# Concurrency, Parallelism and Distributed Systems (CPDS)

## Understanding parallelism

Eduard Ayguadé, Josep-Ramon Herrero, Daniel Jiménez
({eduard,josepr,djimenez}@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

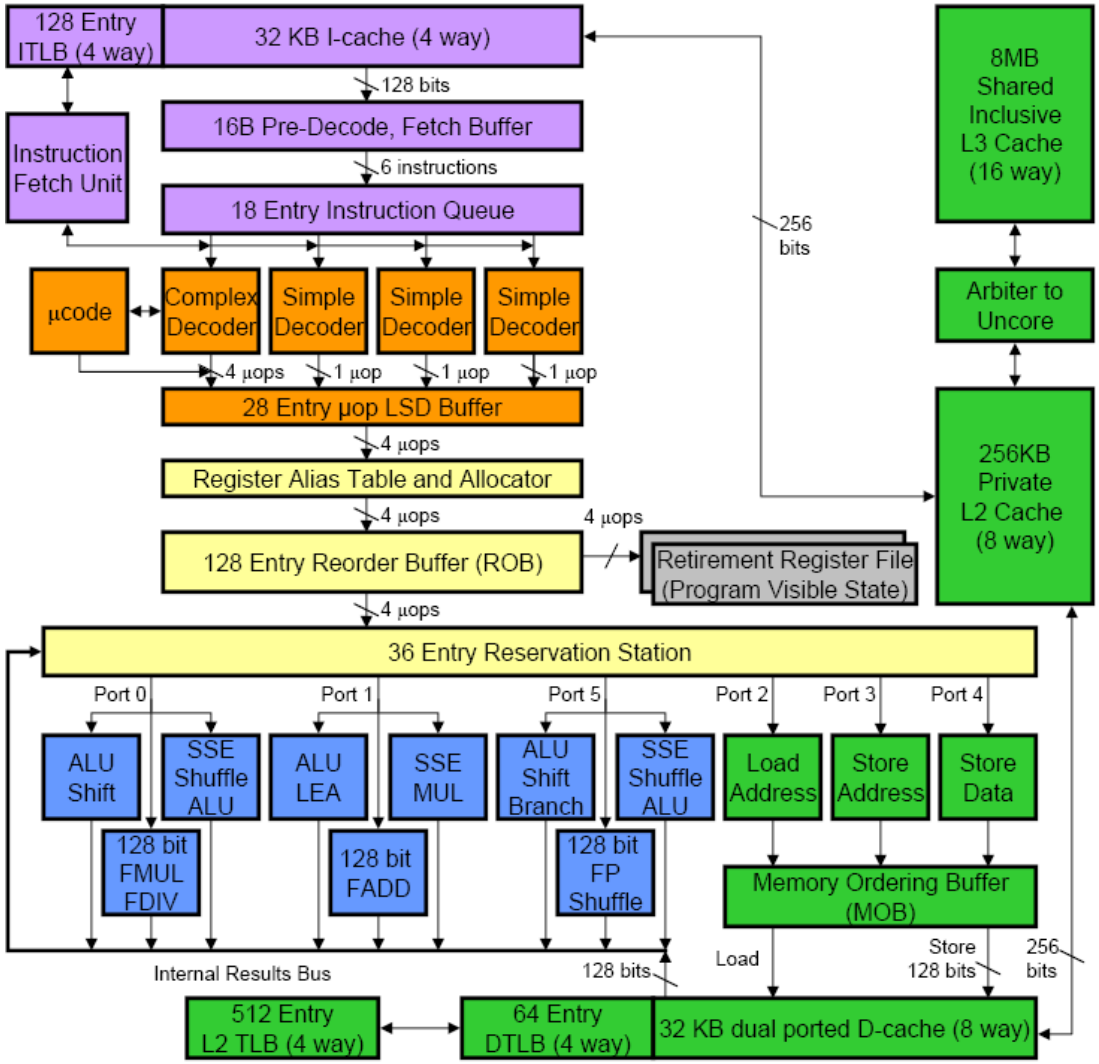
2014/15 - Fall

# Part I

# A brief introduction to parallel architectures

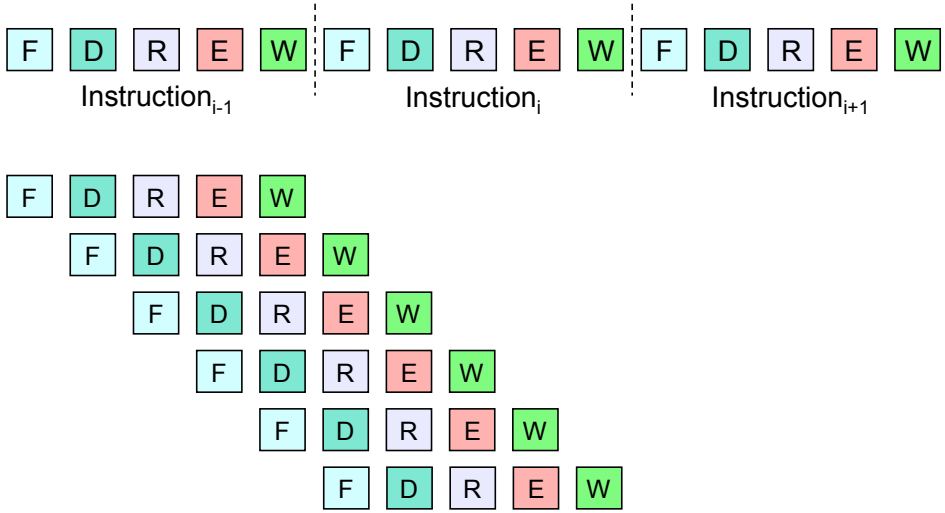
# Outline

# Current uniprocessor architecture: Intel Nehalem i7



# Pipelining

- Execution of single instruction divided in multiple stages
- Overlap the execution of different stages of consecutive instructions
- Ideal: IPC=1 (1 instruction executed per cycle)




- IPC<1 due to hazards (structural, data, control), preventing the execution of an instruction in its designated clock cycle

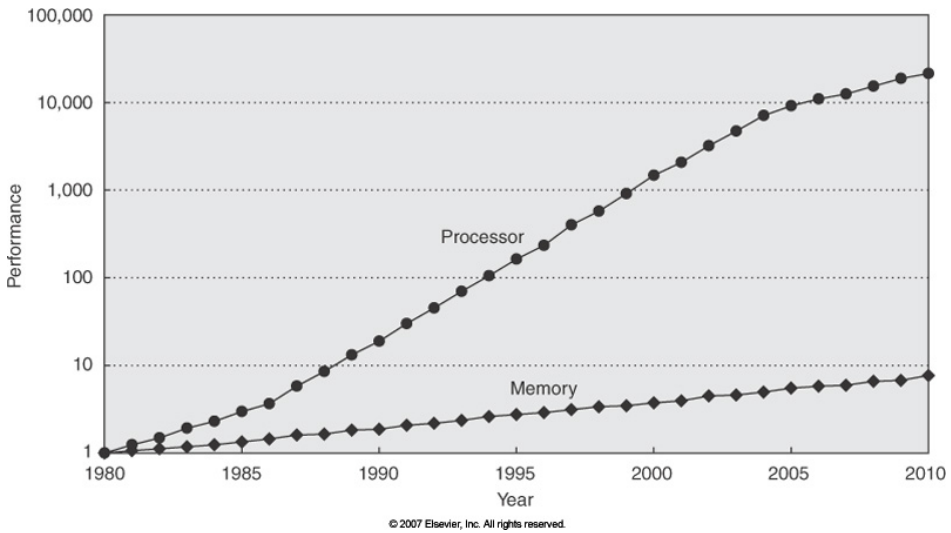
# Sources of parallelism in uniprocessors

- ILP (Instruction-level parallelism)
  - Superscalar architecture: multiple issue slots (functional units)
  - Execution of multiple instructions, from the same instruction flow, per cycle
- TLP (thread-level parallelism)
  - Multithreaded architecture[1]: fill the pipeline with instructions from multiple instruction flows
  - Latency hiding (cache misses, non-pipelined FP, ...)
- DLP (data-level parallelism)
  - SIMD architecture: single-instruction executed on multiple-data in a single word
  - Vector functional unit

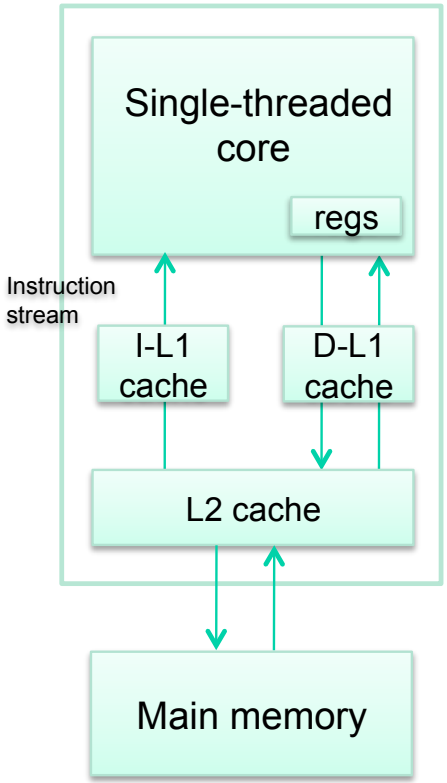
[1] Hyperthreading in Intel terminology

# Memory hierarchy

- Addressing the yearly increasing gap between CPU cycle and memory access times




- Size vs. access time



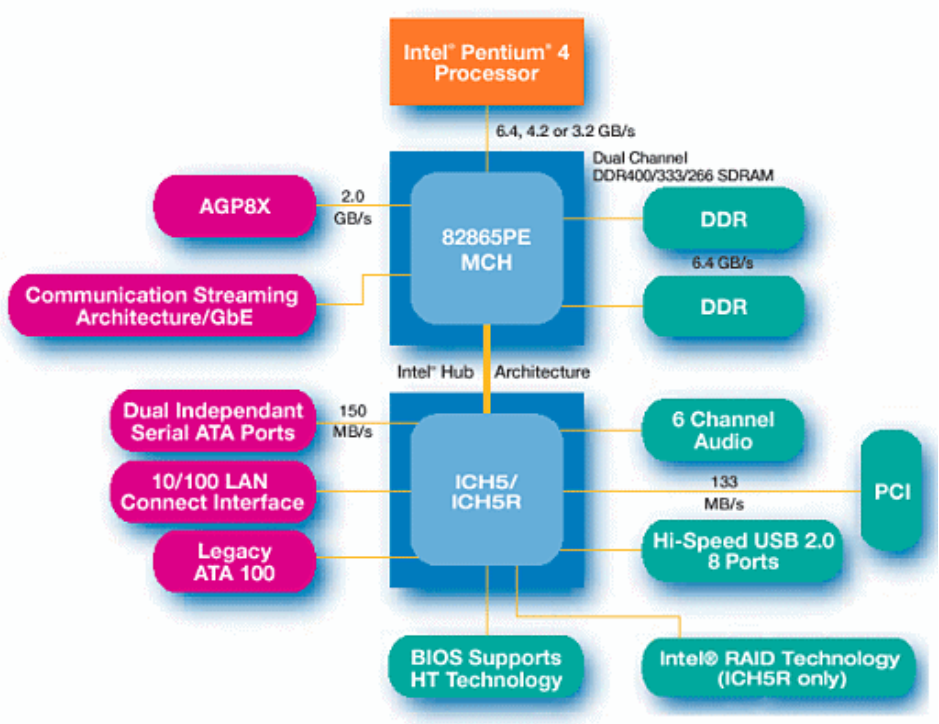

- Non-blocking design

# Memory hierarchy

- The principle of locality: if an item is referenced ...
  - Temporal locality: ... it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial locality: ... items whose addresses are close by tend to be referenced soon (e.g., straight line code, array access)
- Line (or block)
  - A number of consecutive words in memory (e.g. 32 bytes, equivalent to 4 words x 8 bytes)
  - Unit of information that is transferred between two levels in the hierarchy
- On an access to a level in the hierarchy
  - Hit: data appears in one of the lines in that level
  - Miss: data needs to be retrieved from a line in the next level

# Components in a node board

- The processor is connected to memory (and graphics unit or GPU) via the so-called 'northbridge'
- Through the northbridge, the processor is connected to slower I/O devices through the so-called 'southbridge'

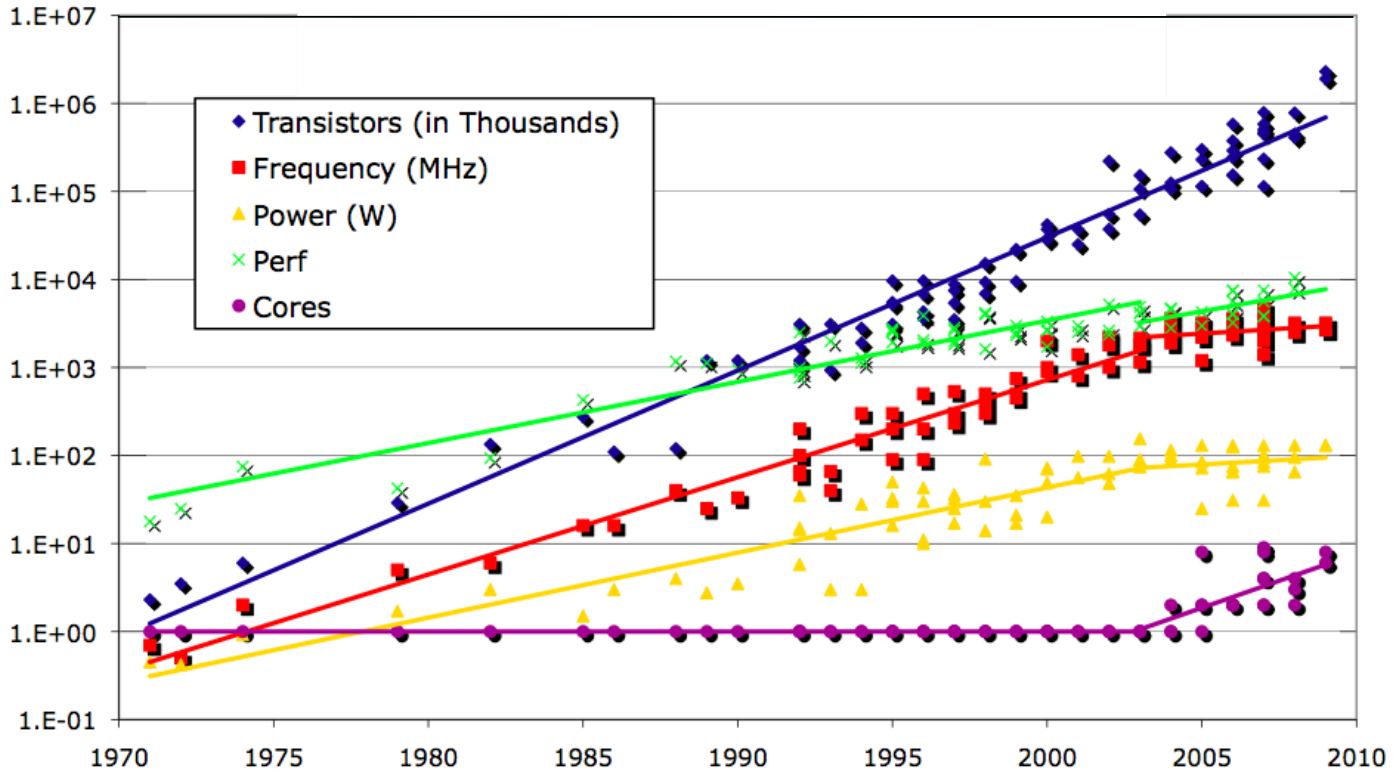


# Who exploits this uniprocessor parallelism?

- In theory, the compiler understands all of this ... but in practice the compiler may need your help:
  - Software pipelining to statically schedule ILP
  - Unrolling to allow the processor dynamicaly exploits ILP
  - Data contiguous in memory and aligned to efficiently exploit DLP
  - Blocking (or tiling) to define a problem that fits in register/L1-cache/L2-cache
  - ...
- Optimized libraries usually help for most commonly used operations

# Transistors, frequency, power, performance and ... cores!

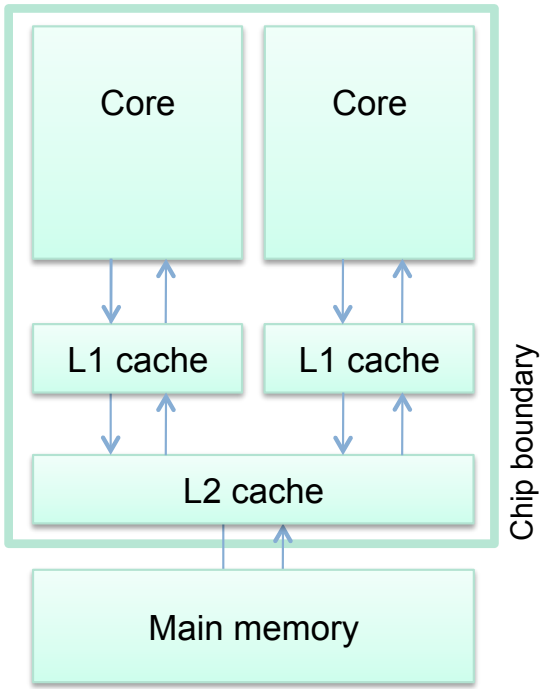
An inflexion point in 2004 ... the power wall[2].



