

High Performance Computing

Course Presentation

Francesco Moscato

Università degli Studi di Salerno
fmoscato@unisa.it

Presentation

Lecturer

Francesco Moscato

Contacts

e-mail: fmoscato@unisa.it



This Course



Course

High Performance Computing (6cr.)

Lessons

On-line on MS Teams

Both Theoretical and Practical with on-line exercises and projects assignments

Resources

E-Learning Platform: <https://elearning.unisa.it/course/view.php?id=XXXXandYYYY>



Examinations

Verification

During the Course, some checkpoints (results are not part of the final grade)

Final Project

Single Student - Final Projects (Design and Implementation of Parallel Software)

Final Project bases during the course

Exams

Defending a dissertation (Presentation, demo and testing) on your Final Project some questions about the whole program



Prerequisites



Mandatory

C Programming, C compiler, Unix shell, Basic Unix system calls, Asymptotic Notation and evaluation of algorithms Computational Complexity, Linear Algebra basics, Algorithms and Data Structures.

Optional

Make and makefiles, GIT, Complex Algorithms, Compilers Architecture, “Many other stuffs I do not remember know”

Really Mandatory

Patience¹

¹Guns N' Roses, G N' R Lies (1989)



Course Outline

- Parallel Architectures, Models and Techniques for parallel algorithm design and evaluation
- Shared Memory Systems and Programming (multi-threaded and OpenMP)
- Message Passing Systems and Programming (MPI)
- GPGPU parallel Architectures and Programming (CUDA)



Course Material

Books

- Peter Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann
- I. Foster. Designing and Building Parallel Programs. Addison-Wesley
- Jason Sanders, Edward Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley

Other Stuff

Manuals, Standards, APIs

Source Code

...



Enjoy ...

You are at the end of a narrow tunnel opening in an immense cavern. A HUGE Red Dragon is looking at you ...



Introduction to HPC

Parallel Architectures and Reference Models

Francesco Moscato

HIGH PERFORMANCE COMPUTING

LM Ingegneria Informatica

Università degli Studi di Salerno

e-mail:fmoscato@unisa.it

Outline



① Introduction

Once Upon a Time

② Parallel Architectures

③ Reference Models

④ Prerequisites



Once Upon a Time: Moore's Law

First Moore Law (1965)

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.



Complexity of IC

measured in numbers of transistors per chip, doubles up every 18 months.

Improvements



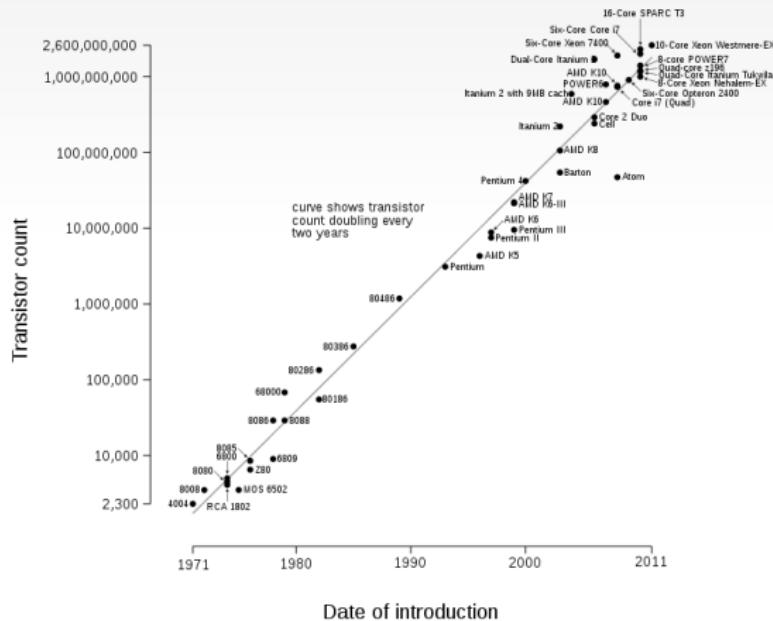
Memory Capacity, CPU components, Frequency, etc.

Performances ...



Moore's Law During the Years

Microprocessor transistor counts 1971-2011 & Moore's law



(image of Wg simon, CC BY-SA 3.0. <https://commons.wikimedia.org/w/index.php?curid=15193542>)



Moore's law problems

Physical limits

Too High Scale of Integration (and frequency) \implies passive quantic noises.

The Speed of Light!

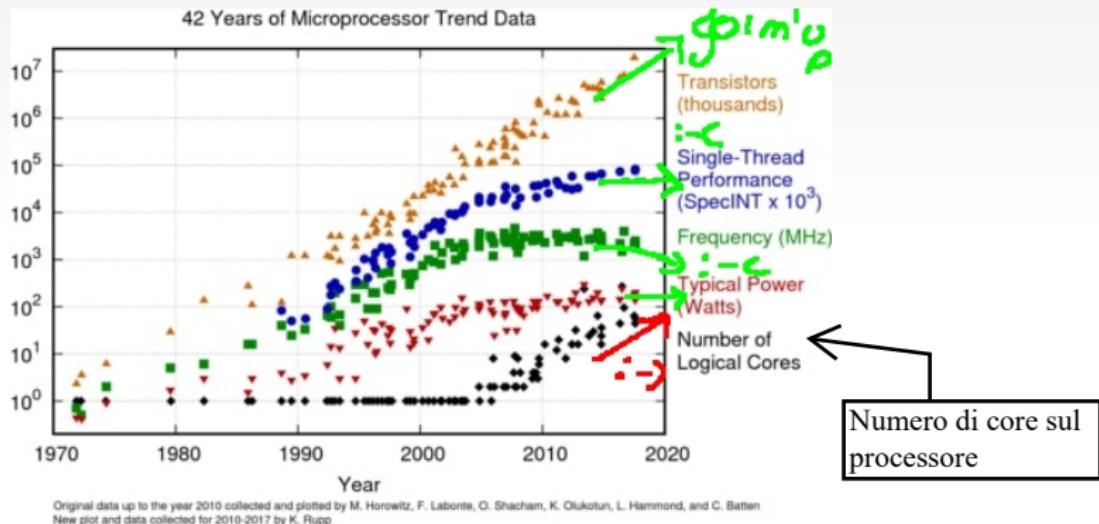
You cannot increment number of transistors to increase *speed* of the same chip.

Once Upon a Time ...

In the Age of *Pentium 4*, we reached that limit



What's happening: not only transistors



(Plot by Karl Rupp from his microprocessor trend data (CC BY 4.0 license))



The Second Moore's Law



Second Moore's Law

The cost of a semiconductor chip fabrication plant doubles every four years

Parallelism

It is more *affordable* to improve performances by using more units of simpler chips

New Age...

... of multicore / multithread CPUs

Parallele **sullo stesso chip**,
hanno la memoria in comune.

(not so) New Age...

... of multicompilers



Little bit of History of Supercomputers

FLOPS: Floating Point Operations / Seconds [in pipeline]

- (1993): Fujitsu Numerical Wind Tunnel - 124.50 GFLOPS;
- (1996): Hitachi CP-PACS 2048 - 368.2 GFLOPS;
- (2002): NEC Earth Simulator - 35 TFLOPS;
- (2020): NVIDIA RTX 3090 - 35 TFLOPS;
- (2007): IBM Blue GeneL - 478.2 TFLOPS;
- (2012): Cray Titan - 17.89 PFLOPS;
- (2016): Sunway TaihuLight - 93.01 PFLOPS;
- (2018): IBM Summit - 122.3 PFLOPS;
- (2020): Fugaku - 415.53 PFLOPS.
- many and many others ...



Parallel Computing

Parallel Computing is more than something to increase performance.

More and More Scaling

It is *Vision* on how to scale from a single processor to virtually limitless computing power.

Why We Need Ever-Increasing Performances

More and more complex models and simulations (climate modeling, protein folding, DNA mapping, Energy research, Data Analysis)



Parallel Programming



Why We need Parallel Programming

Models of Parallel machines are not the same of sequential machines (i.e. Von Neumann - based models)

Parallel Machine Models

depend on Parallel Architectures (we will see some models later)

Parallel Algorithms and Programs

Parallel Algorithms try to exploit parallel architectures features.
Parallel Programs usually use languages and libraries designed to implement *parallel algorithms* for specific *parallel architectures* based on proper *parallel systems and frameworks* (i.e. HW, SW, OS, libraries, languages and compilers etc.)



Flynn Taxonomy

A Taxonomy to Classify Computer Architectures

it mainly depends on instruction flows and data flows;
it was extended to include different memory architectures.

Asynchronous Architecture

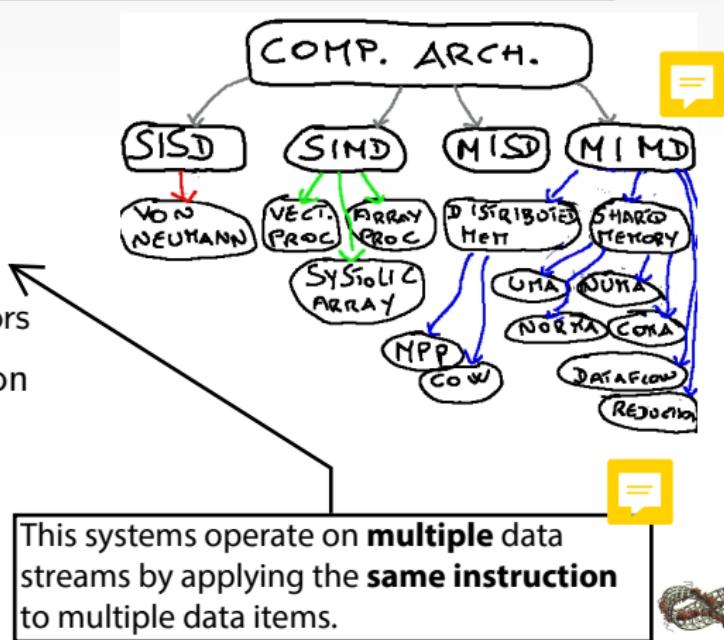
In 2003 DEC developed an *Asynchronous* CPU working without *clock*. Since they are auto-synchronized, they are outside Flynn Taxonomy.



Flynn Taxonomy (2)

it executes a **single instruction** at a time and, in most cases, computes a single data value at a time (processi lavorano su dato atomico)

- **SISD**: Single Instruction, Single Data
- **SIMD**: Single Instruction, Multiple Data
 - Vector Processors
 - Array Processors
 - Systolic Array Processors
- **MISD**: Multiple Instruction Single Data



La prima lettera fa riferimento ai flussi che vengono eseguiti parallelamente nel sistema: o è uno, o è più di uno.

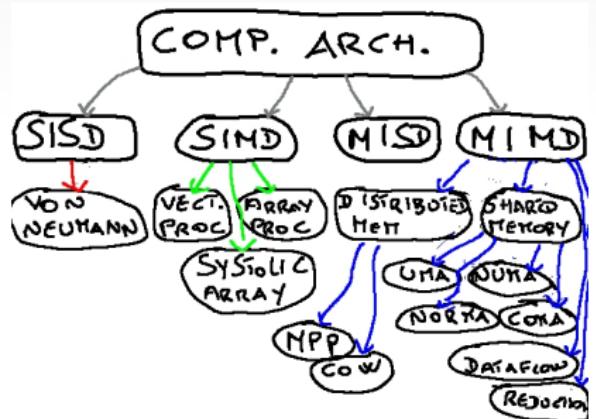
La seconda relativamente al fatto che l'istruzione lavora su scalari, o su elementi più complessi

Flynn Taxonomy (3)

- **MIMD:** Multiple Instruction
Multiple Data

- Distributed Memory Systems:
 - MPP: Massively Parallel Processing
 - COW: Clusters of Workstations
- Shared Memory Systems
 - UMA: Uniform Memory Access
 - NUMA: Non UMA (No Cache - NUMA; Cache Coherent - NUMA)
 - NORMA: NO Remote Memory Access
 - COMA: Cache Only Memory Access
- Dataflow Machines
- Reduction Machines

Multiple Instruction: più processi in parallelo, in maniera reale, non virtualizzata, ma offerta dall'architettura stessa del calcolatore.



Flynn Taxonomy (4)

SIMD

No Parallelism: Sequential execution of instructions. This usually include *pipelined* CPUs even with Out of Order execution of instructions, but only with instructions set with *scalar* operands.

SIMD

The same instruction executes on different data.

Vector Processors: *different data* are on **Vector registers**. The instruction is executed on all the element of the vector: this needs high speed links to memory, that usually is aligned in vectors too (thus we can duplicate buses, buffers, etc.). E.G. *Intel MMX extensions of x86 architecture* (further extended in 2011 with Advanced Vector Extensions); Graphic CPU etc.



Flynn Taxonomy (5)

SIMD

Array Processors: They do not have scalar instructions, The Control Unit of the CPU manages arrays of Processor Elements. PE and Memory are connected each other by matrix-based links (each PE communicates with his 4 neighbors etc.). The CU *routes instructions* to the PEs that manages a part of the vector. An evolution is the *Connection Machine* where complex *cells* substitute PEs.



Flynn Taxonomy (6)

SIMD

Systolic Array Processors: Manages (vector) instructions and data that flow in a *predictable* way. Computation in arrays depends only on input data and processor status. Output data will be managed by other connected elements. This computation is usually executed during signal elaboration.

MISD

More instructions flows on the same data. The grain is on the processes ... Practically never used.



Flynn Taxonomy (7)

MIMD

Multiple Instructions executed on Multiple, different Data, this is the reference model for Cluster computers (many independent computers connected by high speed networks).

Distributed Memory: Every Computing Element (CE) has its own slice of memory. In order to exchange data among private memories, CEs must use **Message Passing** (on networks).

Massively Parallel Processing architectures are Distributed Memory -based with hundreds of thousands of CEs connected by high speed networks; **Clusters of Workstations** are similar to these latter, but CEs are common computers connected each other.

I Clusters of Workstations, sono "figli piccoli" della MPP, in quanto i CEs sono tipicamente dei GP-PC, mentre le reti di comunicazione sono reti generalmente utilizzate (ad es. LAN, Wi-Fi), mentre nella MPP ci sono un sacco di CEs in collegamento con reti ad altà velocità appositamente progettate.



Flynn Taxonomy (8)

MIMD

Shared Memory: CEs share the same (centralized) memory area. **UMA** have a central memory where access time to any memory block is constant for all CEs. This is difficult to implement (the Bus arbiter is complex) and it usually does not support more than few dozens of CEs. **NUMA** allows non-uniform access time to memory block (usually because of caches). **NORMA** architecture allows only private caches for CEs. They use message passing to share local data.



Flynn Taxonomy (9)

Dataflow machines

There is no Program Counter ... Programs are not sequences of instructions, Dataflows (diagrams) define instruction and data management precedences. They require proper languages that implement a data-driven approach. There are some prototypes in literature.

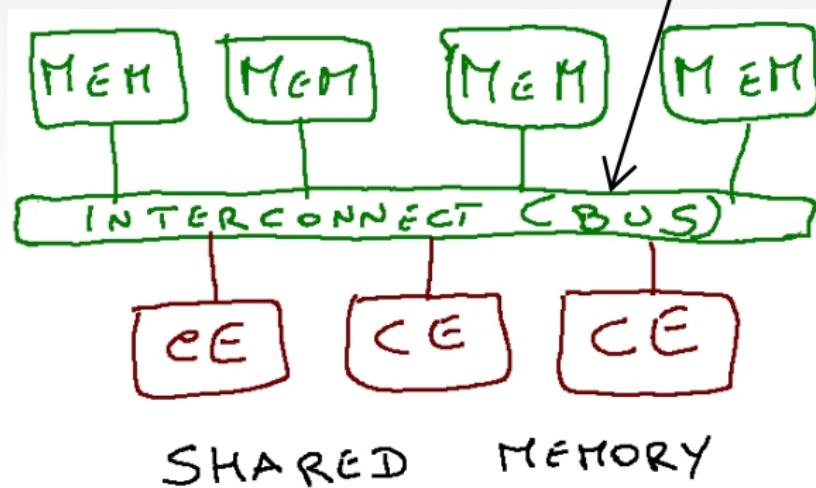
Reduction machines

Similar to Dataflow machines, but they use a demand-driven policy to dataflow executions.



Shared Memory

Nel caso di bus vecchi, se più CE vogliono accedere alla stessa locazione di memoria, devono attendere che la risorsa condivisa (il bus) si liberi, oppure i bus devono essere duplicati, ma chiaramente non è possibile duplicarli in maniera molto estesa (sono HD)

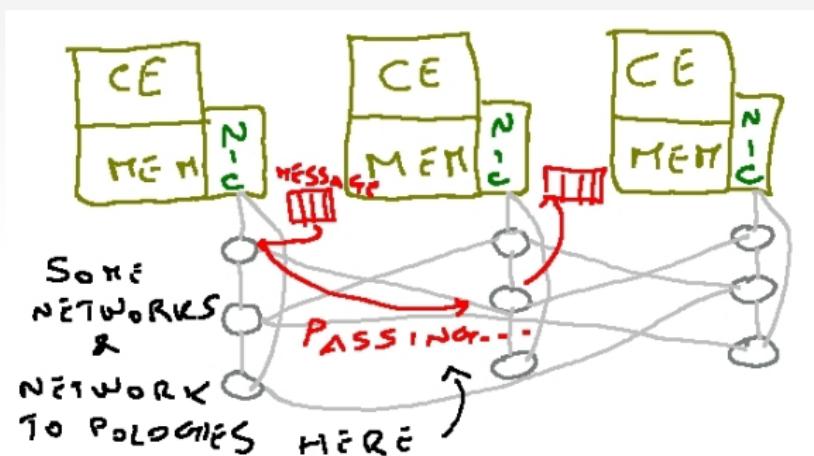


La memoria potrebbe essere unica (come sul libro), oppure diversi banchi di memoria condivisi come questi. I buffer di connessione vengono realizzati in hardware sui principali processori su singolo wafer (come quelli General Purpose), da esso dipendono le prestazioni del sistema.



Distributed Memory

NIC: Network Interface Card



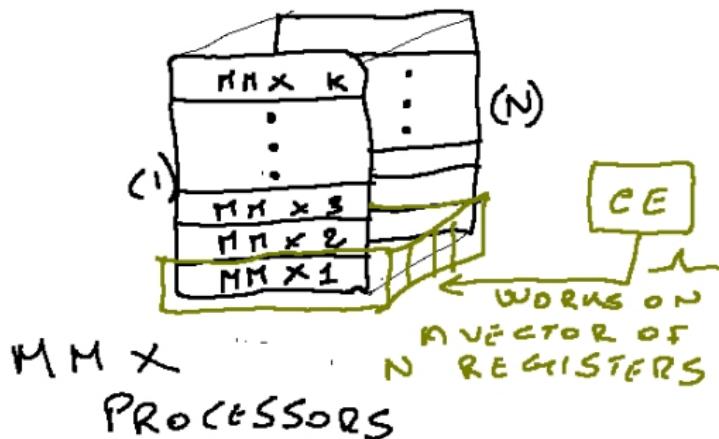
DISTRIBUTED MEMORY

Il Message Passing realizza l'astrazione della memoria condivisa attraverso l'invio di dati presenti nella memoria di un CE ad un altro CE che ne fa richiesta. A seconda della topologia di rete, è possibile ottimizzare l'algoritmo in funzione della topologia che si sta utilizzando, o viceversa scegliere la migliore topologia di rete per la sua implementazione (ad es. andare su un token ring piuttosto che a stella)



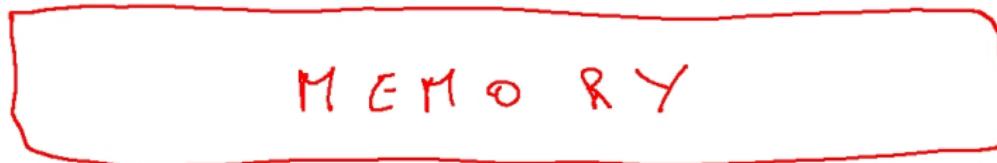
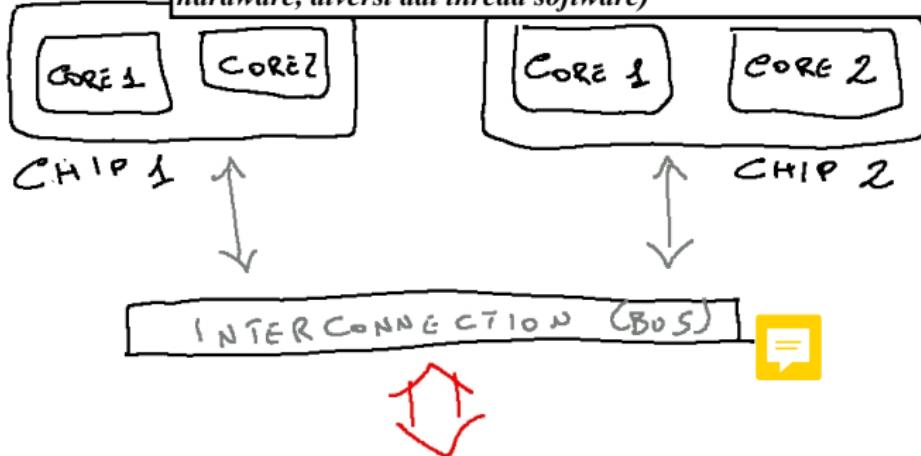
MMX

Ci sono N banchi di registri duplicati in vettori di N registri ciascuno, su cui il CE esegue l'istruzione.



UMA Multicore

Sullo stesso wafer, ci sono più CEs che lavorano in parallelo ognuno che fa girare un suo processo. Intel mischia multicore e multithread (*a livello hardware, diversi dai thread software*)



UMA

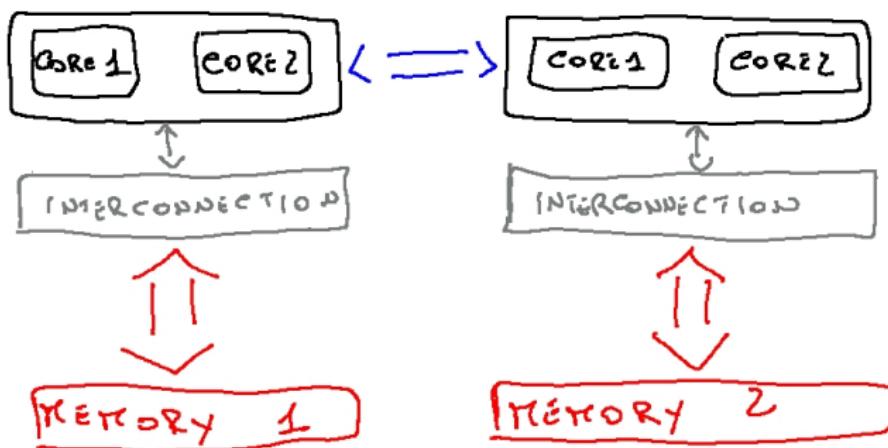
MULTI CORE



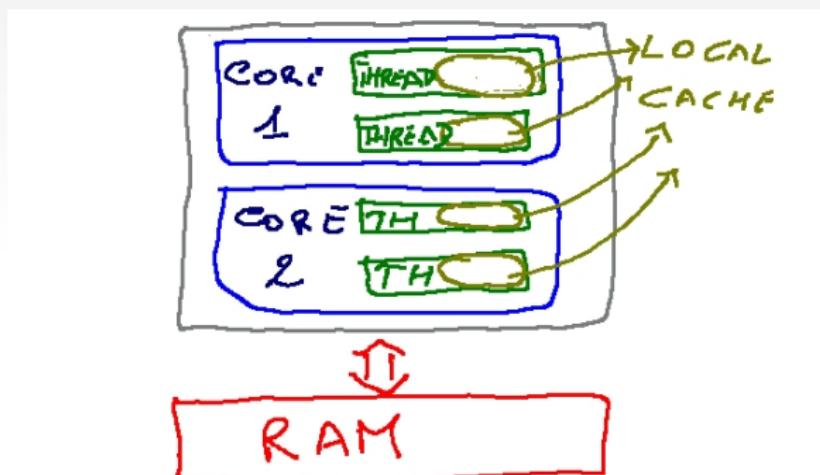
NUMA Multicore

Localmente ho un'architettura UMA, ma non viene mantenuto l'uniformità dell'accesso alle memorie su cui accedono i singoli core. Sulle proprie memorie, ogni core lavora come se fosse UMA, ma, tra di loro, ci sono accessi non uniformi alle memorie di cui non sono proprietari

NUMA MULTICORE SYSTEM



Multicore - Hyperthreading

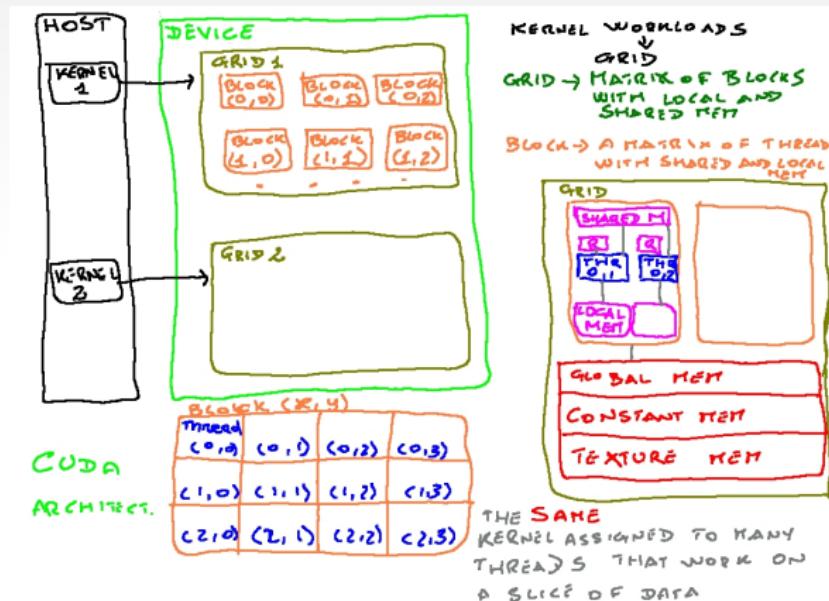


MULTICORE - HYPERTHREAD

In questo caso, i thread si comportano come se fossero UMA in quanto hanno delle loro memorie (local cache), i vari chip accedono poi alla RAM in maniera più o meno uniforme



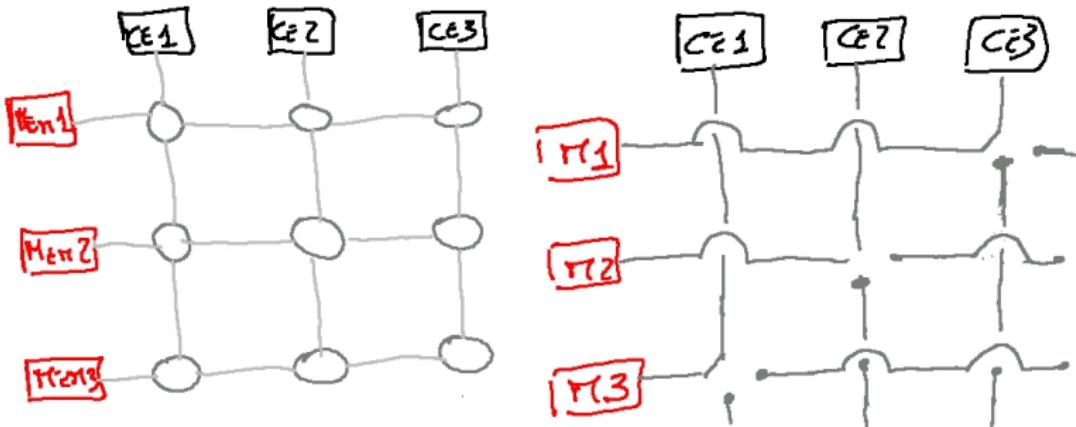
GP-GPU CUDA



Sfruttano le architetture GPU per calcoli General Purpose. Ho tantissimi processori che sanno fare poche cose. Ogni processore fa poche istruzioni su pochi dati, scrivendo il risultato sulla matrice. Fanno quindi più o meno le stesse istruzioni in parallelo su dati diversi.

