Fundamentals of Parallel Processing

Dishant Ailawadi
IIT Roorkee
Tutor – Dr. Wellein and Dr. Gotz

8th Indo-German Winter Academy, Dehradun, December 2009

Outline

- Need for parallel computing
- Types of parallelism
- Parallel computer memory architectures
- Parallel programming models
- Key parallel processing steps & challenges
- Classes of parallel computers
- CUDA and NVIDIA GPUs
- Cluster Computing & MPI
- Cloud computing
- Summary

Need for Parallel Computing

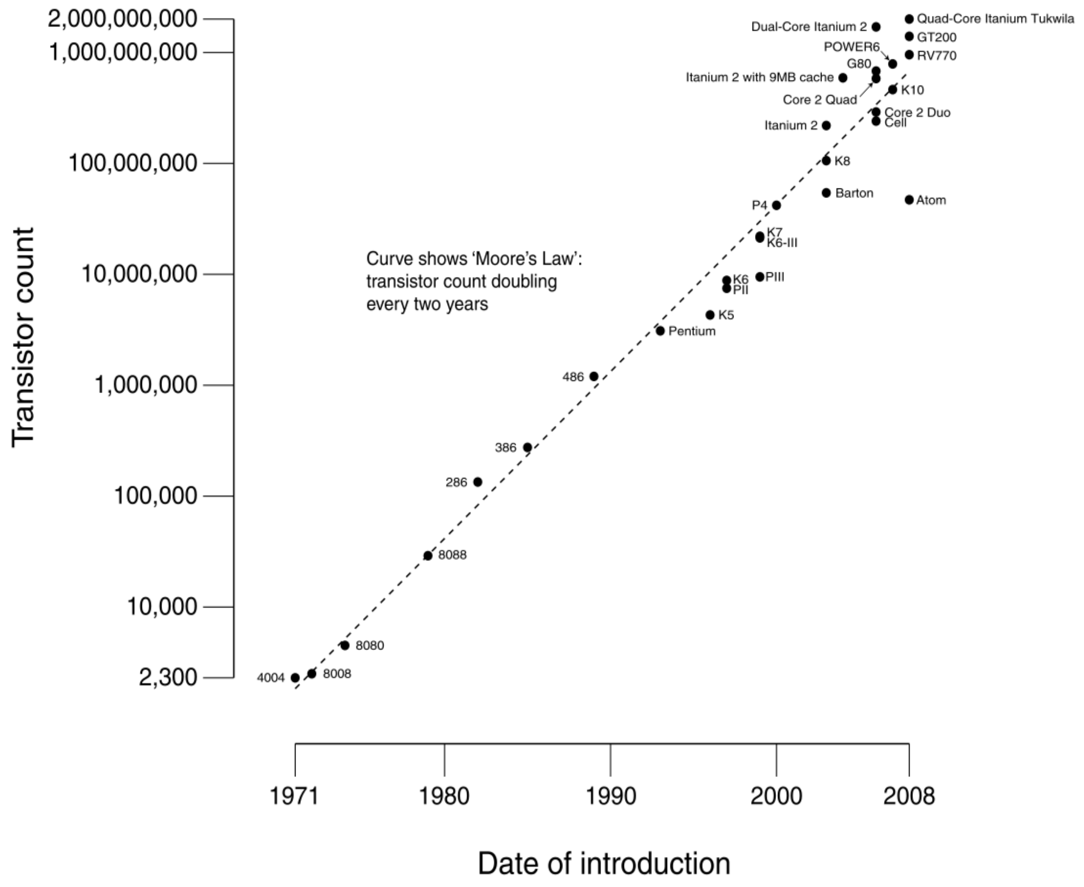
Moore's Law:

Chip performance doubles every 18-24 months

Limits to serial computing

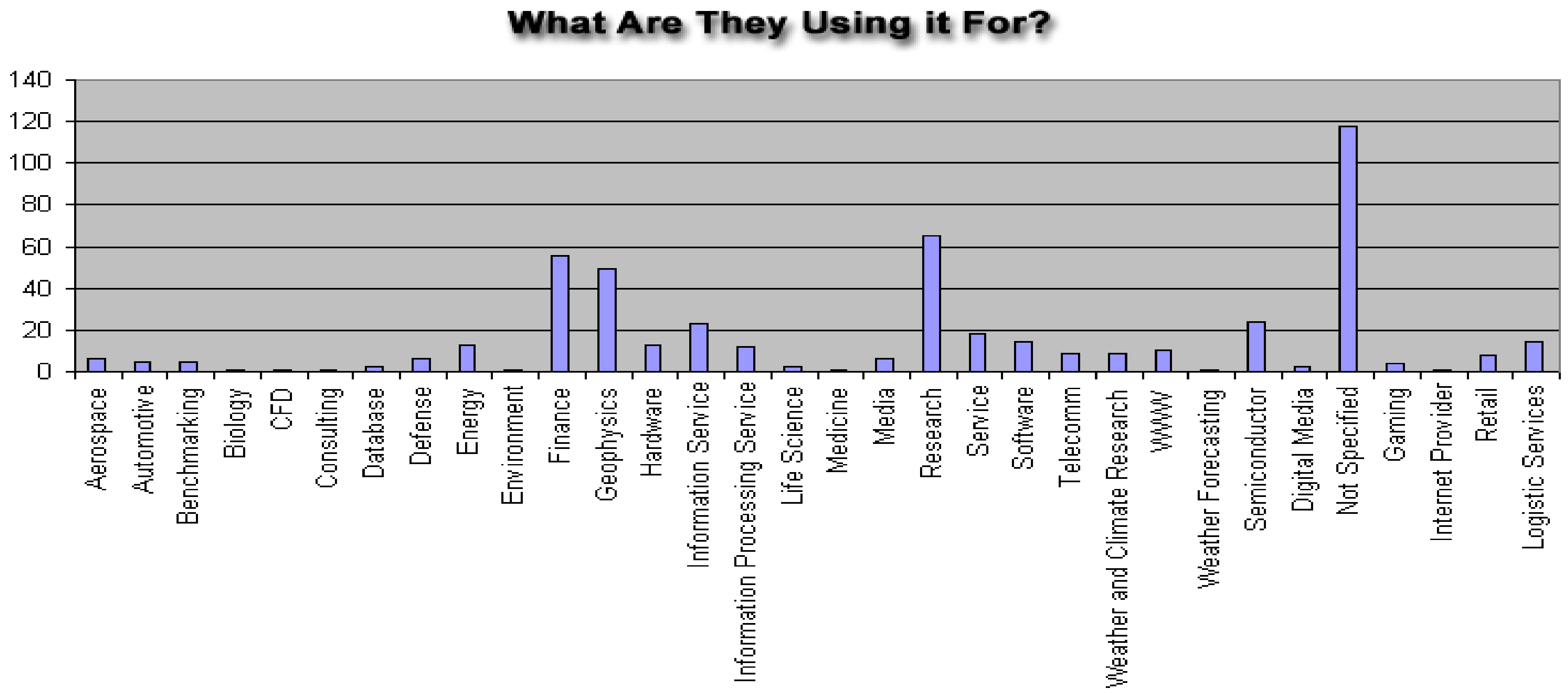
- Transmission speeds
- Limits to miniaturization
- Economic limitations
- Heating $P = C \times V^2 \times F$
- Frequency scaling