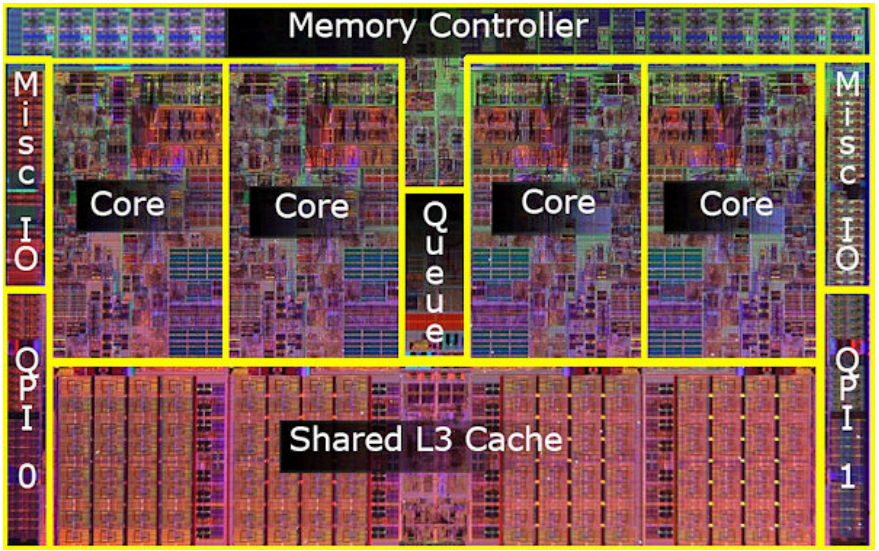

[2] Data from K. Olukotun, L. Hammond, H. Sutter, B. Smith, C. Batten, and K. Asanovic.

# Multicore era

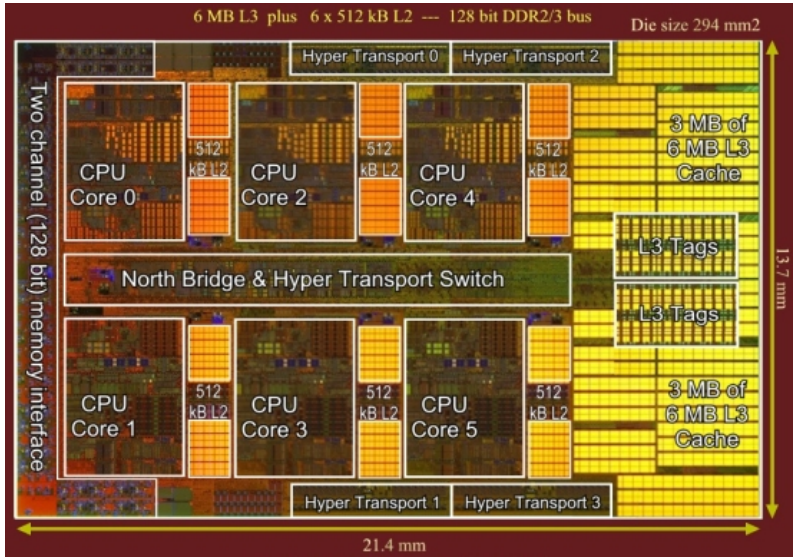
- The number of transistors on a chip is continuing to increase to accommodate multiple processors (cores) on a single chip
- Multicore = Chip Multi-Processor (CMP)



# Multicore era (examples)

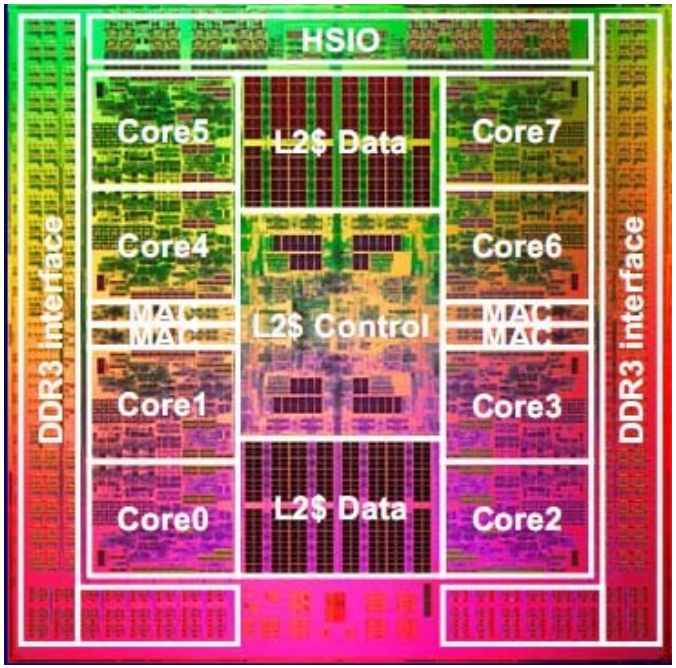



Intel's Nehalem i7 4-core (2 threads/core), 32/32 KB L1 and 256 KB L2 per core. Unified 4 or 8 MB L3

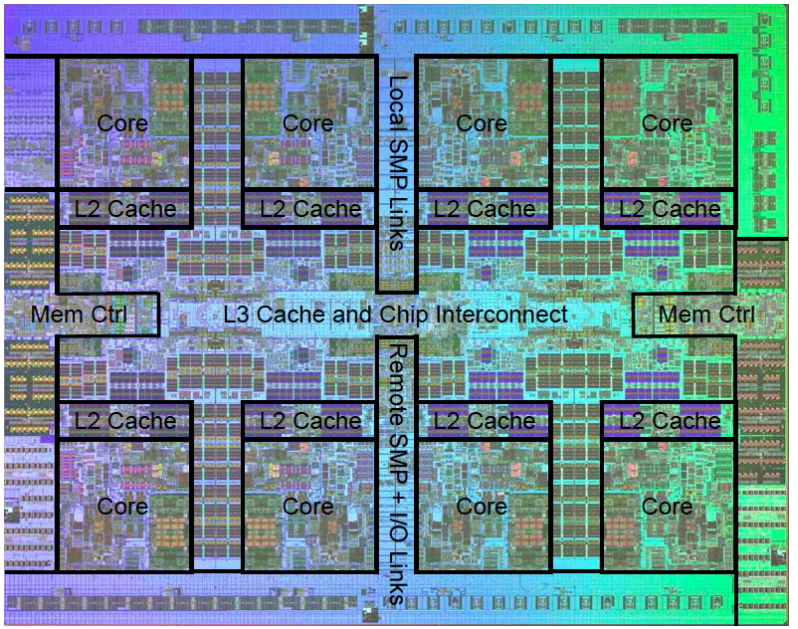



AMD's Istanbul 6-core, 512KB L2 per core and 6 MB shared L3

# Multicore era (examples)



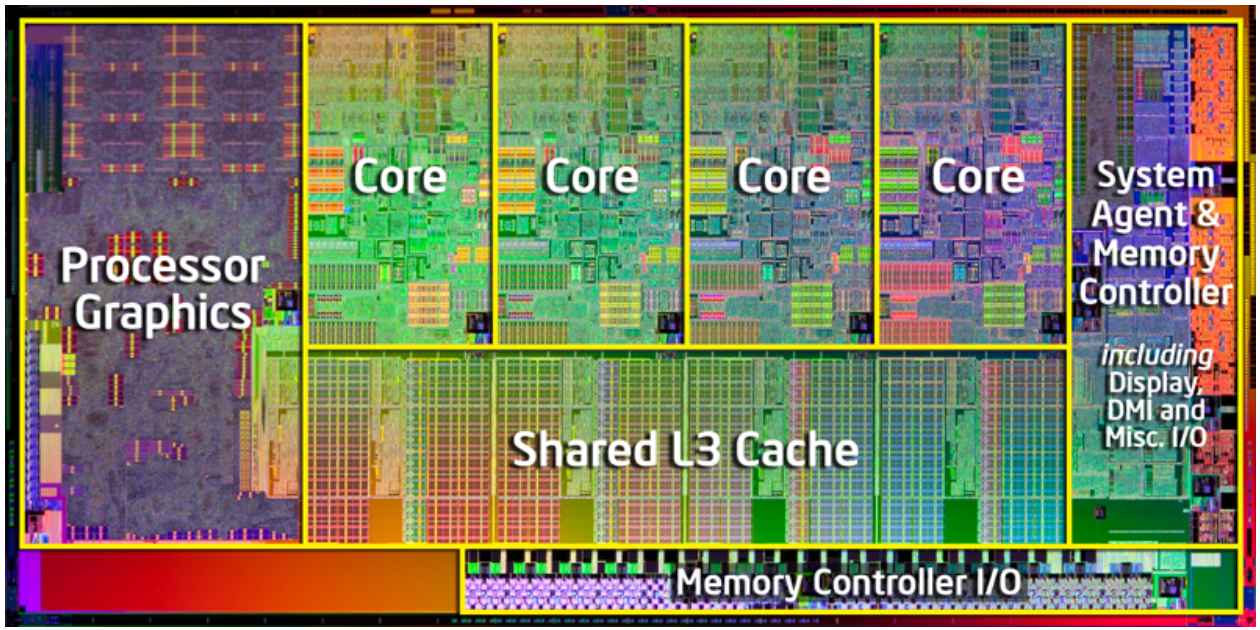

Sun's SPARC64-VIIIfx 8-core, 32/32K L1 per core and 5 MB shared L2. 2 GHz.. 760 M transistors




IBM's Power7 8-core (4 threads/core), 32/32K L1 and 256KB L2 per core. 32 MB shared L3. 2.4 GHz. 1.2 billion tansistors

# Multicore era (examples)

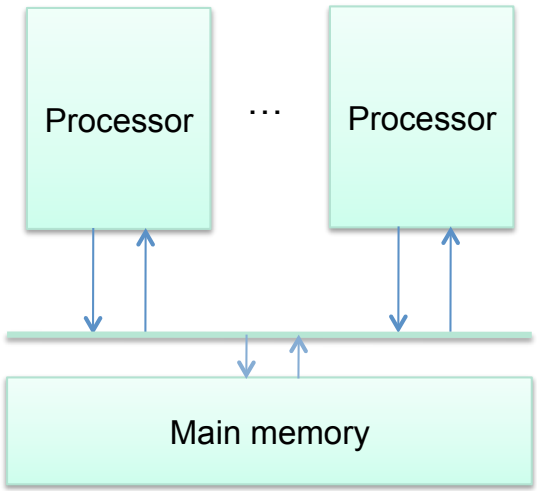
- Homogeneous multi-core systems include only identical cores, heterogeneous multi-core systems have cores with different functionalities, performances, ... (e.g. Intel SandyBridge)



# Outline

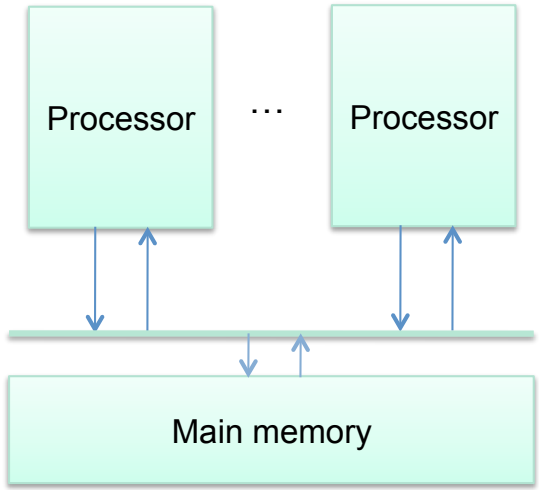
# Shared-memory multiprocessors

- SMP (Symmetric Multi-Processor)
  - Two or more identical processors are connected to a single shared main memory
  - Interconnection network: any processor can access to any memory location
- Symmetric multiprocessing: a single OS instance on the SMP
- Asymmetric multiprocessor (e.g. high/low ILP processors, ...) and multiprocessing (e.g. some processors running OS, others user code)



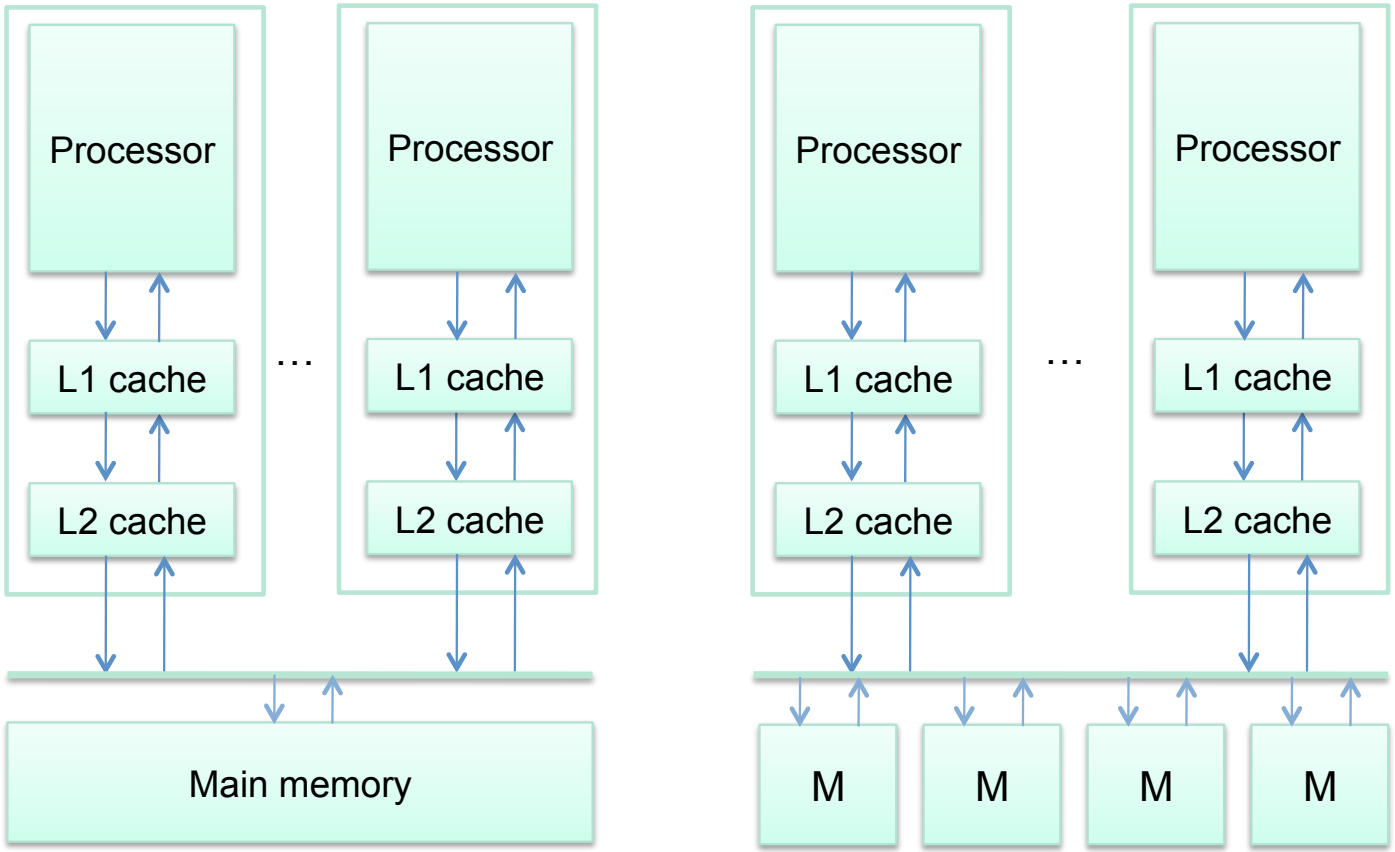
# Shared-memory multiprocessors

- SMP (Symmetric Multi-Processor)
  - Access to shared data with load/store instructions
  - Uniform Memory Access (UMA): access time to a memory location is independent of which processor makes the request or which memory chip contains the data
- The bottleneck in the scalability of SMP is the 'bandwidth' of the interconnection network and the memory