Interconnection Examples (1)

Per parallelizzare, devo fare in modo che i vari CEs non si sovrappongano.

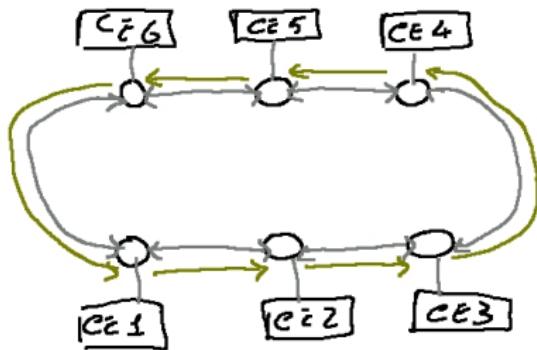


CROSS BAR
(SHARED MEMORY)

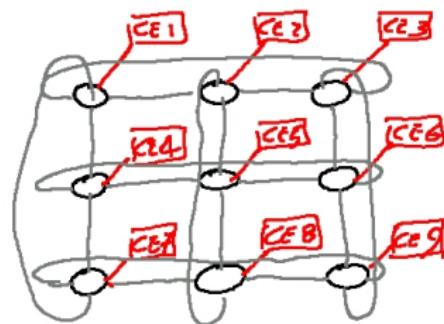
I vari switch vengono collegati in base alle operazioni **read** and **write** dei processori



Interconnection Examples (2)



BIDIRECTIONAL
RING
NON-DIRECTIONAL



TOROIDAL MESH

DISTRIBUTED MEMORY



An Example: Sum up elements in a vector

SUM OF THE ELEMENTS IN AN ARRAY

let A be array of N
 $\text{sum} \leftarrow 0$
 for $i \leftarrow 1$ to n do:
 $\text{sum} \leftarrow \text{sum} + A[i]$

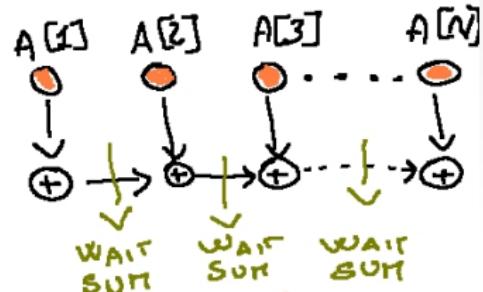
IF $t_{add} = 1$
 $T_{sum} \geq t_{add} \cdot N$

PRAT: PARALLEL RATE
 MEM

CEs



Ci sono anche le istruzioni per gestire il for



This uses only 1 CES

Assunzione: tutti i CES possono accedere contemporaneamente alla memoria

Reduce complexity (1)

L'ipotesi è quella di avere N/2 Ces.

LET'S TRY TO
REDUCE PROCESSING TIME . . .

Reduce complexity (2)

LET'S TRY TO
REDUCE PROCESSING
TIME

...

NICE!

+ IS ASSOCIATIVE
 $a + (b + c) = (a + b) + c$
 $(= a + b + c)$

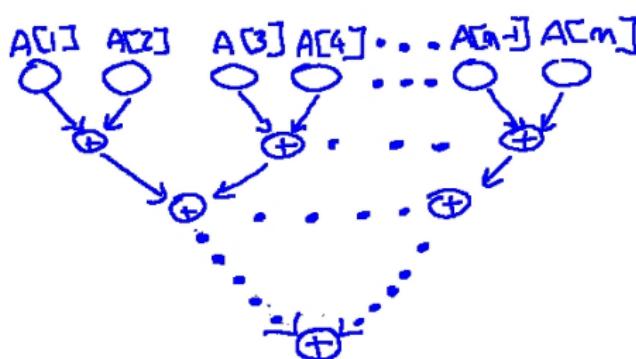
WE CAN DIVIDE THE
PROBLEM AND ASSOCIATE
PARTIAL RESULTS!

Reduce complexity (3)

Accorpo a 2 a 2 perché la **add** nei processori classici ha 2 operandi

LET'S TRY TO
REDUCE PROCESSING
TIME

...



NICE!

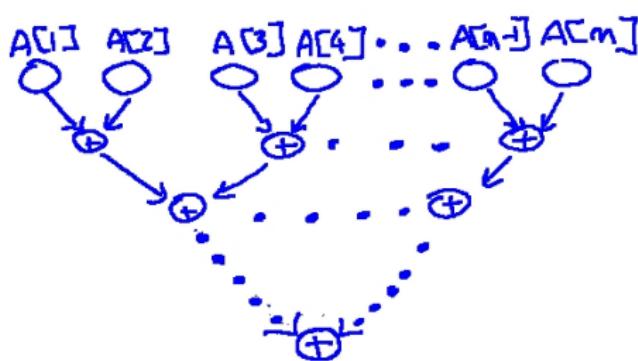
+ IS ASSOCIATIVE
 $a + (b + c) = (a + b) + c$
 $(= a + b + c)$

WE CAN DIVIDE THE
PROBLEM AND ASSOCIATE
PARTIAL RESULTS!

Reduce complexity (4)

LET'S TRY TO
REDUCE PROCESSING
TIME

...



NICE!

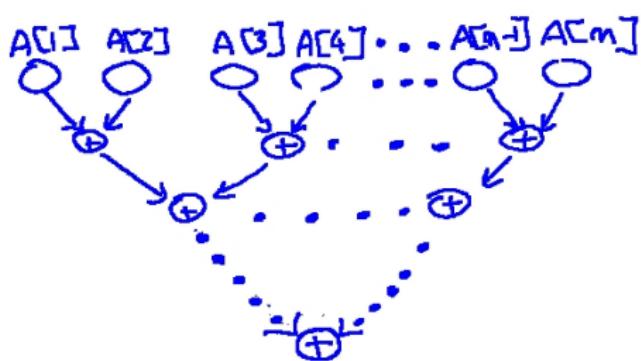
+ IS ASSOCIATIVE
 $a + (b + c) = (a + b) + c$
 $(= a + b + c)$

WE CAN DIVIDE THE
PROBLEM AND ASSOCIATE
PARTIAL RESULTS!

WHAT IS THE MINIMUM
EXECUTION TIME
WITH N CES?

Reduce complexity (5)

LET'S TRY TO
REDUCE PROCESSING
TIME

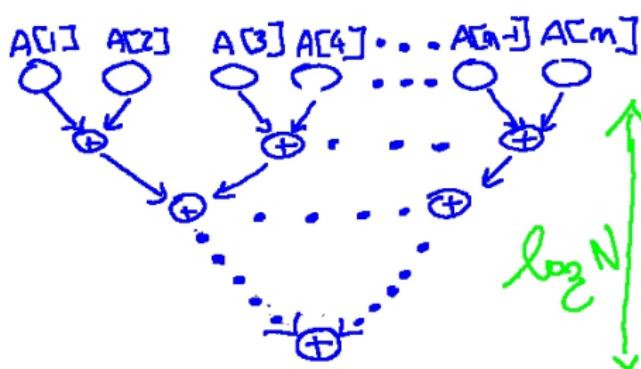


TIME ON N CES
(t_{ADD})

The equation $\sum_i^n 1$ represents the total processing time on n cores. The term 1 is enclosed in curly braces above the summation symbol, indicating that each core processes one unit of work. The summation symbol \sum is followed by a question mark, suggesting a query about the total time or the number of cores required.

Reduce complexity (6)

LET'S TRY TO
REDUCE PROCESSING
TIME

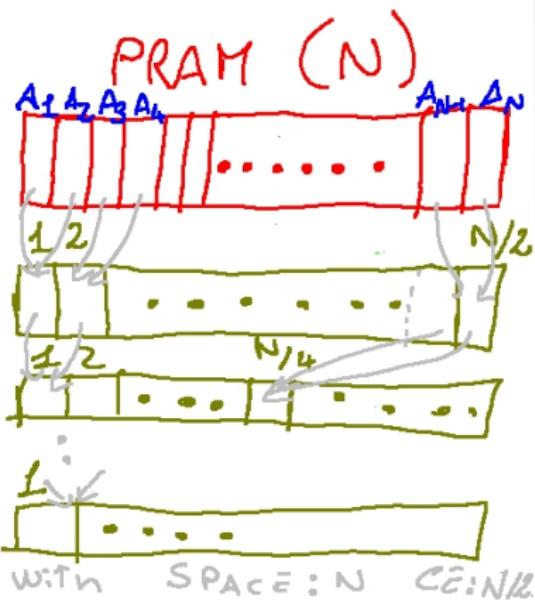
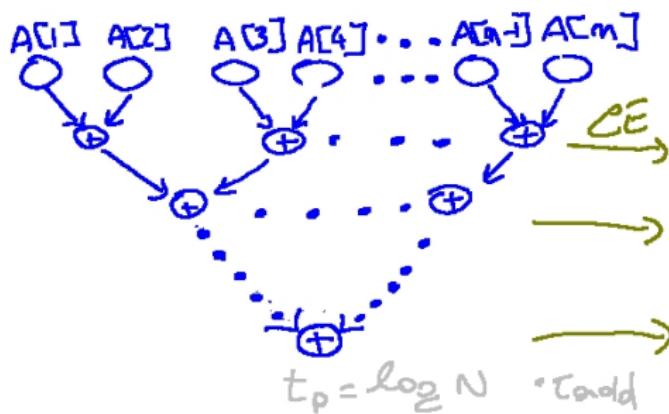


TIME ON N CES
(t_{ADD})



Reduce complexity (7)

LET'S TRY TO
REDUCE PROCESSING
TIME



Missed Anything ?

This model is too simple

What about *Taxonomy*?

Where are *Network Topologies* and *CEs links*?

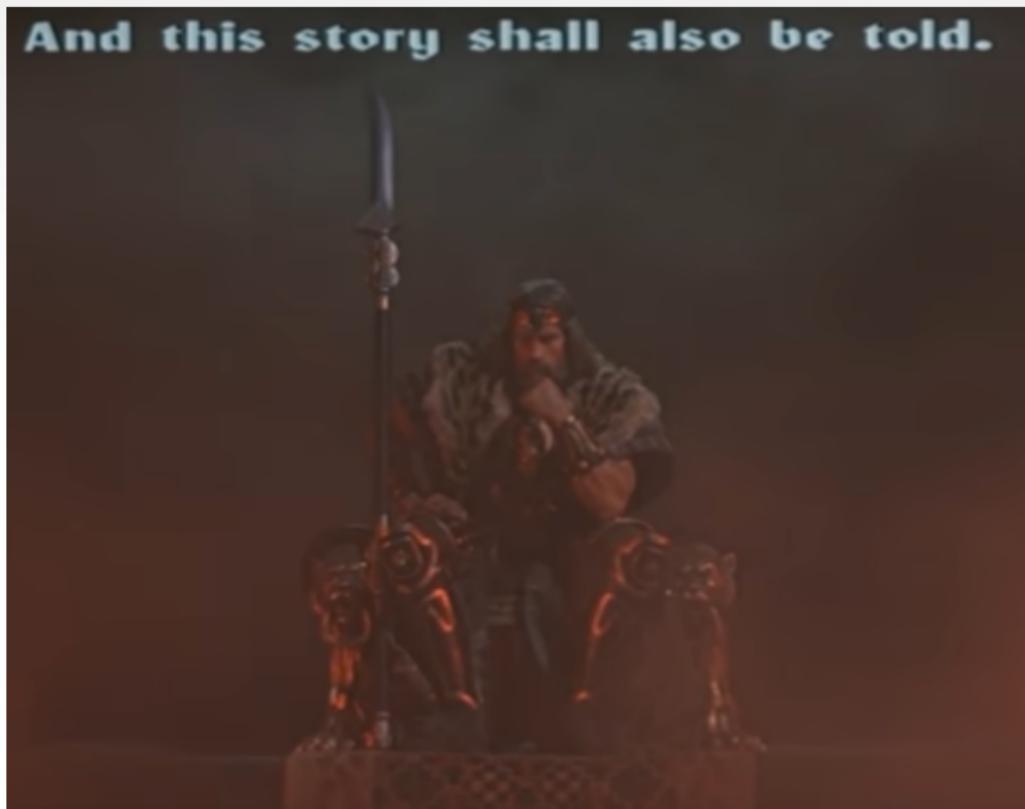
Do we take into account only execution time of an *add* operation ?

And how long does it take ?



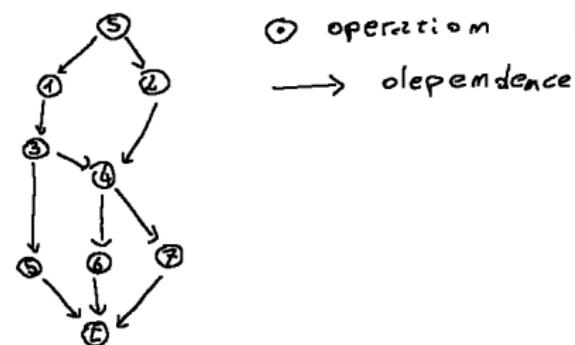
..and Conan Rules

And this story shall also be told.



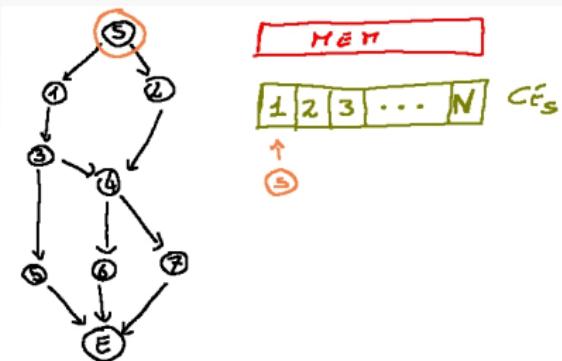
DAG and Scheduling (1)

- We are usually not so lucky like in summing up elements of a vector
- A more general model to describe control and data flows is a Directed Acyclic Graph (DAG)
- nodes are operations
- edges model data or control dependencies
- We assume we have here a simple DAG both for control and data flows.



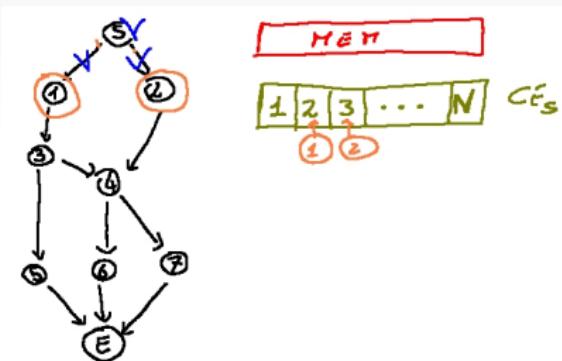
DAG and Scheduling (2)

- The scheduling problem is the problem of assigning CEs to operations in the DAG
- We assign an operation if the operation is ready (all precedences checked), and if we have enough CEs free in the pool.
- Scheduling ends when execute the **E** operation



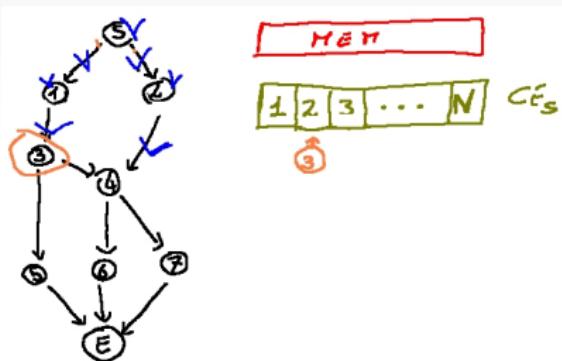
DAG and Scheduling (3)

- The scheduling problem is the problem of assigning CEs to operations in the DAG
- We assign an operation if the operation is ready (all precedences checked), and if we have enough CEs free in the pool.
- Scheduling ends when execute the **E** operation



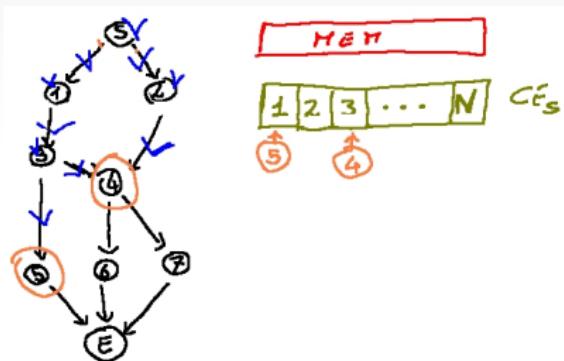
DAG and Scheduling (4)

- The scheduling problem is the problem of assigning CEs to operations in the DAG
- We assign an operation if the operation is ready (all precedences checked), and if we have enough CEs free in the pool.
- Scheduling ends when execute the **E** operation



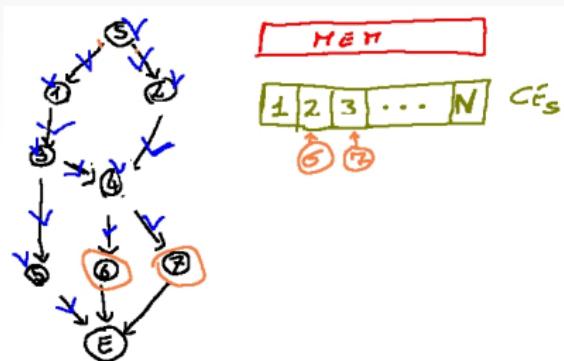
DAG and Scheduling (5)

- The scheduling problem is the problem of assigning CEs to operations in the DAG
- We assign an operation if the operation is ready (all precedences checked), and if we have enough CEs free in the pool.
- Scheduling ends when execute the **E** operation



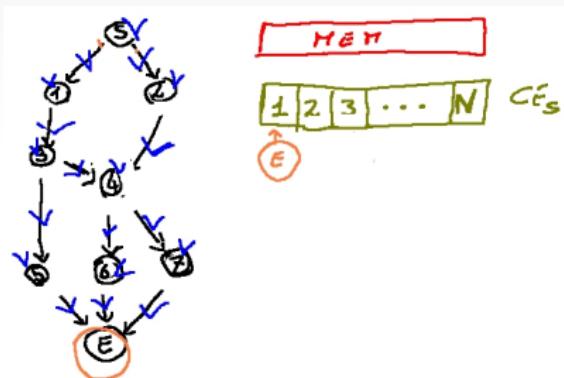
DAG and Scheduling (6)

- The scheduling problem is the problem of assigning CEs to operations in the DAG
- We assign an operation if the operation is ready (all precedences checked), and if we have enough CEs free in the pool.
- Scheduling ends when execute the **E** operation



DAG and Scheduling (7)

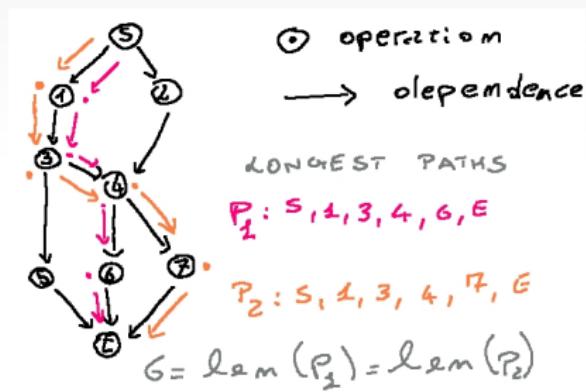
- The scheduling problem is the problem of assigning CEs to operations in the DAG
- We assign an operation if the operation is ready (all precedences checked), and if we have enough CEs free in the pool.
- Scheduling ends when execute the **E** operation



DAG and Scheduling (8)

Gli archi potrebbero essere anche pesati, in tal caso si assume che tutti gli archi abbiano lo stesso costo

- The total execution time depends on the longest path on the DAG
- We have here two longest paths
- in this example we assume the same operation time for each node, and the same (unary) weight on the edges.

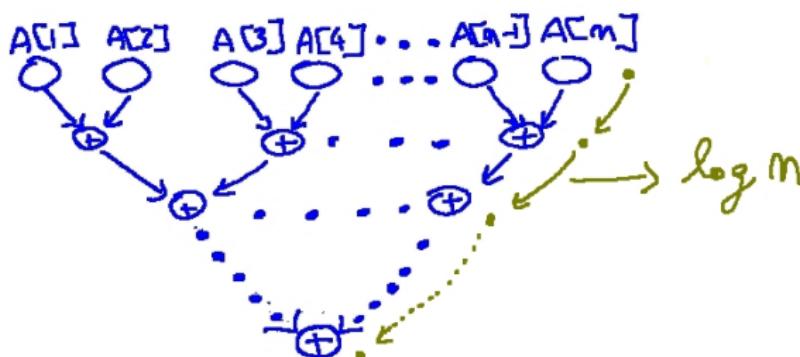


Devo stare attento a considerare i percorsi critici, cercando di ottimizzare al meglio i percorsi critici



DAG and longest Path on SUM

LONGEST PATH
FOR "SUM"



USED CEs

- $m/2$
- $m/4$
- $m/8$
- ⋮
- ⋮
- 1

WHAT IS THE MINIMUM
EXECUTION TIME
WITH N CEs?

The Core Rules in Parallel Program Design

DIVIDE ET IMPERA

- Julius Caesar

Parallel Program Design (Foster's Methodology)

Foster, pag. 26

Partitioning

Divide the computation in small(er) tasks. Focus attention of identification of tasks that can be run in parallel (i.e. the ones that have all resolved dependencies)

Communication

Determine what communication needs to be enacted among tasks

Aggregation

Aumenta la "grana" del dato

Combine tasks and tasks outputs with their communication, in order to increase the grain of data to manage at next steps. This needs the design of synchronization points in parallel algorithms

Mapping

Assign composite tasks to processes/threads. One of the constraints is to minimize communication (i.e. make tasks the more "local" as possible)



Some Assumptions

1

You Know (you are able to read) Asymptotic notation and other math notations

$O(n \log(n))$; $O(n^n)$; $\lceil \frac{n}{p} \rceil$; $\binom{n}{p}$; ...

2

You know how to estimate a complexity of a sequential algorithm
This is from Algorithm and Data Structures course ...

3

~~You have familiarity with definition and solution of Diophantine equations~~

Ehm ... no ... Its ok!



Any Question ?



¹image from: [https://pigswithcrayons.com/illustration/
dd-players-strategy-guide-illustrations/](https://pigswithcrayons.com/illustration/dd-players-strategy-guide-illustrations/)



Introduction to HPC - 2

Parallel Architectures and Reference Models

Francesco Moscato

HIGH PERFORMANCE COMPUTING

LM Ingegneria Informatica

Università degli Studi di Salerno

e-mail:fmoscato@unisa.it

Outline

① Parallelization and Communications

② Performance Models

③ Overview

④ Questions



Parallel Algorithms, Architectures and Nodes Topology

Algorithms Design

Algorithms have different performances on different architectures and node topology.