CPU Transistor Counts 1971-2008 & Moore's Law



Advantages

- Save time and/or money (Clusters)
- Overcoming memory constraints
- Solve larger problems (Web searching/CFD)
- Provide concurrency
- It’s the future of computing



Types of parallelism

Bit-level parallelism :

- Increase in word size from 1970s to 1986.
- An 8-bit processor requires two instructions to add 16 bit integer.

Instruction-level parallelism :

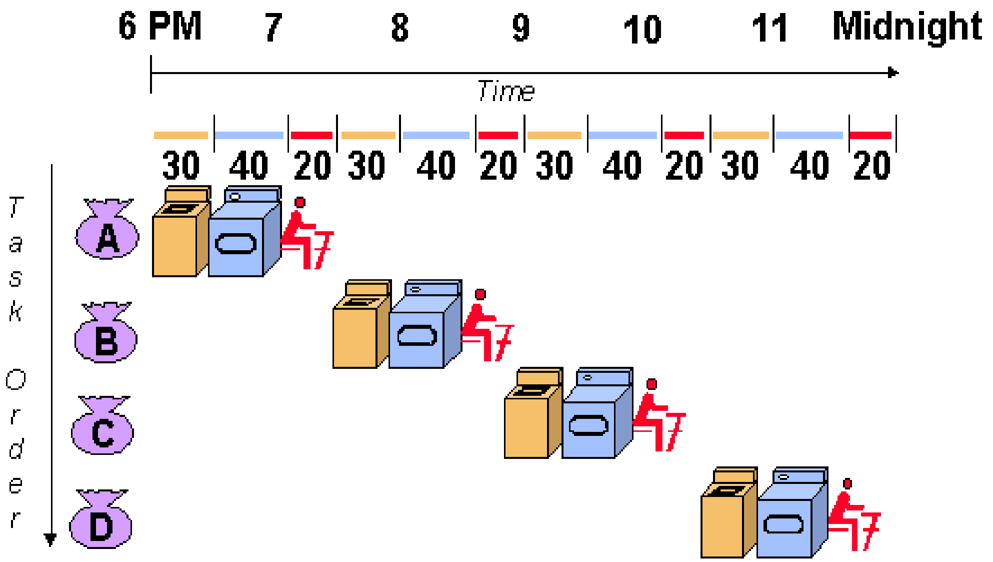
Multistage instruction pipelines.

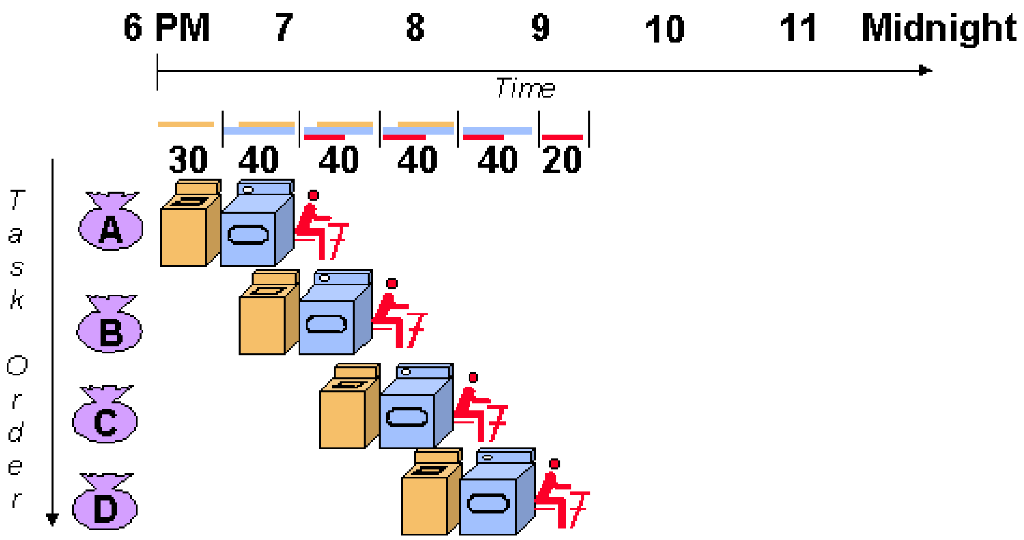
Data parallelism:

distributing the data across different computing nodes to be processed in parallel.(Same calculations).

Task Parallelism:

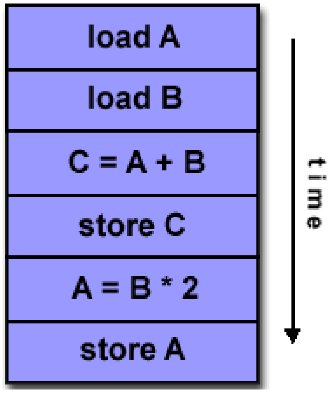
entirely different calculations can be performed on either the same or different sets of data