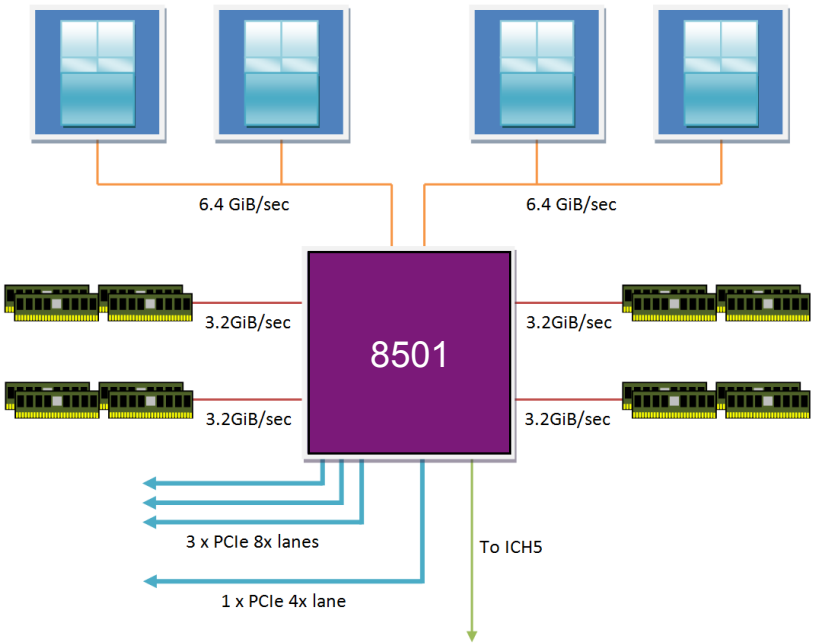
# Shared-memory multiprocessors

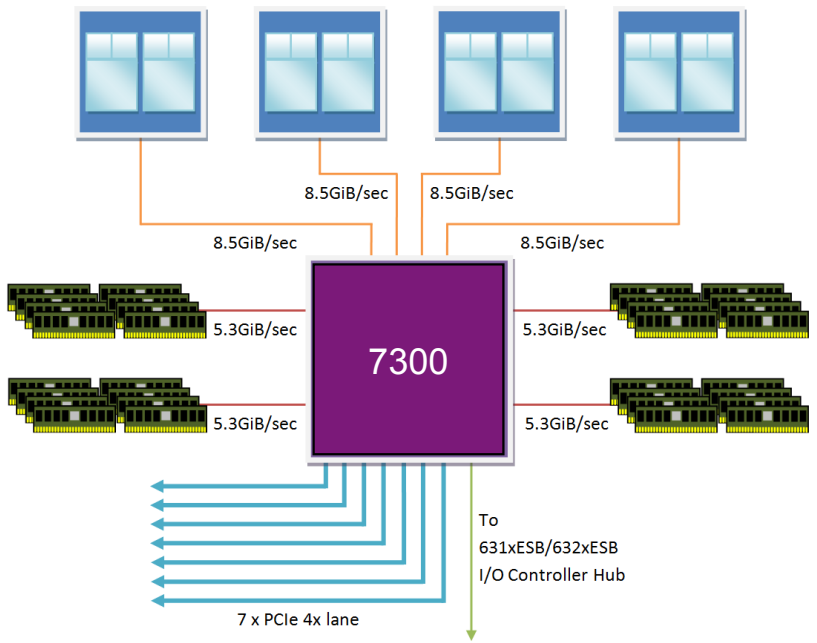
Local caches and multi-banked (interleaved) memory



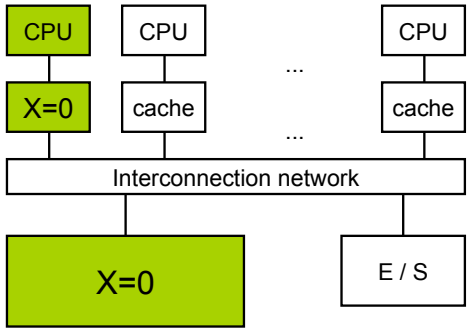
# Front-Side Bus (FSB)

- Processor-northbridge connection (64-bit, 1-4 transfers/cycle)
- A number of processors can be connected to a single FSB sharing its total bandwidth
- Northbridge with several FSB ports

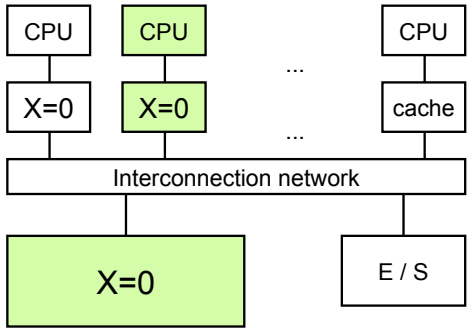


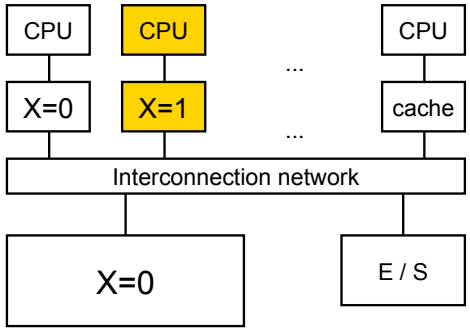
# Shared-memory multiprocessors: coherence



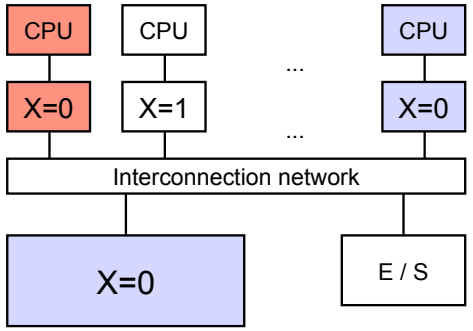

CPU1 reads x




CPU2 reads x




CPU2 writes x

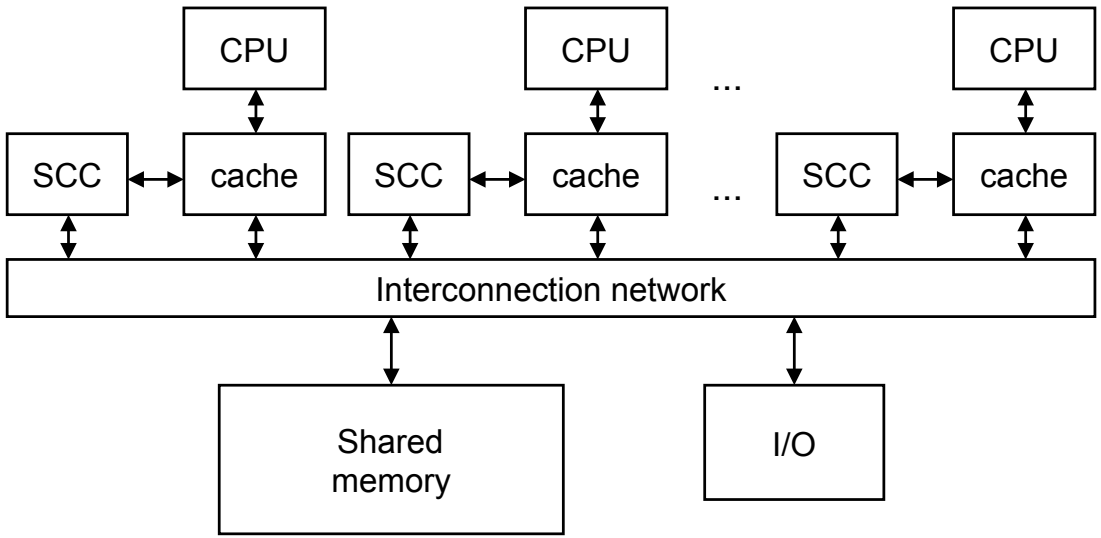



CPU1 or CPU3
read an incorrect x

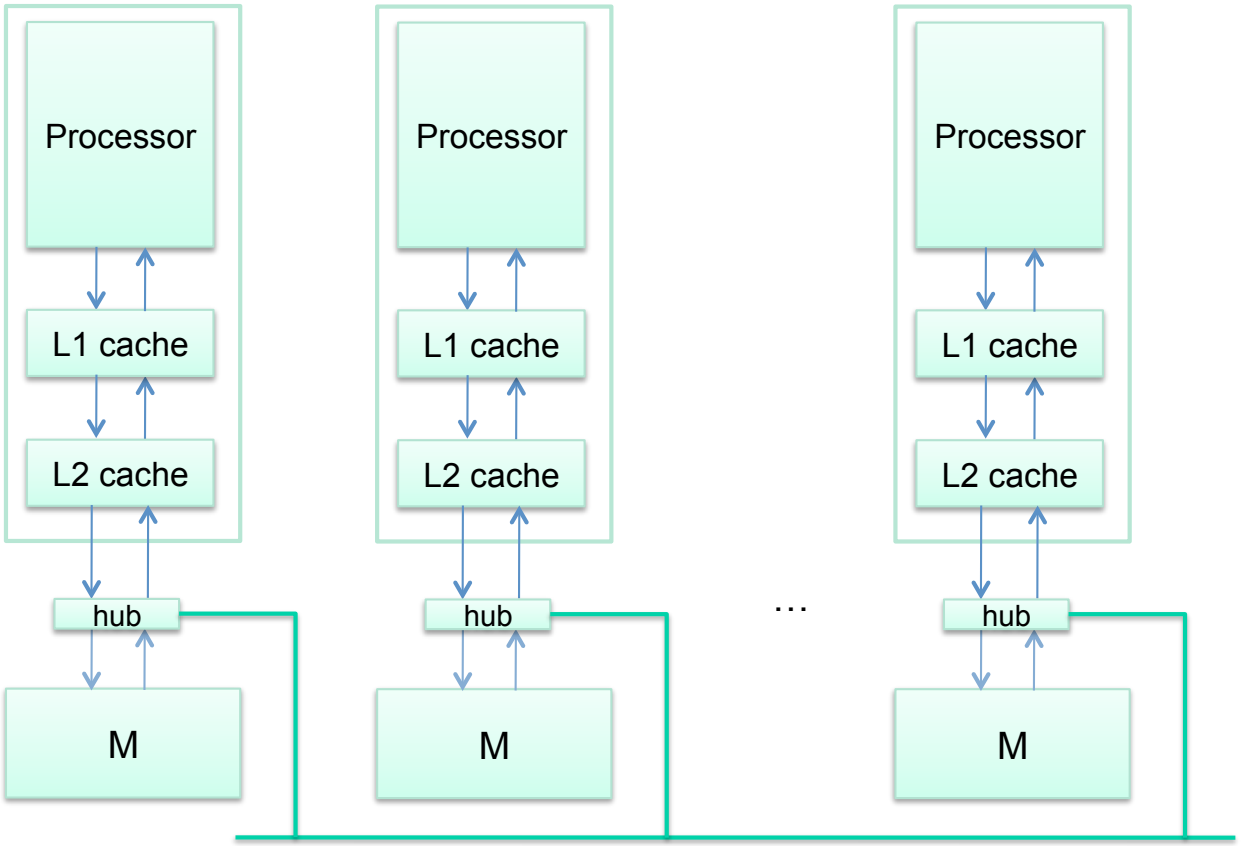
# Snooping-based coherence protocol

Shared bus is a broadcast medium

- Transactions on bus are visible to all processors
- Processors or their representatives can snoop (monitor) bus and take action on relevant events (e.g. change state)



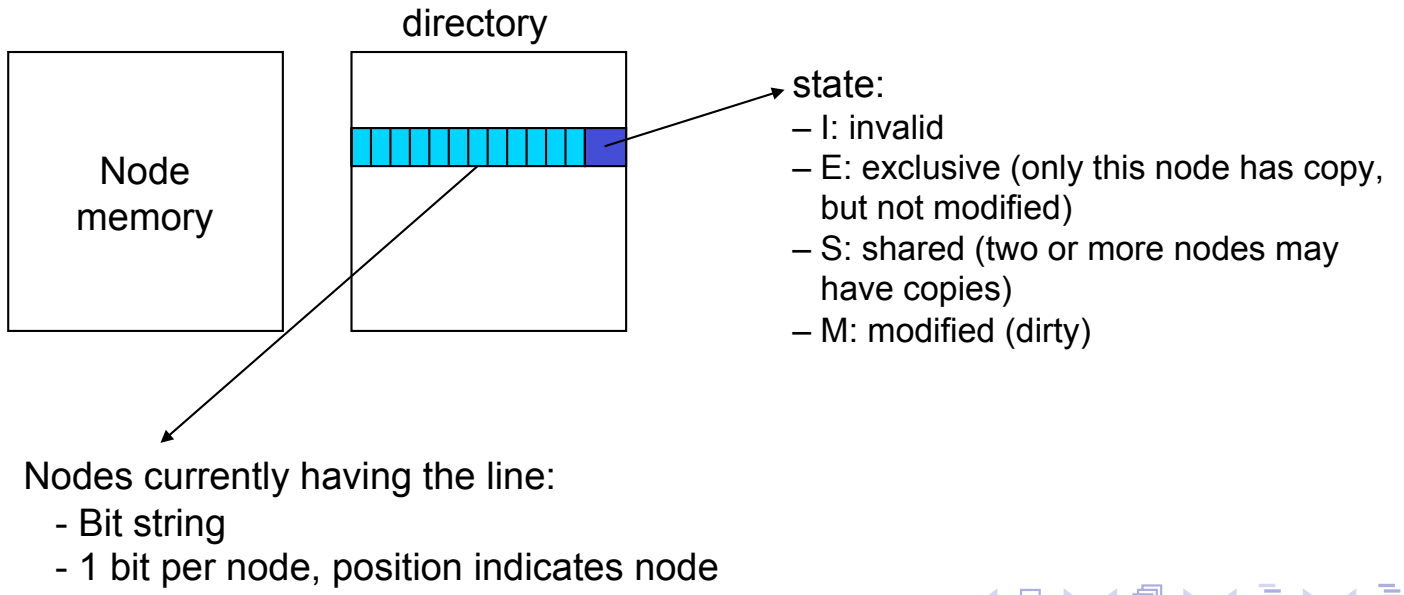
# Escalability? Non-Uniform Memory Architectures




- Hub enables cache-coherent NUMA

# Directory-based cache coherency protocol

- Directory-based is more scalable than Snoopy (bus)
- Only processors with copies recieve info ⇒ Reduce Communication
  - It tracks the state of the data stored in its memory
  - If a block is shared, it must track which processors have it
  - If a block is valid only in one node, it should know which one



# Shared-memory multiprocessors: synchronization

- Need hardware support to guarantee atomic (indivisible) instruction to fetch and update memory
  - User-level synchronization operations (e.g. locks, barriers, point–to–point, ...) using these primitives
- `test-and-set`: read value in location and set to 1

```
lock:  t&s r2, flag
       bnez r2, lock     // already locked?
```

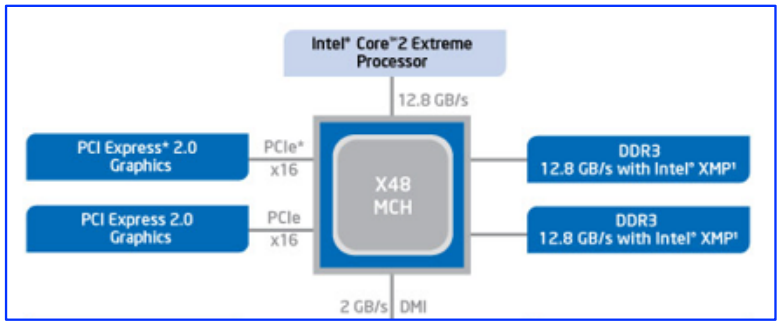
- Atomic exchange: interchange of a value in a register with a value in memory
- `fetch-and-op`: read value in location and replace with increment/decrement

