Parallel programming

Luis Alejandro Giraldo León

Topics

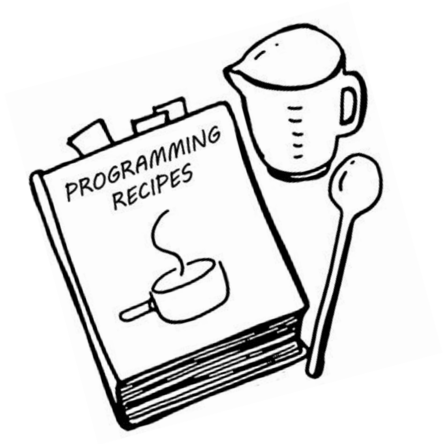
1. Philosophy
2. KeyWords
3. Parallel algorithm design
4. Advantages and disadvantages
5. Models of parallel programming
6. Multi-processor architectures
7. Common Languages
8. References

Philosophy

VEGETABLES
juicy
scrumptious
FRESH
tasty
DELECTABLE

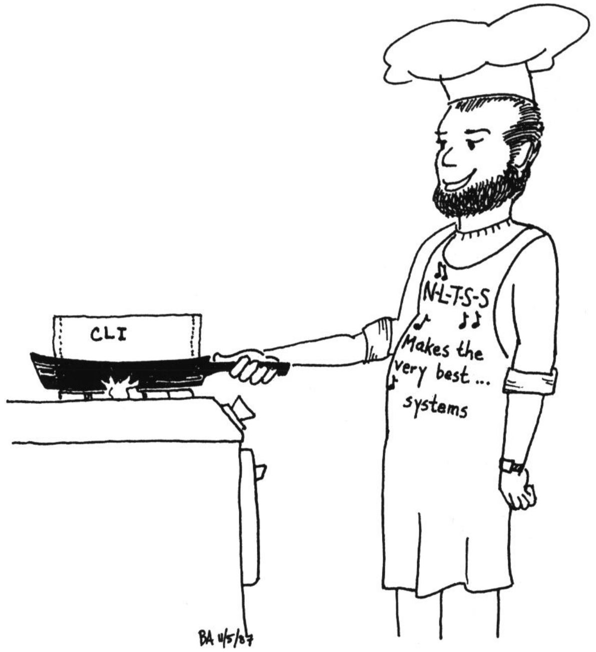
© MARK ANDERSON
WWW.ANDERTOONS.COM
CHEDDAR
SHARP
EXTRA SHARP
RAZOR SHARP
ANDERSON

Parallel vs. Asynchronous



*"Parallel programming is to use more and more threads effectively.
Asynchronous programming is to use less and less threads effectively."*

"Parallelism is one technique for achieving asynchrony, but asynchrony does not necessarily imply parallelism." -Eric Lipert