Algorithm → Architecture

You can design your algorithm and then you can choose best suiting Architecture and Node Topology

Of course: you have to configure and tune

Architecture → Algorithm

You can design your Algorithm Depending on architecture and node topology you have

Of course: This is a constraint that can limit algorithm speedup

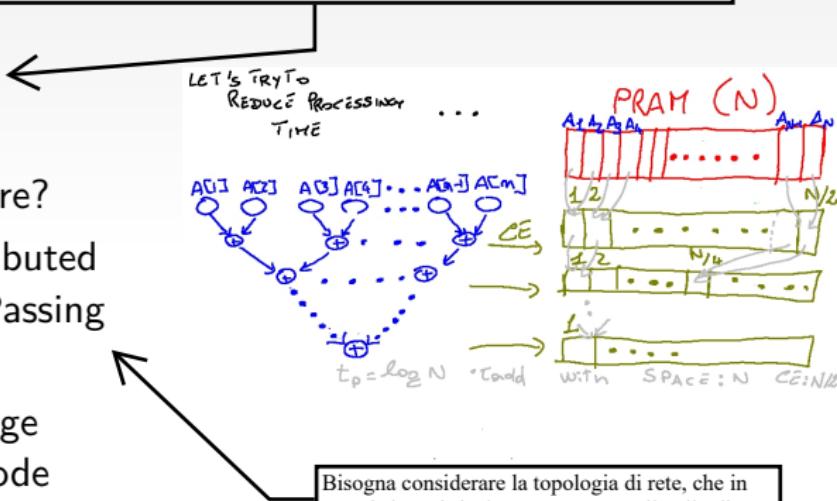
Of course: You have to configure and tune.



Sum Algorithm

Nei sistemi multithread e multicore i CEs sono ridotti, e la RAM è limitata, anche se il tempo di accesso resta uniforme.

- What if we have a Shared-Memory with Multi-Thread Architecture?
- What if we have a Distributed Memory with Message Passing Approach?
- And What if with Message passing with different Node Topology ?



Bisogna considerare la topologia di rete, che in questi sistemi risulta essere sempre il collo di bottiglia.

I nodi stessi sono multicore, si potrebbe usare un approccio ibrido. Per la RAM, c'è l'astrazione della distributed RAM (idealemente ho quella di tutti), ma l'accesso non è uniforme.

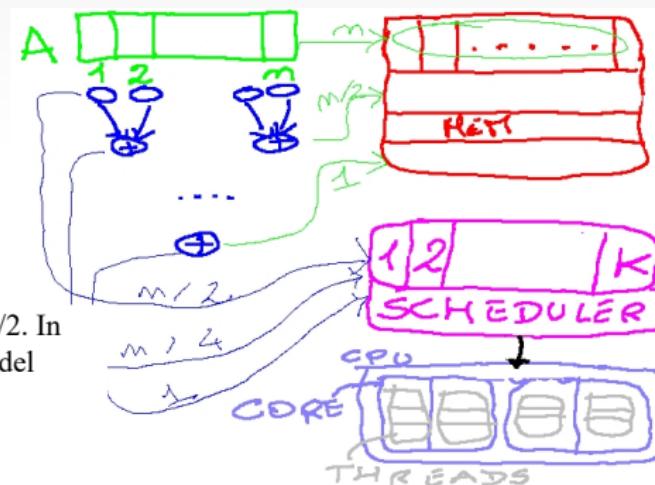


Sum on Multi-core / Shared Memory

- All ok if we have enough memory 
- All ok if we have enough "cores"
- Operating systems (scheduler) virtualize processors.
- What if A is 42 Petabytes ?

In questo caso ho k CPU, ma l'algoritmo ne richiede $N/2$. In questo caso, intervengono i meccanismi di scheduling del sistema operativo, che impiega comunque tempo (eventualmente trascurabile).

Se $A > \text{Dim(RAM)}$, si salva sul disco, in cloud, o altre soluzioni.



Performances issues

Performances

Scheduling (context switches), Hardware Architecture, caching mechanisms and cache sizes, interface to memory (as well as other hardware properties) influence performances

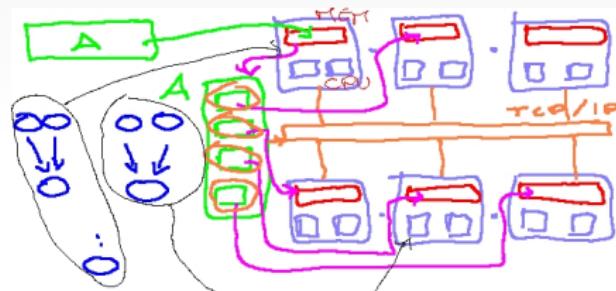
Massive Data and Computation

Massive Computation and/or Massive Data require different architectures: they do not fit into a single CPU core



Sum on MIMD architectures

- If we have a Bus-based network with all-to-all connections
- Input Data exist somewhere ... it has to be routed to nodes
- Intermediate data have to be routed and managed to build result
- We can share input if it is **huge**.
- What about work partitioning?



Nel caso di molti dati, o molta computazione da fare, nel caso di sistemi chiusi, si va su architetture MIMD dato che i sistemi sono "*chiusi*". Attenzione ai tempi di trasmissione in questi sistemi.

Su queste architetture, di solito le istruzioni vanno agglomerate, tenendo in considerazione la topologia e la trasmissione sulla topologia di rete utilizzata. Va capito anche come dividere i dati tra i nodi (parallelismo dei dati)

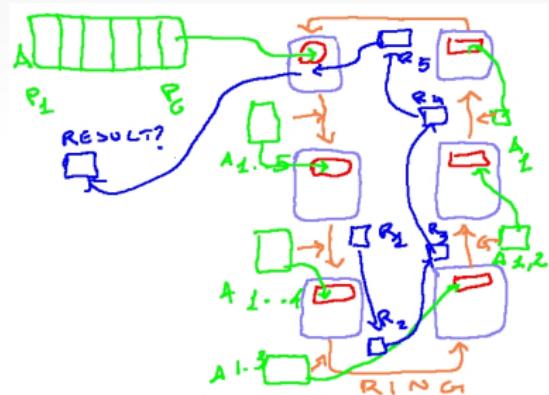
Some Issues on MIMD Architectures

- Bus: the examples shows an all-to-all connection based on TCP/IP network. This is obviously a cheap solution, but some performance problems exist. We can rely on more reliable and *fast* networks (Infiniband, Myrinet, etc.)
- Input Data: We still have the need to distribute input data among nodes. Anyway collection of input data can be shared among the same nodes.
- Partial Results: Algorithms may require distribution or routing of Partial (and final) results.
- Communication: it requires **time**



Sum With Ring Bus

- On a Ring Bus, we need more effort to route input data (distribution of data is $O(N)$ where N is the number of chunks)
- More effort too for routing of partial results.
- Communication** here requires **more** time.



The Need for a Model

A Model

We need a model to analyze parallel/distributed algorithms and their execution on different architectures

Analysis

We need a model general enough to analyze parallel algorithms and their application to parallel architectures

Evaluation

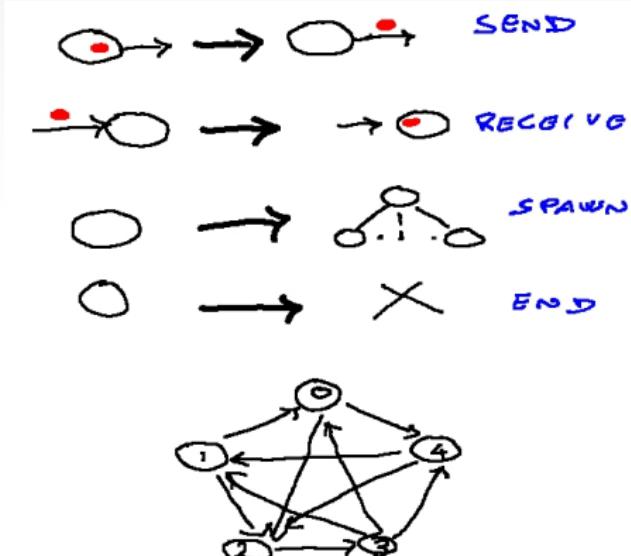
We need methods and techniques to evaluate real performances and improvements at run-time on HPC infrastructures.



An Abstract Model for Message Passing

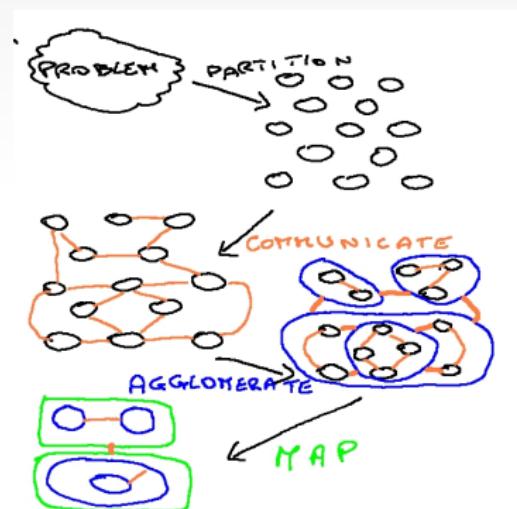
Lets start with an abstract model for

- Nodes for tasks, Edges for communication link
- send-receive of messages
- spawn and termination of tasks.



Design Methodology (By Foster)

- Partitioning: Define a large number of small tasks (divide). Both Data and Computation are decomposed. We isolate tasks that can be executed in parallel, or data that can be processed in parallel. Practical issues are ignored (number of processors, network etc.)
- Communication: This includes messages for coordination (control flow) and data flow as well.
- Agglomeration: tasks and communication structures are evaluated with respect to performance requirements and implementation costs. If necessary, tasks can be agglomerated in larger ones in order to improve performances and reduce costs.
- Mapping: Each Task is assigned to a processor in a network. Some goals to reach are CPU-time maximization, minimization of communication latency, delays and costs. Complex scheduling and load balancing algorithms can be applied here (i.e. mapping can be dynamic).



Partitioning

- Depends on independent tasks
- Decomposition on data (good is having many independent tasks on small chunks of data)
- Domain-based decomposition depend on nature of the problem (e.g. in a 3-D array, you can decompose on 1-D slices)
- Functional decomposition depends on feature-based decomposition



Communication

- **local vs global:** With local communication, tasks communicate only with a small set of other tasks; global communication requires each task to communicate with many (eventually all other) tasks
- **structured vs unstructured:** structured communication has *regular* comm graphs (e.g. a tree, a grid etc.); unstructured communications have arbitrary graphs.
- **static vs dynamic:** in static communication, interaction of tasks are known at design time; in dynamic communication, communication graphs change depending on data and on tasks behaviors and results.
- **synchronous vs asynchronous:** in the first a consumer gets data when a producer generates it; in the second kind of communication, a consumer can get data without the cooperation of a producer.



Agglomeration

- Agglomerating tasks, reduces communication overhead
- sending / receiving data requires time (creation of messages, use of proper data structure in the Operating Systems, sends may be blocking or receives may require active waits etc.)
- the less data we transmit, the more *fast* is the algorithm)
- the more tasks we have, the higher is the cost for their creation (resources, time etc.)



Mapping

- place tasks that can execute concurrently on *different* processor;
- place tasks that communicate a lot on the same processor in order to improve *locality*.
- Mapping Problem is *NP-Complete*
- Dynamic Load Balancing



Speedup

Speedup

is a measure of relative performance of two Algorithms/systems when processing the same problem.

Example

Saying that an Algorithm/System on 12 Processors(nodes) has a Speedup of 10, means that the algorithm/program executed on one processor(node), is ten times slower than the same algorithm/program executed on 12 processors(nodes).

Amdahl's Law

The concept of speedup is connected to the Amdahl's Law



Amdahl's Law

Theoretical SpeedUp

The Amdahl's Law gives a *theoretical speedup* in the latency of execution of a task with fixed workload. It expresses how much performances improves when the resources on the executing system improve (i.e. increases in number).

Sequential part

It assumes that a program has a part that cannot be partitioned to execute in parallel (i.e. a **sequential** part). Since the sequential part cannot be executed in parallel, the maximum theoretic speedup depends on the fraction of sequential part on the whole program.



Amdahl's Law



Speedup

If the sequential part of an algorithm accounts for $1/s$ pf the whole program execution time, then the maximum speedup that can be achieved on a parallel computer is s .

Example

if the sequential part is 5 percent, then the maximum speedup is 20



Theoretical and Real Speedups

Measuring and Evaluating Speedup

When performing experiments to evaluate real speedup, performances and speedup are characterized by sentences like:
The Algorithm implemented on parallel system X achieves a speedup of 10.8 on 12 processors with a problem size $N = 100$.

Narrow region characterization

This information is not “good”. What if N changes ? What if communication costs changes ? What if the number of processors changes?



Algorithm Complexity and Speedup

Speed up changes with complexity

Let us assume that we have an asymptotic time complexity (for the optimal sequential algorithm) of $\mathcal{O}(N + N^2)$, with an execution time function T on a number of processors P with a problem size of N

Three Examples

Let us assume that we have 3 algorithms that *somewhere* have a speedup of 10.8 on $P = 12$



Algorithm Complexity and Speedup

Example

- ① $T = N + N^2/P$: This partitions the $O(N^2)$ component, but replicate the $O(N)$ part.
- ② $T = (N + N^2)/P + 100$: This partitions the whole algorithm but introduces an additional cost of 100;
- ③ $T = (N + N^2)/P + 0.6P^2$: This introduces an additional cost of $0.6P^2$

Speedup

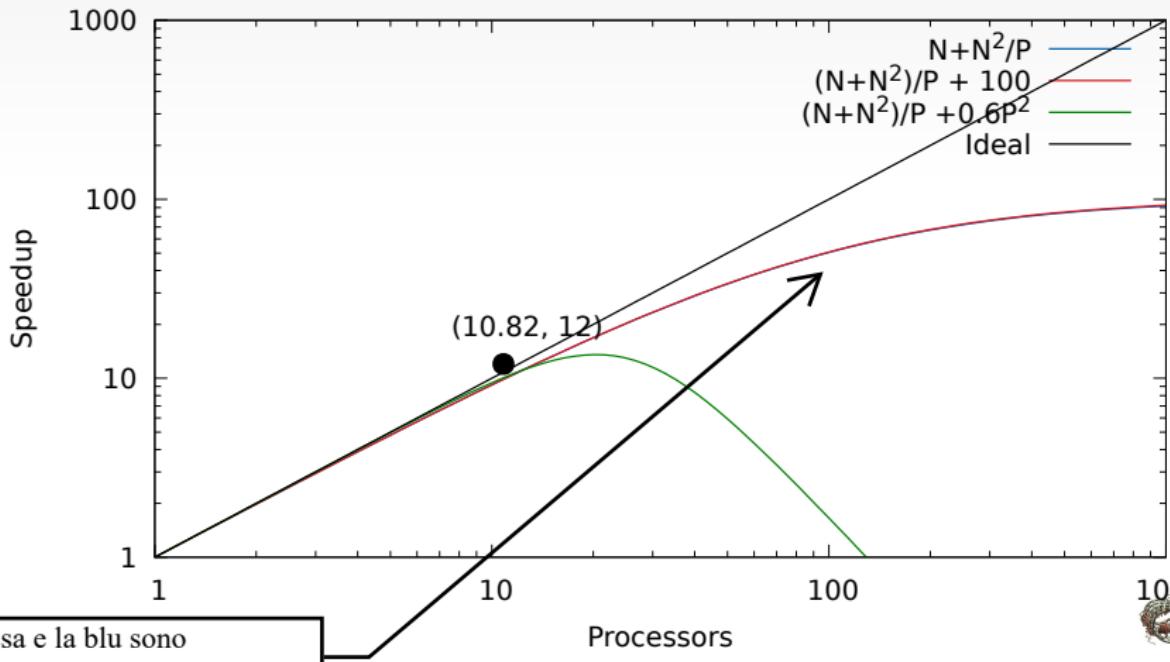
Speedup with $N=100$, $P=12$ is almost the same (10.82) for the three algorithms but ...



Example: Speedup (GNU)Plotted with N=100

Qui è stato plottato $1/[N+N^2/P]$, non il plot diretto!

N=100

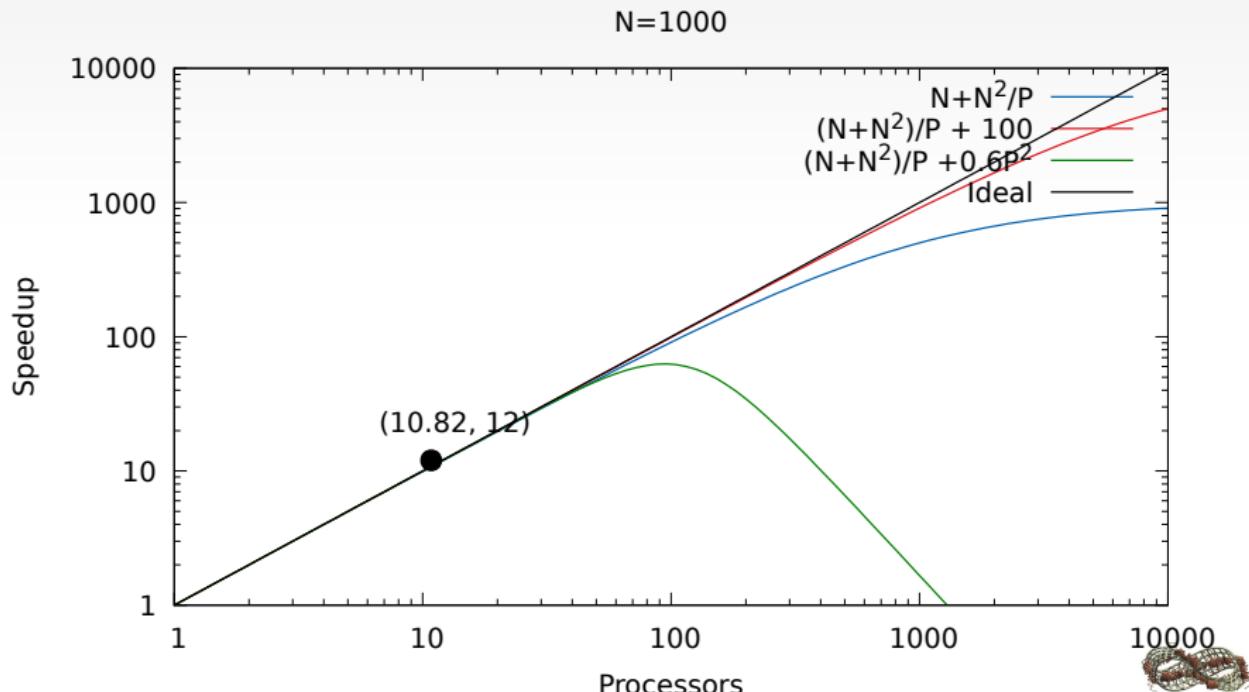


La rossa e la blu sono
sovraposte

icato



Example: Speedup (GNU)Plotted with N=1000



Asymptotic Analysis

Asymptotic Analysis

It is Asymptotic ... And this can be a Problem because it ignores lower-order terms in formulas

Example

The (asymptotic) cost of an algorithm $10N + N^{\log N}$ ignores the term $10N$, that is significative for $N < 1024$ (i.e. the longer time(or space) is taken by the ignored term. This is a problem if we can divide data in little chunks, or if the input is “short”

Communication time

In this example we are not considering Communication time.



The Need for a Model

Other Model

The model based on observations and the asymptotic analysis are not enough accurate in order to explain observed values or to predict future circumstances.

Performance Model

Performance model should consider: Dimension of the Problem; Number of Tasks; Number of Processors; Communication; Operating System, Libraries and other software issues; Hardware ...

$$T = f(N, P, U, \dots)$$



Execution Time

1:30

Definition

The time elapsed between start of first processor and completion of last processor

Component

It has three components: Computation Time (T_{comp}), Communication Time (T_{comm}) and Idle Time (T_{idle}). This is for each Processor executing the parallel program.

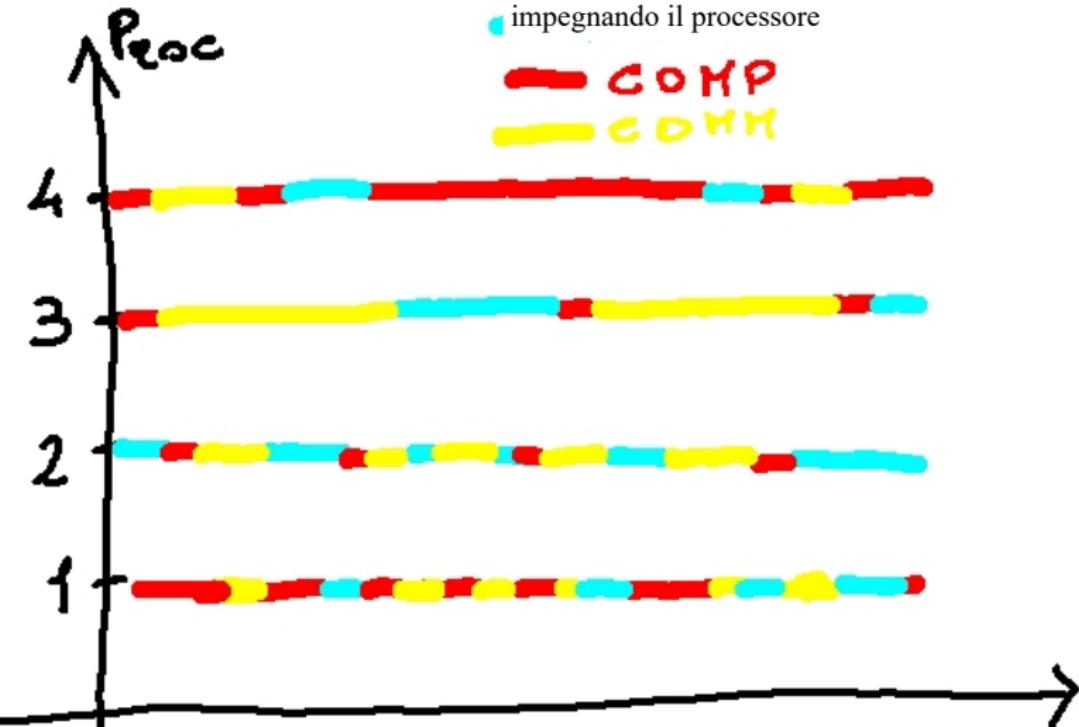
$$T = T_{comp} + T_{comm} + T_{idle}$$



Example of Execution Time

Nota:

Nel caso in cui il mittente ha riempito il buffer nel corso di una sessione TCP/IP, è possibile liberare il processore e far fare la comunicazione al DMA, non impegnando il processore



Execution Time

For balanced distribution of processes:

$$\begin{aligned} T &= \frac{1}{P} (T_{comp} + T_{comm} + T_{idle}) \\ &= \frac{1}{P} \sum_{i=1}^P (T_{comp}^i + T_{comm}^i + T_{idle}^i) \end{aligned}$$



Computation Time

Depends on :

- Problem Size (N)
- Number of tasks or processors
- Memory System, Cache Configuration and Performances
- Processor Architecture, Instructions Set, Pipeline etc.

Performances Changes

Be Aware: Computation Time changes changing Computing Architecture (You cannot simply scale up by comparing GFLOPS)



Communication

Sending a Message needs:

- Setup Communication (t_s)
- Time to send a chunk of data (let's say a word) (t_w)
- Sending as many words as we need to transmit the message (e.g. L)

Sendind a Message

$$T_{msg} = t_s + Lt_w$$

Inter-Intra Processor

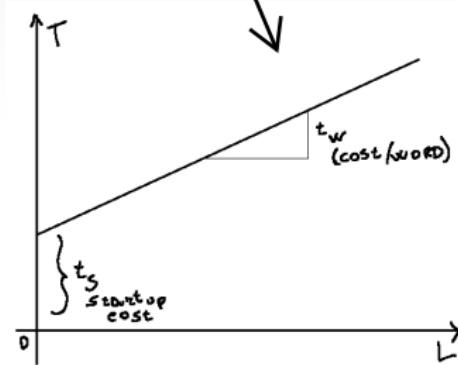
We have to consider time both in the case of Inter-processors as well as of Intra-processors communication. We assume here they are comparable



Sending a Message

- Many Network protocols have irregularity in first stages of communication
- t_w changes depending on word size
- t_w also depends on hardware, Operating systems, implementation of communication protocols etc.

La curva potrebbe variare a seconda del protocollo di rete utilizzato



Idle time

- lack of computation and/or data
- waiting synchronization
- can be avoided overlapping communication and data (making computation *during* communication or while waiting a message), by creating multiple tasks and proper scheduling



Efficiency and Speedup

Efficiency

Can provide a useful measure of parallel algorithm quality

Relative Efficiency

$$E_r = \frac{T_1}{PT_P}$$

T_1 is the time to execute the program on one processor; T_P is the time to execute a parallel task on one processor

Relative Speedup

$$S_r = PE_r$$



Relative Speedup

Example

$$E_r = \frac{T_1}{PT_P}; S_r = PE_r$$

Example 1: $P = 1000$; $T_1 = 10000s$; $T_P = 20s$: for $P = 1000$

$$S_r = 500$$

Example 2: $P = 1000$; $T_1 = 1000s$; $T_P = 5s$: for $P = 1000$

$$S_r = 200$$

Problem in relative Speedup Metric

The second algorithm is better in the range 1-1000

We should have an algorithm-independent metric other than execution time



Not-Ideal Efficiency and Speedup

Overhead

Parallelization introduces an overhead (communication, system call and resources to create and manage parallel tasks, parallel library overhead etc.)

Modification to T_P

$$T_P = T_1/P + T_{overhead}$$



Amdahl's Law again

Evaluation

Suppose we have a sequential program, that we are able to parallelize at 90%; that the parallelization is *perfect* (it scales linearly with P) and that $T_1 = 20$ seconds.