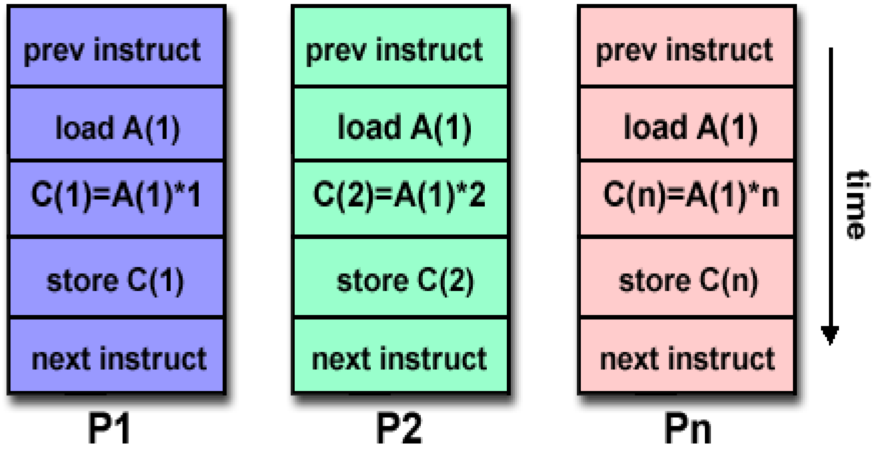


Flynn’s Classical Taxonomy

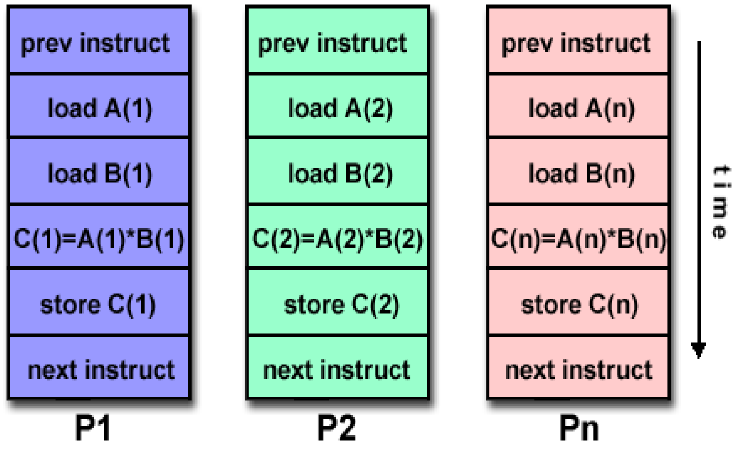
- SISD: A serial computer



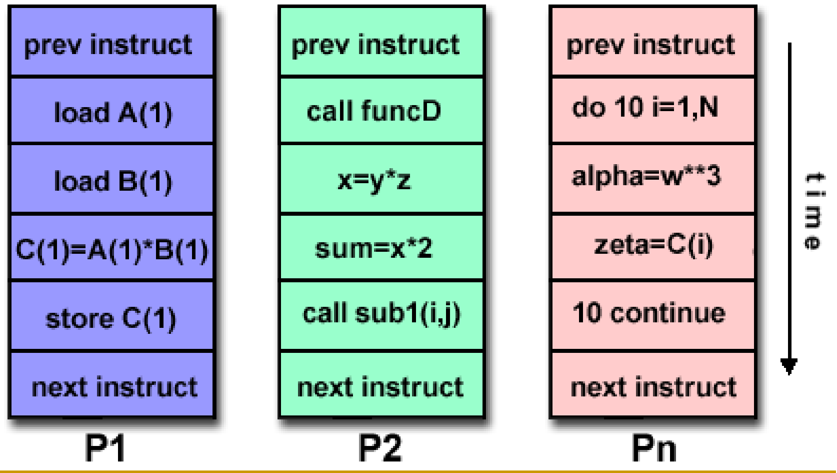
MISD: Cryptographic Decoding



- SIMD: GPUs



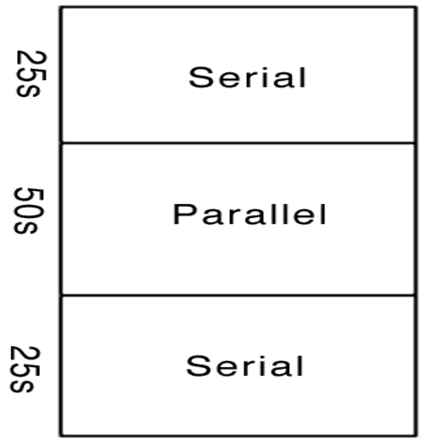
MIMD: Clusters, Supercomputers



Background

Amdahl's law:

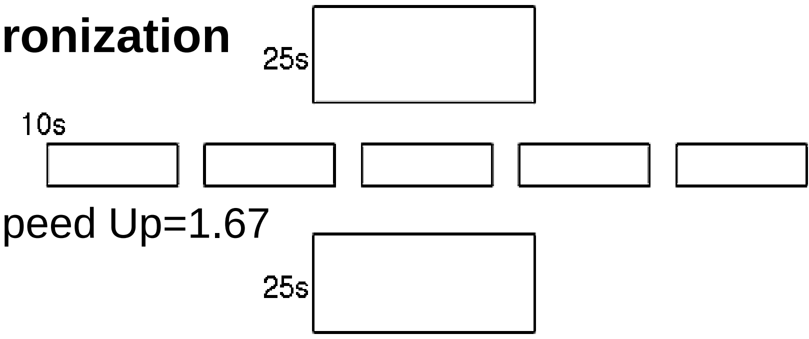
$$Speed\ Up = \frac{1}{(1-P)+\frac{P}{N}}$$



Dependencies:No program can run more quickly than the longest chain of dependent calculations .

Race conditions, mutual exclusion & synchronization

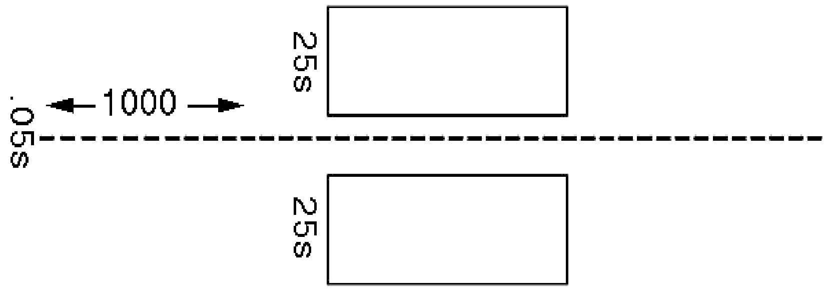
Processor 1	Processor 2
Read X	Read X
X=X+1	X=X+1
Write X	Write X