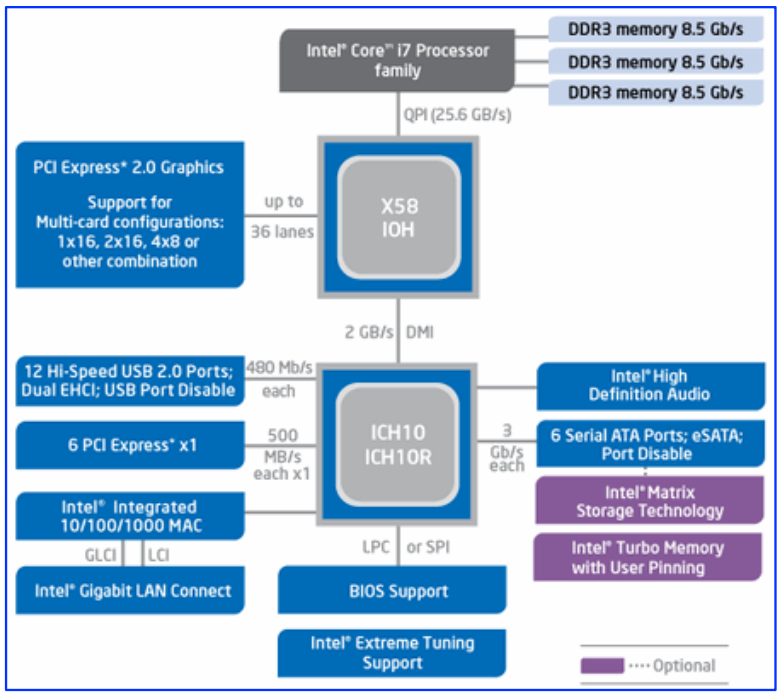
# Shared-memory multiprocessors: synchronization

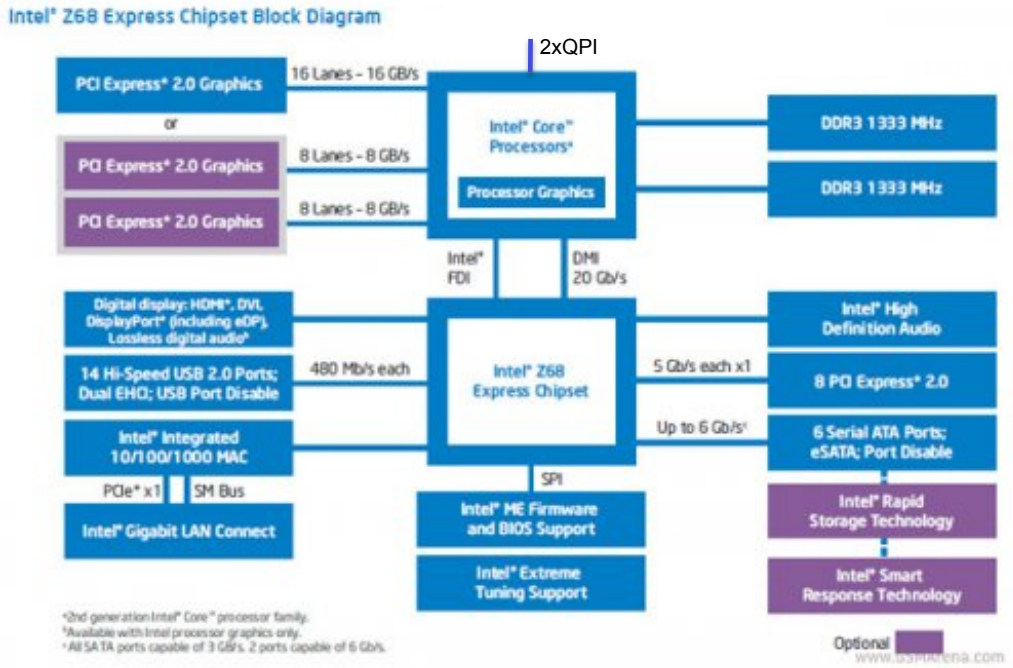
- Atomicity difficult or inefficient in large systems. Idea: assume optimistic atomic access
- Load-linked Store-conditional `ll-sc`
  - `ll` returns the current value of a memory location
  - `sc` stores a new value in that memory location if no updates have occurred to it since the `ll`; otherwise, the `sc` is guaranteed to fail

```
lock: ll r1, location
      bnz   r1, lock
      r2 = f (r1)             // f can be test-and-set, fetch-and-op, ...
      sc location, r2
      beqz  lock              // Z flag set to 1 if atomicity has occurred
```

# Northbridge evolution



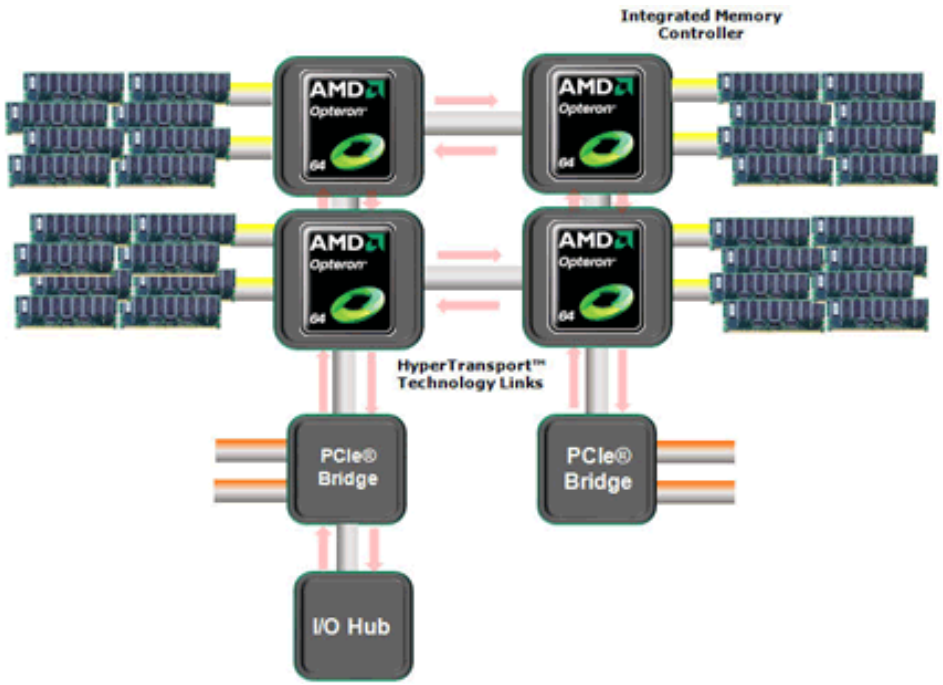




- Memory controller inside the processor chip (e.g. Intel Nehalem)
- Graphics unit inside the processor chip (e.g Intel Sandybridge)

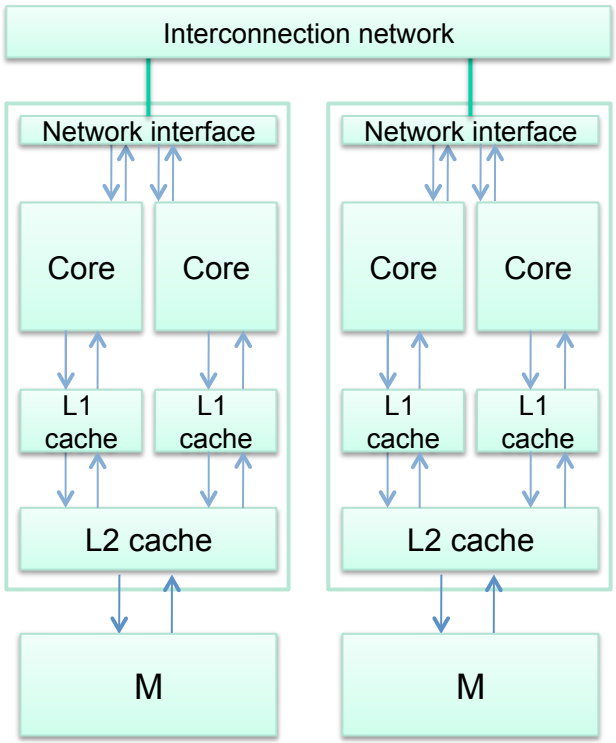
# Quick Path Interconnect (QPI) and Hypertransport (HT)

- Point-to-point processor interconnects, replacing the FSB
- Processor chips with integrated memory controllers
- Connect one or more processors and/or one or more chipsets in a network: non-uniform memory access (NUMA)



# Outline

# Why hardware needs to provide data sharing?

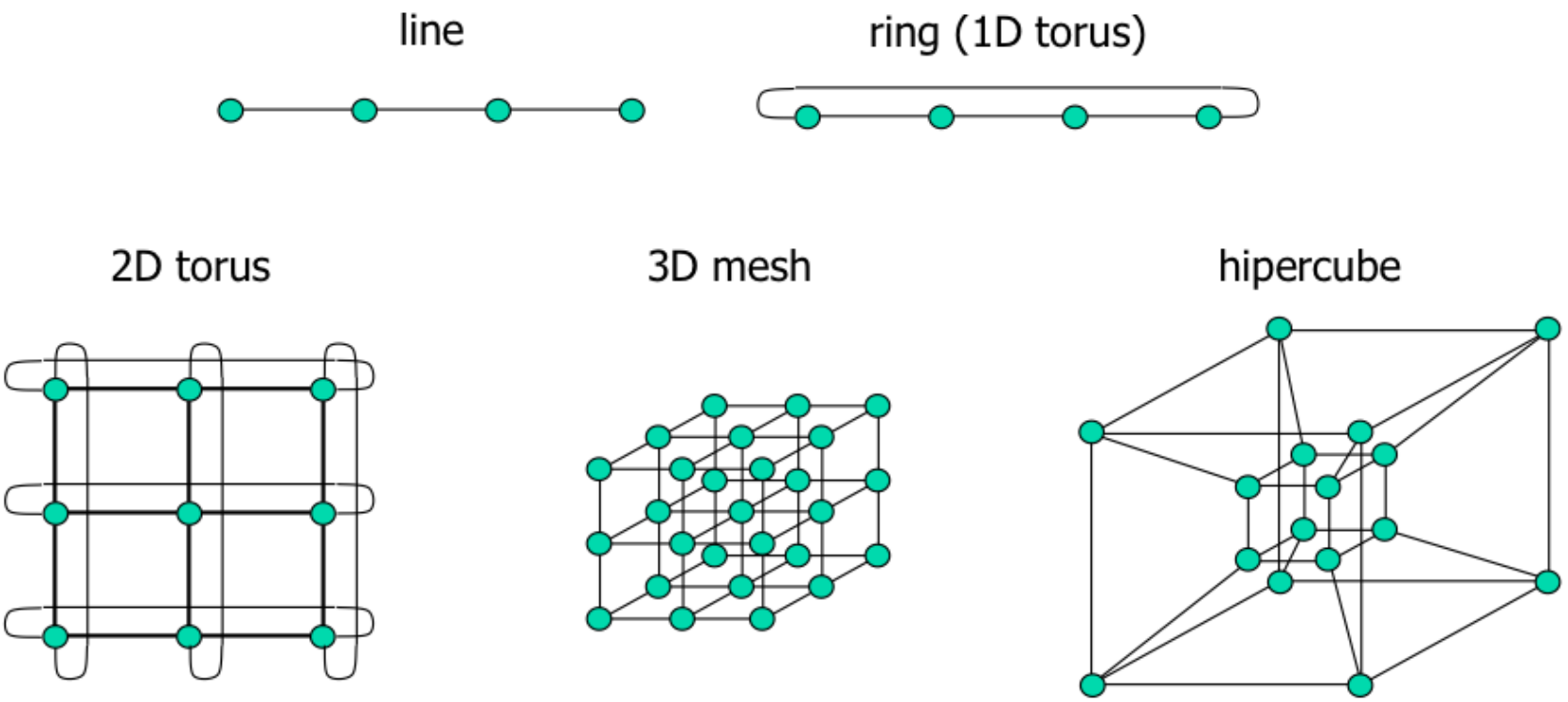



- Simple design: distributed-memory architectures
- Each node has its own (local) exclusive memory, accessible through load/store instructions (no cache coherency among nodes)
- Network interface and routers to exchange data between nodes (interconnection network)

# Interconnection networks

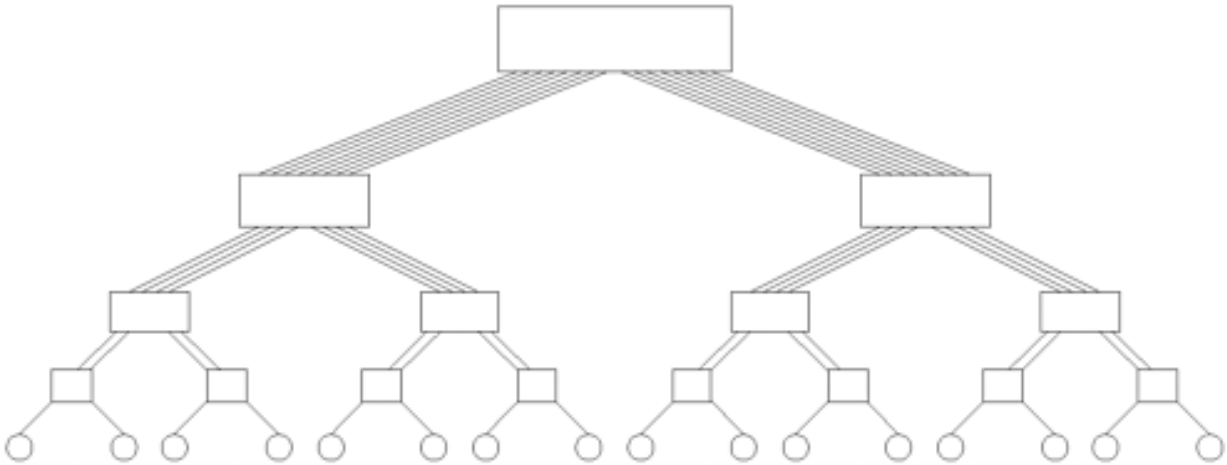
- Interconnection networks are build up of switching elements
  - Switches: devices that contain multiple input and output ports with a crossbar interconnection between them (i.e. any input to any output path available)
- Topology is the pattern in which the individual switches are connected to other switches and to processors and memories (nodes).
  - Direct topologies connect each switch directly to the network interface of a node
  - In indirect topologies at least some of the switches connect to other switches
- Influence on latency, banwidth and congestion

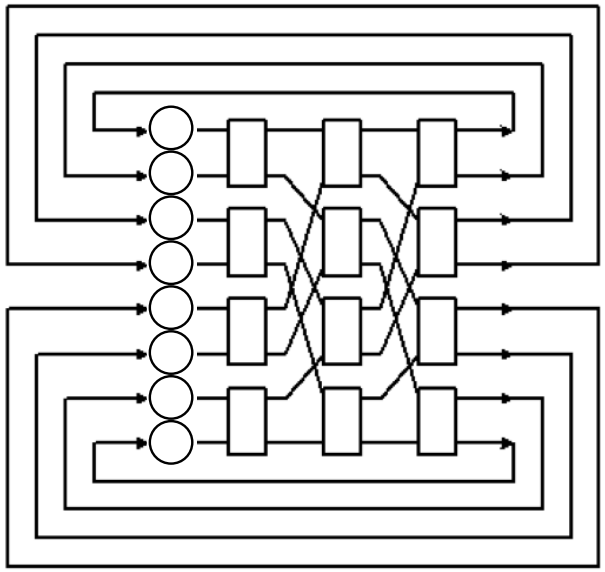
# Interconnection networks: direct topologies



# Interconnection networks: indirect topologies

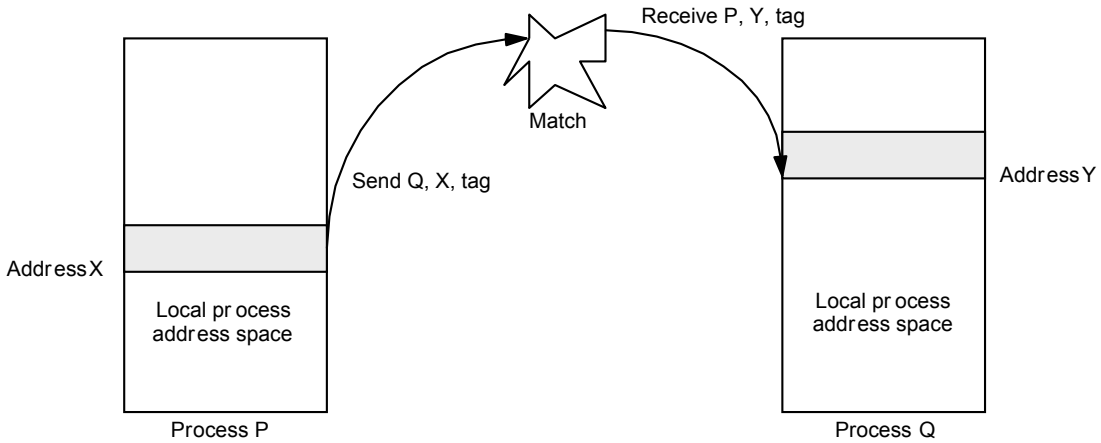
E.g. fat tree (left) and Omega multistage (right) networks