Evaluation

$$T_P = 0.9T_1/P + 0.1T_1 = 18/P + 2$$

$$S = \frac{T_1}{0.9T_1/P+0.1T_1} = \frac{20}{18/P+2}$$

with the Amdahl's law stating that the speedup is less the serial execution time divided the time of the program that cannot be parallelized:

$$S' \frac{T_1}{0.1T_1} = 20/2 = 10$$



Scalability

Idea

Scalability is the property of dealing with *ever-increasing* problem size

Efficiency and Scalability

If we find when increasing the size of the problem, **E is constant**, then the program is **scalable**



Scalability

Example

Let be $T_1 = k$ seconds; $N = k$ (yes ... the same) is the problem size; $T_P = n/P + 1$

$$E = \frac{k}{P(k/P+1)} = \frac{k}{k+P}$$

Let us increase k and P by two factors (α and β respectively)

$$\frac{\alpha k}{\alpha k + \beta P}$$

if $\alpha = \beta = a$

$$\frac{ak}{a(k+P)} = \frac{k}{k+P}$$

In this case ... the program is **scalable**



Homeworks

Parallelize (Shared Memory and Message Passing) and evaluate Performances, Efficiency, Speedup and Scalability of the following algorithm:

Parallel Generation and Sum

```
1 Let A[N] be a vector of N random integer values
2 //Shuffle A[N] with Fisher-Yates algorithm:
3 for i: N downto 1 do
4     p = random number in [1..i]
5     swap A[i],A[p]
6 done
7 //Sum Elements of A
8 sum=0
9 for i:1 to N do
10    sum = sum +A[i];
11 done
```



Main Frameworks/API/Tools for Parallel Programming

In this course...

- **OpenMP**
- **MPI**
- **CUDA**
- **OpenCL**

Many others in literature ...



OpenMP

OpenMP

- Shared Memory Parallel System
- OPEN API specification¹ (it is not a framework or a tool/compiler)
- Implemented and supported by many tools²
- defines a set of **compiler directives, run-time routines and environmental variables**
- based on a **multi-thread** model
- cooperation is through shared memory (and thread-based synchronization directives)
- it allows for transparent scalability
- programming model based on pre-compiling directives (e.g. **#pragma**)

¹<https://www.openmp.org>

²<https://www.openmp.org//resources/openmp-compilers-tools/>



Message Passing Interface (MPI)



- Library Specification for Message-Passing³
- designed for High Performances on massively parallel clusters
- many implementors (e.g. openMPI library)
- Based on Process definition and message passing among processes.
- it contains all directives to manage inter-process communications (send and receive, topology definition, packets managements etc.)

³<https://www.mcs.anl.gov/research/projects/mpi/>



CUDA



- NVIDIA Platform and Library⁴
- Application of General Purpose parallel programming to GPU
- C - C++ Libraries
- based on concept of kernels of computation, that are spawned on groups of threads on shared memory
- manages communication and synchronization

⁴<https://developer.nvidia.com/cuda-zone>



OpenCL



OpenCL

- Open Standard for parallel programming **across heterogeneous devices** (CPU, GPU, FPGA ,...)⁵
- Many Partners (Apple, Nvidia, Intel, Arm, AMD, Samsung, Sony ...)
- task-based and data-based parallelism
- an abstract programming model to support more platforms
- private/local/global/shared memory model
- macro - based programming model (C language)

⁵<https://www.khronos.org/opencl/>



Any Question ?



¹image from: <https://pigswithcrayons.com/illustration/dd-players-strategy-guide-illustrations/>



Shared Memory Programming (1)

Main Concepts, Libraries and Tools
Introduction to OpenMP

Francesco Moscato

Università degli Studi di Salerno
fmoscato@unisa.it

Outline

① Shared Memory

② MT Architectures

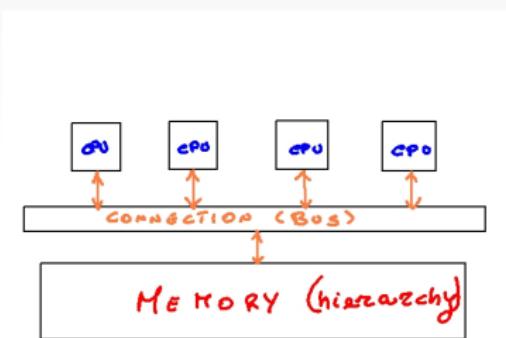
③ Multi Threading Programming

④ OpenMP



Shared Memory Basic Architecture

- Processing Units share memory (address space)
- Connection usually by buses (high speed and throughput)



Shared Memory



Memory Hierarchies

Caching improves performances, but introduces coherency problems too, even in Mono-CPU systems. The problem is more complex the more processing unit share the memory (but not caches). **Hardware Solutions** have to be provided!

Operating System

Modern O.S.s virtualize memory and CPUs. They provide Processes Management, Shared Memory, IPC, Synchronization and Event Management (Signals)



Why Shared Memory Programming (1)



Even if your HW has not HW Shared Memory ...

Natural

your O.S. gives you a Virtual Memory Shared among processes

Performance Improvements

you have performance improvements in multi-process /
multi-threaded programming because you can overlap IO/Bound
and CPU/Bound operations

Easy

Sometimes it easier to design solutions in terms of different
workers *dedicated* to different tasks



Why Shared Memory Programming (2)

Many Frameworks, Languages, and tools supporting SM programming, analysis etc.

- PThread Posix Library 
- C++, Java and many other languages have standard *Thread-based* classes and libraries;
- Almost all operating systems have system calls to manage processes (and light-weight processes, and even “threads”).

Hardware Too

Many *modern* CPUs have Architectures supporting HW multi-processing and multi-threading (yes ... it is a little bit different from Pthreads or Unix Processes ...) 



MultiThread MultiCore Architectures



- This is a *kind recall to HW architecture of Multithreading - MultiCore architectures*
- As you well remember, the evolution of MT-MC Architectures is an effect of saturation of Moore's Law: if it is a problem to make faster Processing Units, Let's gather more PUs... **Unity is Strength!**
- This is the natural way to continue the way of **instruction level parallelism** aimed by pipelines
 - vector-based instructions, dynamic and static ILP, more pipeline stages; more pipelines;
 - instruction re-ordering; fetch and pre-fetch;
 - loop roll and unroll
 - reservation stations;
 -

I sistemi CISC hanno molte istruzioni complesse che sono composte da più microistruzioni, le pipeline di questi processori sono più grandi. Poi sono state inserite diverse pipe per Floating Point e interi.

In hardware, nei sistemi multi thread/multicore, il parallelismo che si vuole sfruttare è quello di lanciare istruzioni contemporaneamente su due pipe libere (vedi slide successive)

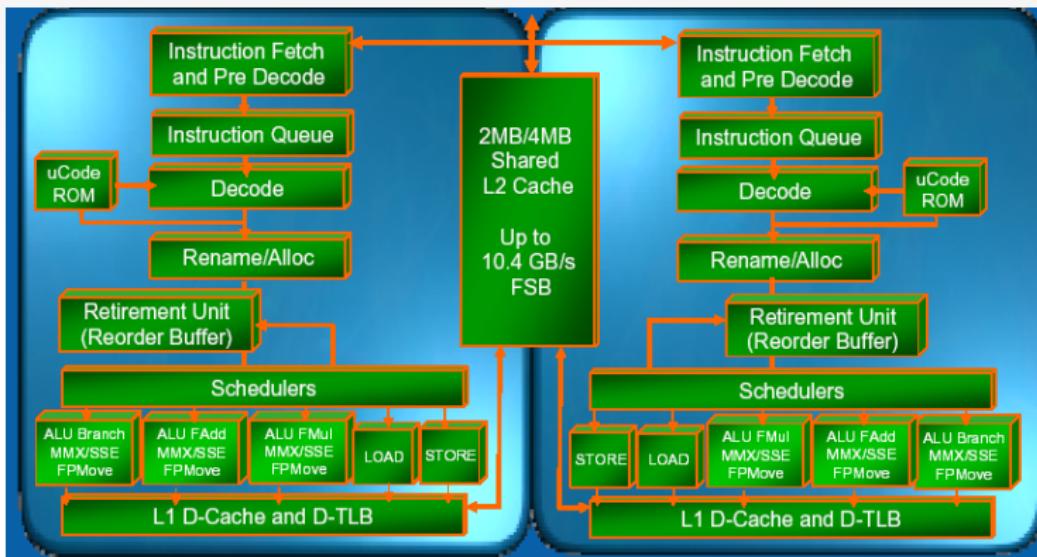


Why Multi-Core / Multi-Threading

- Problems in increasing clk freq
- heat problems in pipelines
- too many pipes (i.e. efficiency - stall problems)
- general trends:
multi-threading
programming and Parallel Architectures



Example from the Past: Core 2 Duo (Intel)

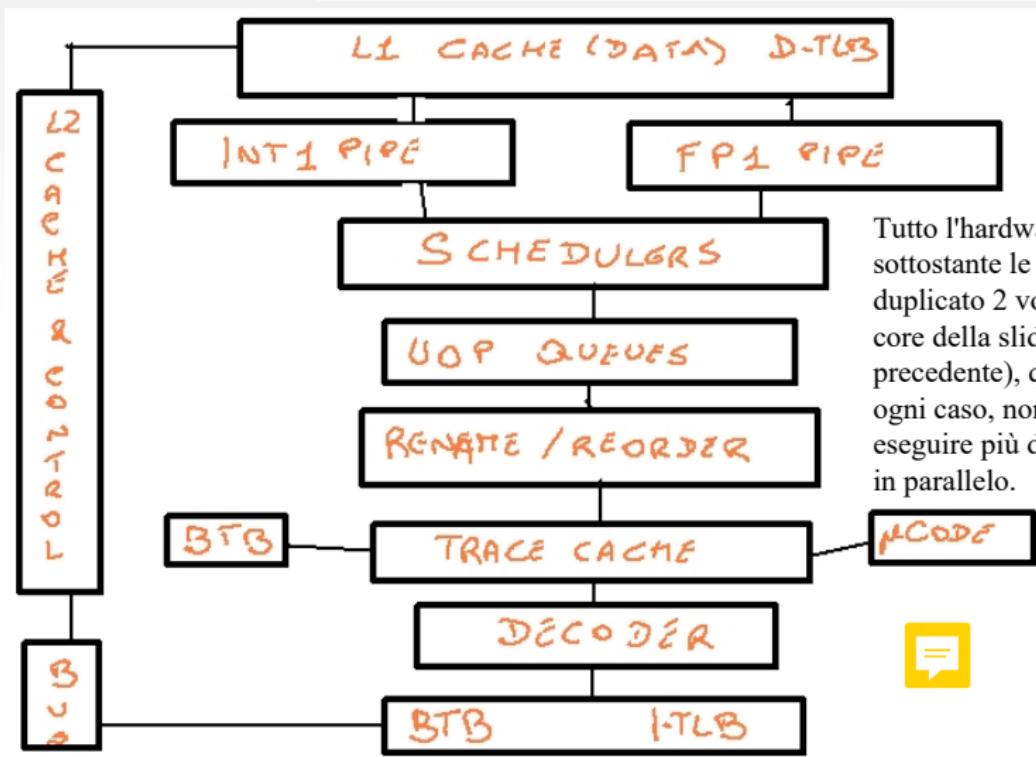


In questo caso, l'architettura è stata duplicata, e le due CPU condividono una cache di secondo livello mediante un bus ad alto throughput



No SMT (1)

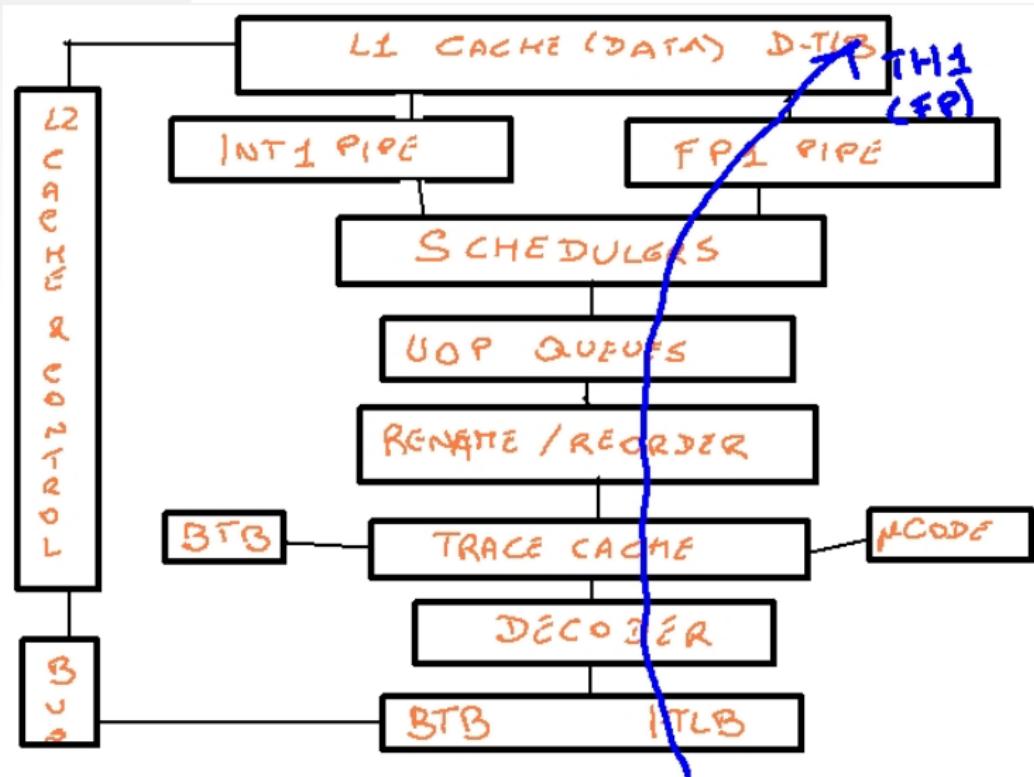
In questo caso, il compilatore parallelizzato prendeva tutte le istruzioni che operavano su floating point indipendenti da quelle che operavano sugli interi, li raggruppava e li mandava in esecuzione.



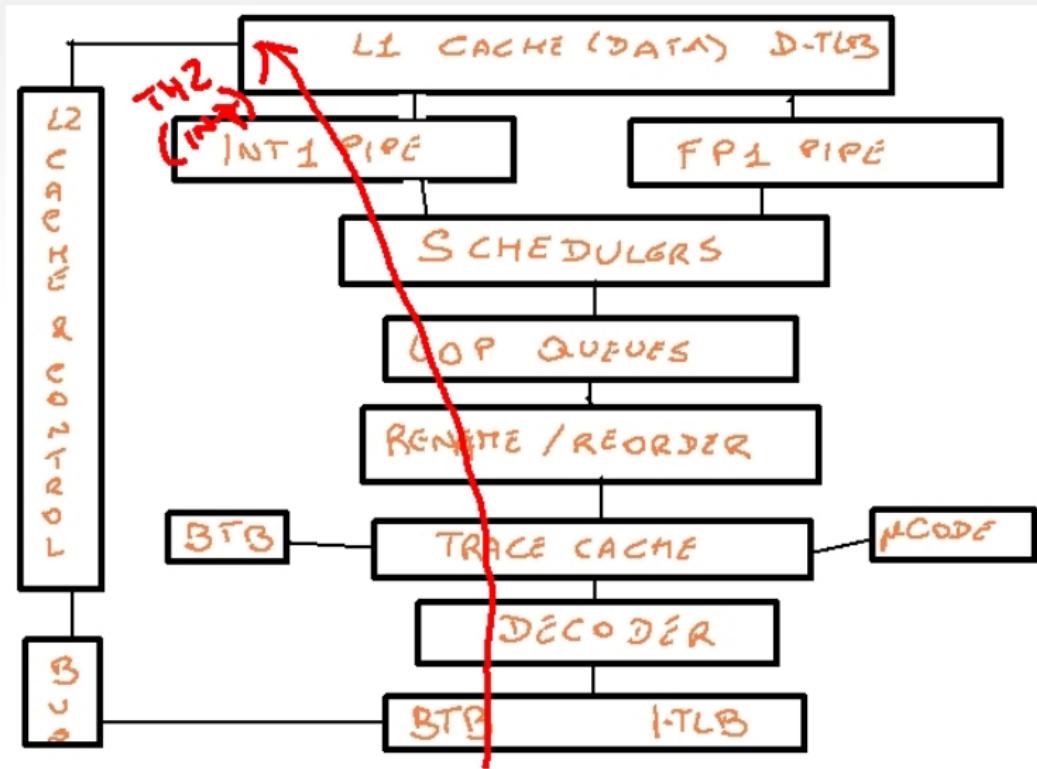
Tutto l'hardware sottostante le 2 pipe è duplicato 2 volte (sono i 2 core della slide precedente), quindi in ogni caso, non è possibile eseguire più di due flussi in parallelo.



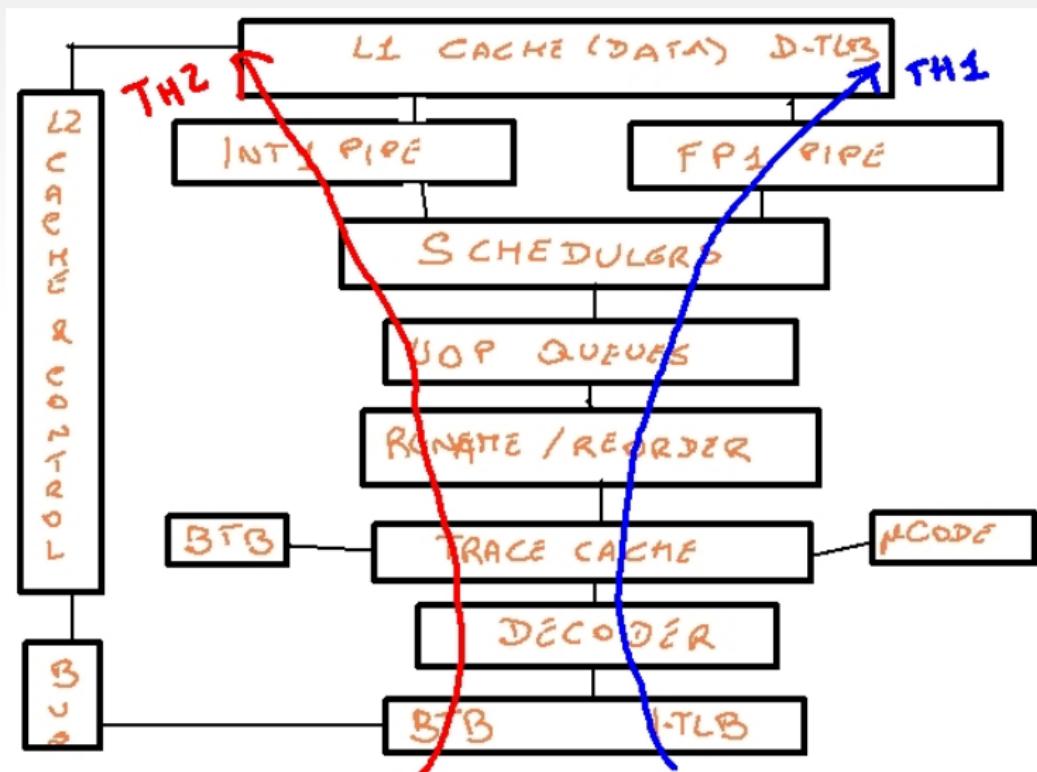
Simultaneous MultiThreading (SMT) is a technique for improving the overall efficiency of superscalar CPUs with hardware multithreading. SMT permits multiple independent threads of execution to better utilize the resources provided by modern processor architectures.



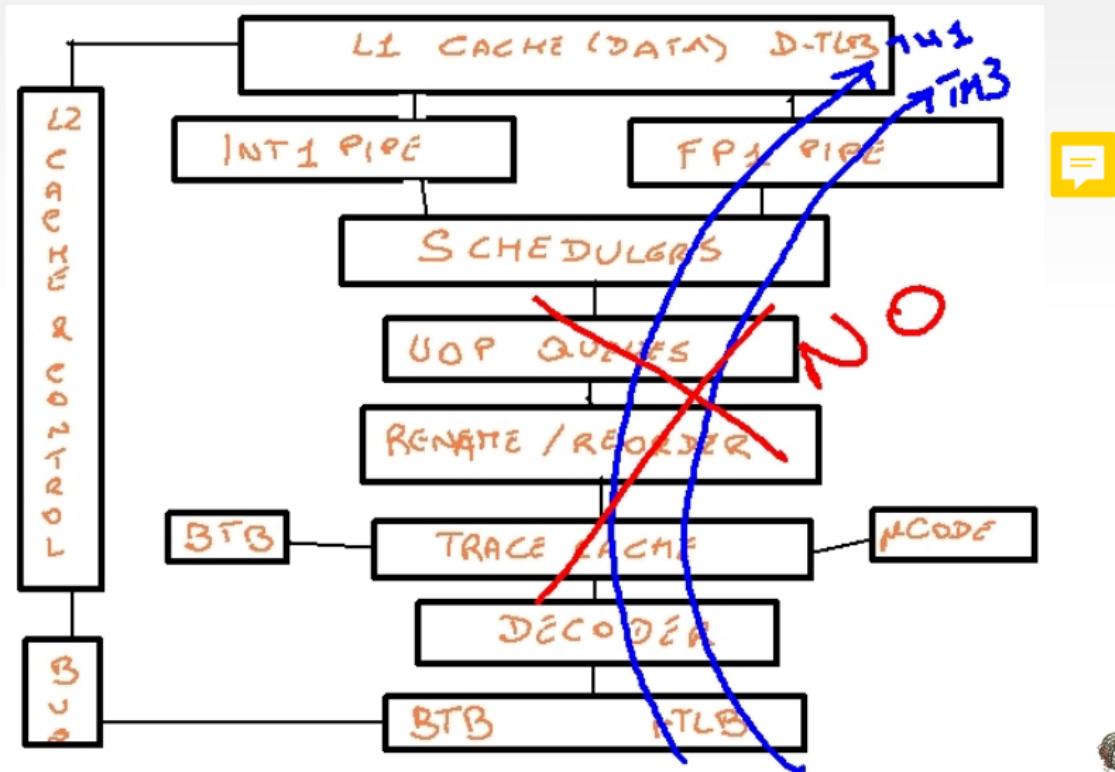
No SMT (3)



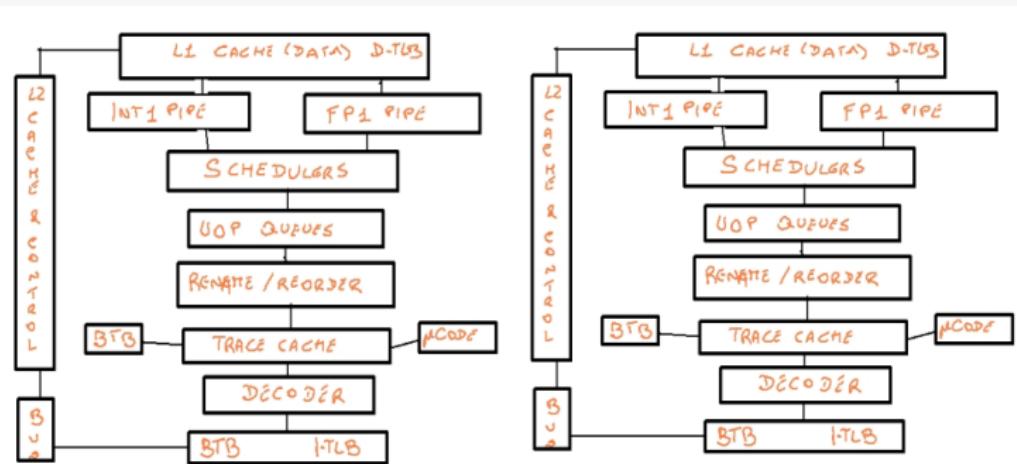
With SMT (1)



With SMT (1)

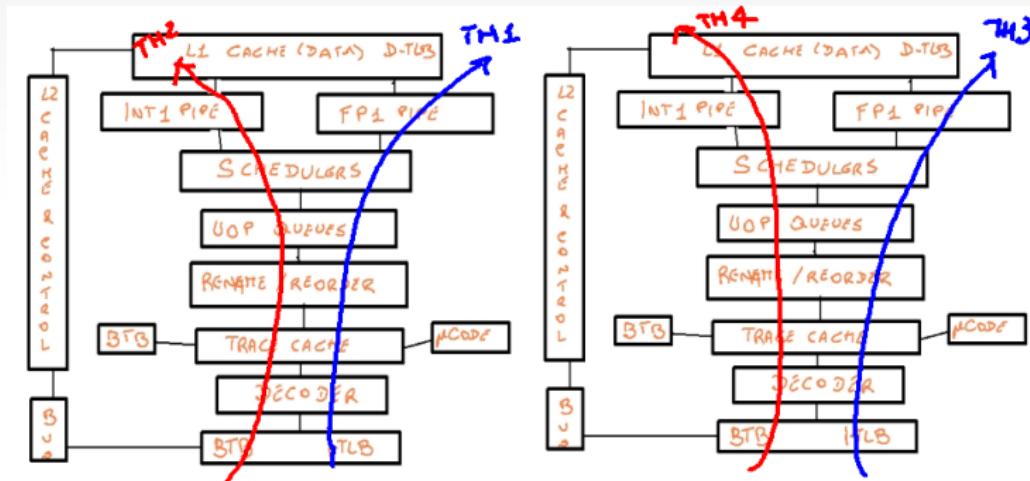


Multi-Core (with or without SMT) (1)



Multi-Core (with SMT) (2)

I thread che partono in parallelo non sono quelli creati dal programmatore, ma questi flussi di istruzioni che partono in parallelo grazie al compilatore



General MT Scheduling

- **Coarse-grained:** (Intel Itanium 2) Switch occurs when a thread is blocked by an high latency event (e.g. cache miss). Switch must be executed in one clock cycle and many CPU are replicated.
- **Fine-grained:** (some MIPS, Sun UltraSparc ...) Switch occurs at any clock cycle (preemptive multi-tasking). Many component are replicated.
- **Simultaneous Multi Threading:** (Intel Hyper-Threading, IB; Power5, ...): In parallel as you saw before.



Multi Threading and Shared Memory

Multi Thread Programming

is one of the main used programming model for Shared Memory

OpenMP and Multi Threading

OpenMP is based on Multi-Thread Programming. Its APIs and libraries allow for transparent implementation and management of **optimized** multi-threaded software.