Speed Up=1.67

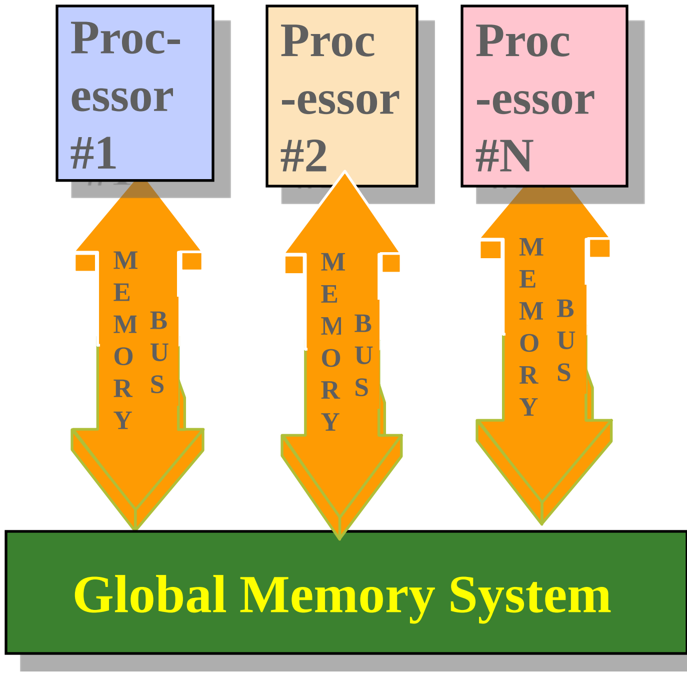
- **Fine-grained :** Excessive Comm.
- **Coarse-grained**
- **Embarrassing parallelism :** Rare

Easily parallelizable



Speed Up= 2

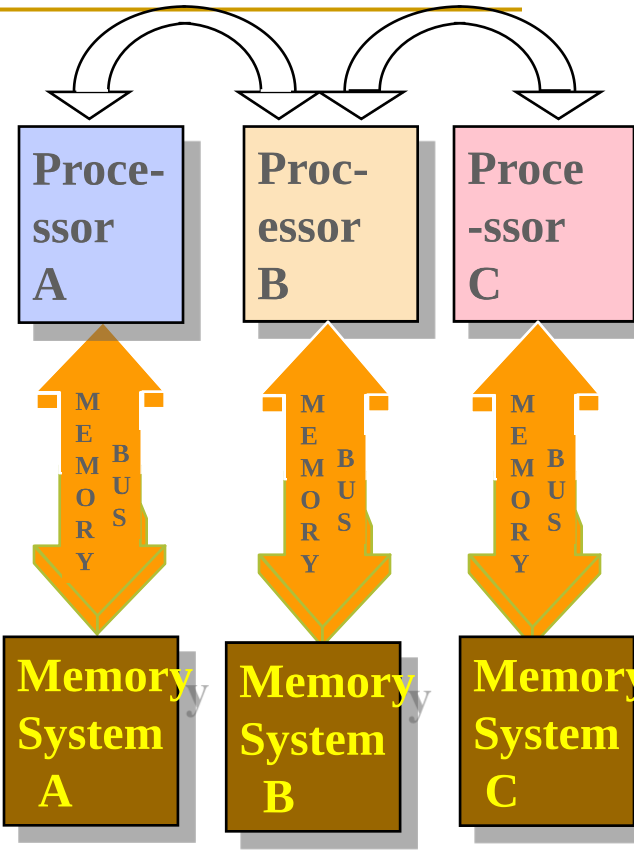
Shared Memory Systems



Source Processor writes data to GM & destination retrieves it.

- Easy to build, conventional OS of SISD can easily be ported
- Limitation : reliability & expandability. A memory component or any processor failure affects the whole system.
- Adding more processors increase traffic.
- Global address space proves user-friendly to programmers.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.
- Expensive: Difficult to design when more processors
- Programmer responsibility to insure "correct" access of global memory.(Race conditions)

Distributed Memory



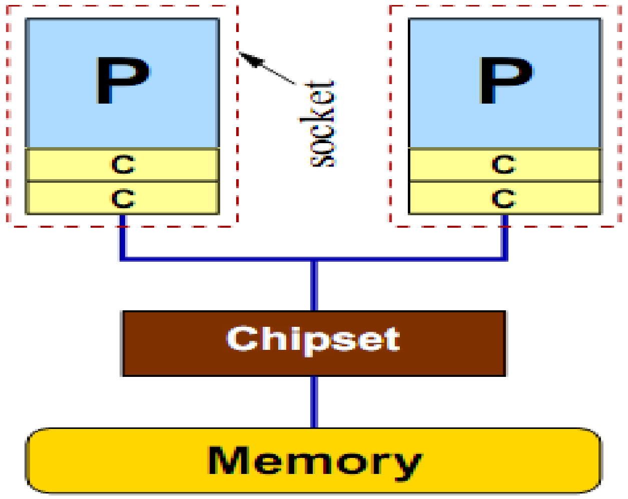
- Network can be configured to Tree, Mesh, Cube.
- Unlike Shared MIMD
  - ➔ **easily/ readily expandable**
  - ➔ **Highly reliable (any CPU failure does not affect the whole system)**
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the shelf processors and networking.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- The programmer is responsible for many of the details associated with data communication between processors.

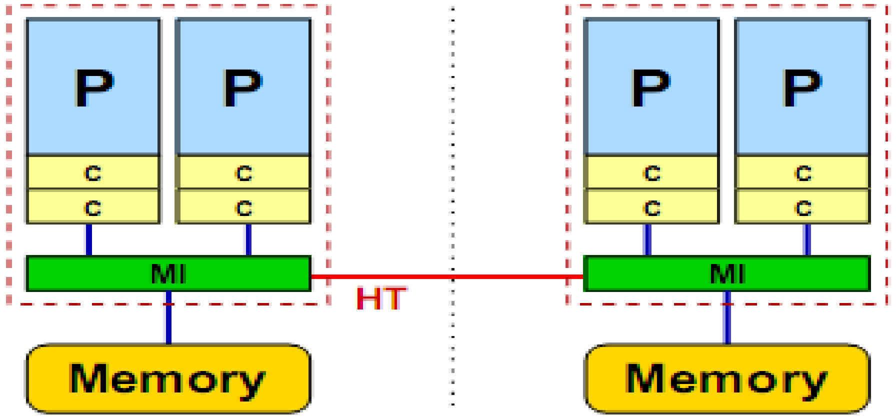
Uniform memory access(UMA)

- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

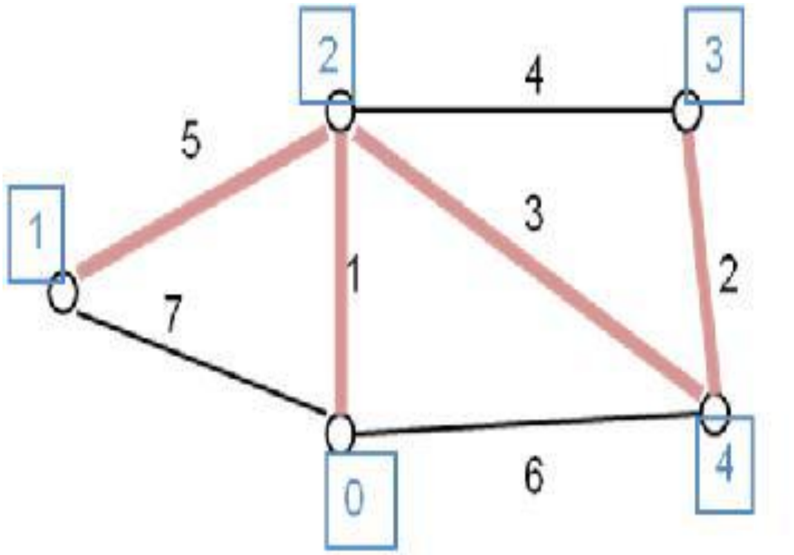
Non-Uniform Memory Access

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA





Key Parallel Programming Steps



Minimum Spanning Tree:

Baruvka's Algorithm:

- Assign Each node and it's edges to a processor
- Find shortest Edge from that node
- Combine connected components. Repeat

- Parallel computing requires

  To find the concurrency in the problem

1) To structure the algorithm so that concurrency can be exploited
2) To implement the algorithm in a suitable programming environment

Kruskal Algo: Select shortest edge of graph if it doesn't make cycle. Repeat.

Sequence- (0,1) (3,4) (2,4) (1,2)

Prim's Algo: Add chosen node to the set. Select edge with least weight originating from the set. Add edge to MST. Repeat.

Sequence- (0,1) (2,4) (3,4) (1,2)

Challenges of Parallel Programming

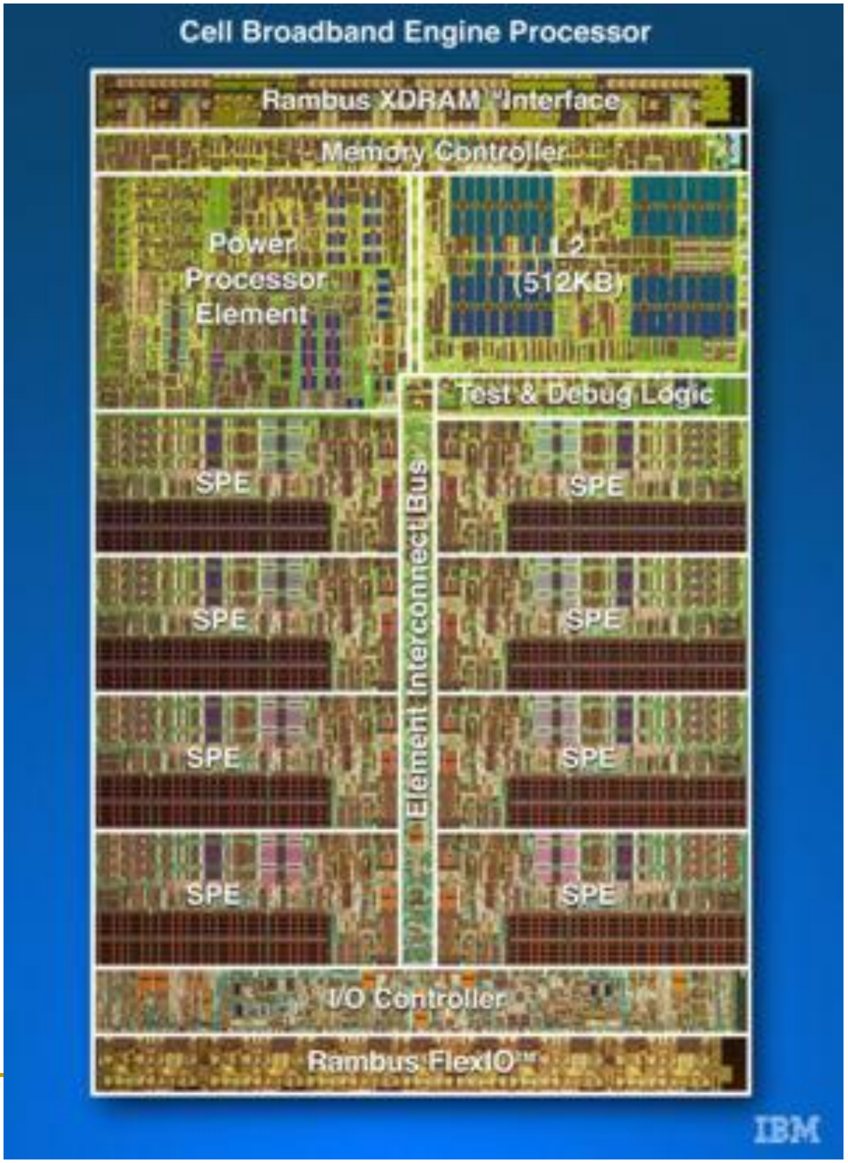
- Finding and exploiting concurrency often requires looking at the problem from a non-obvious angle
  - Computational thinking
- Dependences need to be identified and managed
  - The order of task execution may change the answers
    - Obvious: One step feeds result to the next steps
    - Subtle: numeric accuracy may be affected by ordering steps that are logically parallel with each other
- Performance can be drastically reduced by many factors
  - Overhead of parallel processing
  - Load imbalance among processor elements
  - Inefficient data sharing patterns
  - Saturation of critical resources such as memory bandwidth

Classes of Parallel Computers

- **Multi-core computing**

  A multi-core processor is an integrated circuit to which two or more processors have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks

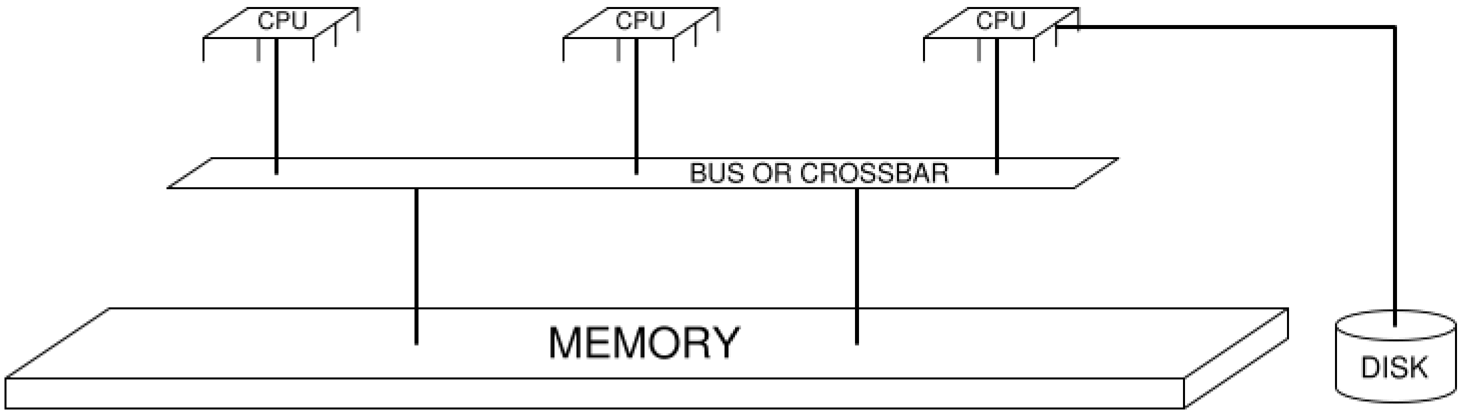
1. cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods
2. All cores are identical in *homogeneous* multi-core systems and they are not identical in*heterogeneous* multi-core systems.