# Communication model

Data exchange using send and receive primitives




- Send specifies buffer to be sent and receiving process
- Receive specifies sending process and application storage to receive into
- Optional tag on send and matching rule on receive
- Optional implicit synchronization (e.g. blocking receive)

# Who does communication?

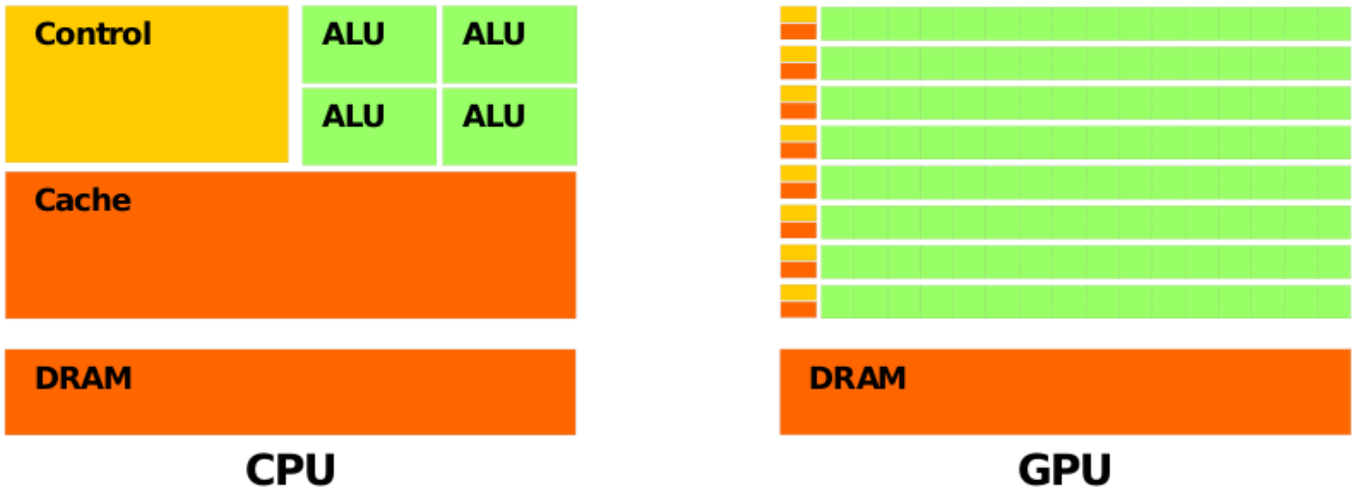
- Software DSM (distributed-shared memory)
  - Software layer that implements data sharing (and coherence)
  - Transparently to programmer
  - Usually based on page faults (OS involved, high overhead), which uses the communication model to move pages between nodes
- Message-passing paradigm
  - User-level library exporting the communication model to the programmer
  - Programmer moves data when necessary, assuming a data distribution

# Outline

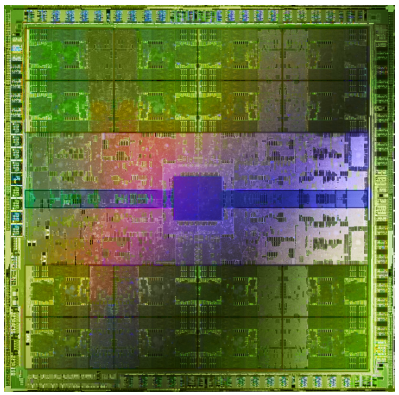
# Why GPU architectures

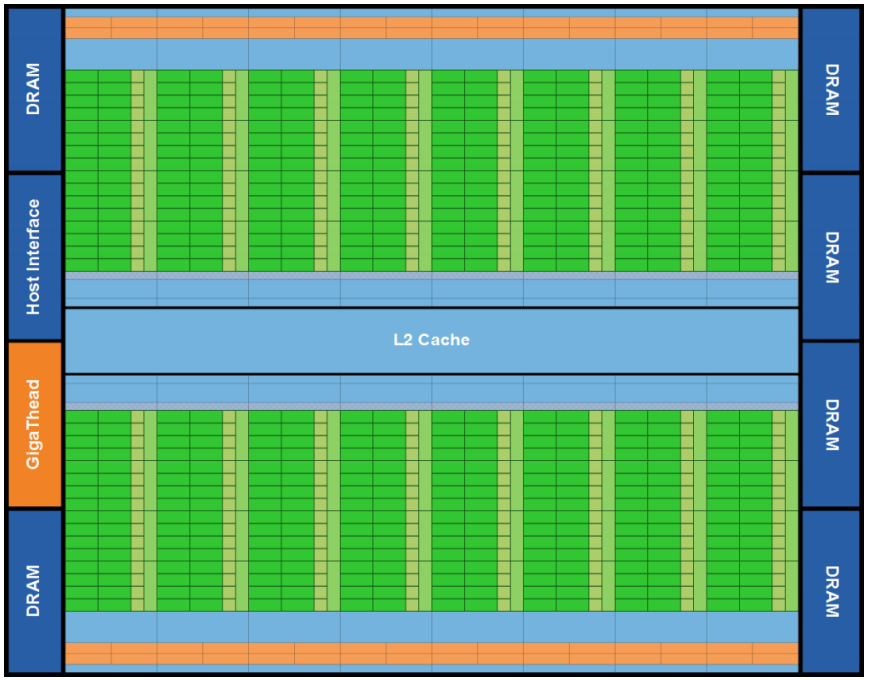
GPU are specialized for highly parallel, compute-intensive computation

- Simple control: same computation on all compute units
- Massive number of threads to reduce impact of long latency memory accesses



# NVIDIA GPU architecture



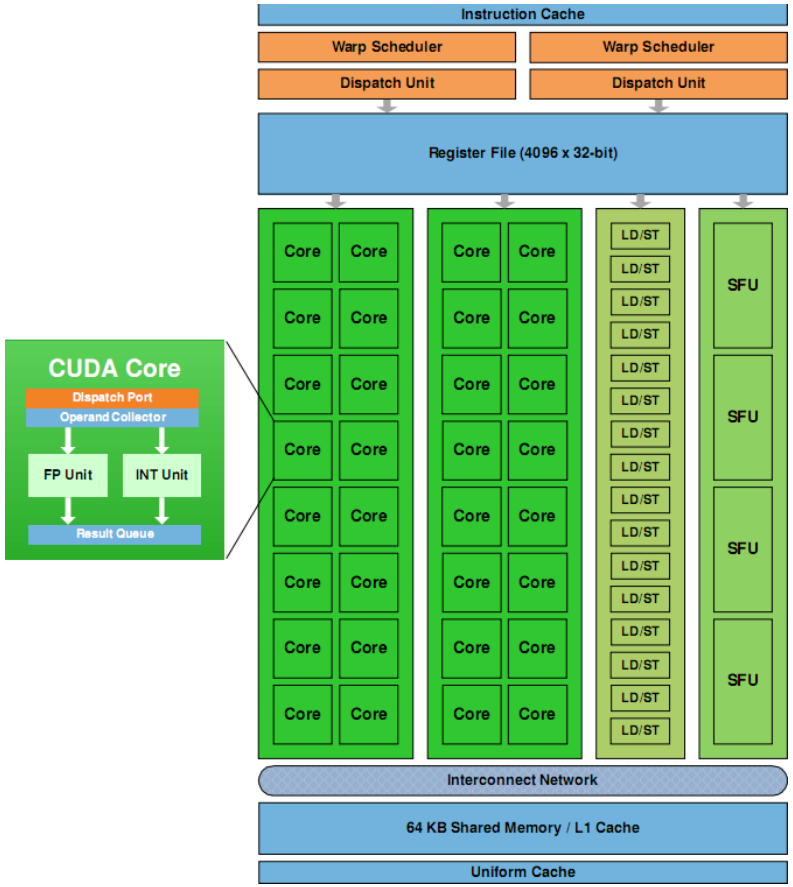



Die and aerial view of the NVIDIA Fermi architecture, with the graphics-specific parts (raster engine, polymorphic engines, ...) omitted.

# NVIDIA GPU architecture (cont.)

The NVIDIA Fermi architecture consists of:

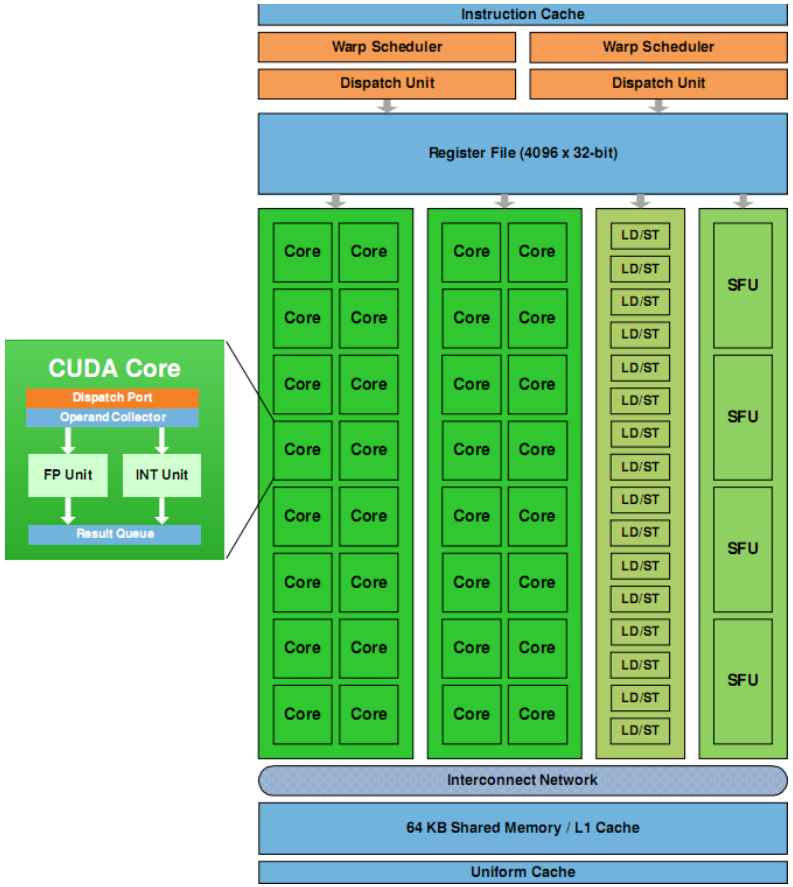
- 16 streaming multiprocessors (SM) interconnected via a 768k L2 cache crossbar
- Each SM has 32 cores and 4096 registers, sharing 64 KB (48K L1 and 16K shared memory or viceversa).
- Each core has one FP unit and one integer unit



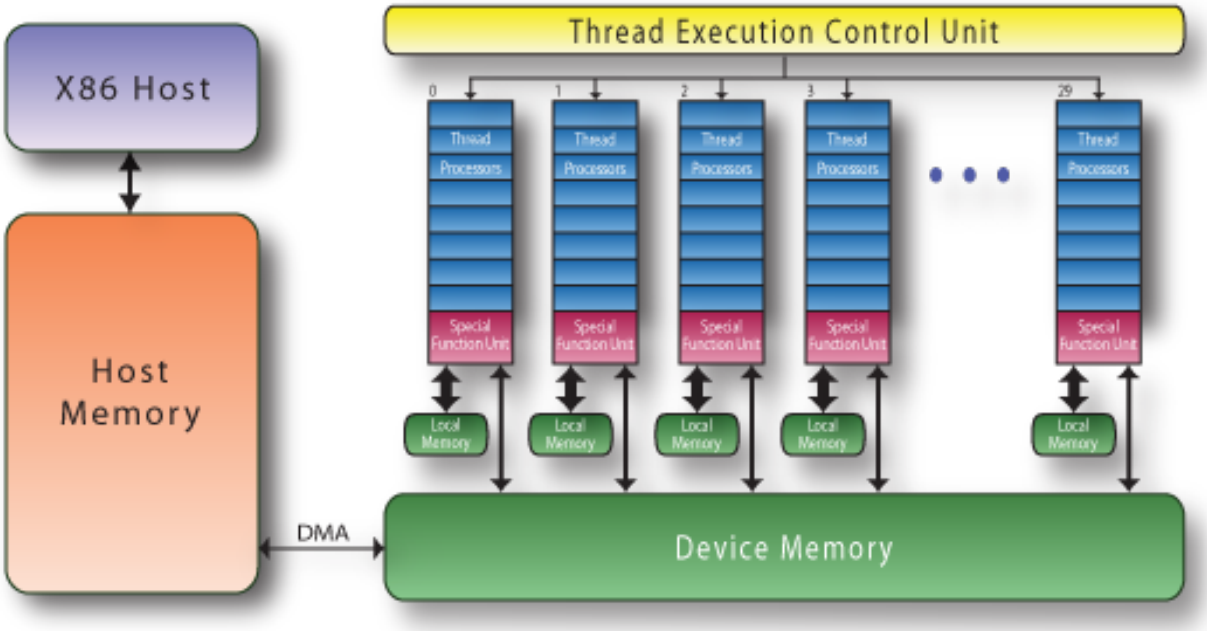
# NVIDIA GPU architecture (cont.)

The NVIDIA Fermi architecture consists of:

- Each SM executes a *warp* of 32 threads, can multiplex 48 active warps
- Each SM has 16 load/store units, allowing source and destination addresses to be calculated for sixteen threads per clock
- Each SM also has 4 Special Function Units (SFU) that execute complex instructions such as sin, cosine, reciprocal, and square root



# CPU-GPU block diagram




Separate address spaces between CPU (host memory) and GPU (device memory). Communication using IO commands and DMA memory transfers (e.g. PCI Express)

# Part II

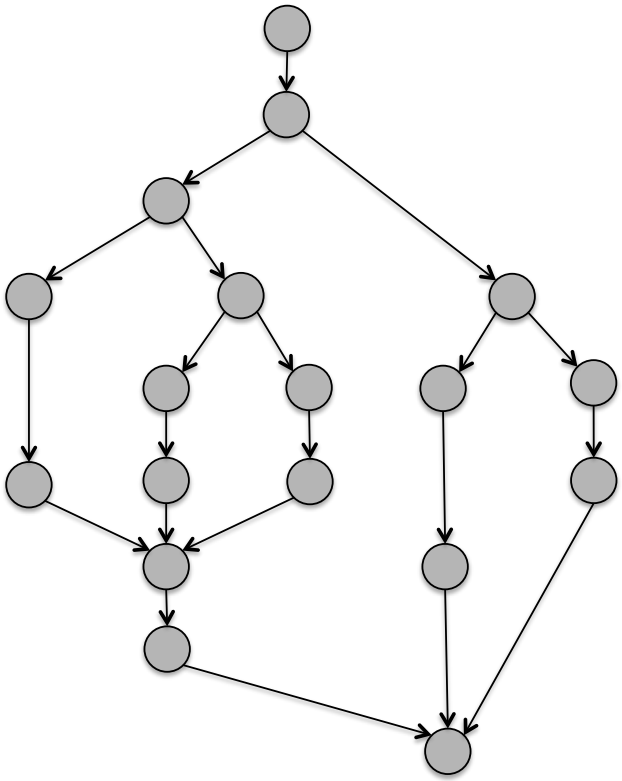
# Parallelism: concept and metrics

# Outline

# Finding concurrency/parallelism

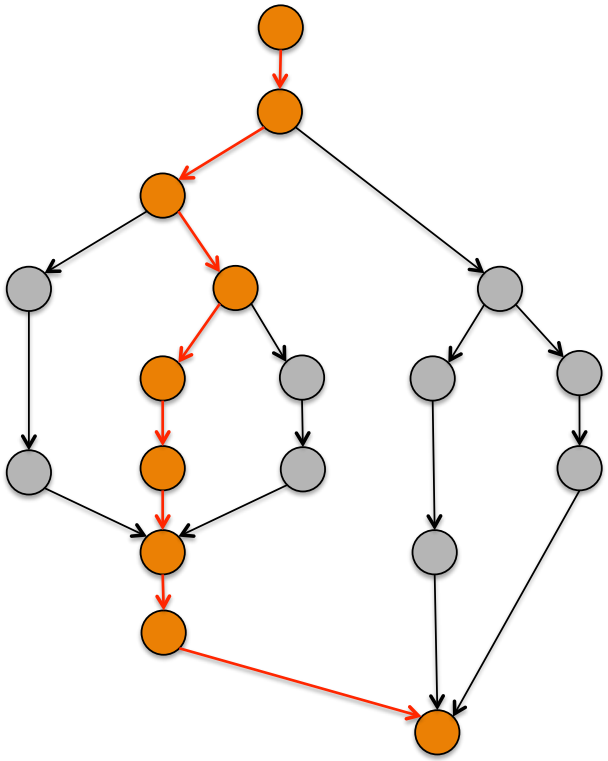
- Can the computation be divided in parts?
  - Based on the processing to do:
    - Task decomposition (e.g. functions, loop iterations)
  - Based on the data to be processed:
    - Data decomposition (e.g. matrix) (implies task decomposition)
- There may be data dependencies between tasks
- The decomposition determines the potential paralellism that could be obtained