OpenMP

You can write parallel software for Shared Memory with OpenMP almost without any knowledge of Multi-Thread programming. The code is smaller, more easy to read and many times, OpenMP implementation is faster since it may force front-end and back-end optimization.



PThreads

Posix threads

Anyway, knowledge of pthreads and their programming model is useful for better understanding of OpenMP directives, scheduling policies etc.

Hence ...

We propose here a(nother) *kind* remind of Pthreads

Old But Gold

B.Lewis, D.J. Berg: PThread Primer. A guide to Multithreaded Programming, SunSoft Press, A Prentice Hall Title. (1996)



PThread Basic Primitives (1)

PThread Library

```
1 #include <pthread.h>
2 ...
3
4 gcc -pedantic -Wall -o ... ... -lpthread
```



PThread Basic Primitives (2)

Creating and Joining

```
1 #include <pthread.h>
2
3 void * entry_point(void *arg)
4 {...}
5
6 int main(int argc, char **argv)
7 {
8     pthread_t thr;
9     if(pthread_create(&thr, NULL, &entry_point, NULL))
10         { Create a Thread }
11
12     if(pthread_join(thr, NULL))
13         { Wait for termination }
14 ...
15 }
```

Paradigma
Thread &
Join

PThread Basic Primitives (3)

Exit

```
1 #include <pthread.h>
2
3 void * entry_point(void *arg)
4 {
5     pthread_exit(NULL); //exits this thread
6 }
7
8 int main(int argc, char **argv)
9 {
10    pthread_t thr;
11    if(pthread_create(&thr, NULL, &entry_point, NULL))
12        { ... }
13
14    if(pthread_join(thr, NULL))
15        { ... }
16 ...
17 }
```



PThread Synchronization

- Barriers: Computations stops until they meet up all together.
(this can be done with semaphores, but barriers are more convenient in many cases.)
- Mutexes: standard mutexes with lock and unlock primitives
- Semaphores: Pthread does not provide semaphores. However separate POSIX standard defines them (semaphore.h)



PThread Barrier

Barrier

```
1 #include <pthread.h>
2 ...
3
4 pthread_barrier_t barr;
5
6 void * entry_point(void *arg)
7 {
8     do something...
9     int rc = pthread_barrier_wait(&barr);
10    if(rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD)
11    {
12        printf("Could not wait on barrier\n");
13        exit(-1);
14    }
15 }
```



PThread Barrier

Barrier

```
1 ...
2 int main(int argc, char **argv)
3 {
4     pthread_t thr[THREADS];
5
6     // Barrier initialization IN MAIN
7     if(pthread_barrier_init(&barr, NULL, THREADS))
8     {
9         printf("Could not create a barrier\n");
10        return -1;
11    }
12    for(int i = 0; i < THREADS; ++i)
13        if(pthread_create(&thr[i], NULL, &entry_point, (void*)i))
14            ...
15    for(int i = 0; i < THREADS; ++i)
16        if(pthread_join(thr[i], NULL))
17            ...
```



PThread Mutexes

Mutex and Critical Region

```
1 #include <pthread.h>
2 // A shared mutex
3 pthread_mutex_t mutex;
4 void* theother(void *arg)
5 {
6     for(int i = 0; i < ITERATIONS; ++i){
7         // Lock the mutex
8         pthread_mutex_lock(&mutex);
9         target -= target * 2 + tan(target);
10        // Unlock the mutex
11        pthread_mutex_unlock(&mutex);
12    }
13
14    return NULL;
15 }
```



PThread Mutexes

Mutex and Critical Region

```
1 ...
2 int main(int argc, char **argv)
3 {
4     pthread_t other;
5     // Initialize the mutex
6     if(pthread_mutex_init(&mutex, NULL))
7     {...}
8     if(pthread_create(&other, NULL, &theother, NULL))
9     {...}
10    for(int i = 0; i < ITERATIONS; ++i)
11    {
12        pthread_mutex_lock(&mutex);
13        target += target * 2 + tan(target);
14        pthread_mutex_unlock(&mutex);
15    }
```



PThread Mutexes

Mutex and Critical Region

```
1 ...
2     if(pthread_join(other, NULL))
3     {...}
4     // Clean up the mutex
5     pthread_mutex_destroy(&mutex);
6 ...
7 }
```



PThread Semaphores

Semaphores Wait and Post

```
1 #include <semaphore.h> // This is not sys/sem.c !
2 #include <pthread.h>
3
4 sem_t OKSem;
5 int resource;
6
7 void* consume(void *arg)
8 {
9     sem_wait(&OKSem);
10    if(!resource)
11    {
12        // Get some resources
13        ++resource;
14    }
15    sem_post(&OKSem);
16    return NULL;
17 }
```



PThread Semaphores

Semaphores Wait and Post

```
1 ...
2 int main(int argc, char **argv)
3 {
4     pthread_t threads[THREADS];
5
6     resource = 0;
7     // Init sem to 1. If the second arg is 0, sem is
8     // is shared between threads (and not processes).
9     if(sem_init(&OKSem, 0, 1))
10        { ... }
11     for(int i = 0; i < THREADS; ++i)
12     {
13         if(pthread_create(&threads[i], NULL, &consume, NULL))
14             {...}
15     }
16 ...
```



PThread Semaphores

Semaphores Wait and Post

```
1 ...
2 for(int i = 0; i < THREADS; ++i)
3 {
4     if(pthread_join(threads[i], NULL))
5         {...}
6 }
7 sem_destroy(&OKSem);
8 ...
9 }
```



PThread Other Elements

- Condition Variables
- rw locks
- Thread Safety
- cache, cache coherency and false sharing



Exercise

Parallel Sum

Design a MultiThreaded (PThread)/ Shared Memory solution for the problem of summing up elements of a random vector, whose elements have been randomly shuffled.



What is OpenMP

OpenMP

is a shared-memory API; it facilitates shared-memory programming.

OpenMP

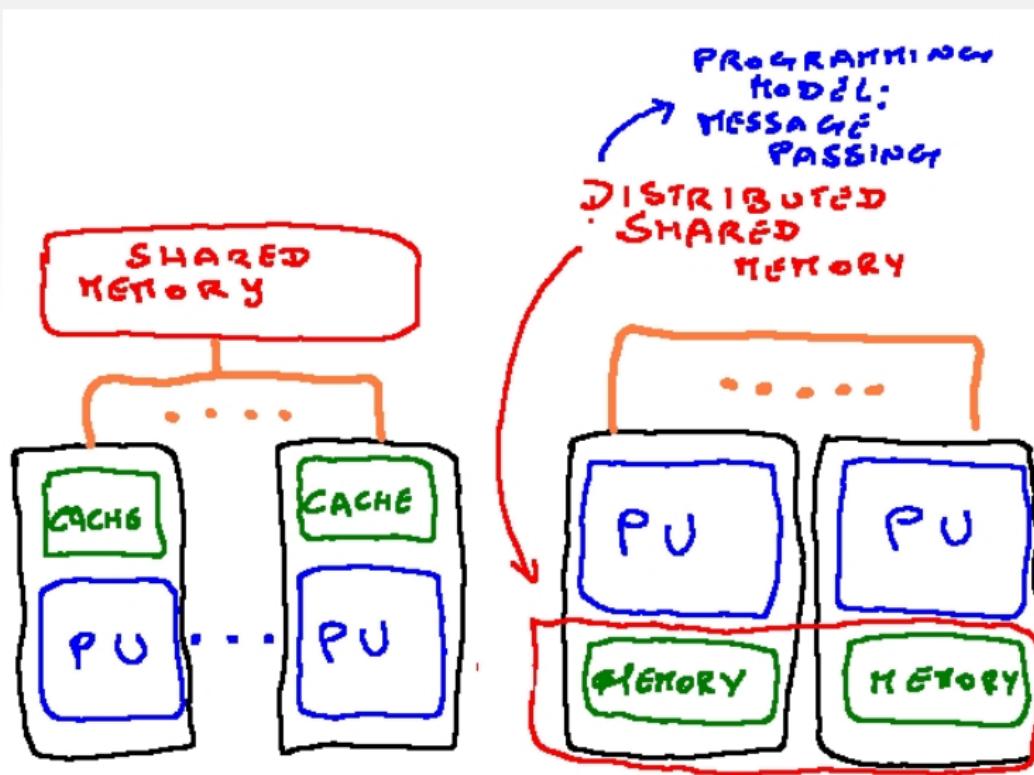
is NOT a programming language. It is a **notation** that can be added to a sequential program in Fortran, C, C++

OpenMP

has strong emphasis on **structured** parallel programming; it is **easy to use**.



Distributed and Shared Memory



Example

Dot-Product

Design and Analyze a parallel version (both SMP with PThreads and Message Passing) of the Dot-Product of two vectors of *double*



Dot Prod: Sequential

Dot Product

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define N 256
4
5 inline int init_with_multiples (double * a, int dim, double val);
6
7 int main (int argc, char * argv[]){
8     double a[N], b[N];
9     init_with_multiples (a,N,0.5);
10    init_with_multiples (b,N,2.0);
11    double sum = 0;
12    ...
```



Dot Prod: Sequential

Dot Product

```
1 ...
2     for (int i = 0; i<N; ++i)
3         sum += a[i]*b[i];
4
5     printf("Sum = %f\n", sum);
6     return 0;
7 }
8
9 int init_with_multiples (double * a, int dim, double val){
10    for (int i = 0; i<dim; ++i)
11        a[i]=i*val;
12 }
```



Dot Prod: PThreads

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define NTHREADS 4
6 #define N 256
7 double sum;
8 double a[N], b[N];
9 int status;
10 pthread_t threads[NTHREADS];
11 pthread_mutex_t mutexsum;
12 void *dotprod(void * arg);
13 int main (int argc, char * argv[]){
14     pthread_attr_t attr;
15     for (int i=0;i<N; ++i) a[i]=i*0.5; b[i]=i*2.0;
16     pthread_mutex_init(&mutexsum, NULL);
17     pthread_attr_init (&attr);
18     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
19     for (int i=0; i<NTHREADS;i++)
20         pthread_create ( & threads[i], & attr, dotprod, (void *) i);
21     pthread_attr_destroy(&attr);
22     for (int i=0;i<NTHREADS; i++)
23         pthread_join(threads[i],(void**) & status);
24     printf("sum= %f \n", sum);
25     pthread_mutex_destroy (&mutexsum);
26     pthread_exit(NULL);
27 }
```



Dot Prod: PThreads

```
1 ...
2 void *dotprod(void * arg)
3 {
4     int myid, i, first_here, last_here;
5     double sum_local=0;
6     myid = (int)arg;
7     first_here = myid*N/NTHREADS;
8     last_here = (myid +1) * N/NTHREADS;
9     for (int i=first_here; i<last_here; i++)
10        sum_local += a[i]*b[i];
11     pthread_mutex_lock(&mutexsum);
12     sum += sum_local;
13     pthread_mutex_unlock (&mutexsum);
14     pthread_exit((void*) 0);
15 }
```



OpenMP Implementation

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 #define N 256
5 double sum = 0;
6 int main (int argc, char * argv[]){
7     double a[N], b[N];
8     int status;
9     for (int i = 0; i<N; ++i){
10         a[i]=i*0.5; b[i]=i*2;
11     }
12     sum=0;
13 #pragma omp for reduction(+:sum)
14     for (int i=0; i<N; ++i) sum += a[i]*b[i];
15
16     printf("sum = %f\n", sum);
17 }
```



OMP #pragma

- tells compiler to parallelize the loop
- identifies *sum* as reduction variable on a sum operation.
- transparent to assignment of loop iterations to threads
- few code than pthread version
- (for more complex problems) probably better performances than pthreaded versions.
- **Fork-Join** programming model: the program starts as a single thread, then it forks many threads, and finally it joins them like with a barrier.
- **initial thread** is the name used for the “parent” thread.



OpenMP Feature Set

- creation of teams of threads for parallel execution
- specification of work sharing policies in a team
- definition of shared and private variable
- threads synchronization and exclusive execution (i.e. without interference of other threads)

C Style

In the following we use C-Style directives and features



OMP Features Set

- Sharing work among threads
- if no specification, all threads redundantly execute all the code
- implicit synchronization at the end of work-sharing construct.
- most common work sharing is in loops
- OpenMP tries to assign the more disjoint sets of iterations as possible (more complex policies are possible)



OMP Memory Model

- By default, data is shared among threads and it is visible to all of them.
- **private** variables allow for thread-specific management (in a memory called thread stack)
- transparent synchronization on shared memory variables
- memory synchronization by **flush** primitive



OMP Thread Synchronization

- Small and simple set of synchronization features
- reduced synchronization errors
- barrier
- critical regions
- mutexes
- Number of Threads ... and Thread Numbers



Performance and Speedup

Parallel or not Parallel ...

Virtually all programs contain some regions that are suitable for parallelization and other regions that are not.

Amdahl's Law

$$S = \frac{1}{(f_{par}/P + (1-f_{par}))}$$

where: f_{par} is the parallel fraction of the code and P is the number of Processing Unit.

Exercise

Estimate f_{par} of the last OMP code chunk.



Open MP directives

General form

```
#pragma omp directive-name [clause[,]clause] ... new-line
```

Programming in C

- include `<omp.h>`
- compile **AND** link (both) with `-fopenmp` option



Exercise



Matrix Times Vector

Parallelize the operation of multiplying a matrix $M[m, n]$ for a vector $v[n]$:

$$r_i = \sum_{j=1}^n M_{i,j}v_j \quad i = 1, \dots, m$$

Waiting for Questions



Exercise

Matrix Times Vector

Parallelize the operation of multiplying a matrix $M[m, n]$ for a vector $v[n]$:

$$r_i = \sum_{j=1}^n M_{i,j} v_j \quad i = 1, \dots, m$$

Waiting for Questions

...



Exercise

Matrix Times Vector

Parallelize the operation of multiplying a matrix $M[m, n]$ for a vector $v[n]$:

$$r_i = \sum_{j=1}^n M_{i,j}v_j \quad i = 1, \dots, m$$

Waiting for Questions



Exercise

Matrix Times Vector

Parallelize the operation of multiplying a matrix $M[m, n]$ for a vector $v[n]$:

$$r_i = \sum_{j=1}^n M_{i,j} v_j \quad i = 1, \dots, m$$



Waiting for Questions

...



Exercise

Matrix Times Vector

Parallelize the operation of multiplying a matrix $M[m, n]$ for a vector $v[n]$:

$$r_i = \sum_{j=1}^n M_{i,j}v_j \quad i = 1, \dots, m$$

Waiting for Questions

What Directive Can I Use

Don't Mind for the moment ... simply underline loops you want to parallelize ... and variables to share



Take a Look at code

Sequential Code

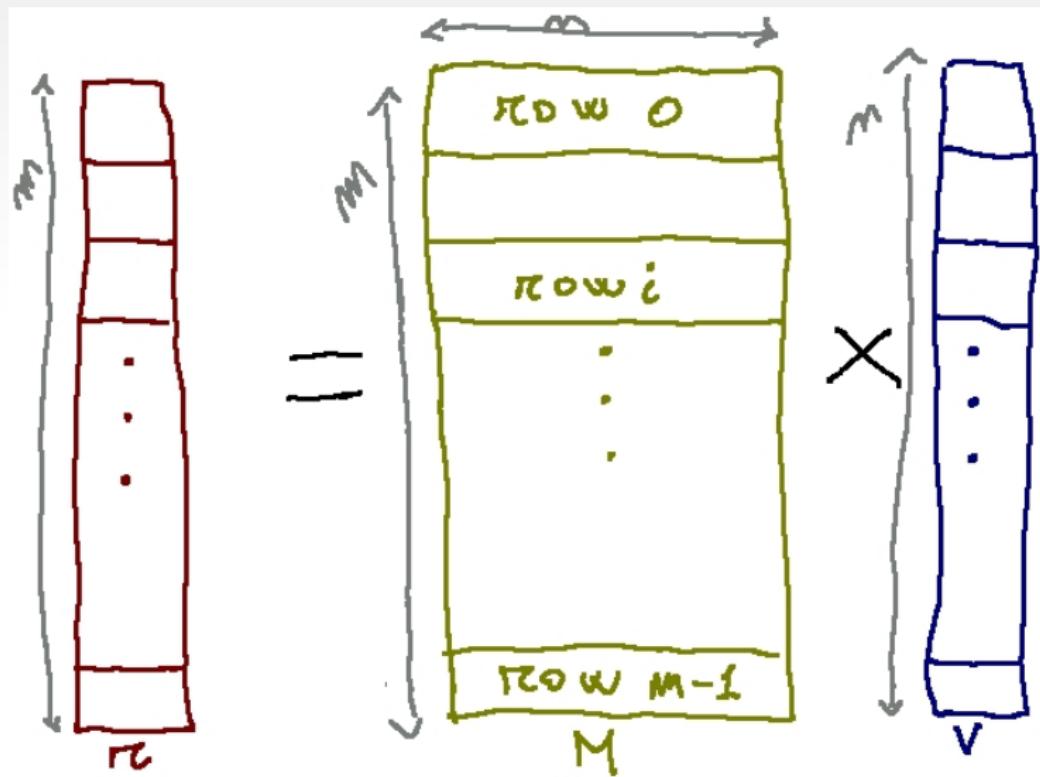
Click Here!

OpenMP Code

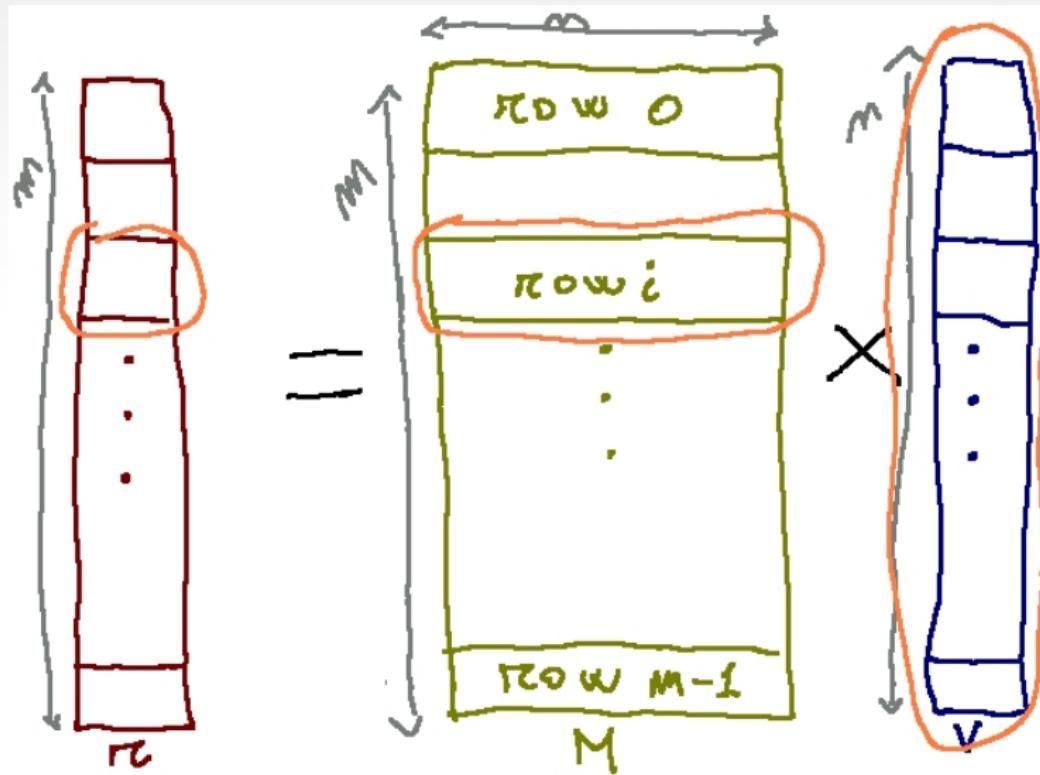
Click Here! Where is Parallelization ?



Example Parallelization

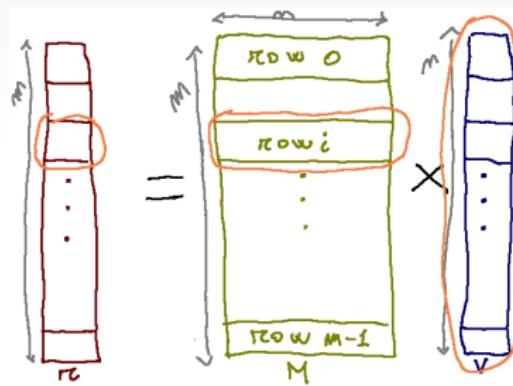


Example Parallelization



Example Parallelization

- we parallelize rows dot vector operations
- we use a single directive because we parallelize only one loop



Some remarks

- Data in OMP either is shared by threads in a team, or is private
- when private, each team owns a copy of the variable
- clauses *shared* and *private* defines scope of variables in the team
- OMP provides built-in data sharing attribute
- the clause *default(none)* informs the OMP compiler that we do not want to rely on automatic assignment of variable scope: we prefer to declare explicitly private and shared variables.
- for performance issues, it is better to have private variables (if we can ...)
- each thread in a parallel region that completes its portion of work, encounters a *barrier*



Conditional Compilation

You can use this macro to compile both sequential and parallel version of your program (i.e. if you use or not the -fopenmp option)

```
1 #ifdef _OPENMP
2     #include <omp.h>
3 #else
4     #define omp_get_thread_num() 0
5 #endif
6 ...
7 int TID = omp_get_thread_num();
```

Try It

... with hello world OMP program



OMP Constructs

- Parallel
- Loop (Work-Sharing)
- Sections (Work-Sharing)
- Single (Work-Sharing)
- Data-Sharing, No Wait, Schedule Clauses
- Barrier
- Critical
- Atomic
- Locks
- Master



Parallel Construct

#pragma omp parallel [clause[,]clause] ...] structured block

```
1 #pragma omp parallel
2 {
3     printf ("Parallel Region executed by %d thread\n",
4            omp_get_thread_num());
5     if (omp_get_thread_num() == 1)
6         printf(" I'm thread %d and I'm different\n",
7                omp_get_thread_num());
8 } //end of parallel region
```



parallel clauses

Parallel Clauses

if(*scalar-expression*)
num_threads(*integer-expression*)
private(*list*)
firstprivate(*list*)
shared(*list*)
default(*none|shared*)
copyin(*list*)
reduction(*operator : list*)



Sharing work

Functionality	Directive
Distribute iterations over the threads	#pragma omp for
Distribute independent work units	#pragma omp sections
Only one thread executes the code block	#pragma omp single



Work Sharing with parallel and for

```
1 #pragma omp parallel shared(n) private (i)
2 {
3     #pragma omp for
4     for (i = 0; i<n; ++i)
5         printf("Thread %d, executes iteration %d\n",
6             omp_get_thread_num(), i);
7 }
```

work sharing

A parallel region that shares inner for iterations among its threads.

Try it!



Loop

aim

Implicit Barriers ensure that results are available when needed (?)

```
1 #pragma omp parallel shared (n,a,b) private (i)
2 {
3     #pragma omp for
4     for (i=0;i<n;++i) a[i] = i;
5     #pragma omp for
6     for (i=0;i<n;++i) b[i]= 2*a[i];
7 }
```



Loop clauses

Loop Clauses

private(*list*)
firstprivate(*list*)
lastprivate(*list*)
nowait
ordered
schedule(*kind*[, *chunk_size*]
reduction(*operator* : *list*)



Sections

aim

If two or more threads are available, one thread invokes funcA(), and another calls funcB(). Other threads are idle!