KEY WORDS

Secuencial programming

Parallel programming

Thread

Task

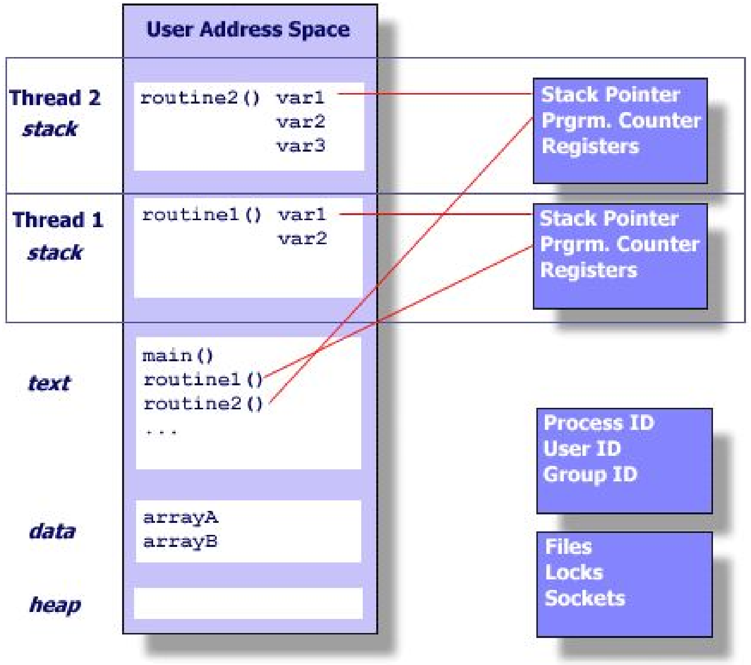
Pipelining

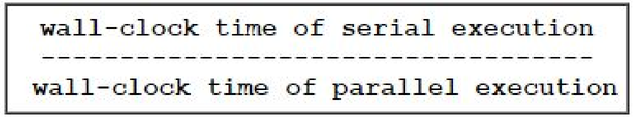
Shared Memory

Distributed Memory

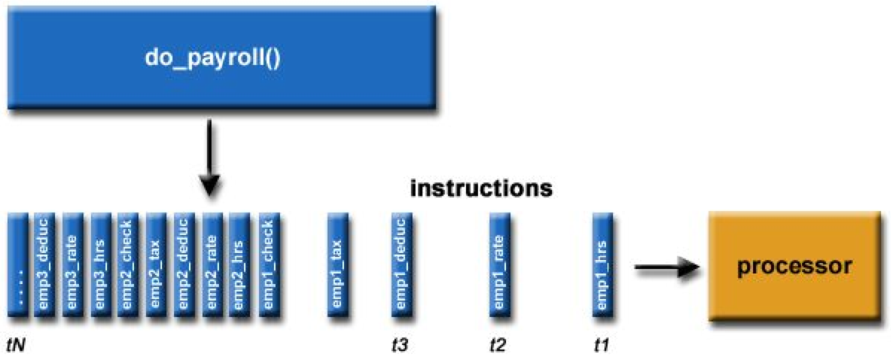
Speedup

Parallel Overhead

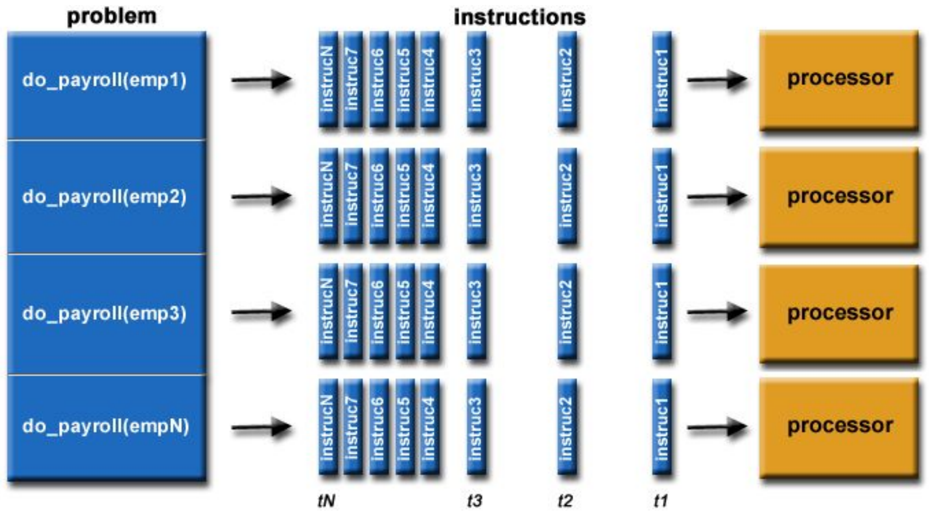



```
 wall-clock time of serial execution
 -----------------------------------
wall-clock time of parallel execution
```

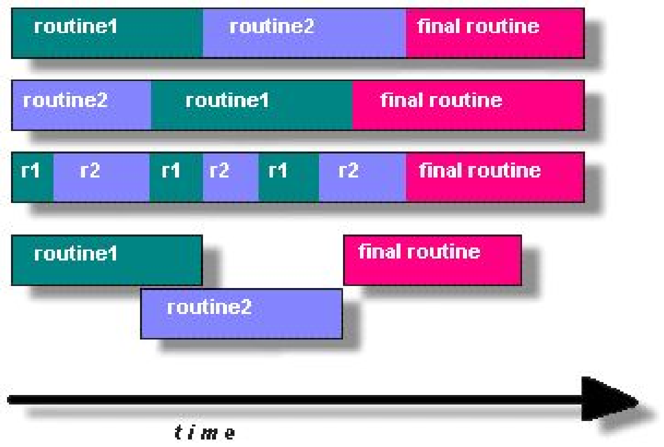
Goals

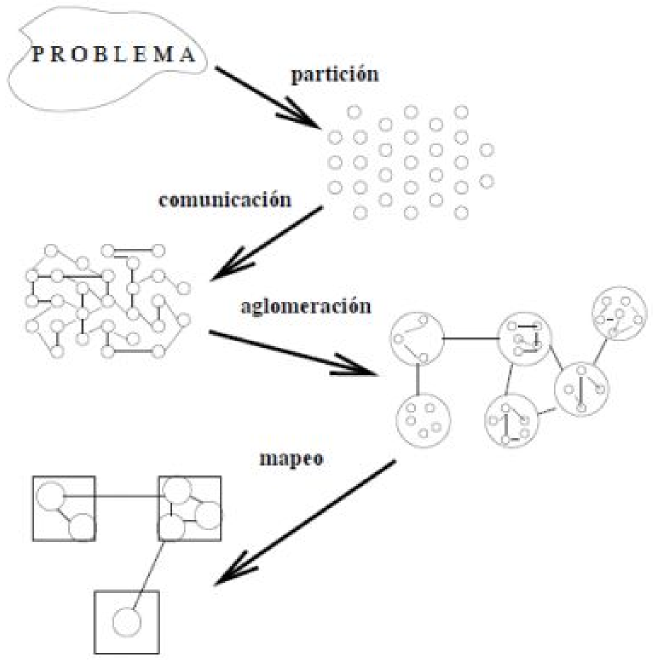


- Be broken apart into discrete pieces of work that can be solved simultaneously;
- Execute multiple program instructions at any moment in time;
- Be solved in less time with multiple compute resources than with a single compute resource.