Classes of Parallel Computers

- Symmetric multiprocessing (SMP)

  SMP involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory. Most common. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.
- Allow any processor to work on any task no matter where the data for that task are located in memory with proper operating system support,
- SMP systems can easily move tasks between processors to balance the workload efficiently.

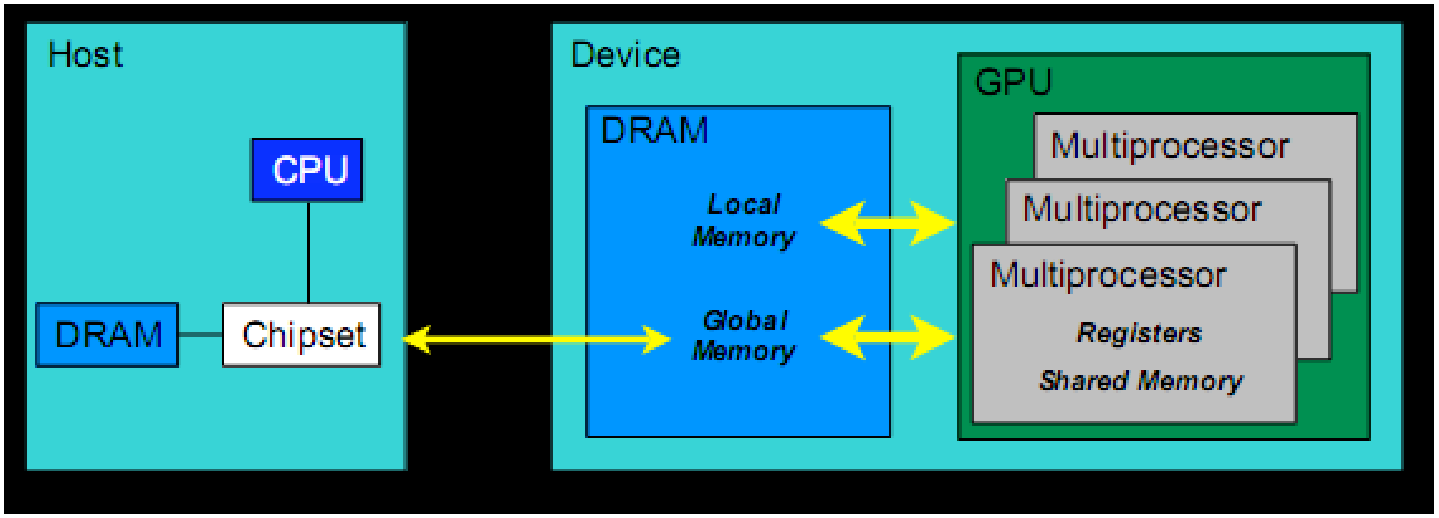


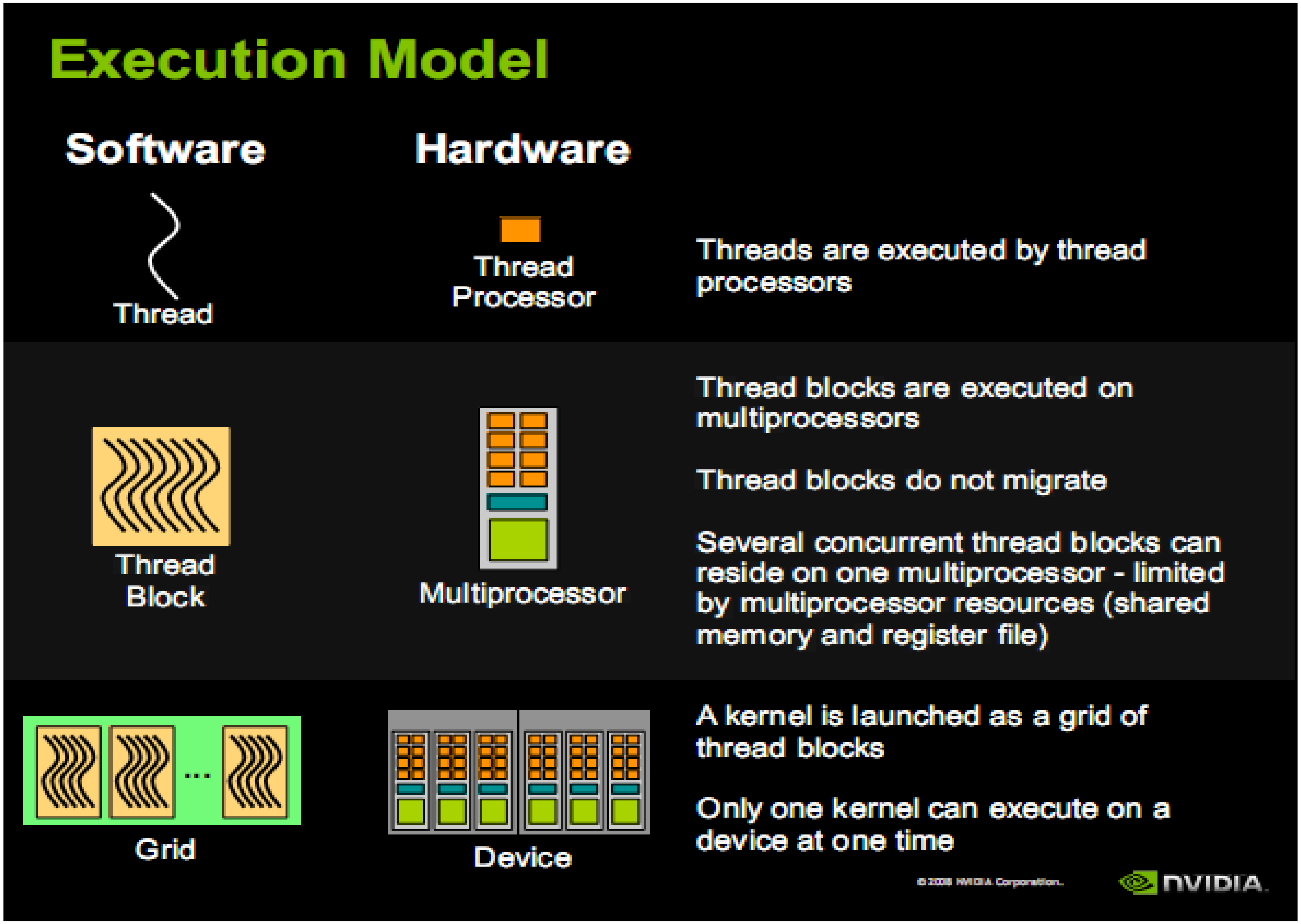
Classes of parallel computers

- A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network.
- Clusters are composed of multiple standalone machines connected by a network.
- MPP: a single computer with many networked processors. Have specialized interconnect networks. Each CPU has it's own memory & copy of OS. Eg. Blue Gene
- Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet .deals only with embarrassingly parallel problems.
- General-purpose computing on graphics processing units is a fairly recent trend in computer engineering research. Act as co-processors.

CUDA and NVIDIA GPUs

- CUDA is a software environment for parallel computing with minimal extensions to C/C++.
- Parallel portions of an application are executed on device as kernels, executing many threads.
- Enables heterogeneous systems (CPU+GPU)





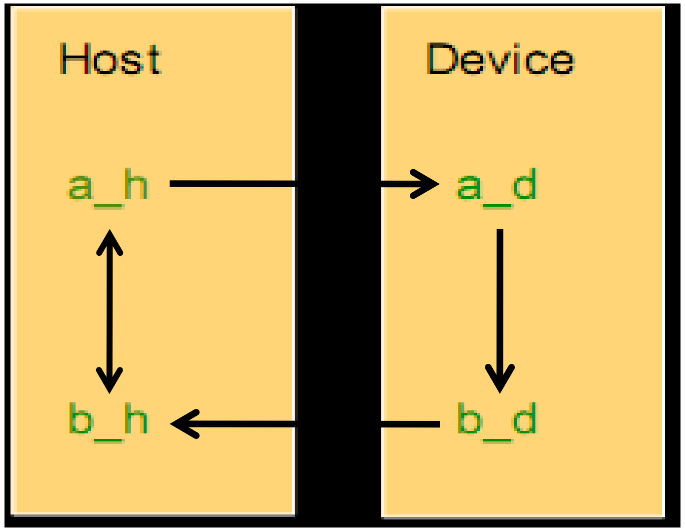
Execution Model

Software	Hardware	
Thread	Thread Processor	Threads are executed by thread processors
Thread Block	Multiprocessor	Thread blocks are executed on multiprocessors Thread blocks do not migrate Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
Grid	Device	A kernel is launched as a grid of thread blocks Only one kernel can execute on a device at one time

NVIDIA

Data Movement Example

```
int main(void) {
float *a_h, *b_h;  // host data
float *a_d, *b_d;  // device data
int N = 14, nBytes, i ;