```
1 #pragma omp parallel
2 {
3     #pragma omp sections
4     {
5         #pragma omp section
6             (void) funcA();
7         #pragma omp section
8             (void) funcB();
9     }
10 }
```



Sections clauses

Sections Clauses

private(*list*)

firstprivate(*list*)

lastprivate(*list*)

nowait

reduction(*operator* : *list*)



Single

#pragma omp single [clause[,]clause] ... structured block

```
1 #pragma omp parallel shared(a,b) private (i)
2 {
3     #pragma omp single
4     {
5         a=42; printf("Single here by %d\n",
6             omp_get_thread_num());
7     }
8     /*Implicit barrier here*/
9     #pragma omp for
10    for (i=0;i<n;++i) b[i]=a;
11 }
12 printf("After parallel : \n");
13 for (i=0;i<n;++i)
14     printf("b[%d]=%d\n",i,b[i]);
```



Master

Like *single*, but the execution is demanded to master thread only.



Combined Constructs

Full Version	Combined
<pre>#pragma omp parallel { #pragma omp for for-loop }</pre>	<pre>#pragma omp parallel for for-loop</pre>
<pre>#pragma omp parallel { #pragma omp sections { [#pragma omp section <i>structured block</i>] [#pragma omp section <i>structured block</i>] ... }</pre>	<pre>#pragma omp parallel sections { [#pragma omp section <i>structured block</i>] [#pragma omp section <i>structured block</i>] ... }</pre>



Other Clauses

- lastprivate: allows for access to private variables after the end of a parallel region
- firstprivate: allows of pre-initialization of private variables in a parallel region
- default: gives variables referenced in the construct a default visibility (i.e. private, shared, none). if *none* is specified, programmer must address all variables.
- nowait: removes implicit barriers at the end of parallel regions



Schedule Clause

Loop Clause

It Controls the way loop iterations are distributed over the threads.
(This have a major impact on performances)

Loop Syntax

schedule(kind[, chunk_size])

Chunk size

Represents the granularity of workload assigned to the threads in the team. It must be a constant: any loop invariant integer expression with a positive value is allowed.



Schedule Clause Kinds

- static: Iterations are divided into chunks. They are assigned to the threads statically in round robin. If no chunk size is specified, the iteration space is divided equally.
- dynamic: Iterations are assigned to threads as the threads request them. Threads execute a chunk, then request another chunk. If no chunk size is specified, default is 1.
- guided: like dynamic; if chunk size is 1, the size of chunk is proportional to the number of unassigned iterations, divided by the number of threads; if chunk size is “ k ”, the size is evaluated as defined before, but chunks must contain *at least k iterations*.
- runtime: decision is made at runtime. The schedule and (optional) chunk size are set through *OMP_SCHEDULE* environment variable.



Other Constructs

Barriers at some points

#pragma omp barrier

Execution in sequential order

#pragma omp ordered *structured-block*

Critical Regions definition

#pragma omp critical [(*name*)] *structured-block*

Mark Atomic updates of Shared Variables

#pragma omp atomic *structured-block*



Locks

Locks

Low-level, general-purposes locking runtime, similar to semaphores.
More flexible than *critical* and *atomic*.

Syntax

```
void omp_func_lock (omp_lock_t * lck)
```

func values

init, destroy, set, unset, test

for nested locks: init_nest, destroy_nest, set_nest, unset_nest,
test_nest

lock variables

lock variables type: **omp[*_nest*]*_lock_t***



Other Clauses

- if : for parallel region, define condition to execute parallelization (sequential code is executed otherwise)
- num_threads: in parallel construct, specifies how many threads should be in the team executing the parallel region.
- ordered: executes the statement in order independently from executing thread.



Reduction Clause

Syntax

reduction (operator:list)

Meanings

executes the selected operation on the list of variables in the parallel for region, that have been updated locally by each thread.

Operators and initialization

+ : (0); * : (1); - : (0); & : (\sim 0); | : (0); ^ : (0); && : (1); || : (0);



Next Step

Performance (speedup etc.) and measurements



Any Question ?



¹image from: <https://pigswithcrayons.com/illustration/dd-players-strategy-guide-illustrations/>



Shared Memory Programming (2)

Performances Issues and Analysis
Introduction to OpenMP

Francesco Moscato

Università degli Studi di Salerno
fmoscato@unisa.it

Outline

- ① Performance Issues
- ② Measuring OpenMP Performances
- ③ Best Practices
- ④ Questions



Sequential Issues: Memory Management

Exploit Locality and Cache !

Memory Access (C version) good Memory Access

```
1 for (int i=0; i<n; i++)  
2     for (int j=0; j<m; j++)  
3         sum += a[i][j];
```

Memory Access (C version) bad Memory Access

```
1 for (int j=0; j<m; j++)  
2     for (int i=0; i<n; i++)  
3         sum += a[i][j];
```



Translation-Lookaside Buffer

- Memory Addresses are (usually) virtualized
- Logical Addresses arranged in virtual pages
- Pages size is usually 4 or 8 KByte (but much larger are possible)
- TLB improves performances when finding pages
- TLB have performances issues when addressing unreferenced pages (just like CPU caches)

The better is having heavily referenced pages



Loops

- Programs spend the most of their time in loops
- loops can be managed (and modified) in order to improve performances
- problems include: references of memory locations, overhead of loops management
- Loops can be restructured, transformed (loop exchange)

Loop Unrolling is widely used



Short Loop nest

Short loops have high relative management overhead

```
1 for (int i=1; i<n; i++) {  
2     a[i]=b[i]+1;  
3     c[i]= a[i]+a[i-1] + b[i-1];  
4 }
```

Unrolled Loop by a factor of 2 reduces overhead

```
1 for (int i=1; i<n; i+=2) {  
2     a[i]=b[i]+1;  
3     c[i]= a[i]+a[i-1] + b[i-1];  
4     a[i+1]=b[i+1]+1;  
5     c[i+1]= a[i+1]+a[i] + b[i];  
6 }
```



Short Loop nest

Outer loop Unrolling

```
1 for (int j=0; j<n; j+=2) {  
2     for (int i=0; i<n; i++)  
3         a[i][j] = b[i][j] +1;  
4     for (int i=0; i<n; i++)  
5         a[i][j+1] = b[i][j+1] +1;  
6 }
```

No benefits from loop unrolling

```
1 for (int j=0; j<n; j++)  
2     for (int i=0; i<n; i++)  
3         a[i][j] = b[i][j] +1;
```

Unroll and jam

```
1 for (int j=0; j<n; j+=2)  
2     for (int i = 0; i<n; i++) {  
3         a[i][j] = b[i][j] +1;  
4         a[i][j+1] = b[i][j+1] +1;  
5     }
```

Loop Fusion

2 Loops both accessing a

```
1 for (int i=0; i<n ; ++i)
2   a[i]=b[i] *2;
3 for (int i=0;i<n;i++) {
4   x[i] = 2 * x[i];
5   c[i] = a[i] +2;
6 }
```

Loop Fusion

```
1 for (int i=0;i<n;i++) {
2   a[i] = b[i] * 2;
3   c[i] = a[i] +2;
4   x[i] = 2 * x[i];
5 }
```



Loop Fission

Poor Cache use and Bad Memory Access

```
1 for (int i = 0 ; i<n; i++){  
2     c[i] = exp(i/n);  
3     for (j=0; j<m; j++)  
4         a[j][i] = b[j][i] + d[j] *  
5             e[i];  
6 }
```

Loop Fission

```
1 for (int i=0; i<n; i++)  
2     c[i] = exp (i/n);  
3  
4 for (int j=0; j<m; j++)  
5     for (int i = 0; i<n; i++)  
6         a[j][i] = b[j][i] + d[j]*e[  
    i];
```



First of all: bintime

time

Use the command to evaluate: *real*, *user* and *sys* times

Do not forget to evaluate overhead

Measure time **both** for Sequential (i.e. without OpenMP) and Parallel on 1 CPU (i.e with OpenMP but only with 1 thread)



Example

Version	Number of Processors	CPU time (s)	Elapsed time (s)	Speedup	Efficiency (%)
Serial	1	10.20	10.20	1.00	100
Parallel	1	10.40	10.40	0.98	98
Parallel	2	10.61	5.76	1.77	88
Parallel	4	11.02	3.75	2.72	68
Parallel	8	11.83	3.35	3.04	38

Example of results about parallel performance and speedup with a fraction of parallelized code of 0.95 and 2% overhead (evaluated on elapsed time)



What is to evaluate **before** speedup evaluation

- Evaluate **base** primitives overhead when number of threads or number or size of chunks increase: for each pragma (parallel, for, parallel for etc.)
- Evaluate different workloads (type and size)
- provide “canary” code for regions
- estimate overhead on your system
- Eventually study performances of inner regions in order to analyze load balancing

Overhead

Evaluation helps in choosing the best number of threads for your application

Exercise

Do this now ...



Best Practice

- use compilers optimization options (-Ox), all of them ... evaluate the best option
- Evaluate the use of shared, private, lastprivate and firstprivate
- optimize barrier use (remember implicit barriers)
- avoid the *ordered* construct if not necessary (it reduces benefits from optimizations)
- avoid large critical regions, maximize critical regions
- avoid parallel regions in Inner Loops
- Address poor load balance
- avoid *false sharing*
- choose a “good” technique to allocate arrays
- **Do NOT print nothing and Do NOT include debug symbols**
- **Evaluate performances of CORRECT programs (test and debug come first)**



Exercise

Matrix dot Vector

Parallelize (with OpenMP) and evaluate performances, speedup, efficiency of your parallel version of Matrix dot Vector algorithm.

Final Project

This is part of your Final Project ... Produce a Report motivating your choices and obtained results.

Source Code

Include in the report, your full source code, test cases, input data, makefiles, eventually APIs and inner code documentation.

Copyrights and other License issues

Please read the Requirements reported on the E-Learning Platform



Any Question ?



¹image from: [https://pigswithcrayons.com/illustration/
dd-players-strategy-guide-illustrations/](https://pigswithcrayons.com/illustration/dd-players-strategy-guide-illustrations/)



Distributed Memory and Message Passing Programming

Programming with MPI

Francesco Moscato

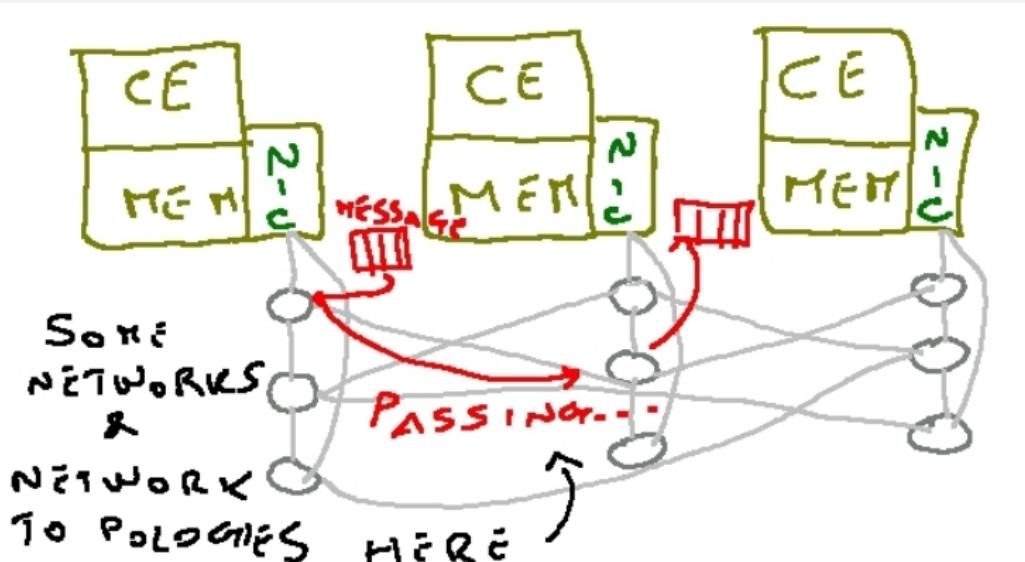
Università degli Studi di Salerno
fmoscato@unisa.it

Outline

- ① MPI Basics
- ② Point to Point Communication
- ③ Collective Communication
- ④ Derived DataTypes
- ⑤ IO
- ⑥ Questions



Distributed Memory Model



DISTRIBUTED MEMORY



Distributed Memory and Message Passing

Distributed Memory

By Message Passing, Nodes with local Memory create an *Abstract* Memory that is distributed among nodes

Passing Data

Data in local memories are shared among nodes in the sense that nodes that own data in their local memories can send them to other nodes by using proper communications

Nodes and Communication

Nodes access rapidly to their local memory and other nodes access to the same data through communication networks (usually high-speed networks)



Message Passing Interface (MPI)

MPI

MPI is a *standard, vendor-independent, portable library*

Supported Languages

C, C++, Fortran, Fortran90, Fortran 2008

Some wrappers for other Interpreted languages:

e.g.: Python with mpi4py

MPI is not ...

A Language;

a framework for automatic parallelization



Common Implementation

MPICH

<http://www.mpich.org>

OpenMPI

<http://www.open-mpi.org>

Intel MPI

<https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html>



Running MPI

Single Machine or Clusters

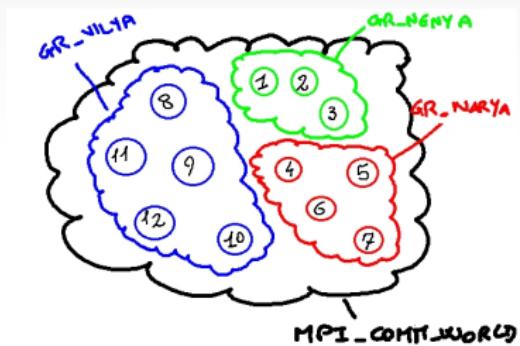
MPI creates *processes*: You can have some speedup on single, multi-core CPUs: performances do not improve when you use more processes than cores.

Obviously you can install the MPI middleware to run on many Nodes on a cluster or even on different nodes on local networks or internet.



Communicators and Groups

- MPI clusters processes into **groups** for communication
- A **Communicator** is a container of processes that can communicate together.
- First **Groups** are created and *then* a Group is used to create a Communicator
- ***MPI_COMM_WORLD*** is the default communicator for *ALL* processes.
- Processes in a Communicator are identified by a unique **rank** (i.e. an ID)
- The **size** of a communicator is the number of processes it manages. It cannot be changed after Communicator creation



First MPI Example

Hello World

```
1 #include <mpi.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main (int argc, char * argv[])
6 {
7     int rank, size;
8     MPI_Init(&argc, &argv); //init MPI environment
9     //get rank and size of communicator
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11    MPI_Comm_size(MPI_COMM_WORLD, &size);
12
13    printf("Hello! I am rank # %d of %d processes\n", rank, size);
14
15    MPI_Finalize(); //Terminate MPI execution env.
16    exit(EXIT_SUCCESS);
17 }
```



Compile and Run First Example

MPICH version on Linux

```
1 $ mpicc firstmpi.c -o firstmpi
2 $ mpirun -np 2 ./firstmpi
3
4 Hello! I am rank # 1 of 2 processes
5 Hello! I am rank # 0 of 2 processes
```



Blocking or Non Blocking ?

Synchronous Communications

Implement blocking p2p communications

E.G.: *MPI_Send()* and *MPI_Recv()*

Asynchronous Communications

Implement non-blocking communications

E.G.: *MPI_Isend()* and *MPI_Irecv()*

Not p2p communications

Collective communication calls like: *MPI_Broadcast*,
MPI_Reduce, *MPI_Barrier* ...



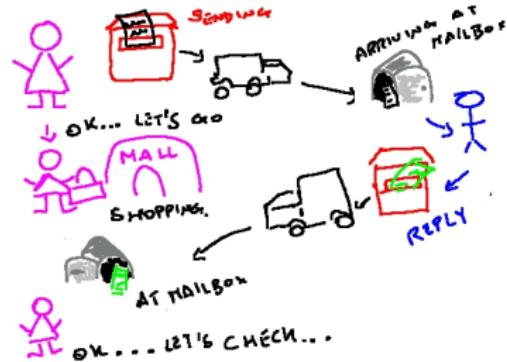
Blocking (Synchronous) calls



- Peer is waiting for a message
- Message is sent
- Sender waits for a reply
- Peer sends the reply
- Sender gets the reply ack or receipt



Non-Blocking (Asynchronous) calls



- Sender uses a Mailbox to send the message, than it can *do other*
- Message arrives at peer's mailbox
- Peer reads the message
- Peer replies by using a Mailbox
- Message arrives at sender's mailbox
- Sender reads the message from the mailbox



Blocking Send

MPI_Send

```
1 int MPI_Send(void *buf, int count, MPI_Datatype datatype,
2                 int dest, int tag, MPI_Comm comm)
```

- ***buf** points to a memory buffer with data
- **count** is the number of elements in buf
- **datatype** is the (MPI) type of data in buf
- **dest** is the rank of dest process
- **tag** is an int that identifies the type of communication (it can be used to define channels in communicators)
- **comm** is the Communicator where process sends data



MPI Datatypes

- *MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_LONG*
- Array of DataTypes
- indexed arrays of blocks of datatypes
- arbitrary structure of datatypes
- Custom Datatypes ...



MPI_Send() and MPI_Ssend()

Blocking calls, with some differences:

- **MPI_Send()**: returns to the application when the buffer is available to (re)use (e.g. when a small buffer has been copied to an internal buffer, before the receiving process executes the receive).
- **MPI_Ssend()**: always waits for the receiver to complete the Recv call, even if the message is buffered internally



Blocking Receive

MPI_Recv

```
1 int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
2               int source, int tag, MPI_Comm comm, MPI_Status *
3               status)
```

- ***buf** points to a memory buffer with data
- **count** is the number of elements in buf
- **datatype** is the (MPI) type of data in buf
- **source** is the rank of sending process
- **tag** is an int that identifies the type of communication (it can be used to define channels in communicators)
- **comm** is the Communicator where process sends data
- **status** is a data structure containing info on received message



Example

array sending

Let's have two processes, one owning a vector of positive numbers, another owning a vector of negative numbers.
Let's these processes exchange their vectors.

runinng

`mpirun -np 2 ./exchange`



Examples

Wrong use of Send and Recv

Click Here!

Good Use of Send and Recv

Click Here!



Send and Receive call

MPI_Sendrecv

```
1 int MPI_Sendrecv(void* sendbuf, int sendcount, MPI_Datatype  
    senddatatype, int dest, int sendtag, void* recvbuf, int  
    recvcount, MPI_Datatype recvdatatype, int src, int recvtag,  
    MPI_Comm comm, MPI_Status * status);
```

- **sendbuf** send buffer
- **sendcount** is the number of elements in sendbuf
- **senddatatype** is the (MPI) type of data in sendbuf
- **dst** is the rank of recipient process
- **sendtag** tag for send message
- **recvbuf** buffer at receive point
- **recvcount** is the number of elements in recvbuf
- **recvdatatype** is the type of elements in recvbuf
- **src** is the rank of the sender
- **recvtag** is the tag for receive messages
- **comm** is the Communicator where process sends data
- **status** is a data structure containing info on received message



Example

MPI_Sendrcv

Click Here!



Non Blocking Send

MPI_Isend

```
1 int MPI_Isend(
2     void* data,
3     int count,
4     MPI_Datatype datatype,
5     int destination,
6     int tag,
7     MPI_Comm communicator,
8     MPI_Request* request)
```

- data: pointer to data to be written
- count: number of elems in data
- datatype: type of elements in data
- destination: destination rank
- tag: message tag
- communicator: The communicator



Non Blocking Recv

MPI_Irecv

```
1 int MPI_Isend(
2     void* data,
3     int count,
4     MPI_Datatype datatype,
5     int destination,
6     int tag,
7     MPI_Comm communicator,
8     MPI_Request* request)
```

- data: pointer to data to be written
- count: number of elems in data
- datatype: type of elements in data
- source: source rank
- tag: message tag
- communicator: The communicator



Test

MPI_Test

Tests the status of a request (created by a Isend or a Irecv)

```
1 int MPI_Test(
2     MPI_Request* request,
3     int * flag,
4     MPI_Status* status)
```



Wait

MPI_Wait

Waits the execution of a request (created by a Isend or a Irecv)

```
1 int MPI_Wait(  
2     MPI_Request* request,  
3     MPI_Status* status)
```



Async Communication Example

ISend and IRecv

Click Here!



Exercise

Ring Topology

Realize a Ring Topology by using MPI Synchronous and Asynchronous calls.



Sync Solution

ISend and IRecv

Click Here!



Collective Operations

The most commonly used are:

- Synchronization:
 - *MPI_Barrier()*
- One-To-All Communication
 - *MPI_Bcast()*, *MPI_Scatter()*
- All-to-One Communication
 - *MPI_Reduce()*, *MPI_Gather()*
- All-to-All Communication
 - *MPI_Alltoall()*, *MPI_Allgather()*, *MPI_Allreduce()*



Barrier

Barrier API

```
1 int MPI_BARRIER ( MPI_Comm communicator )
```

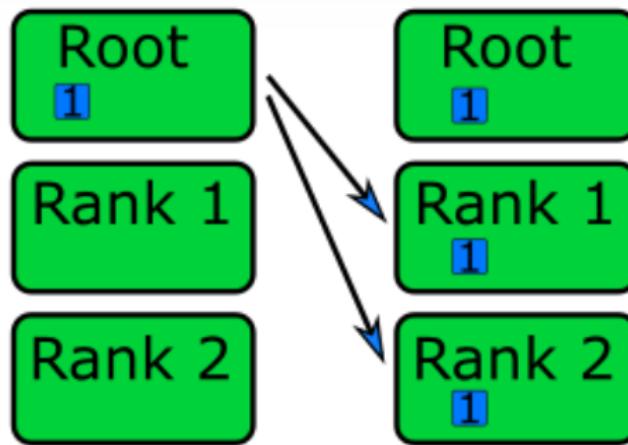
in a barrier all ranks in the communicator wait each other reach the barrier



Broadcast

Broadcast API

```
1 int MPI_Bcast (void * data, int count, MPI_Datatype datatype,  
2                 int root, MPI_Comm communicator)
```



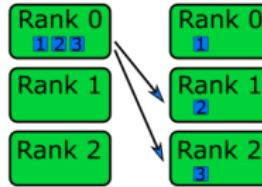
Scatter

Scatter API

```
1 int MPI_Scatter( void* sendbuf, int sendcount, MPI_Datatype  
    sendtype, void* recvbuffer, int recvcount, MPI_Datatype  
    recvtype, int root, MPI_Comm communicator)
```

scatter

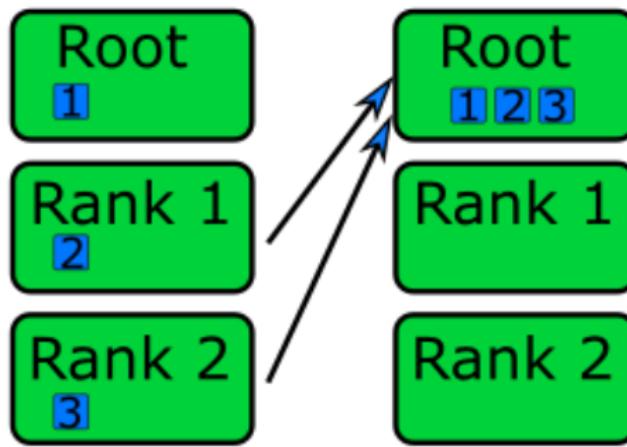
Data in **sendbuf** from **root** is splitted in chunks, each of **sendcount** elements of type **sendtype**. Received data is written to **recvbuf**. Usually $recvcount = N\text{ranks} * sendcount$



Gather

Gather API

```
1 int MPI_Gather( void* sendbuf, int sendcount, MPI_Datatype  
    sendtype, void* recvbuffer, int sendcount, MPI_Datatype  
    recvtype, int root, MPI_Comm communicator)
```



Reduce

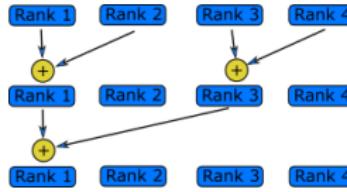
Reduce API

```
1 int MPI_Reduce( void* sendbuf, void* recvbuffer, int count,  
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm  
    communicator)
```

Reduce

Each rank sends a piece of data, which are combined on their way to rank **root** into a single piece of data.

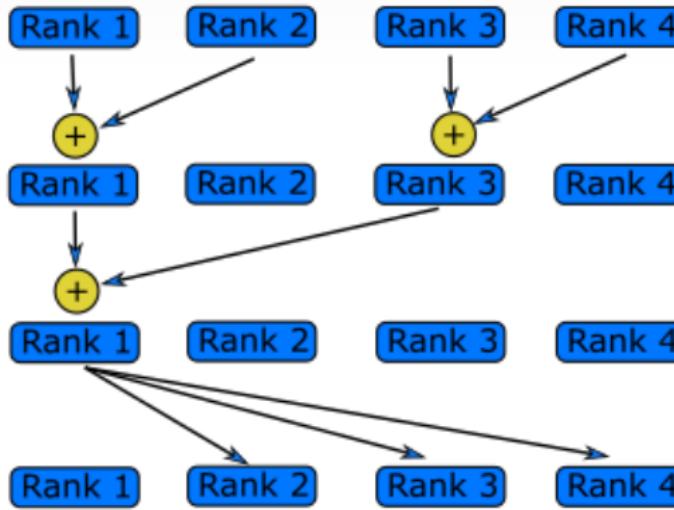
Combination Operations include: *MPI_SUM*, *MPI_MAX*, *MPI_MIN*, *MPI_PROD*,
MPI_MAXLOC, *MPI_MINLOC*



Allreduce

Allreduce API

```
1 int MPI_Allreduce( void* sendbuf, void* recvbuffer, int count,  
                     MPI_Datatype datatype, MPI_Op op, MPI_Comm communicator)
```



Exercise



Gather

Use the Gather API to write a program where each rank sends an “Hello World” to rank0



Possible Solution

Solution

Click Here!



Exercise

Use Reduce and Allreduce

this Code! is wrong: it returns the local sum and maximum.

Modify it by using Allreduce and Reduce to have the right results.



Solution

Solution

Click Here!



Derived Datatypes

- User-defined datatypes
- based on basic MPI datatypes
- Useful when dealing with messages containing non-contiguous data of a single type, or with contiguous or non-contiguous data of mixed datatypes
- user datatypes improve readability and portability



Construction of datatype

- ① Build datatype by using a template. The new datatype has type *MPI_Datatype*
- ② allocate the datatype (*MPI_Type_commit()*)
- ③ use the datatype
- ④ deallocate the datatype



Datatype Construction

Example

```
1 MPI_Datatype new_type; //datatype name declaration
2 ...
3 MPI_Type_XXX(..., &new_type); //construct the datatype
4 MPI_Type_commit (&new_type); //Allocate
5 // ... Some work here
6 ...
7 MPI_Type_free(&new_type); //remove
```



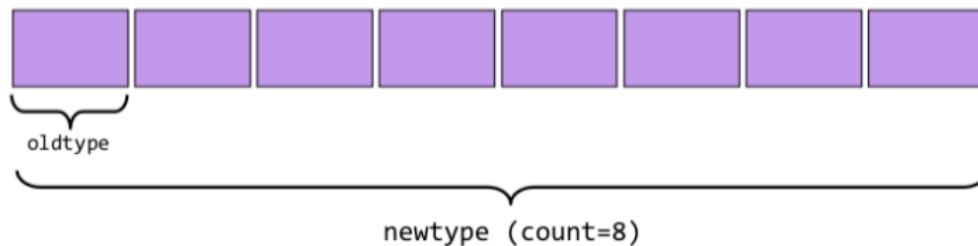
Most used Constructors

- *MPI_Type_contiguous()*: replicates contiguously locations
- *MPI_Type_vector()*: replicates with stride
- *MPI_Type_hvector()*: strides are given in bytes
- *MPI_Type_indexed()* creates a new type from blocks comprising identical elements with varying size and displacement
- *MPI_Type_hindexed* displacements are in byte
- *MPI_Type_create_subarray* creates a datatype corresponding to a distributed multidimensional array
- *MPI_Type_create_struct()* creates a datatype from a generic set of datatypes, displacements and block sizes.



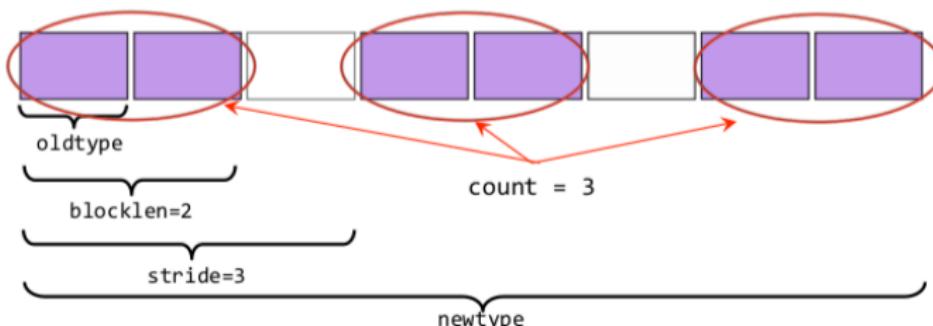
MPI_Type_contiguous()