Parallel algorithm design





Advantages and disadvantages

- SAVE TIME AND/OR MONEY
- SOLVE LARGER / MORE COMPLEX PROBLEMS
- Threads also have their own private data

- The primary intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required. For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.
- The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.
- For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.
- The algorithm may have inherent limits to scalability.
- Programmers are responsible for synchronizing access (protecting) globally shared data.

Models of parallel programming

1. Parallel Computing
2. Distributed Computing
3. Hybrid Distributed-Shared computing

Shared Memory

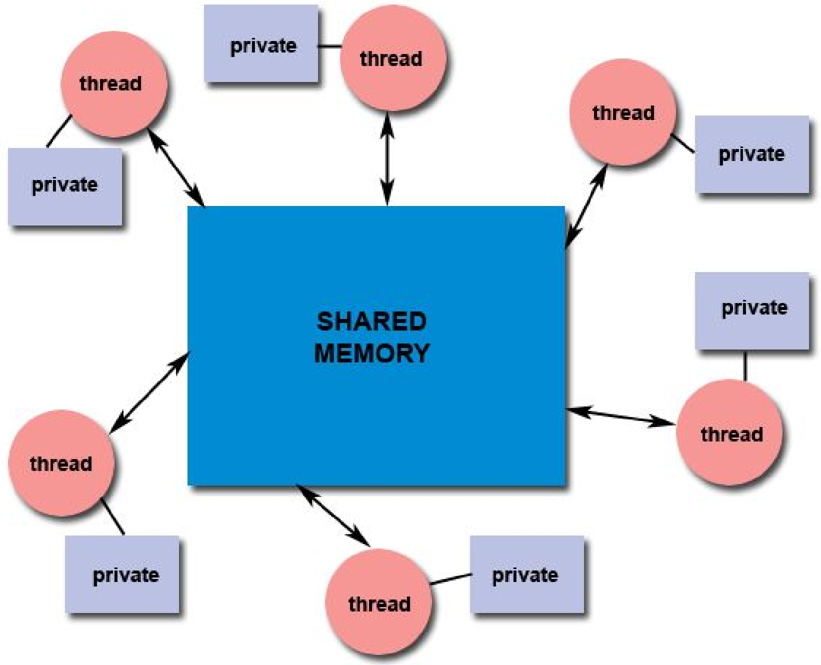
Sharing resources

- ***Manager/worker:***
- ***Pipeline:***
- ***Peer:***

Advantages and disadvantages parallel computing

1. Facilidad al compartir datos
2. Arquitectura flexible
3. Un solo espacio de direccionamiento la comunicación entre procesadores

1. Difícil de escalar (incrementa el tráfico en la memoria compartida).
2. Sincronización es responsabilidad del programador

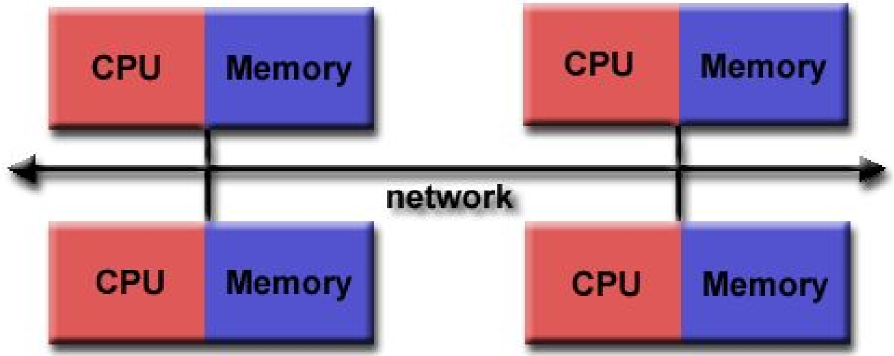


Distributed Memory

Cache coherency does not apply.

Send messages

Own local memory



Advantages and disadvantages distributed computing

1. Scalable memory
2. Each processor can rapidly access its own memory without interference and without the overhead
3. Cost effectiveness

1. The code and data must be physically transferred to the local memory of each node before execution.
2. The results have to be transferred from the nodes to the host system.
3. The programmer is responsible for the data communication between processors.
4. Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.