# Computing the parallelism



- Computation task graph abstraction
  - Directed Acyclic Graph
  - Node = arbitrary sequential computation
  - Edge = dependence (successor node can only execute after predecessor node has completed: dataflow)
- Processor abstraction (simplification)
  - P identical processors
  - Each processor executes one node at a time

# Computing the parallelism



- $T_1 = \sum_{i=1}^{nodes}(work\_node_i)$
- $T_\infty = \sum_{i \in criticalpath}(work\_node_i)$, assuming sufficient resources
- $Parallelism = T_1/T_\infty$, independent of number of processors $P$
- $P_{min}$ is the minimum number of processors necessary to achieve $Parallelism$

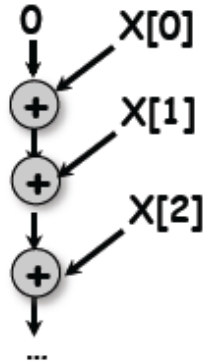
# Example: vector sum

Compute the sum of elements `X[0] ... X[n-1]` of a vector `X`

Sequential algorithm.

```
sum = 0; for ( i=0 ; i< n ; i++ ) sum += X[i];
```

- Computation graph



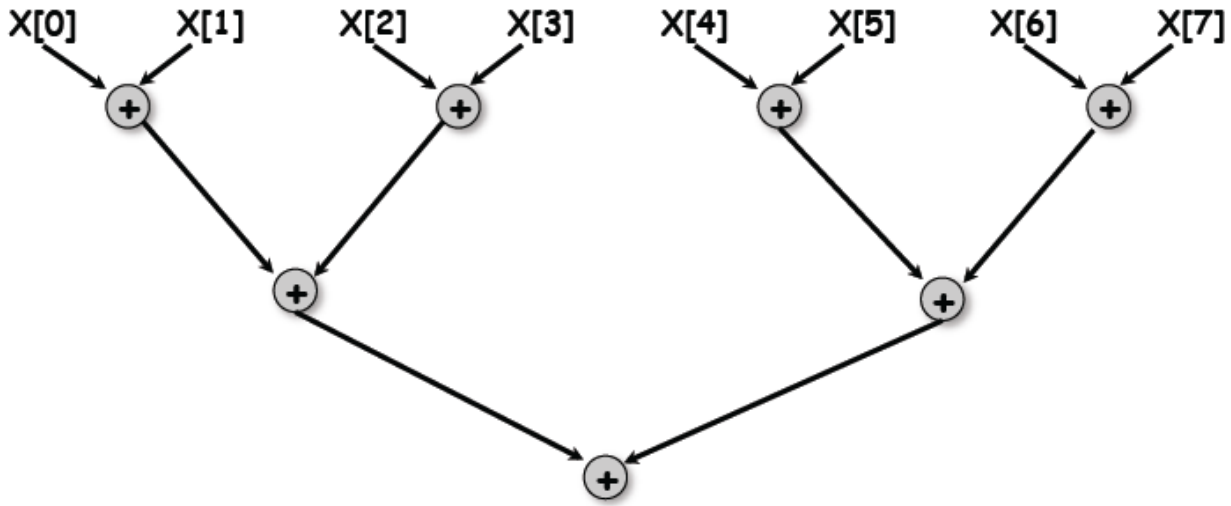

$$T_1 = O(n)$$
$$T_\infty = O(n)$$
$$Parallelism = O(1)$$

How can we design an algorithm (computation graph) with more parallelism?

# Example: vector sum

Compute the sum of elements `X[0] ... X[n-1]` of a vector `X`

- An alternative computation graph



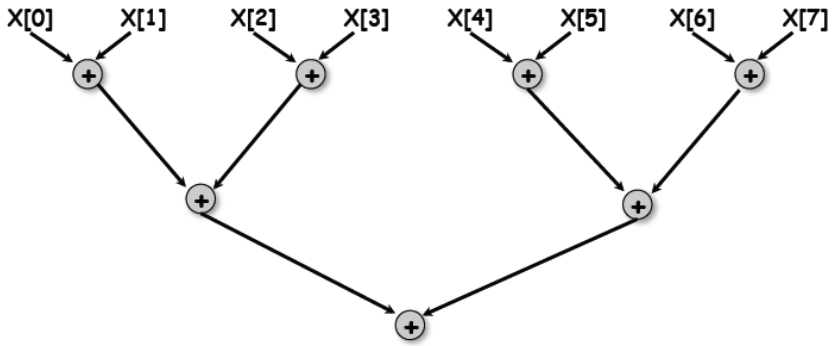

- $T_1 = O(n)$; $T_\infty = O(log\ n)$; $Parallelism = O(n/(log\ n))$
- How to restructure the sequential algorithm to have this computation graph? (recursive solutions)

# Example: vector sum

Compute the sum of elements `X[0]` ... `X[n-1]` of a vector `X`

Recursive algorithm:

```
int recursive_sum(int *X, int n)
{
   int ndiv2 = n/2;
   int sum=0;

   if (n==1) return X[0];

   sum = recursive_sum(X, ndiv2);
   sum += recursive_sum(X+ndiv2, n-ndiv2);
   return sum;
}

void main()
{
   int sum, X[N];
   ...
   sum = recursive_sum(X,N);
   ...
}
```

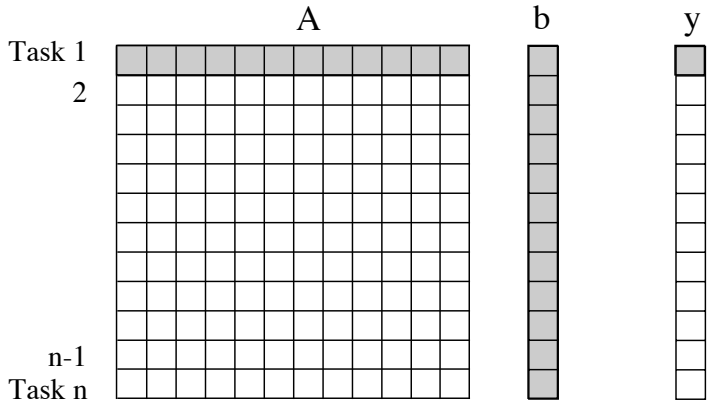


# Granularity and parallelism

- Each node in the computation task graph represents a sequential computation
- The granularity of the decomposition is determined by the size of each node in the computation graph
- Fine-grained tasks vs. coarse-grained tasks: the degree of parallelism increases as the decomposition becomes finer in granularity and vice versa

# Granularity and parallelism: fine-grained decomposition

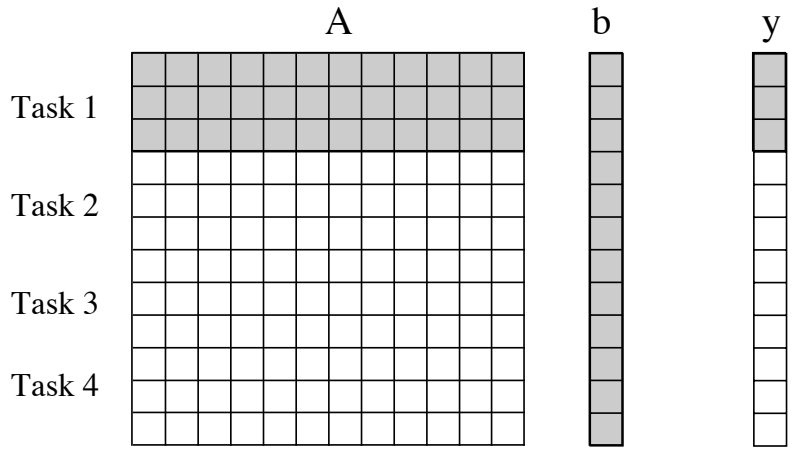
Example: matrix-vector product ($n$ by $n$ matrix):




- A task could be computing the whole vector $y$
- A task could be each individual $\times$ or $+$ in the dot product that computes an element of $y$ ($y[i] = y[i] + A[i][j] * b[j]$)
- A task could also be each complete dot product to compute an element of $y$ ($y[i] = \sum_{j=1}^{j=n} (A[i][j] * b[j])$)

# Granularity and parallelism: coarse-grained decomposition

A coarse grained counterpart to the matrix-vector product example. Each task in this example corresponds to the computation of three elements of the result vector.

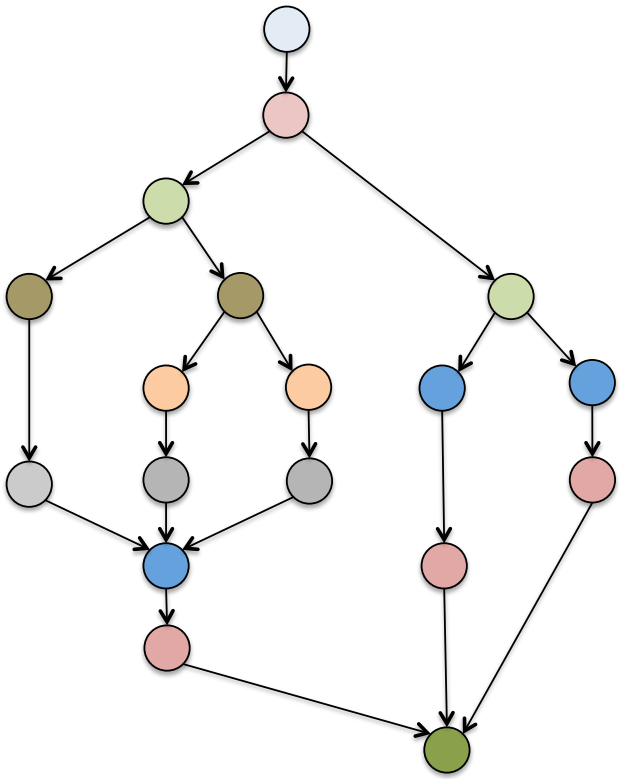


# Granularity and parallelism: fine vs. coarse-grained

- It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity but...
  - Inheret bound on how fine the granularity of a computation can be
    - *e.g. matrix-vector multiply:* $(n^2)$ *concurrent tasks.*
  - Tradeoff between the granularity of a decomposition and associated overheads (sources of overhead commented later, e.g. creation of tasks, synchronization, exchange of data between tasks, ...)
  - The granularity may determine performance bounds

# Outline

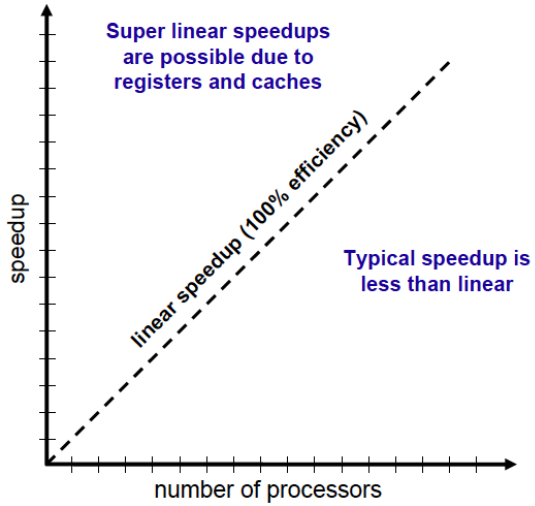
# Speedup



- $T_p$ = execution time on $P$ processors (depends on the schedule of the graph nodes on the processors)
- Lower bounds
  - $T_p \geq T_1/P$
  - $T_p \geq T_\infty$
- Speedup on $P$ processors: $S_p = T_1/T_p$

# Speedup vs. efficiency

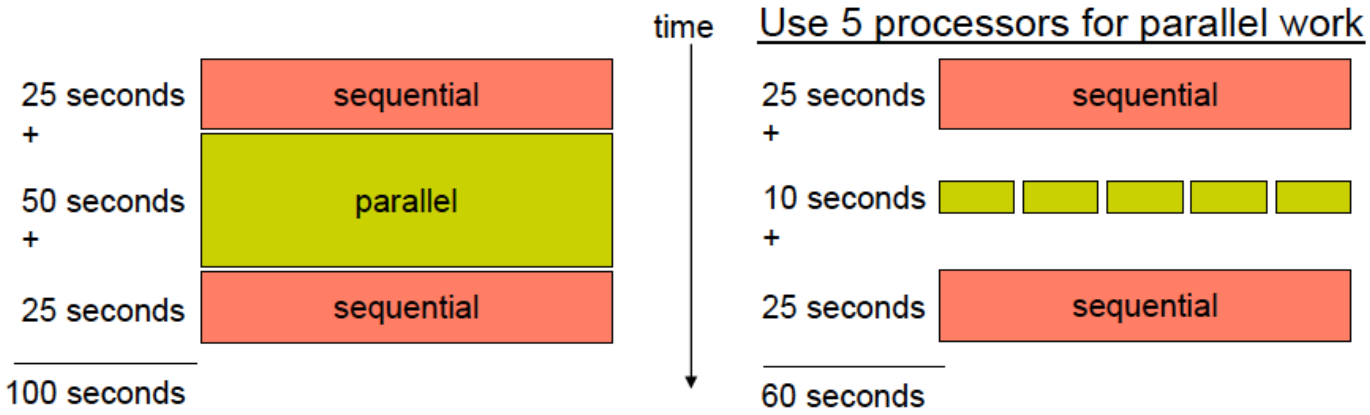
- Speedup $S_p$: relative reduction of execution time when using $P$ processors with respect to sequential




- Efficiency $Eff_p$: it is a measure of the fraction of time for which a processing element is usefully employed
  - $Eff_p = T_1/(T_p \times P)$
  - Also, $Eff_p = S_p/P$

# Amdahl's law

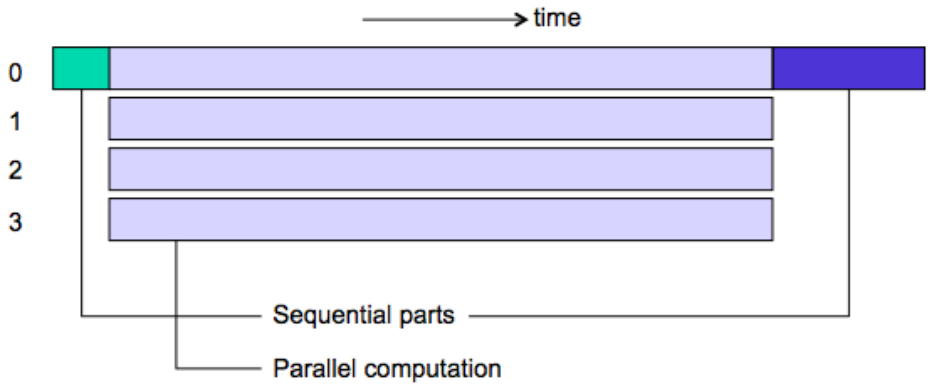
The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used




- Parallel part is $5$ times faster: $Speedup_{parallel\_part} = 50/10 = 5$
- Parallel version is just $1.67$ times faster: $S_p = 100/60 = 1.67$