```
1 int MPI_Type_contiguous(int count, MPI_Datatype oldtype,  
                         MPI_Datatype *newtype)
```



MPI_Type_vector

```
1 int MPI_Type_vector(int count, int blocklen, int stride,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```



MPI_Type_hvector

```
1 int MPI_Type_vector(int count, int blocklen, int stride,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

The same of previous datatype, but it is heterogeneous and strides is specified in “bytes”



Example

Create two datatypes in order to:

- Exchange a given row and a given column in a matrix
- process rank 0 owns the matrix, process rank 1 has to receive one row and one column
- Let the Matrix be $M \times N$
- Since C stores elements row by row: row type is contiguous; col type must be striped

Solution

Click Here!



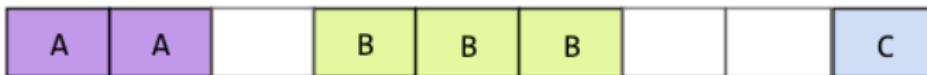
MPI Structure

```
1 int MPI_Type_create_struct(int nblocks, const int  
array_of_blocklen[], const MPI_Aint array_of_displacements  
[], const MPI_Datatype array_of_types[], MPI_Datatype *  
newtype)
```

- **nblocks**: number of blocks. A block is a collection of data of the same type
- **array of blocklen**: an array of int with the size of each block
- **array of displacements**: array that specifies the offset of each block (in bytes)
- **array of types**: array with (old) datatypes
- **newtype**: handle for new datatype



MPI Structure Example



- nblocks: 3
- array of blocklen: 2,3,1
- array of displacement: 0, $3 * \text{sizeof}(A)$,
 $3 * \text{sizeof}(A) + 5 * \text{sizeof}(B)$
- array of types: A,B,C (where A,B,C can be any MPI basic type)

auto alignment

compiler may insert one or more empty bytes to pad structure (e.g.
when mixing chars with int or double)



Safety and Portability

Use *`MPI_Get_address`* to get displacements ...



Example

```
1 typedef struct st {float x; float y; int type; } ST;
2
3 int nblocks = 2, blocklen[] = {2, 1};
4 oldtypes[] = {MPI_FLOAT, MPI_INT};
5 MPI_Aint displ[] = {0, 8}; // Manual setting (not very recommended)
6 MPI_Datatype MPI_ST;
7 ST s;
8 //...
9 MPI_Get_address(&(s.x),&displ[0]);
10 MPI_Get_address(&(s.type), &displ[1]);
11 displ[1] -= displ[0]; displ[0] -= displ[0];
12 MPI_Type_create_struct (nblocks, blocklen, displ, oldtypes, &MPI_ST);
13 MPI_Type_commit(&MPI_ST);
14 s.x = ... // Initialize record here
15 int dst = 0, src = 1;
16 if (rank == src) MPI_Send (&s, 1, MPI_ST, dst, 10, MPI_COMM_WORLD);
17 else MPI_Recv (&s, 1, MPI_ST, src, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



SubArrays

```
1 int MPI_Type_create_subarray(int ndims, const int sizes[], const  
    int subsizes[], const int starts[], int order, MPI_Datatype  
    oldtype, MPI_Datatype * newtype)
```

- `ndims`: number of array dimensions
- `sizes`: number of elements of type `oldtype` in each dimension of the full array
- `subsizes`: number of elements of type `oldtype` in each dimension of the subarray
- `starts`: starting coordinates of the subarray in each dimension
- `order`: array storage order flag (in C: `MPI_ORDER_C`)
- `newtype`: the new datatype handler



Example

$$A = \left(\begin{array}{cccccc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \end{array} \right)$$

Diagram illustrating the distribution of matrix A across two processes. The matrix is 5x6. The first row (a11 to a16) is partitioned into two subvectors: a11 to a13 (size 3) and a14 to a16 (size 2). The second row (a21 to a26) is partitioned into two subvectors: a21 to a23 (size 3) and a24 to a26 (size 2). The third row (a31 to a36) is partitioned into two subvectors: a31 to a33 (size 3) and a34 to a36 (size 2). The fourth row (a41 to a46) is partitioned into two subvectors: a41 to a43 (size 3) and a44 to a46 (size 2). The fifth row (a51 to a56) is partitioned into two subvectors: a51 to a53 (size 3) and a54 to a56 (size 2).

starts = {1, 3}

subsizes[0] = 3
subsizes[1] = 2

sizes[0] = 5

sizes[1] = 6



Managing Files in MPI

- MPI has many routines to manage data from files
- We see here some basic routines
- Properties in basic routines
 - Positioning (with MPI file pointers)
 - Synchronization (blocking or non-blocking)
 - Coordination (local or collective)



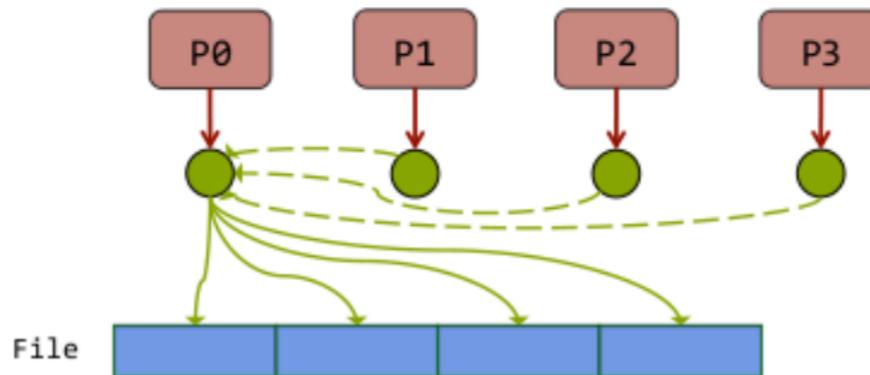
I/O in Parallel Programs

Three different approaches:

- Master-Slave (or sequential)
- Distributed I/O on local files
- Fully parallel I/O



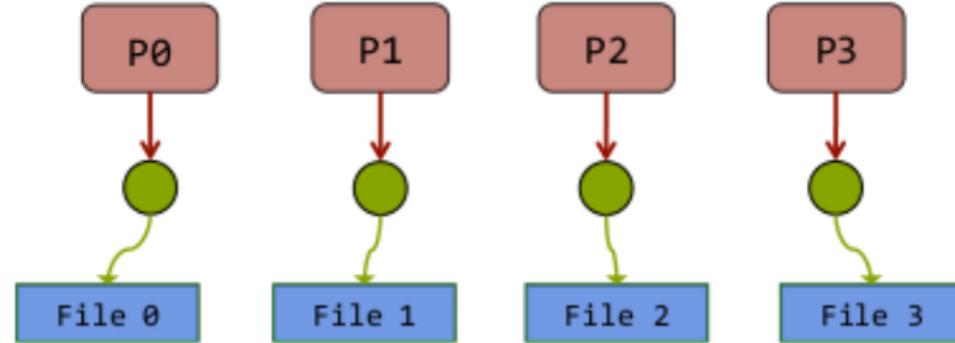
Master-Slave



- Pros: data consistency, parallel machines may have disks only on one node
- Cons: lack of parallelism, lots of communications



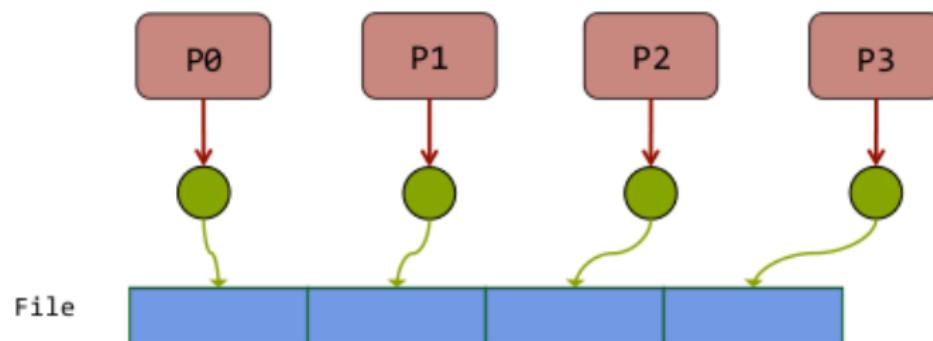
Distributed I/O on Separate Files



- Pros: Scalable, no communications
- Cons: not very usable: too much files if you have lots of processes



Fully Parallel I/O



- Pros: High Performance, avoid communication, single file
- Cons: extra coding



MPI I/O functions

<code>MPI_File_open()</code>	Opens a file on all processes in the communicator group
<code>MPI_File_close()</code>	Closes a file on all processes in the communicator group
<code>MPI_File_delete()</code>	Deletes a file
<code>MPI_File_write()</code> <code>MPI_File_write_all()</code> <code>MPI_File_write_ordered()</code> <code>MPI_File_write_at()</code> <code>MPI_File_write_shared()</code>	Write using individual file pointer; Collective write using individual file pointer; Collective write using shared file pointer; Write using explicit offset. Write using shared file pointer
<code>MPI_File_read()</code> <code>MPI_File_read_all()</code> ...	Read using individual file pointer; Collective read using individual file pointer;
<code>MPI_File_seek()</code>	Updates the individual file pointer
<code>MPI_File_set_view()</code>	Changes the process's view of the data in the file



File Opening

```
1 int MPI_File_open(MPI_Comm comm, char *filename, int amode,  
    MPI_Info info, MPI_File *fh)
```

- amode is the opening mode
- info provides additional information (it is system dependent and you can use *MPI_INFO_NULL*)
- the call is a *collective* routine. All processes must provide the same amode and the same filename
- This supports **ONLY Binary I/O**



amode

MPI_MODE_RDONLY	Open in read only mode
MORE_MODE_RDWR	Open for read/write modes
MPI_MODE_WRONLY	Open in write only mode
MPI_MODE_CREATE	Create file if it does not exist
MPI_MODE_EXCL	Generate error if creating an existing file
MPI_MODE_DELETE_ON_CLOSE	Delete file when closed (used for temporary files).
MPI_MODE_UNIQUE_OPEN	File will not be opened elsewhere by the system
MPI_MODE_SEQUENTIAL	File will not have file pointer moved manually
MPI_MODE_APPEND	Move file pointer to end of file when opening.



Shared and Local File Pointers

MPI supports read/write ops with:

- Shared fp: one rank at a time owns the shared pointer for r/w. This may lead to performances loss. All functions are collective (e.g. *MPI_Write_shared()*, *MPI_Write_ordered()*, *MPI_File_seek_shared()* etc.)
- Local fp: each rank has its own fp. There are both collective and non-collective operations (e.g. non-collective : *MPI_File_write()*, *MPI_File_read()*; collective: *MPI_File_write_all()*)
- File Views: Map data from multiple processors to the file representation on disk.



I/O and Shared Pointers

```
1 int MPI_File_write_ordered(MPI_File fh, void *buf, int count,  
    MPI_Datatype datatype, MPI_Status *status)
```

- collective access using shared fp
- accesses ordered by ranks
- fp moves as processes access to the file
- the same view has to be used on all processes
- read with *MPI_File_read_ordered()*



I/O and local pointers

```
1 int MPI_File_seek(MPI_File mpi_fh, MPI_Offset offset, int whence)  
2  
3 int MPI_File_write(MPI_File mpi_fh, void *buf, int count,  
4                     MPI_Datatype datatype, MPI_Status *status);  
5 int MPI_File_write_all();
```

- Seek operations update local pointer
- whence is the update mode (*MPI_SEEK_SET*, *MPI_SEEK_CUR*, *MPI_SEEK_END*)
- *MPI_File_write()* is not collective, the collective version is *MPI_File_write_all()*



File Views

File View

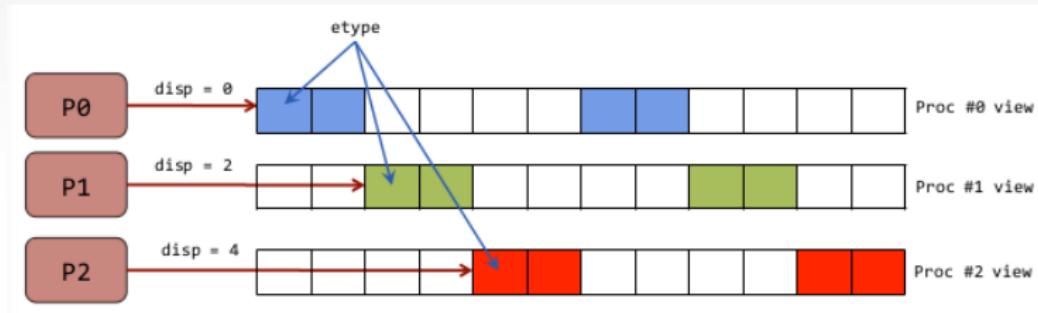
Defines the part of the file that is *visible* to a process, as well as the type of data in the file.

Read and Write

processes access to **bytes** (Binary I/O)



View consist of



displacement: the number of bytes from the beginning of file

etype: the basic unit of data access

filetype: the type of elements in the visible part

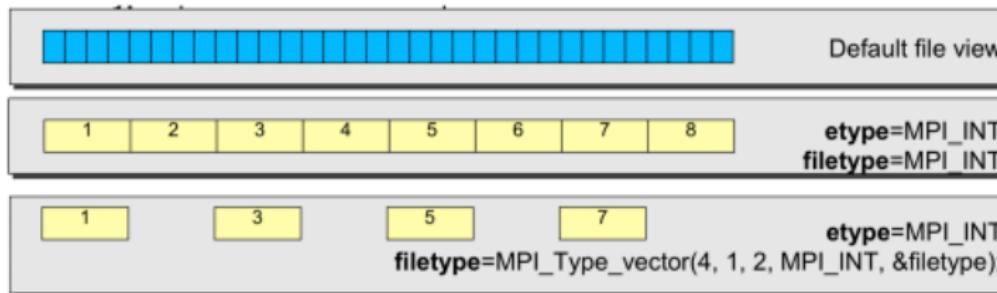


Views setting

```
1 int MPI_File_set_view(MPI_File mpi_fh, MPI_Offset disp,  
    MPI_Datatype etype, MPI_Datatype filetype, char *datarep,  
    MPI_Info info);
```

datarep is the data representation (string)

info describes the object (handle)



datarep in File view

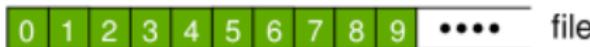
- native: (default) use the memory layout without conversion.
No precision loss, no portability
- internal: layout implementation-dependent. It is portable within the same MPI implementation
- external32: This uses an MPI Standard (32-bit big endian IEEE). It is portable, it has some conversion overhead, it is not implemented everywhere

Internal and external32 portability is guaranteed only when using correct MPI datatypes and not MPI_BYE



Default File View

- the default view is defined by the `MPI_File_open()`
- `disp = 0; etype= MPI_BYTE = filetype;`



view of process 0



view of process 1



view of process 2



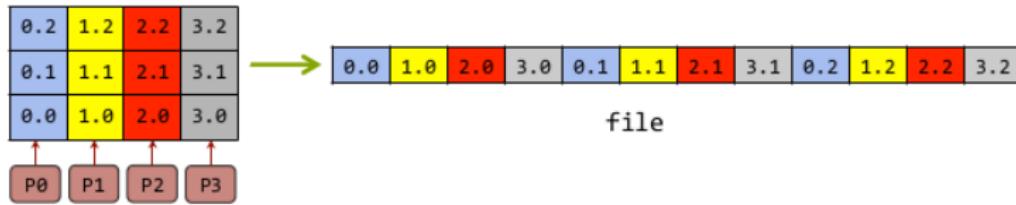
Example

Click Here!



File Views and Non-Contiguous Data

File views are good when writing non-continuously on files



In the example the file view has: count=3, blocklen=1, stride=4



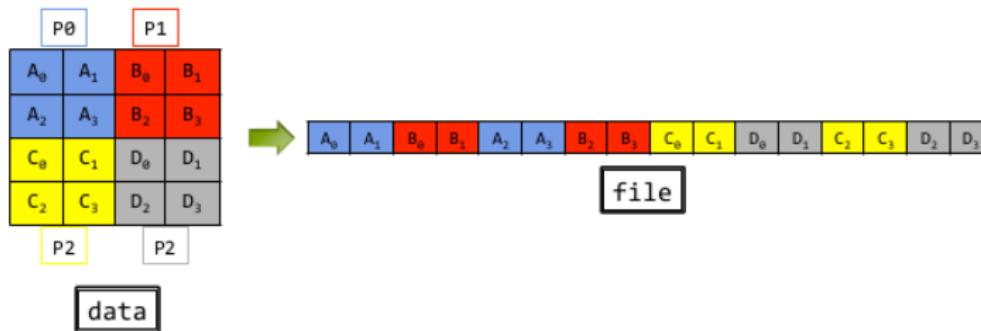
File Views

```
1 for (i = 0; i < NELEM; i++) buf[i] = rank + 0.1*i; // Fill buffer
2 MPI_Datatype vec_type;
3 MPI_Type_vector(NELEM, 1, size, MPI_DOUBLE, &vec_type); // Create
              vector type
4 MPI_Type_commit(&vec_type);
5 disp = rank*sizeof(double);
6 // Compute offset (in bytes)
7 MPI_File_set_view(fh, disp, MPI_DOUBLE, vec_type, "native",
                  MPI_INFO_NULL);
8 // Set view
9 MPI_File_write(fh, buf, NELEM, MPI_DOUBLE, MPI_STATUS_IGNORE);
10 // Write
11 MPI_Type_free(&vec_type);
```



Multidimensional Arrays

- I/O on Multidim arrays should be managed independently from decomposition
- datafiles should be written in a “serial order” (e.g.: row major order in C)
- use a **subarray** datatype
- Use a *Cartesian Decomposition*



Cartesian Decomposition

Cartesian Decomposition

is a parallelization method whereby different portions of the domain are assigned to individual processes

Cartesian Decomposition

Maps a rank to a Coordinate



```
1 int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims []
, const int periods[], int reorder, MPI_Comm * comm_cart)
```



Cartesian Decomposition

- comm_old: input communicator
- ndims: number of dimensions of Cartesian grid (integer)
- dims: integer array of size ndims specifying the number of procs in each dimension;
- periods: logical array of size ndims specifying periodicity (true) or not (false) in each dimension;
- reorder: ranking may be reordered (true) or not (false)
- comm_cart: communicator with new Cartesian topology



Example

- Click Here!



Any Question ?



¹image from: [https://pigswithcrayons.com/illustration/
dd-players-strategy-guide-illustrations/](https://pigswithcrayons.com/illustration/dd-players-strategy-guide-illustrations/)



Distributed Memory and Message Passing Programming

MPI Examples

Francesco Moscato

Università degli Studi di Salerno
fmoscato@unisa.it

Outline

① Dijkstra Algorithm

② 1D Wave Propagation

③ Questions



Require: N as size of the Graph

```
1: function DIJKSTRA( $s$ )
2:    $OK[s] \leftarrow (s, i)$ 
3:   for all  $i$  do
4:      $D[i] = (\text{s}, \text{i})$ 
5:   end for
6:   loop
7:    $x \leftarrow i$  where  $D[i]$  is minimum for  $OK[i] = FALSE$ 
8:    $OK[x] \leftarrow TRUE$ 
9:   for all  $i$  where  $OK[i] is FALSE$  do
10:    if  $D[i] > D[x] + (x, i)$  then
11:       $D[i] \leftarrow D[x] + (x, i)$ 
12:      Indicate we came to  $i$  from  $x$ 
13:    end if
14:  end for
15: end loop
16: end function
```



Sequential Dijkstra

Click Here!

Try your Own

Parallelize with MPI Dijkstra Algorithm



Parallel Dijkstra

Click Here!

This is a Skeleton to IMPROVE

- Inputs from Files, Output to Files
- Try at least one sparse matrix representation
- What about *path[]* in parallel version ? Figure out how to fix the problem
- Use Datatypes



1D Wave Propagation Resume

[Click Here!](#) for Theory

[Click Here!](#) for Code Docs

[Click Here!](#) for MPI Code



Exercise

(Optional) Detect The Problem

This code has a problem : Detect and Correct!

Evaluate Speedup (theoretical and from real runs)



Any Question ?



¹image from: <https://pigswithcrayons.com/illustration/dd-players-strategy-guide-illustrations/>



CUDA Architecture and Programming Model

CUDA

Francesco Moscato

Università degli Studi di Salerno
fmoscato@unisa.it

Outline

- ① Introduction
- ② GPGPU CUDA Model
- ③ CUDA Examples
- ④ Performances
- ⑤ Memory Management
- ⑥ Streams and Concurrency
- ⑦ Questions



GPU

What's a GPU

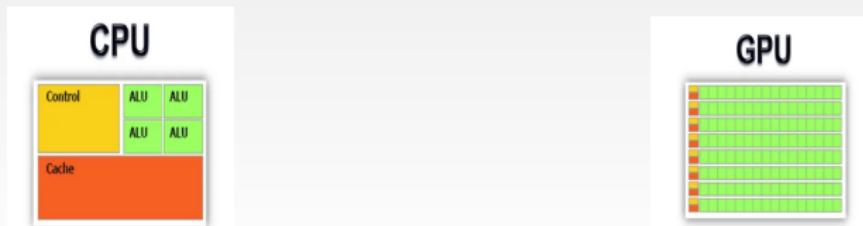
High parallel CPU (*many core*) with private memory and high bandwidth. GPUs are optimized for 3D rendering and other graphical operations, for RT management of high-definition graphics.

Main features

- Intense data-parallel computations
- The *same* algorithm on *many* different elements (in parallel)
- simple control flow (control unit has simple features)
- limited spatial locality (*fine grained parallelism*)
- high arithmetic intensity to hide load/store latencies (GPUs have reduced caches cause they have ... *many cores*)



CPUs vs. GPUs



- Complex Control Logic
- *few* cores
- Large caches (many levels)
- High clock speed
- Shallow pipes

- Many parallel execution units (ALUs)
- Graphics optimizations
- Deep pipes (hundreds of stages)
- High throughput
- simple (but becoming more CPU-like) control flow logic



Normal GPU operations

3D Objects are represented by their 3D vertex. When a point of view in a scene changes:

- 3D coordinates are shifted to the new point of view (scaling, rotations, projections, translations etc.) (*virtual camera*)
- polygons are filled with colors and textures, filters are applied (antialias, smoothing etc), lights and shadows are evaluated (*rendering*)
- 3D scenes are projected to 2D coordinates (pixels) to create images on the screen (*rasterizing*)
- typical scenes have more than 2M vertex, and more than 6M pixels.
- for fluid views, operations have to be applied more than 30-60 times per seconds



General Purpose GPU and GPU computing

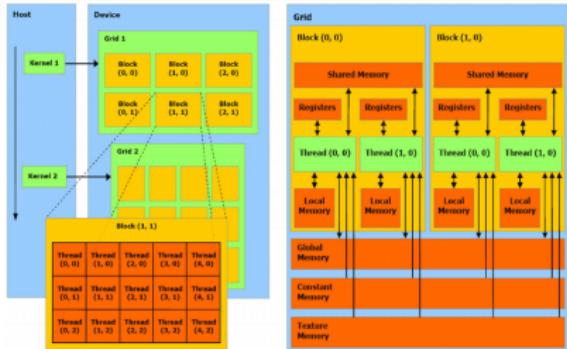
The idea

- Use GPU power to compute *other* than graphics.
- GPU has many ALUs and a device memory
- *Intense data-parallel computations*: the same set of operations executed in parallel on different data ... many times ... in parallel.
- The GPU devotes wafer space to data processing and not to data caching and complex flow control algorithms



CUDA GPU Architectures

- Main Global Memory (high bandwidth)
- Streaming Processor (grouping independent cores and control units)
- Each Unit has:
 - Many ALU-based cores
 - dispatchers for instructions
 - shared memory with high bandwidth and very little access time to data.



CUDA Products

Click Here! for Nvidia Web Site



GPGPU basics

- GPU is used like a coprocessor with thousands of cores and high speed and bandwidth memory
- thousands of threads are executed on the GPU contemporary
- threads are on different GPU cores
- each thread works on different data independently
- GPU parallelism is something like SPMD paradigm
- GPU threads are very *light*. Threads have their registers and there is not context switch.