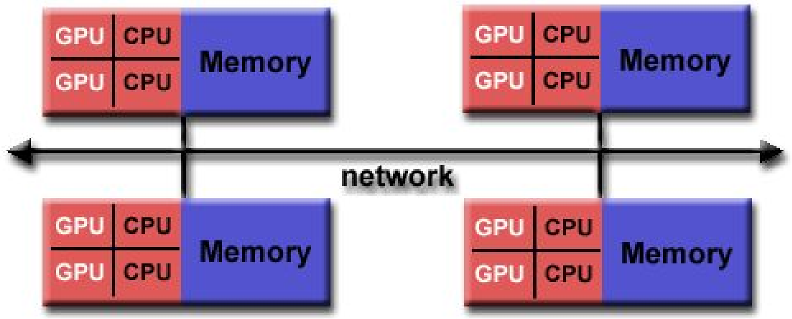
Hybrid Distributed-Shared Memory

The largest and fastest computers

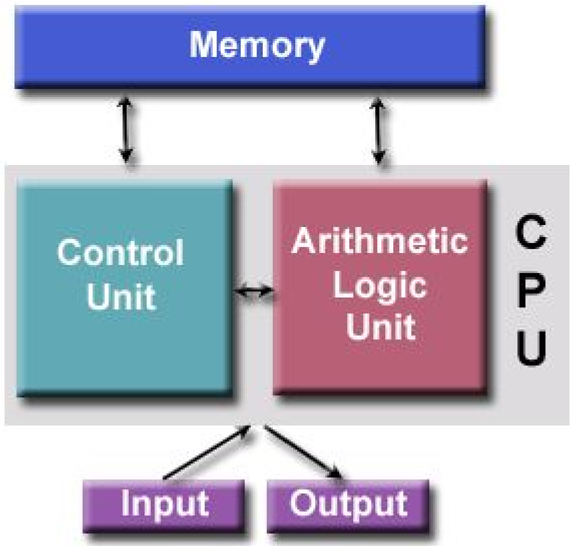
Advantages in common of the architectures.

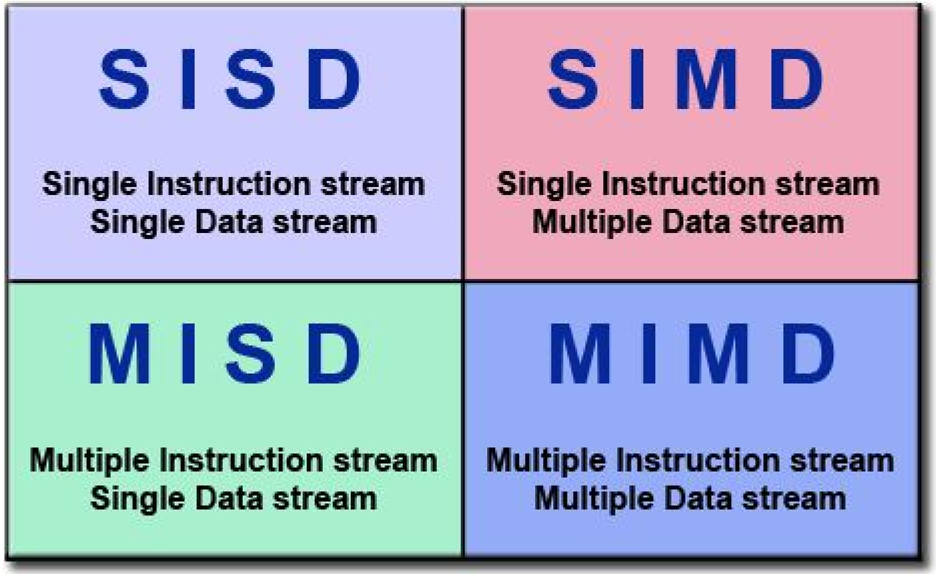
Increased scalability is an important advantage

Increased programmer complexity is an important disadvantage



Multi-processor architectures



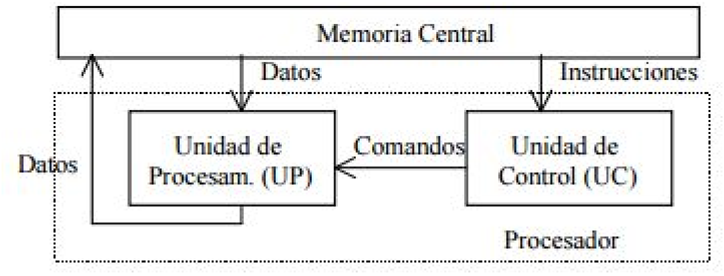


SISD Single Instruction stream Single Data stream	**SIMD** Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	**MIMD** Multiple Instruction stream Multiple Data stream