nBytes = N*sizeof(float);
a_h = (float *)malloc(nBytes);
b_h = (float *)malloc(nBytes);
cudaMalloc((void **) &a_d, nBytes);
cudaMalloc((void **) &b_d, nBytes);
for (i=0, i<N; i++) a_h[i] = 100.f + i;
cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);
for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
 return 0; }
```



Increment Array Example

CPU program

```
void inc_cpu(int *a, int N) {
  int idx;
  for (idx = 0; idx<N; idx++)
    a[idx] = a[idx] + 1;
                                     }


int main() {
  ...     inc_cpu(a, N);
                      }
```

CUDA program

```
__global__ void inc_gpu(int *a, int
    N) {
  int idx = blockIdx.x*blockDim.x
    + threadIdx.x;
  if (idx < N)a[idx] = a[idx] + 1;
}
int main() {   …
  dim3 dimBlock (blocksize);   dim3
    dimGrid( ceil( N /
    (float)blocksize)  );
  inc_gpu<<<dimGrid,
    dimBlock>>>(a, N); }
```

Variable Qualifiers (GPU code)

__device__

Stored in global memory (large, high latency, no cache) Allocated with cudaMalloc (__device__ qualifier implied) Accessible by all threads

Lifetime: application

__shared__

Stored in on-chip shared memory (very low latency) Specified by execution configuration or at compile time Accessible by all threads in the same thread block

Lifetime: thread block

Unqualified variables:

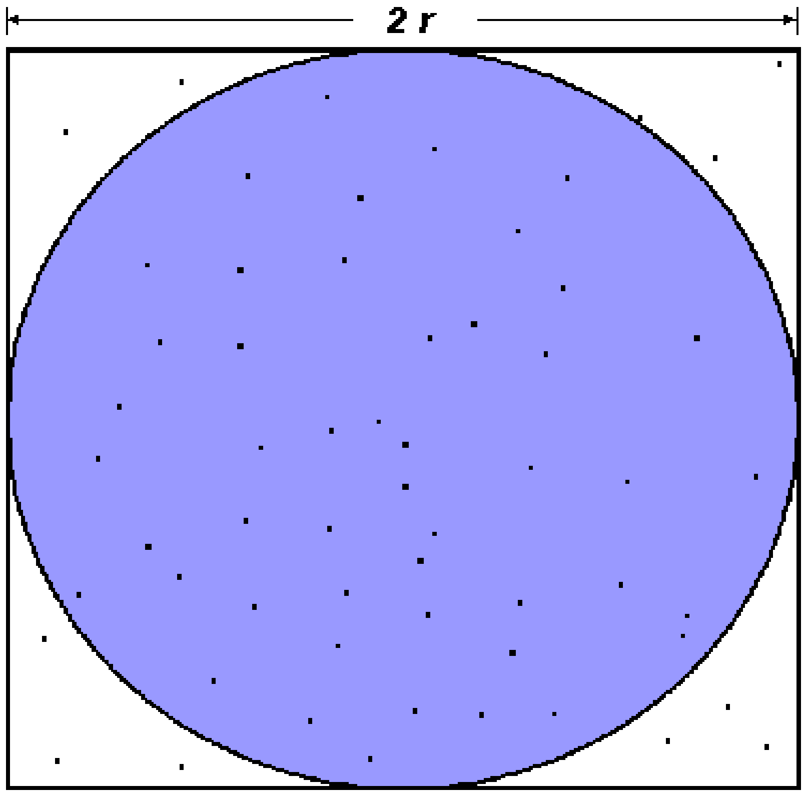
Scalars and built-in vector types are stored in registers What doesn’t fit in registers spills to “local” memory

An Example

- **PI Calculation**

  Serial pseudo code for this

```
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
```



- Note that most of the time in running this program would be spent executing the loop
- Leads to an embarrassingly parallel solution
  - Computationally intensive
  - Minimal communication
  - Minimal I/O

Parallel strategy: break the loop into portions that can be executed by the tasks.