# Amdahl's law

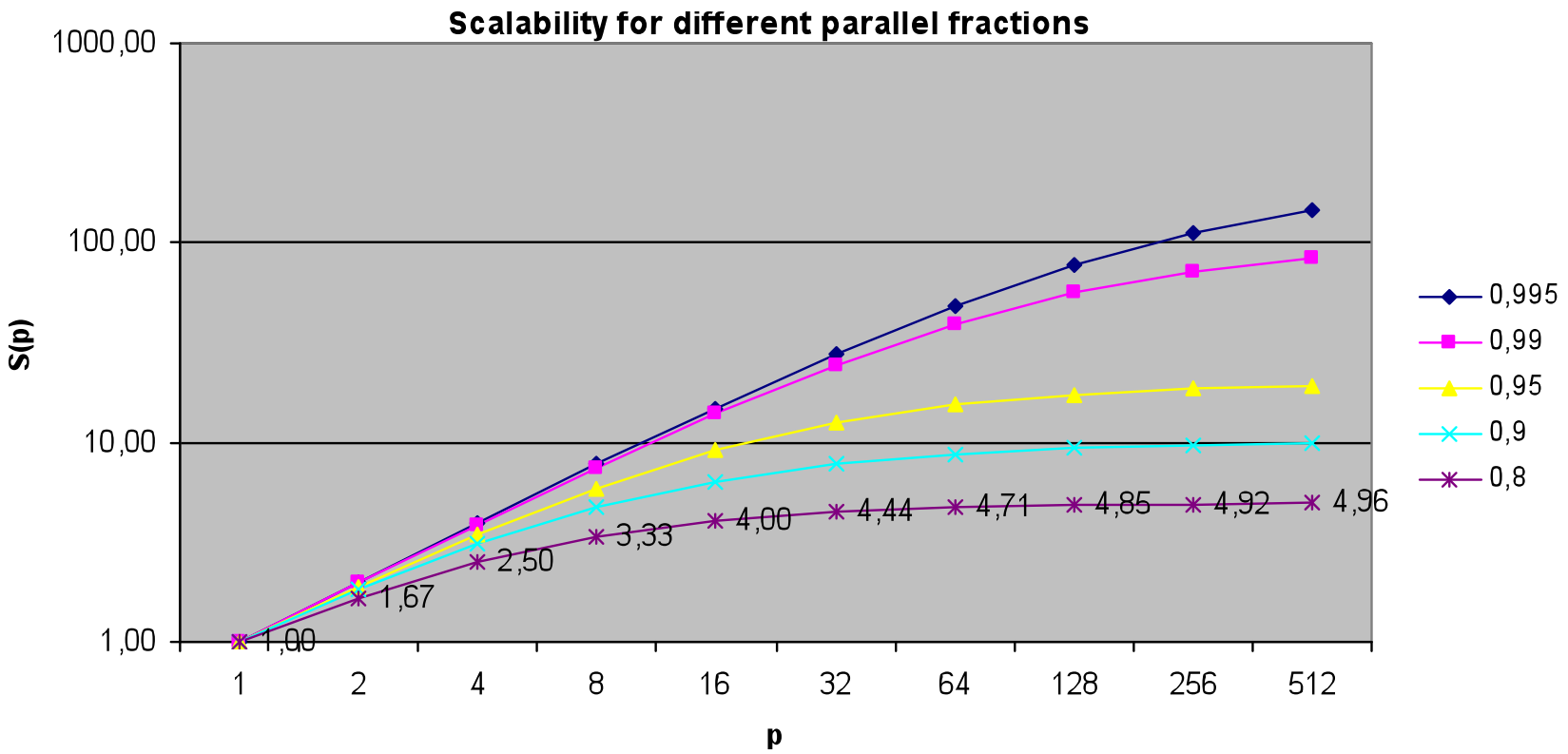
- Performance improvement is limited by the fraction of time the program runs in parallel (e.g. $\varphi$)




$$T_1 = T_{seq} + T_{par} = (1 - \varphi) \times T_1 + \varphi \times T_1$$
$$T_p = (1 - \varphi) \times T_1 + (\varphi \times T_1 / P)$$
$$S_p = \frac{T_1}{T_p}$$

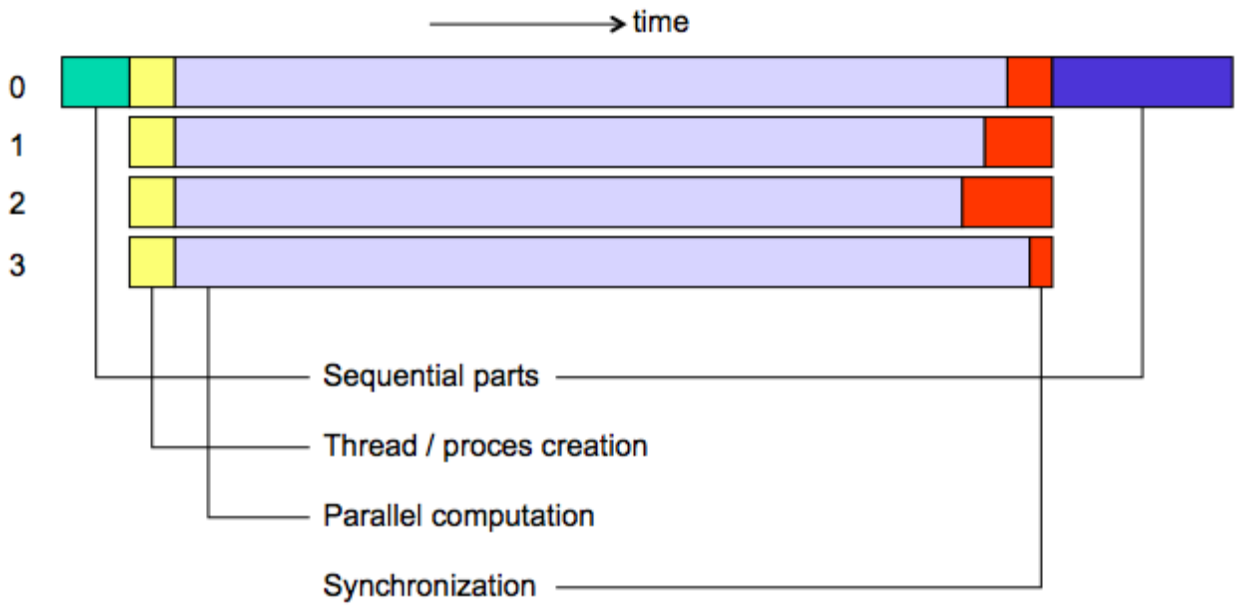
$$S_p = \frac{1}{((1 - \varphi) + \varphi/P)}$$
$$S_p \to \frac{1}{(1 - \varphi)} \; when \; P \to \infty$$

# Amdahl's law


Scalability for different parallel fractions
1000,00
100,00
10,00
1,00
S(p)
1
2
4
8
16
32
64
128
256
512
p
1,00
1,67
2,50
3,33
4,00
4,44
4,71
4,85
4,92
4,96
0,995
0,99
0,95
0,9
0,8

# Sources of overhead

Parallel computing is not free, we should account overheads (i.e. any cost that gets added to a sequential computation so as to enable it to run in parallel)



# Sources of overhead

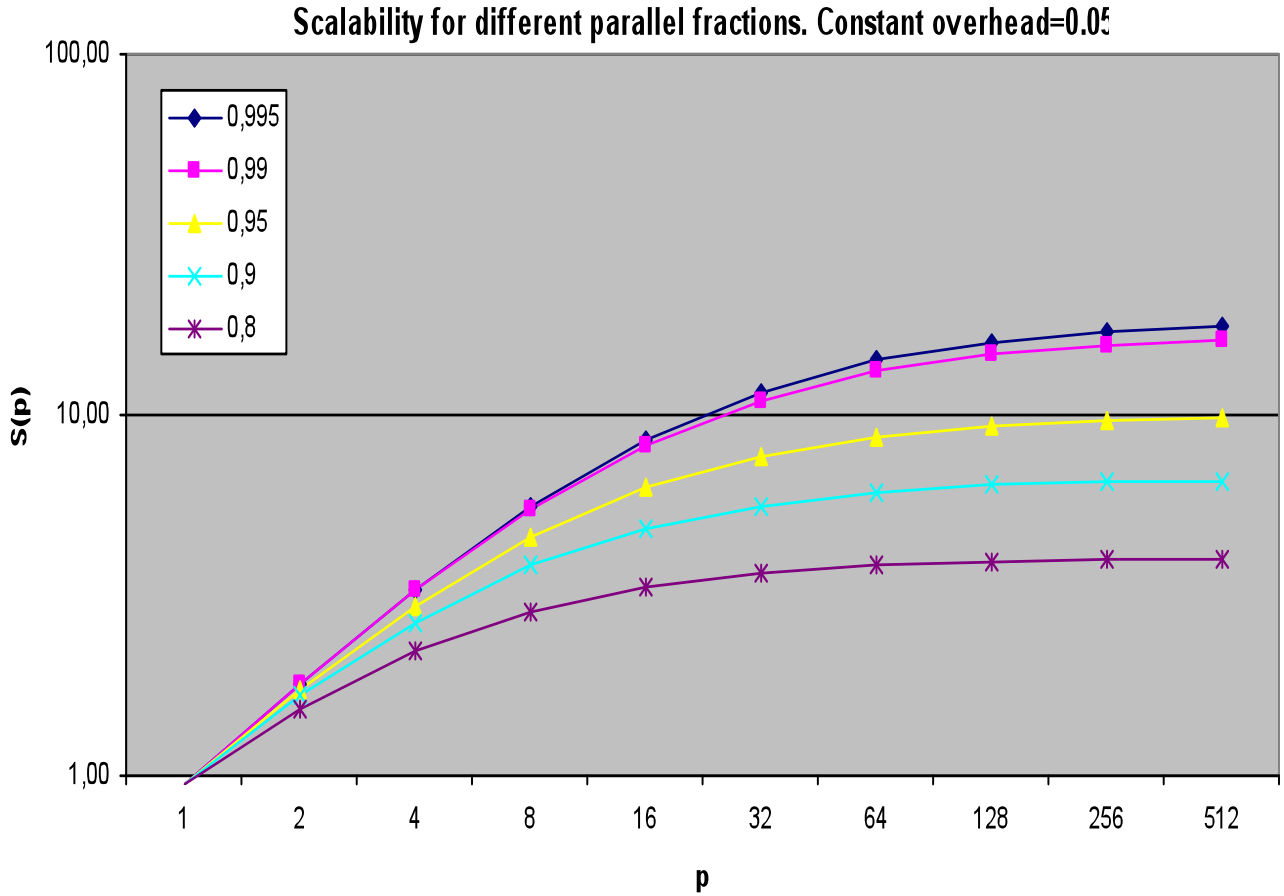
- **Task creation and termination:** extra processing performed at the start and end of each task
- **Synchronization:** extra processing to ensure that dependences in computation graph are satisfied
- **Data sharing:** can be explicit via messages, or implicit via a memory hierarchy (caches)

# Sources of overhead (cont.)

- **Idleness:** thread cannot find any useful work to execute (e.g. dependences, load imbalance, poor communication and computation overlap or hiding of memory latencies, ...)
- **Computation:** extra work added to obtain a parallel algorithm (e.g. replication)
- **Memory:** extra memory used to obtain a parallel algorithm (e.g. impact on memory hierarchy, ...)
- **Contention:** competition for the access to shared resources (e.g. memory, network)

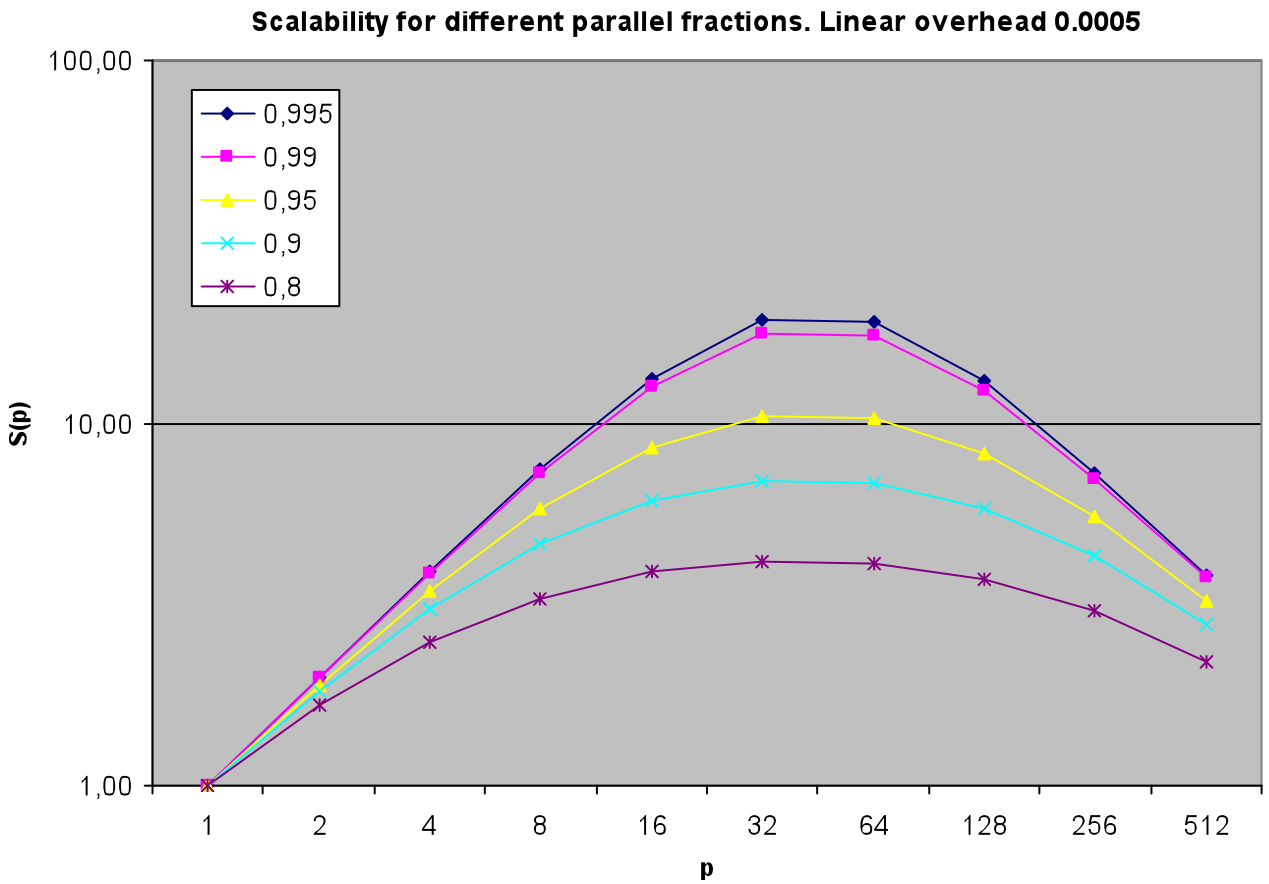
# Amdahl's law (constant overhead)

$$T_p = (1 - \varphi) \times T_1 + \varphi \times T_1/p + overhead$$



# Amdahl's law (linear overhead)

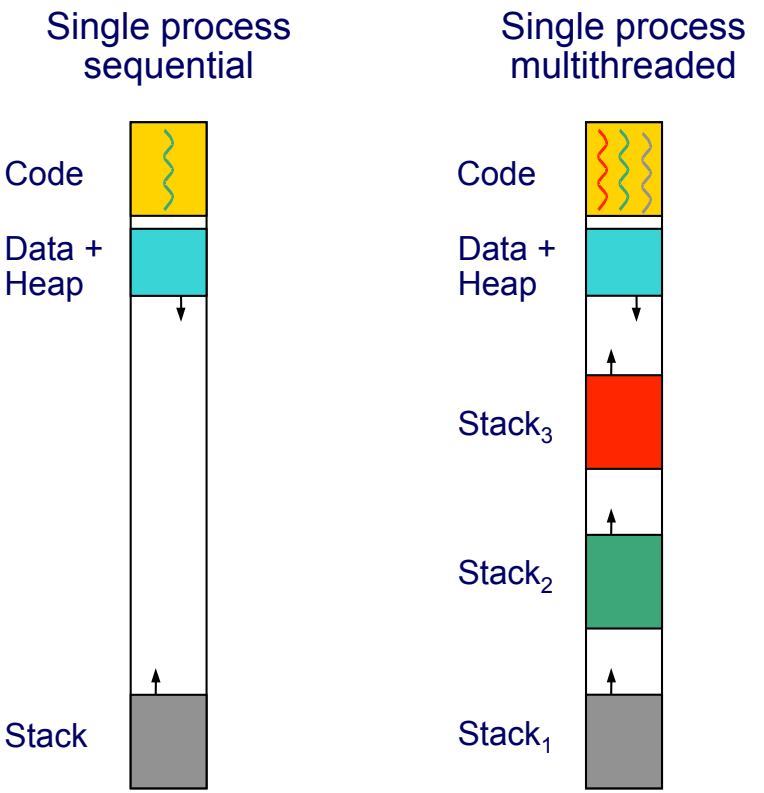
$$T_p = (1 - \varphi) \times T_1 + \varphi \times T_1/p + overhead_p$$



# Part III

# Address spaces in parallel programming

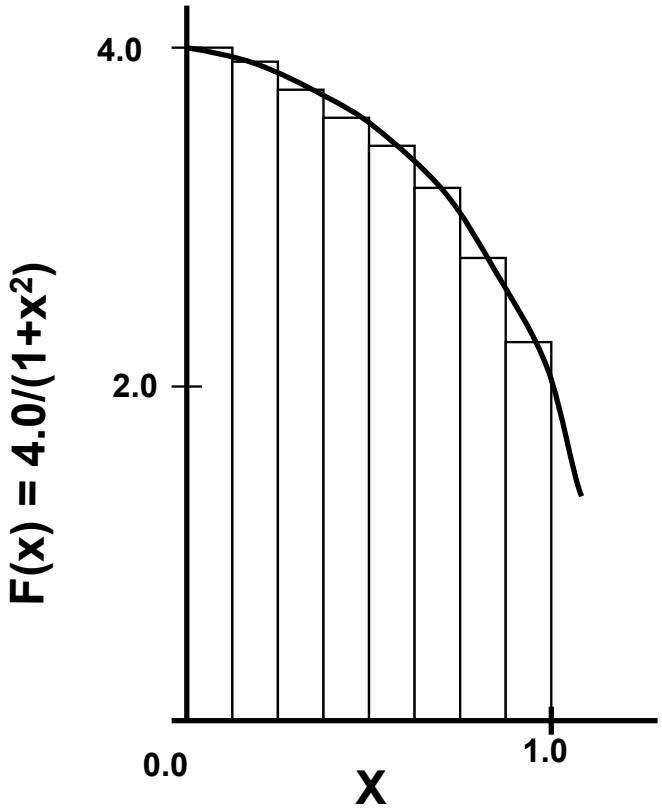
# Shared memory: address space




Programmer needs

- Distribute work
- All threads can access data, heap and stacks
- Use synchronization mechanisms to avoid data races

# Shared memory programming: example




Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width $\Delta x$ and height $F(x_i)$ at the middle of interval i.