Clasificación de Flynn

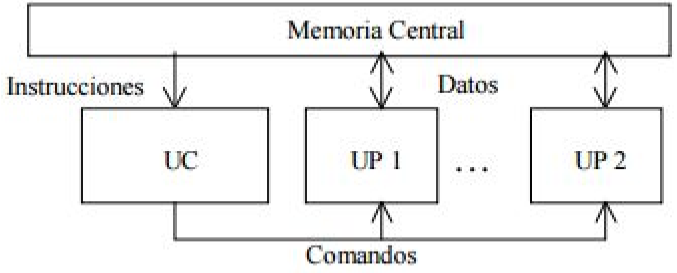
SISD (Single Instruction Stream, Single Data Stream)

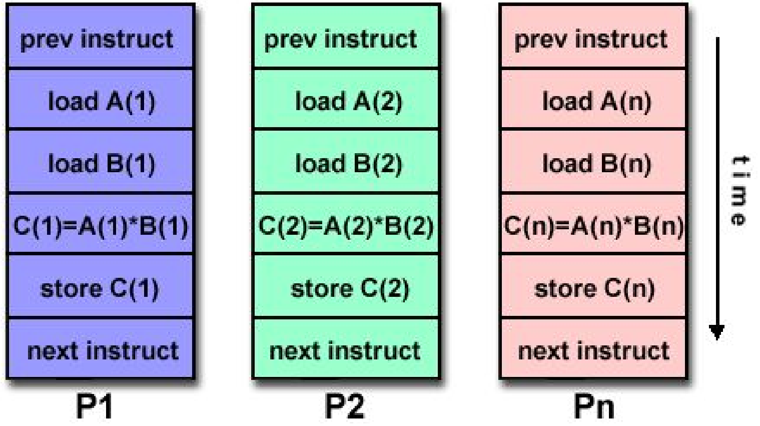
- Deterministic execution
- This is the oldest type of computer
- Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.



SIMD (Single Instruction Stream, Multiple Data Stream)

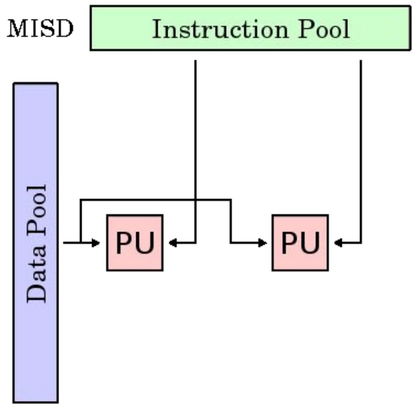
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Ex. arrays sum

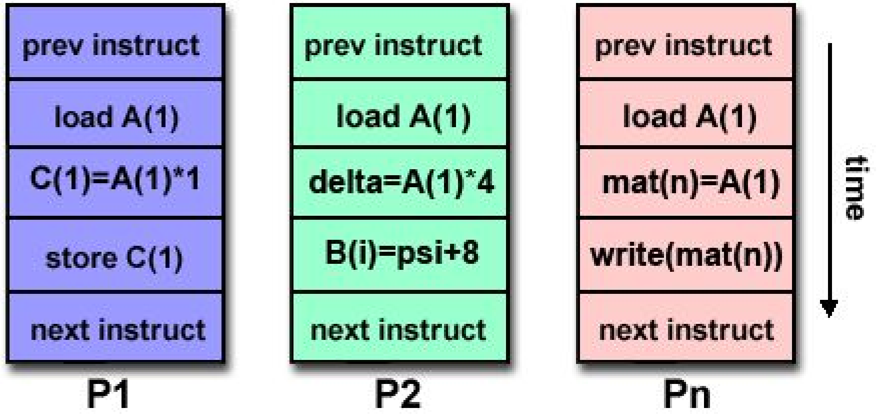




Multiple Instruction, Single Data (MISD):

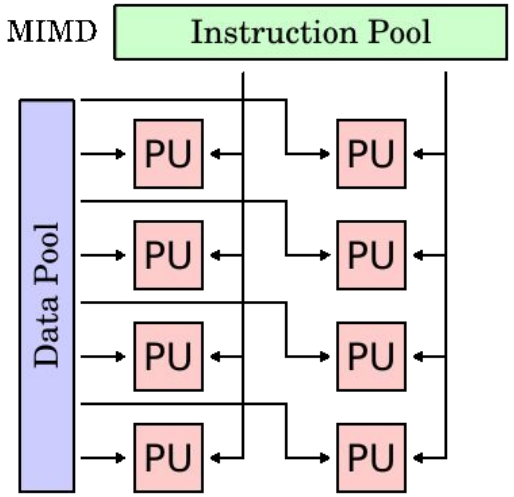
- Few (if any) actual examples of this class of parallel computer have ever existed.
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.