- For the task of approximating PI:
  - Each task executes its portion of the loop a number of times.
  - Each task can do its work without requiring any information from the other tasks
  - Uses the SPMD model. One task acts as master and collects the results.

```
npoints = 10000
circle_count = 0
p = number of procesors     num = npoints/p
find out if I am MASTER or WORKER
do j = 1,num
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
if I am MASTER receive from WORKERS their circle_counts
compute PI (use MASTER and WORKER calculations)
else if I am WORKER send to MASTER circle_count
endif
```

Cluster Computing & MPI

- Group of linked computers working together closely.
- Cost Effective compared to other High Flops SC.
- Excellent for parallel operations but much poorer than traditional SCs for non parallel ops.
- MPI is used, Language independent comm. Protocol
- 1980s -early 1990s: Distributed memory, parallel computing architecture, but incompatible software tools for parallel programs need for a standard arose.
- Final version of draft released in May, 1994
- MPI-2 picked up where the first MPI specification left off, was finalized in 1996. Current MPI combine both.
- **Explicit parallelism:** Programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI

Message Passing Interface

- MPI conforms to the following rules:
- Same program runs on all processes(SPMD) no restriction to MPMD
- Synchronizes well with Data Parallelism.
- No concept of shared memory
- Enables parallel programs in C, Python (though seq. lang.)

Information about which processor is sending the message:

- Where is the data on the sending processor.
- What kind of data is being sent.
- How much data is there.
- Which processor(s) are receiving the message.
- Where should the data be left on the receiving processor.
- How much data is the receiving processor prepared to accept.

“Hello World” MPI Program

```
program mpitest
use MPI
integer rank,size,ierror

call MPI_Init(ierror)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

write(*,*) ’Hello World, I am ’,rank,’ of ’,size

call MPI_Finalize(ierror)

end
```

```
Hello World, I am 3 of 4
Hello World, I am 0 of 4
Hello World, I am 2 of 4
Hello World, I am 1 of 4
```

Automatic parallelization

- Converting sequential code into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously
- Most Focused : LOOPS
- An Example:

```
do i=1 to n                      do i=2 to n               <-NOT Possible
z(i) = x(i) + y(i) enddo         z(i) = z(i-1)*2 enddo

                                  do i=2 to n
                                  z(i) = z(1)*2^(i-1) enddo
```

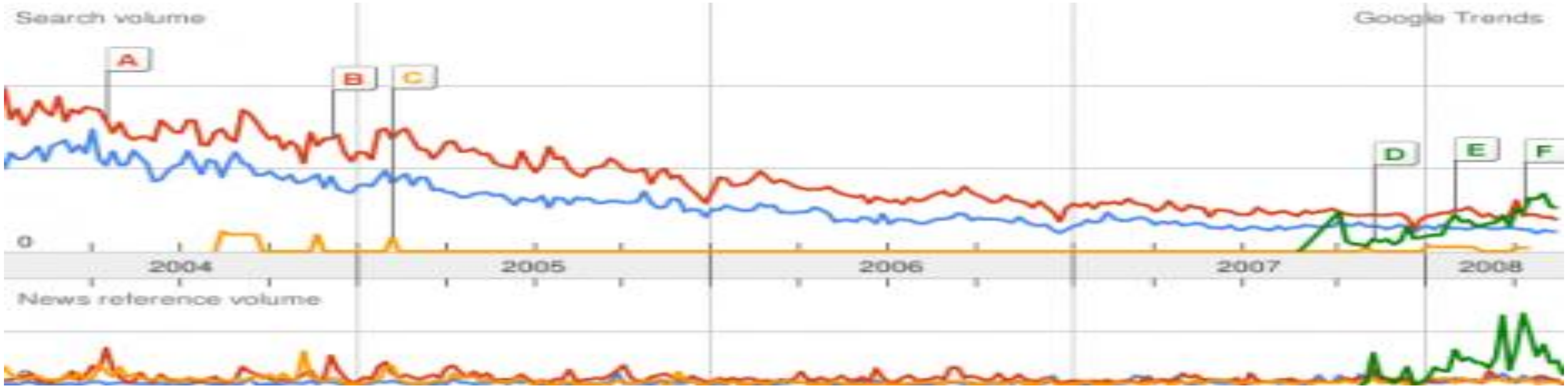
- Parallelizing compilers can be fairly successful with Fortran77 programs.
- Success with C has been more limited.
- Instead of parallelizing at compile time, it may be done at runtime as all the information about variable is available.
- If a compiler is trying to prove a certain relation in order to parallelize a given program loop, and is unable to prove it, the compiler could compile a runtime test into the program that can determine whether the relation is true.

Difficulties

- dependence analysis is hard for code using indirect addressing, pointers, recursion, and indirect function calls
- loops have an unknown number of iterations;
- accesses to global resources are difficult to coordinate in terms of memory allocation, I/O, and shared variables.
- Different architectures
- Attempts:
- Abstract Data and Commnications Library (ADCL) Univ. of Stuttgart
- Automatic performance tuning of MPI-parallel apps. (Runtime)
- Historic Compilers:
- Vienna Fortran compiler
- Polaris Compiler

Cloud Computing

- Instead of installing a suite of software for each computer, you'd only have to load one application.
- That application would allow workers to log into a Web-based service which hosts all the programs the user would need for his or her job.
- User applications like office, database applications etc. can all be put on the cloud.
- Google's app engine allows one to upload his own applications on the net which everybody can use.
- Low Hardware Cost: Just need the cloud computing system's **interface software,**
- Communication costs and latency are too high to allow parallelization across the world. (Remote access of memory is costly)
- **distributed computing grid computing utility computing cloud computing**


Search volume
Google Trends
A
B
C
D
E
F
News reference volume