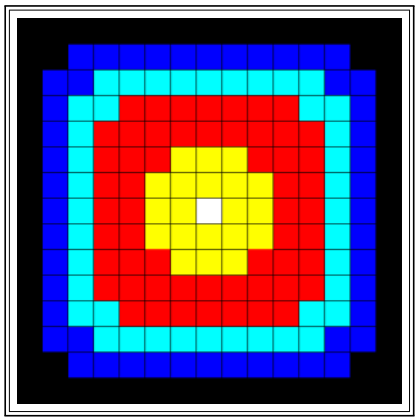
- Limits of static load balaning
  - non efficient with heterogeneous processors
  - when evaluated function is quick for some input (e.g. 0s) and costly on other input, distributing equal-sized portions may not suffice
- Improvement with pool of tasks
- Master: keep the pool of tasks and distributes to workers, collecting results
- Worker: repeat: ask a new task, execute it and send results to master
- Load balancing (granularity of tasks critical for performances)

Solution 2

```
find out if I am MASTER or WORKER

if I am MASTER

  do until no more jobs
    send to WORKER next job
    receive results from WORKER
  end do

  tell WORKER no more jobs

else if I am WORKER

  do until no more jobs
    receive from MASTER next job

    calculate array element: a(i,j) = fcn(i,j)

    send results to MASTER
  end do

endif
```

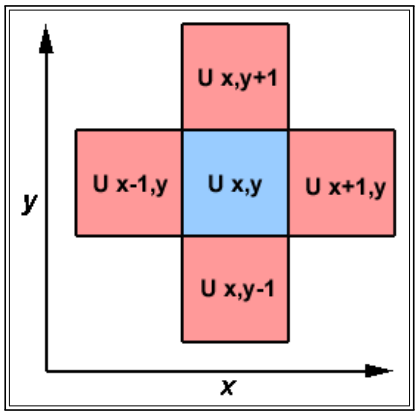
Second example - 1

- The equation describes temperature over time
- High in the middle and zero outside (at time 0)
- Goal: evaluate how it changes
- Communication among tasks is needed



Second example - 2

- Computations on neighbors' values
- $U_{x,y} = U_{x,y} + C_x \cdot (U_{x+1,y} + U_{x-1,y} - 2 \cdot U_{x,y}) + C_y \cdot (U_{x,y+1} + U_{x,y-1} - 2 \cdot U_{x,y})$

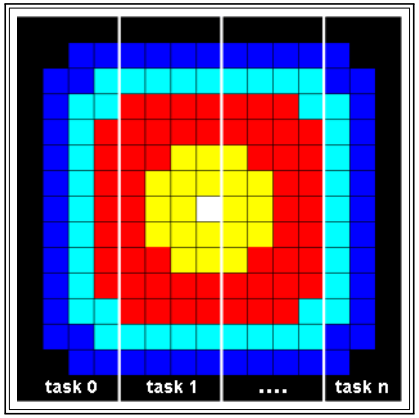


Serial code

```
do iy = 2, ny - 1
do ix = 2, nx - 1
  u2(ix, iy) =
    u1(ix, iy)  +
      cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
      cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
end do
end do
```


Solution 1

- SPMD solution
- Partitioned array
- Interior elements belonging to a task are independent of other tasks
- Border elements are dependent upon a neighbor task's data, necessitating communication.
- Master-worker



Solution 1 - code

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time
    send neighbors my border info
    receive from neighbors their border info

    update my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```

- SPMD
- Convergence shown by stability of results
- In the loop: send, receive, update array...

Critical view of solution 1

- Blocking communication: a bottleneck
- It can be improved with non-blocking communication
- The idea: update interior of own zone, while the communication of border data is occurring

Solution 2 - code

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time

    non-blocking send neighbors my border info
    non-blocking receive neighbors border info

    update interior of my portion of solution array
    wait for non-blocking communication complete
    update border of my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```

- Non-blocking communication is started...
- Then, go to the computations ...
- ...and terminates when data is received from border

Plan

Scalability

- **Scalability**: the property of a solution to a problem to mantain its *efficiency* as the *dimension* grows
- Some keywords to be addressed in the context of parallel programming:
  - Efficiency: speedup over the "corresponding" sequential solution
  - Dimension: processors number, type or interconnection; problem size (memory)
- Big-Oh notation for algorithms: scalability, but only in principle
  - what happens when you fill the current level of the memory hierarchy you are using
  - what happens when number of processors grows to infinity
  - ...

The most influential concepts

- Panel in 2000 IPDPS: find the most influential concepts in Parallel and Distributed Processing field, in the past millennium
- Participants: M. Theys, S. Ali, H.J. Siegel, M. Chandy, K. Hwang, K. Kennedy, L. Sha, K. Shin, M. Snir, L. Snyder, T. Sterling
- The process:
  - Proposal of candidates (≤ 10 per panelist)
  - Formulation of a list of candidates
  - Vote (panelists and audience)

Out of the result top 10 list

All very important and interesting concepts

- Cellular automata
- Client-server
- PRAM
- Priority inversion
- Neural networks
- RPC
- Zero
- Quantum computing
- ...

The result

- 10. Multithreaded (lightweight) program execution
- 9. Cluster computing
- 8. Message passing and packet switching
- 7. Load balancing
- 6. Synchronization (including semaphores)
- 5. Multiprogramming
- 4. Divide and conquer
- 3. Pipelining
- 2. Arpanet and Internet
- **1. Amdahl's law and scalability**

How to conjugate "Scalability"

- Wide area network:
  - cloud computing
- Local area network:
  - cluster computing
- Personal area:
  - desktop/mobile multicore processors
- Is scalability a hopeless battle? (Ask Amdahl...)