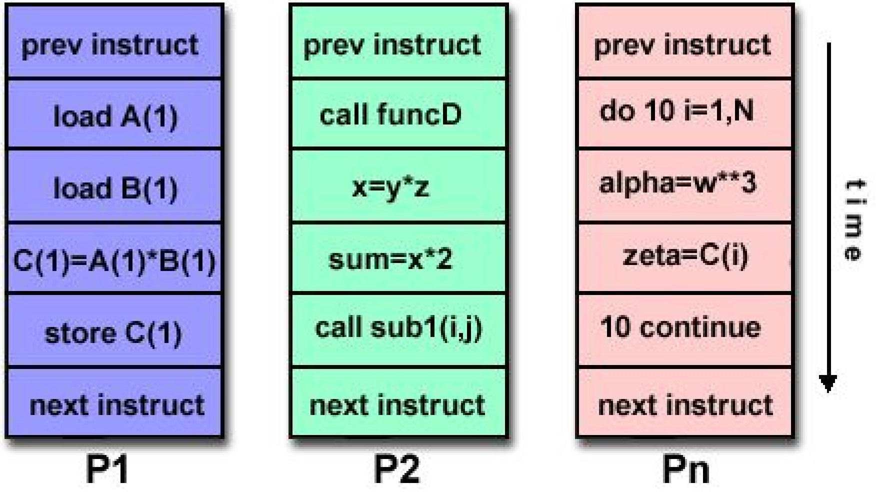




Multiple Instruction, Multiple Data (MIMD)

- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components



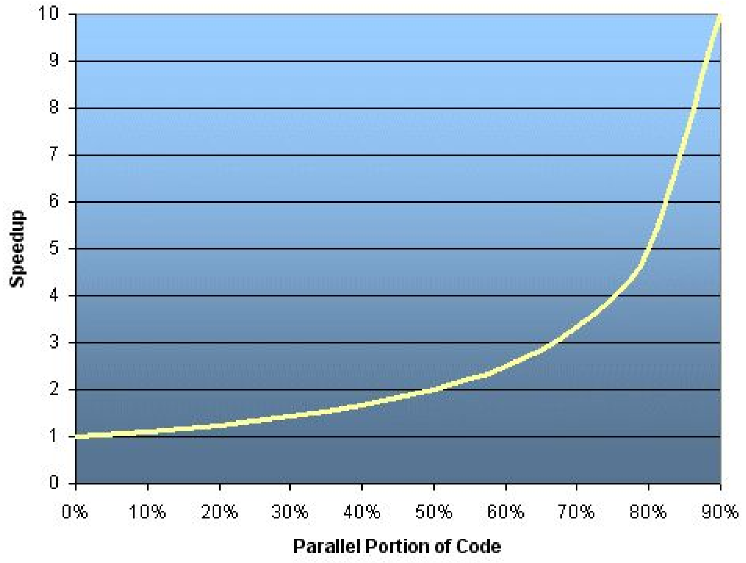


Designing Parallel Programs

Speedup

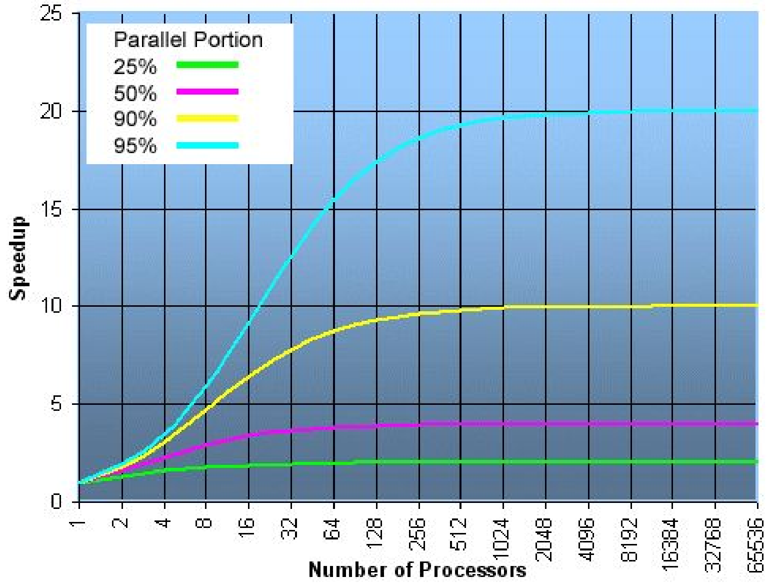
Amdahl's Law

$$\text{speedup} = \frac{1}{1 - P}$$



$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1,000	1.99	9.91	90.99
10,000	1.99	9.91	99.02
100,000	1.99	9.99	99.90



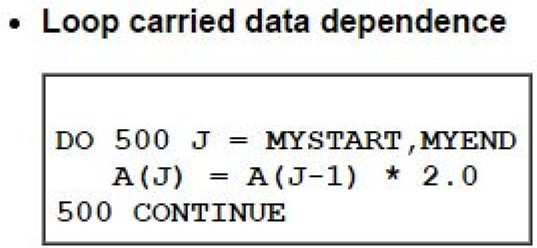
Synchronization

“Often requires "serialization" of segments of the program”