# Shared memory programming: example (cont.)

Sequential code:

```
void main ()
{
        int i;
        double x, pi, step, sum = 0.0;
        long num_steps = 100000;

        step = 1.0/(double) num_steps;

        for (i=1;i<= num_steps; i++) {
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Work distribution: each processor responsible for computing some rectangles (in other words, to execute some iterations of loop `i`). Guarantee that all processors have computed their `sum` before combining into final value

# Shared memory programming: example (cont.)

OpenMP code (not the shortest one, just for illustration purposes):

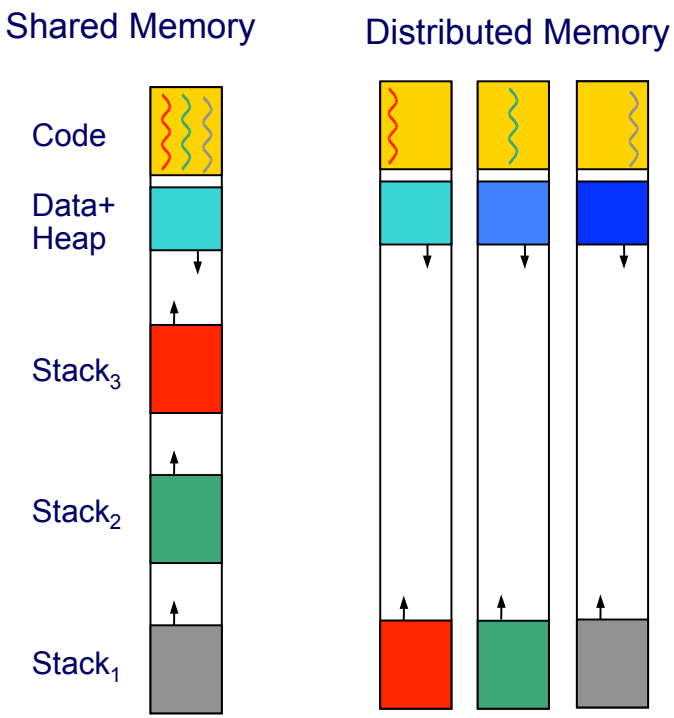
```
#include "omp.h"
#define NUM_THREADS = 4
void main ()
{
        int i, id;
        double x, pi, step, sum[NUM_THREADS];
        long num_steps = 100000;

        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);

#pragma omp parallel private(id)
        {
        id = omp_get_thread_num();
        sum[id] = 0.0;
#pragma omp for private(x,i)              // Implicit barrier synchronization at the end of for
        for (i=1;i<= num_steps; i++) {
                x = (i-0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
        }
#pragma omp single                       // Only one computing final result
        for (i=0, pi=0.0; i<NUM_THREADS; i++)
                pi += step * sum[i];
        }
}
```

# Shared memory programming: example (cont.)

OpenMP code:

```
void main ()
{
        int i;
        double x, pi, step, sum=0.0;
        long num_steps = 100000;

        step = 1.0/(double) num_steps;

#pragma omp parallel for private(x) reduction(+:sum)
        for (i=1;i<= num_steps; i++) {
                x = (i-0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
       pi = step * sum;
}
```

# Distributed memory: address space




Programmer needs

- Distribute work
- Distribute data
- Use communication mechanisms to share data explicitly
- Use synchronization mechanisms to avoid data races

# Distributed memory programming: example

Sequential code:

```
void main ()
{
        int i;
        double x, pi, step, sum = 0.0;
        long num_steps = 100000;

        step = 1.0/(double) num_steps;

        for (i=1;i<= num_steps; i++) {
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Data/work distribution: each processor responsible for computing some rectangles and have private copy of `sum`. Reduce partial (local) copies of `sum` into final value

# Distributed memory programming: example (cont.)

## MPI code:

```
#include <mpi.h>
void main ()
{
        int i;
        double x, pi, step, sum=0.0;
        long num_steps = 100000;
        int numprocs, myid;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);

        step = 1.0/(double) num_steps;

        for (i = myid; i < num_steps; i += numprocs) {
                x = (i + 0.5)*step;
                sum += (4.0 / (1.0 + x*x));
        }
        mypi = step * sum;

        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

# Distributed memory programming: example (cont.)

## Hybrid MPI/OpenMP code:

```
#include <mpi.h>
void main ()
{
        int i;
        double x, pi, step, sum=0.0;
        long num_steps = 100000;
        int numprocs, myid;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myid);

        step = 1.0/(double) num_steps;

#pragma omp parallel for private(x) reduction(+:sum)
        for (i = myid; i < num_steps; i += numprocs) {
                  x = (i + 0.5)*step;
                  sum += (4.0 / (1.0 + x*x));
        }
        mypi = step * sum;

        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

        MPI_Finalize();
}
```

# Distributed memory programming: example (cont.)

## Erlang code:

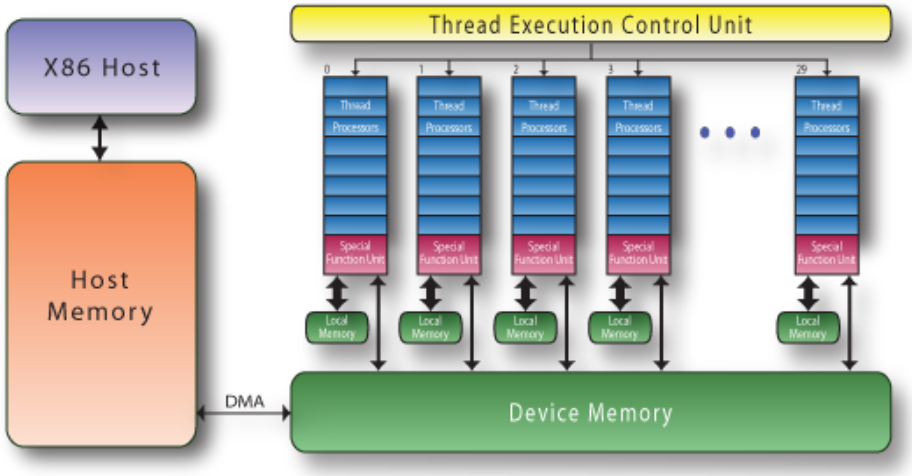
```
-module(pi).
-author("Zvi").
-compile([export_all]).

%% functional and parallel PI calculation
test()->
    N = 1000000,
    Impls = [serial,parallel],
    [ Impl,timer:tc(pi,calc_pi,[Impl,N]) || Impl<-Impls ].

%% 1. serial - tail recursion - increment index
calc_pi(serial,N) ->
    calc_pi_aux(1,N,1/N);

%% 2. parallel SMP version - process per scheduler
calc_pi(parallel,N) ->
    NWorkers = erlang:system_info(schedulers),
    NPerWorker = trunc(N/NWorkers+0.5),
    Step = 1/N,
    Self = self(),
    Pids = [spawn(fun() -> Self !
           sum,calc_pi_aux(I,lists:min([N,I+NPerWorker-1]),Step) end)
               || I<-lists:seq(1,N,NPerWorker) ],
           lists:sum([receive sum,S -> S end || _<-Pids]).
```

# Distributed memory programming: example (cont.)

Erlang code (cont.):

```
%% Auxiliary function
calc_pi_aux(From,To,Step) ->
    calc_pi_aux(From,From,To,Step,0).

calc_pi_aux(To,_From,To,Step,Sum) -> Step*Sum;
calc_pi_aux( I,From,To,Step,Sum) ->
    X=Step*(I-0.5),
    V=4.0/(1.0+X*X),
    calc_pi_aux(I+1,From,To,Step,Sum+V).
```

# GPU-enabled architectures: address space




Programmer needs

- Offload computation: low parallelism in CPU, data-parallel part in GPU
- Transfer data to devices and between devices: explicit communication
- Use synchronization mechanisms to avoid data races (between cores in GPU and between CPU-GPU)

# GPU programming: example

## Computation kernel for GPU:

```
#include <cuda.h>
#include <stdio.h>

// XXX float sums[size];

__global__ void pi_steps( float* sums, int size, float step, int num_steps,
           int steps_per_thread ) {

     int k = blockIdx.x * blockDim.x + threadIdx.x;
     int i = k * steps_per_thread;
     if( k < size ) {
          sums[k] = 0.0f;
          for( int j = 1; j <= steps_per_thread; j++ )
               if( i+j <= num_steps ) {
                    float x = ((float)(i+j) - 0.5f) * (float)step;
                    sums[k] += 4.0f/(1.0f + x*x);
                    }
          }
     }
```

# GPU programming: example

## Computation kernel for GPU (cont.):

```
// XXX: sizeof(reds) MIN size/reds_per_thread

__global__ void pi_reduction( float* reds, float* sums, int size, int reds_per_thread )
     {
     int k = blockIdx.x * blockDim.x + threadIdx.x;
     int i = k * reds_per_thread;
     if( k < size/reds_per_thread ) {
          reds[k] = 0.0f;
          for( int j = 0; j < reds_per_thread; j++ )
               if( i+j < size ) reds[k] += sums[i+j];
          }
     }
```

# GPU programming: example

Host code:

```
#define NUM_STEPS 1048576
#define REDS_PER_THREAD 2
#define STEPS_PER_THREAD 8
#define MIN_RED 256
#define THREADS_PER_BLOCK 256

int main( ) {
     float *cu_sums, *cu_reds;
     float *red = (float*) malloc( MIN_RED * sizeof(float) );

     int num_sums = NUM_STEPS/STEPS_PER_THREAD;
     int reds_size = num_sums/REDS_PER_THREAD;
     cudaMalloc( (void**)&cu_sums, num_sums * sizeof(float) );
     cudaMalloc( (void**)&cu_reds, reds_size * sizeof(float) );
     ...
```

# GPU programming: example

Host code (cont.):

```
...
dim3 block( THREADS_PER_BLOCK );
dim3 grid( num_sums / THREADS_PER_BLOCK );

float step = 1.0f/(float)NUM_STEPS;
pi_steps<<<grid,block>>>( cu_sums, num_sums, step, NUM_STEPS, STEPS_PER_THREAD );

float* t_sums = cu_sums;
float* t_reds = cu_reds;
int t_sums_size = num_sums;
while( t_sums_size > MIN_RED ) {
     dim3 block_red( THREADS_PER_BLOCK );
     dim3 grid_red( reds_size / THREADS_PER_BLOCK );
     pi_reduction<<<grid_red,block_red>>>( t_reds, t_sums, t_sums_size, REDS_PER_THREAD );
     t_sums_size /= REDS_PER_THREAD;
     float* tmp = t_reds;
     t_reds = t_sums;
     t_sums = tmp;
     }
...
```

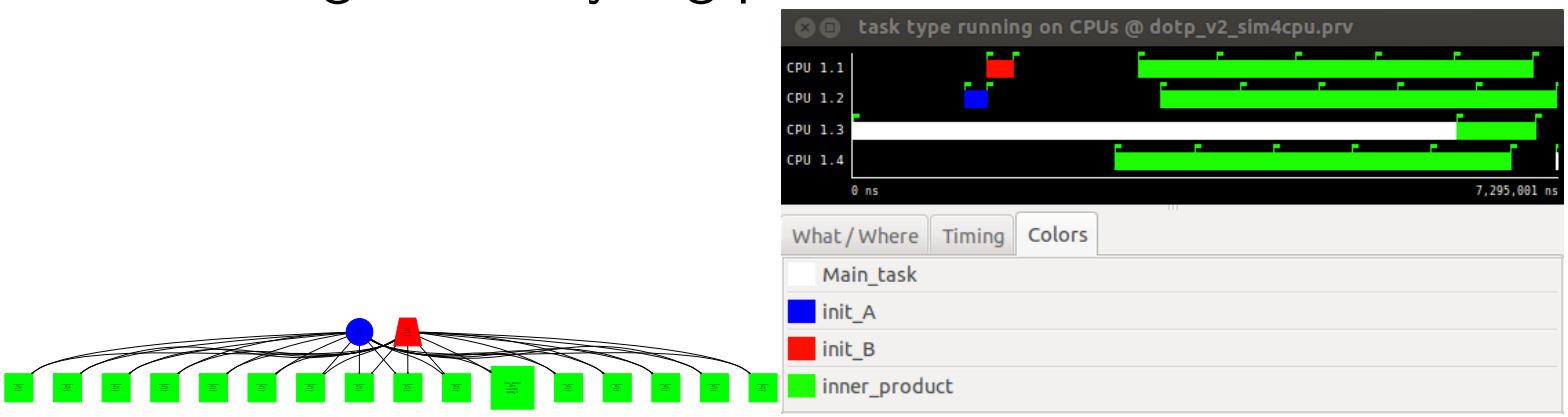
# GPU programming: example

Host code (cont.):

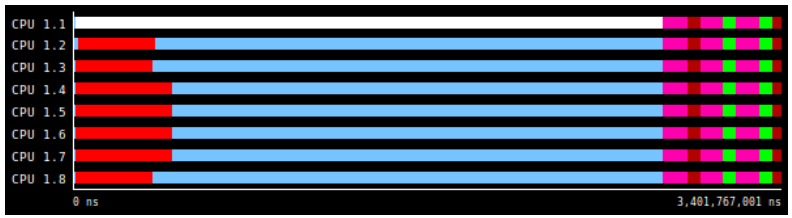
```
    ...
    cudaMemcpy( red, t_sums, sizeof(float) * MIN_RED, cudaMemcpyDeviceToHost );

    float sum_gpu = 0.0f;
    for( int i = 0; i < MIN_RED; i++ )
        sum_gpu += red[i];
    float pi_gpu = step*sum_gpu;
    printf("Pi GPU = %f \n", pi_gpu );
    return 0;
}
```

# Organization of Module II: Parallelism

- Modeling and analyzing performance




- Distributed-memory programming using MPI
- Programming GPU devices for computation acceleration using CUDA



# Concurrency, Parallelism and Distributed Systems (CPDS)

## Understanding parallelism

Eduard Ayguadé, Josep-Ramon Herrero, Daniel Jiménez
({eduard,josepr,djimenez}@ac.upc.edu)

Computer Architecture Department
Universitat Politècnica de Catalunya

2014/15 - Fall