1. Barrier
2. Lock / semaphore
3. Synchronous communication operations

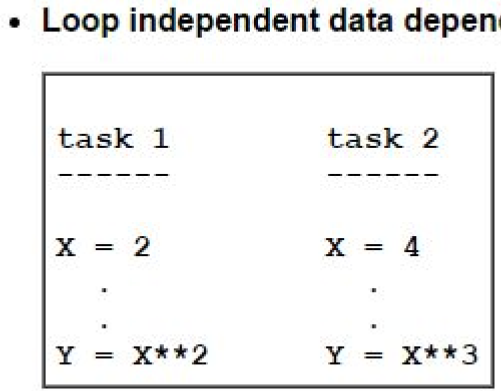
Dependency

Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.



- **Loop carried data dependence**

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



- **Loop independent data dependence**

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

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

Example 1

Is this problem able to be parallelized?

How would the problem be partitioned?

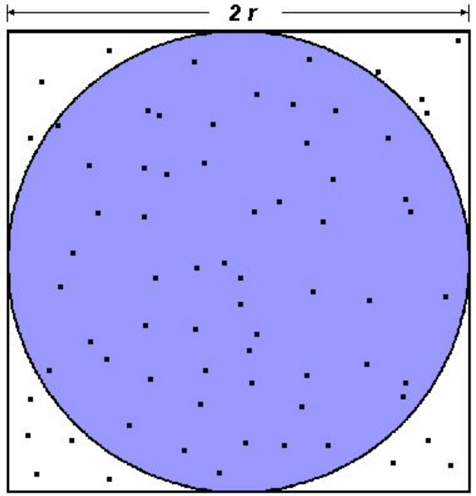
Are communications needed?

Are there any data dependencies?

Are there synchronization needs?

Will load balancing be a concern?

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```

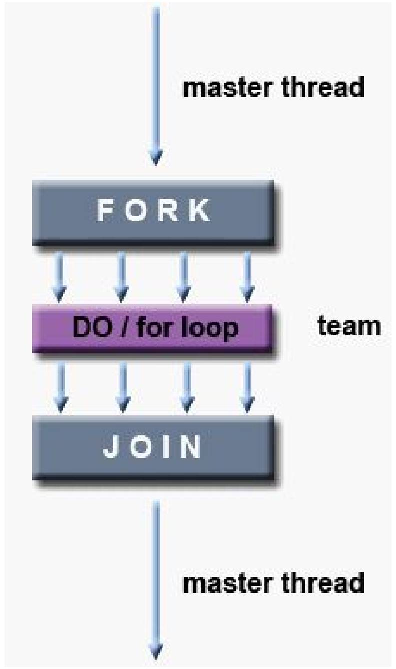


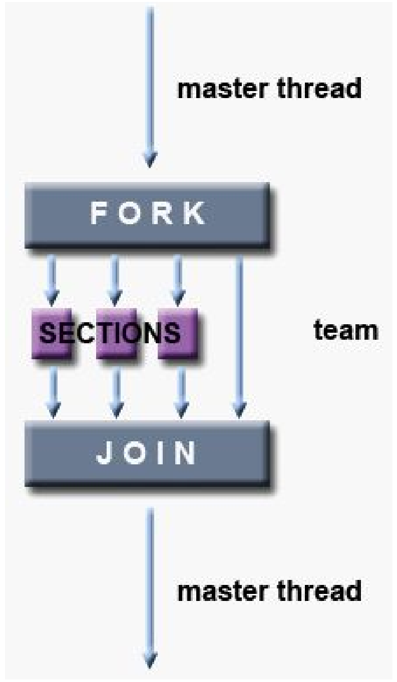
Common Languages

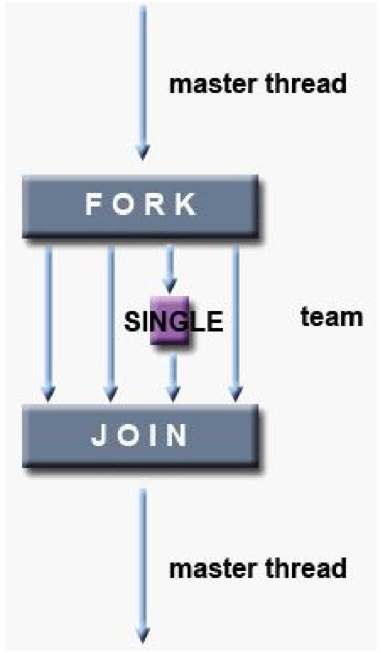
NVIDIA® CUDA®

OpenMP®

OpenMP


master thread
FORK
DO / for loop
team
JOIN
master thread


master thread
FORK
SECTIONS
team
JOIN
master thread


master thread
FORK
SINGLE
team
JOIN
master thread

```c
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[]) {

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 region */

}
```

```c
#include <omp.h>
#define N 1000

main(int argc, char *argv[]) {

int i;
float 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];

    }  /* end of sections */

  }  /* end of parallel region */

}
```

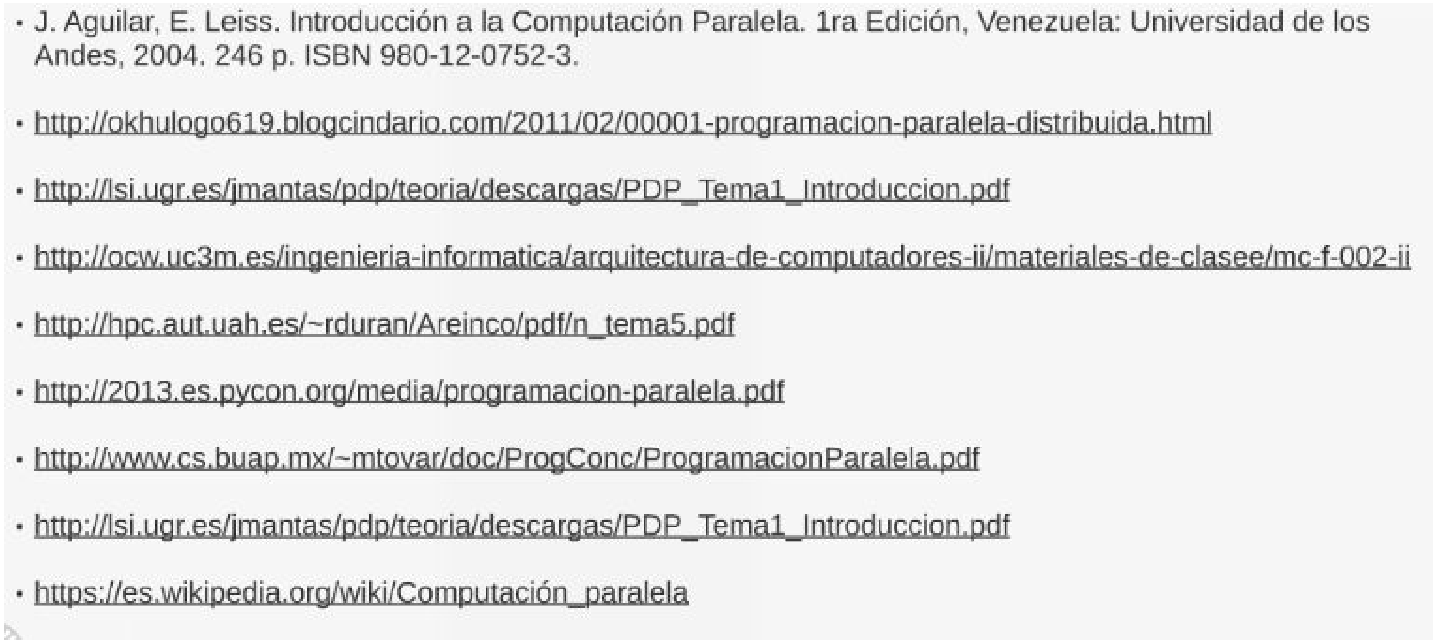
Single

Sincronización

C / C++
barrier parallel - upon entry and exit critical - upon entry and exit ordered - upon entry and exit for - upon exit sections - upon exit single - upon exit

```c
#include <omp.h>

main(int argc, char *argv[]) {

int x;
x = 0;

#pragma omp parallel shared(x)
  {

  #pragma omp critical
  x = x + 1;

  }  /* end of parallel region */

}
```

References



- J. Aguilar, E. Leiss. Introducción a la Computación Paralela. 1ra Edición, Venezuela: Universidad de los Andes, 2004. 246 p. ISBN 980-12-0752-3.
- http://okhulogo619.blogcindario.com/2011/02/00001-programacion-paralela-distribuida.html
- http://lsi.ugr.es/jmantas/pdp/teoria/descargas/PDP_Tema1_Introduccion.pdf
- http://ocw.uc3m.es/ingenieria-informatica/arquitectura-de-computadores-ii/materiales-de-clasee/mc-f-002-ii
- http://hpc.aut.uah.es/~rduran/Areinco/pdf/n_tema5.pdf
- http://2013.es.pycon.org/media/programacion-paralela.pdf
- http://www.cs.buap.mx/~mtovar/doc/ProgConc/ProgramacionParalela.pdf
- http://lsi.ugr.es/jmantas/pdp/teoria/descargas/PDP_Tema1_Introduccion.pdf
- https://es.wikipedia.org/wiki/Computación_paralela