Cuda Architecture

Compute Unified Device Architecture (CUDA)

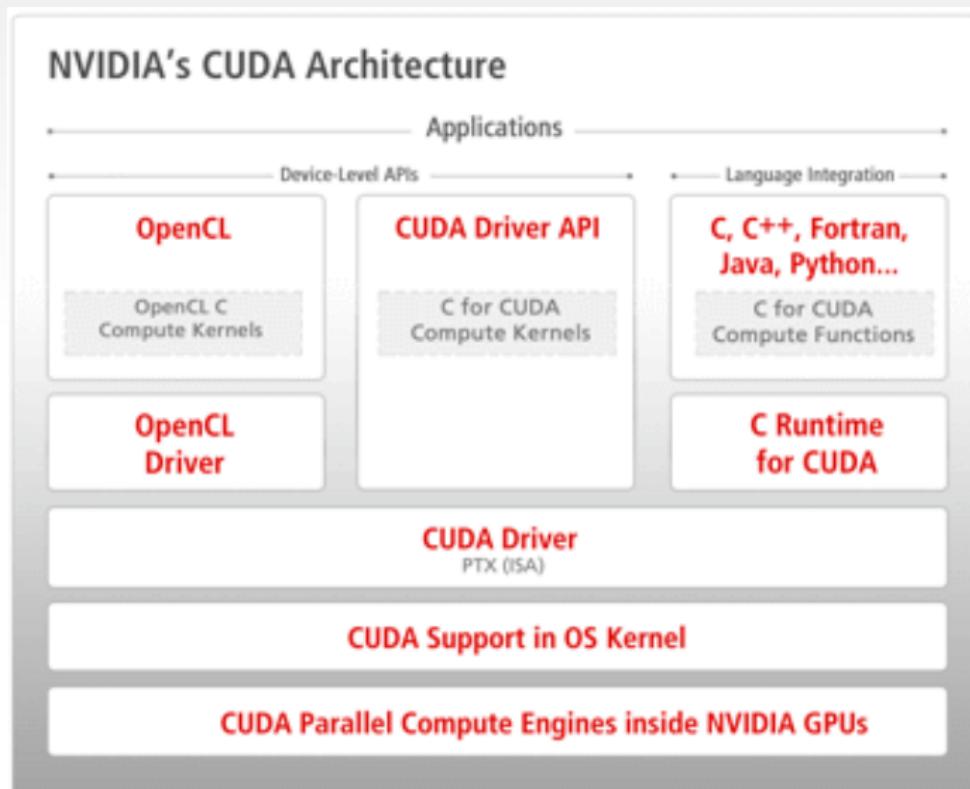
A general purpose architecture and a programming model as well.

It's based on:

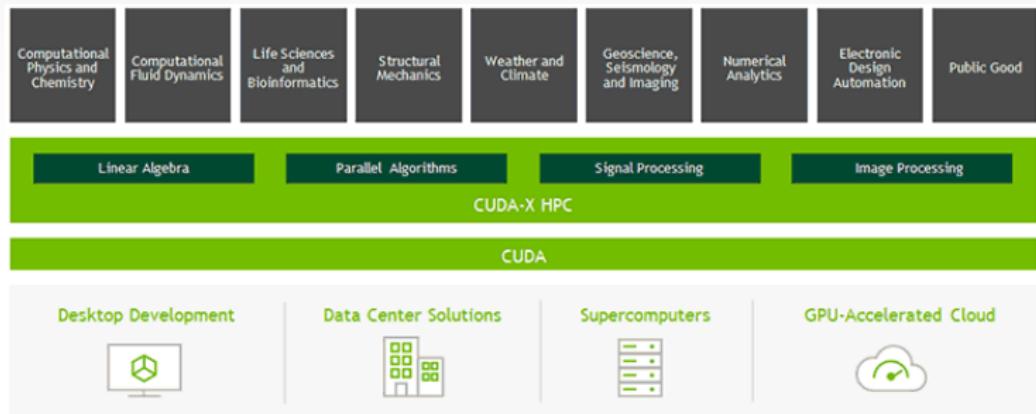
- multi-threading (hierarchical) programming paradigm
- few extensions to C and Fortran languages to describe parallel execution of code
- an instruction for *Parallel Thread eXecution* (PTX)
- a complete framework to develop and profile CUDA programs



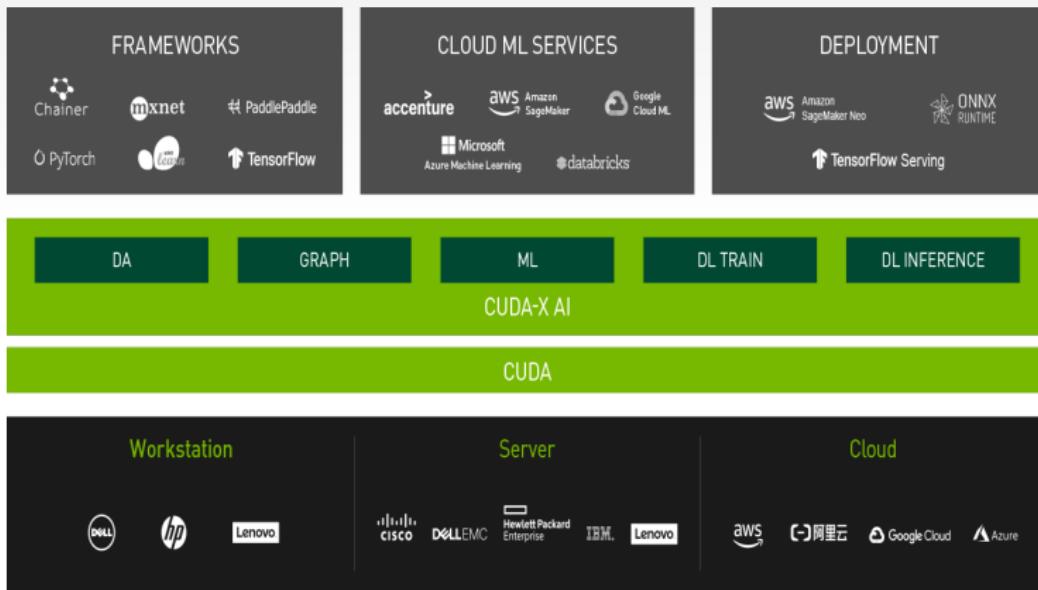
Cuda Architecture main stack



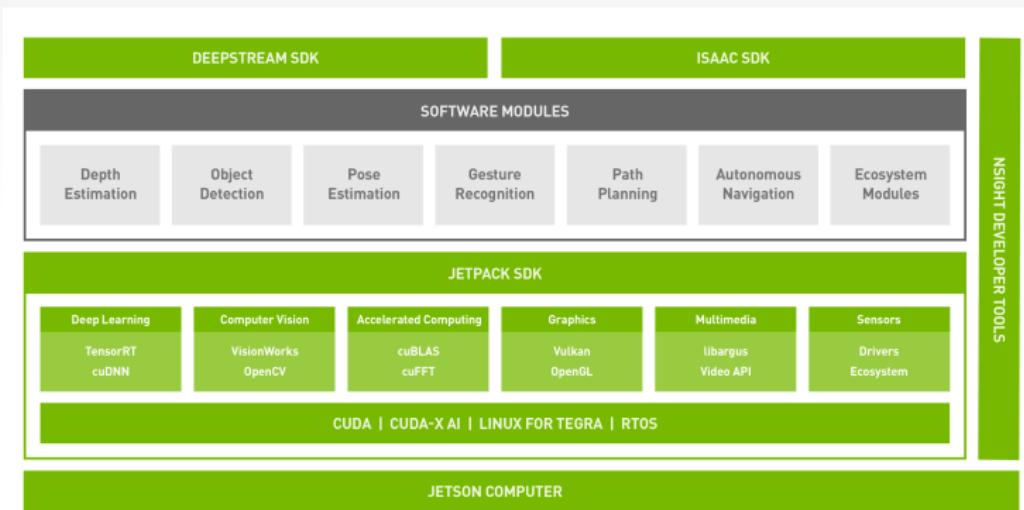
CUDA (current) real view .



CUDA (current) real view ..



CUDA (current) real view ...



CUDA (current) real view



To Infinity ... and Beyond !



CUDA API(s)

CUDA has two levels of APIs (mutually exclusive):

- driver API: low level, requires hard-coding, no “kernel” facilities; finer control over device elements, no need for nvidia compiler
- runtime API: easier programming (need for nvidia compiler): C-for-CUDA language. It lays upon driver API.



Driver API

- the core is in nvcuda dynamic library (cuXXX entries)
- init with culInit() so that a cuda context is created.
- imperative API
- handle-based management of objects
- kernels are loaded as PTX objects by the host code.
- Kernels are launched by using proper API functions



CUDA: Addressing Architecture

Compute Capability

the compute capability of a device is a couple $X.y$: X is the major number and it identifies the chipset architecture; y is the minor number and it identifies the release of the base chipset

Compute Capabilities

- $CC3.7$: A Kepler K80 Architecture
- $CC7.0$: A Volta V100 Architecture
- $CC8.0$: An Ampere A100 Architecture
- $CC8.6$: A GeForce 30xx Architecture



Some Features of Tesla GPUs

Tesla GPU	"Fermi"	"Fermi"	"Kepler"	"Kepler"	"Maxwell"	"Pascal"	"Volta"	"Turing"	"Ampere"
	GF100	GF104	GK104	GK110	GM200	GP100	GV100	TU104	GA100
Compute Capability	2.0	2.1	3.0	3.5	5.3	6.0	7.0	7.0	8.0
Streaming Multiprocessors (SMs)	16	16	8	15	24	56	84	72	128
FP32 CUDA Cores / SM	32	32	192	192	128	64	64	64	64
FP32 CUDA Cores	512	512	1,536	2,880	3,072	3,584	5,376	4,608	8,192
FP64 Units	-	-	512	960	96	1,792	2,688	-	4,096
Tensor Core Units							672	576	512
Threads / Warp	32	32	32	32	32	32	32	32	32
Max Warps / SM	48	48	64	64	64	64	64	64	64
Max Threads / SM	1,536	1,536	2,048	2,048	2,048	2,048	2,048	2,048	2,048
Max Thread Blocks / SM	8	8	16	16	32	32	32	32	32
32-bit Registers / SM	32,768	32,768	65,536	65,536	65,536	65,536	65,536	65,536	65,536
Max Registers / Thread	63	63	63	255	255	255	255	255	255
Max Threads / Thread Block	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024	1,024
Shared Memory Size Configs	16 KB	16 KB	16 KB	16 KB	96 KB	64 KB	Config	Config	Config
	48 KB	48 KB	32 KB	32 KB			Up To	Up To	Up To
			48 KB	48 KB			96 KB	96 KB	164 KB
Hyper-Q	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Parallelism	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Unified Memory	No	No	No	No	No	Yes	Yes	Yes	Yes
Pre-Emption	No	No	No	No	No	Yes	Yes	Yes	Yes
Sparse Matrix	No	Yes							



CUDA programming basics

- Heterogeneous execution (CPU/GPU)
- data-parallel computation: isolated computational slices are called CUDA Kernels
- CUDA Kernels are executed by many different threads in parallel
- Threads Hierarchy
- Threads can compute different data elements independently
- Parallel Architecture: Single Instruction Multiple Threads (SIMT, by NVIDIA), very similar to SIMD.
- no penalty for context switch (each thread has its own registers).
- Programmer chooses the number of threads to run that in turn act on different data element independently



CPU vs GPU

CPU

- Optimized for low-latency accesses to caches
- Complex control logic (out-of-order, prefetch etc.)
- Complex Pipes, Good for reactive, event - driven tasks

GPU

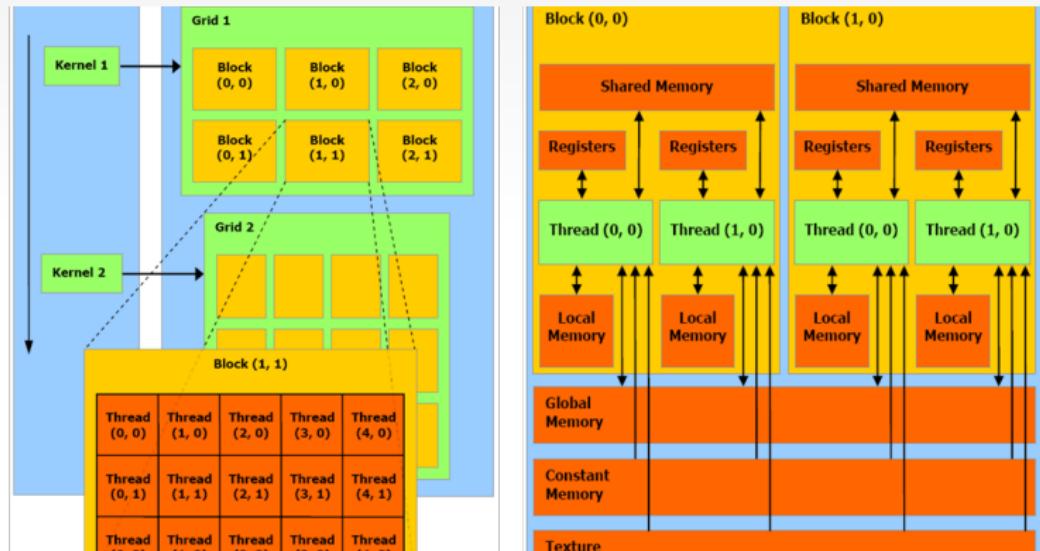
- Optimized for data-parallel tasks
- small slices of instruction to execute in parallel on many data

Hybrid Approach

- Serial parts are executed on CPU (host)
- computational-intensive data-parallel regions on the GPU
- memory transfer from/to Host Memory to/from GPU



GPGPU Thread Hierarchy



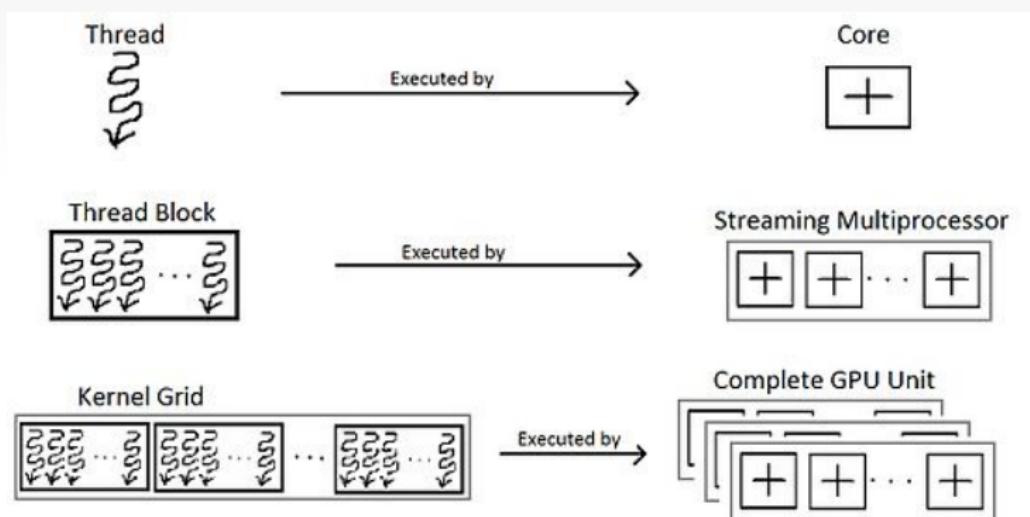
GPGPU Thread Hierarchy

- To Compute N elements on the GPU N threads must be spawned
- Threads are grouped together in Blocks
- Threads in the same block can cooperate exchanging data (they have a shared memory)



Threads and Streaming Multiprocessor

(from en.wikipedia.org)



Threads and Streaming Multiprocessor

- Thread blocks are assigned to SM in round-robin
- different HW generations have different max number of blocks assignable to SM.
- A Block remains on a SM until all threads in the block complete their computations.
- No synchronization among blocks (they are independent)
- Threads in blocks are divided into *Warps* of 32 threads
- a Warp executes one common instruction at a time.



Data Movement

- Host to Device (H2D) and Device to Host (D2H)
- on “classic” systems, usually PCIe and DDR controller drive the movement
- movements only before and after GPU computation
- dedicated buses for data stream exist (NVLink by NVidia) with total data rates of about 300 GByte/s



Warp

- GPU creates and manages warps (groups of 32 threads)
- Threads in a Warp start together at the same address, but evolve depending their own program
- warp executes instructions on SM cores, Load and Store units or Special Function Units (SFUs)



Dealing with Latencies

Latencies

They are the number of cycles to end an operation (e.g.: arithmetic latency 18-24 cycles; memory latency 400-800 cycles)
It's an hardware effect, it cannot be avoided

Solutions to hide latency

- pipeline and saturation of pipelines
- saturation bandwidth in memory bound problems

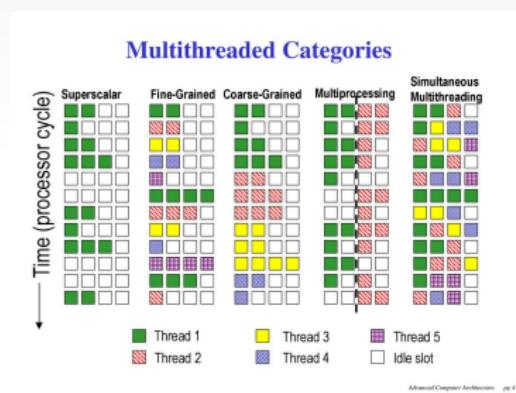
Two possible ways (can be combined)

- Thread-Level Parallelism (TLP)
- Instruction-Level Parallelism (ILP)



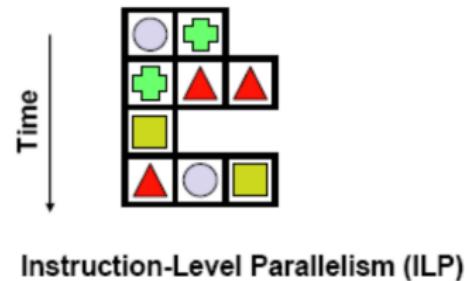
TLP

- provide as much thread as possible to the scheduler can find warps ready to execute with independent operations
- TLP is good if there are low-level independent operations in CUDA kernels.



ILP

- find multiple independent operations in CUDA kernel, allowing kernel to act on different data.
- The scheduler do not select new warp until there are eligible instruction ready to be executed in the current warp since they are independent
- the scheduler stay on the same warp and optimizes load of hardware pipelines



Instruction-Level Parallelism (ILP)



Vector Add: Where is data intensive part?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void initVector(double *u, int n, double c) {
5     for (int i=0; i<n; i++)
6         u[i] = c;
7 }
8 int main(int argc, char *argv[]) {
9     int i;
10    const int N = 1000;
11    double * u = (double *) malloc(N * sizeof(double));
12    double * v = (double *) malloc(N * sizeof(double));
13    double * z = (double *) malloc(N * sizeof(double));
14    initVector((double *) u, N, 1.0);
15    initVector((double *) v, N, 2.0);
16    initVector((double *) z, N, 0.0);
17    // z = u + v
18    for (i=0; i<N; i++)
19        z[i] = u[i] + v[i];
20    printf("%f %f %f\n", z[0], z[1], z[N-1]);
21    return 0;
22 }
```



Vector Add: Where is data intensive part?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void initVector(double *u, int n, double c) {
5     for (int i=0; i<n; i++)
6         u[i] = c;
7 }
8 int main(int argc, char *argv[]) {
9     int i;
10    const int N = 1000;
11    double * u = (double *) malloc(N * sizeof(double));
12    double * v = (double *) malloc(N * sizeof(double));
13    double * z = (double *) malloc(N * sizeof(double));
14    initVector((double *) u, N, 1.0);
15    initVector((double *) v, N, 2.0);
16    initVector((double *) z, N, 0.0);
17    // z = u + v
18    for (i=0; i<N; i++) // <---
19        z[i] = u[i] + v[i]; // <---
20    printf("%f %f %f\n", z[0], z[1], z[N-1]);
21    return 0;
22 }
```



Simple CUDA version

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 ...
4 ...
5 __global__ void gpuVectAdd(double *u, double *v, double *z, int N)
6 {
7     // define index by exploiting thread as index
8     int i = f(threadIdx.x) ;
9     ...
10    z[i] = u[i] + v[i];
11 }
12 ...
13 ...
14 int main(int argc, char *argv[]) {
15 ...
16     // allocate memory on device
17 ...
18     // copy data from host to device
19 ...
20 ...
21     // define the execution configuration: 1 Block of N Threads
22     gpuVectAdd<<<1, N>>>(u_dev, v_dev, z_dev, N);
23 ...
24 }
```



CUDA syntax extensions

- a CUDA Kernel function is defined using the `__global__` keyword before a function
- a kernel is executed N times in parallel by N different CUDA Threads on the device
- The number of CUDA Threads that execute the kernel is declared by the **kernel execution configuration**:
`cudaKernelFunc <<< ... >>> (arg1, arg2, ..., argn)`
- threads have unique ID in the built-in `threadIdx` variable.
- the variables is a 3-component struct (vector) with components `.x, .y, .z`



Kernel execution Configuration

kernelFunction <<< numBlocks, numThreads >>> (...)

where:

- numBlocks: is the grid size in terms of thread blocks along each dimension
- numThreads: is the block size in terms of threads along each dimension

Typical Invocation

```
1 dim3 numThreads(32);  
2 dim3 numBlocks((N-1)/numThreads.x + 1);  
3 kernelFunction<<<numBlocks, numThreads>>>(...);
```



CUDA Threads

- Grouped in Blocks (can be 1D, 2D, 3D sized in threads)
- Blocks can be organized into 1D, 2D, 3D grids of blocks in turn
- Blocks are executed independently
- No assumption of block execution order (they are independent ...)
- Blocks have a unique ID (*blockIdx* variable, that has the usual three components *.x*, *.y*, *.z* that represent block coordinates inside the Grid)
- *threadIdx* contains the coordinates of a thread inside a Block
- *blockDim* is the dimension of a block in terms of thread units
- *gridDim* is the dimension of the grid in block units.



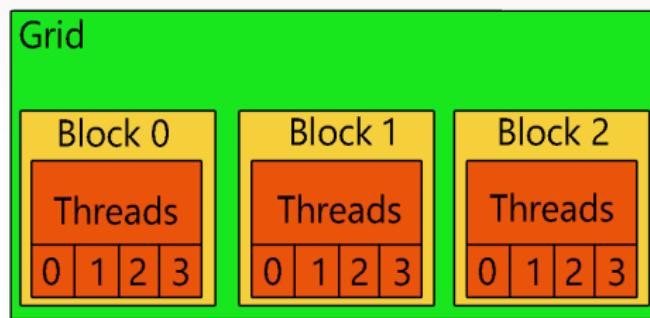
index in 1D grid of 1D blocks

1D Grid of 1D Blocks



index in 1D grid of 1D blocks

1D Grid of 1D Blocks

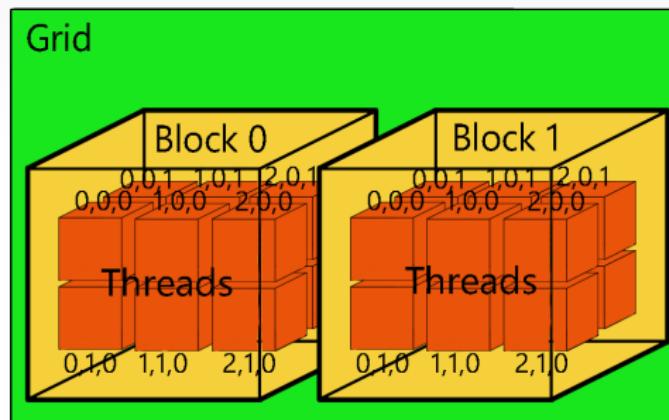


```
int index = blockIdx.x * blockDim.x + threadIdx.x;
```



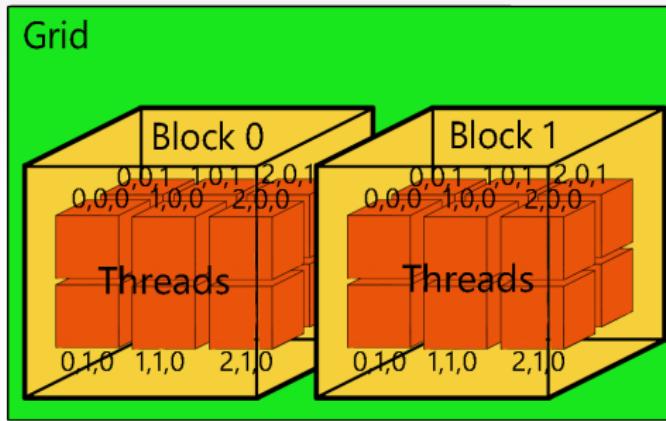
index in 1D grid of 3D blocks

1D Grid of 3D Blocks



index in 1D grid of 3D blocks

1D Grid of 3D Blocks



```
int index = blockIdx.x * blockDim.x * blockDim.y *  
blockDim.z + threadIdx.z * blockDim.y * blockDim.x +  
threadIdx.y * blockDim.x + threadIdx.x;
```

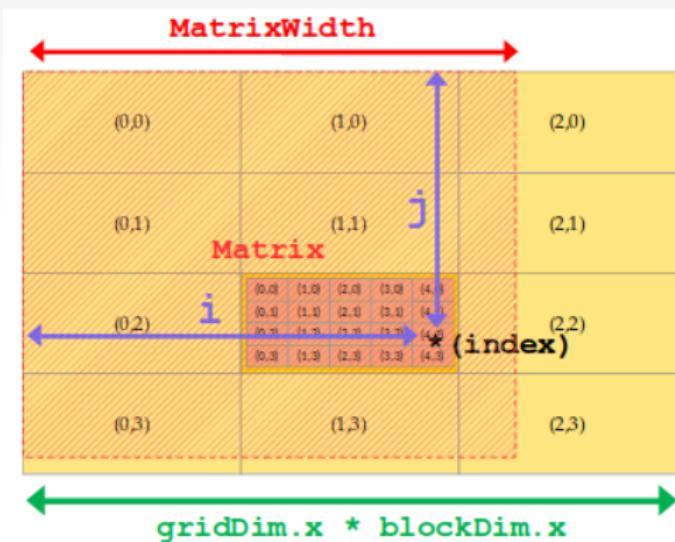


CUDA VectorAdd - 1D thread 1D block

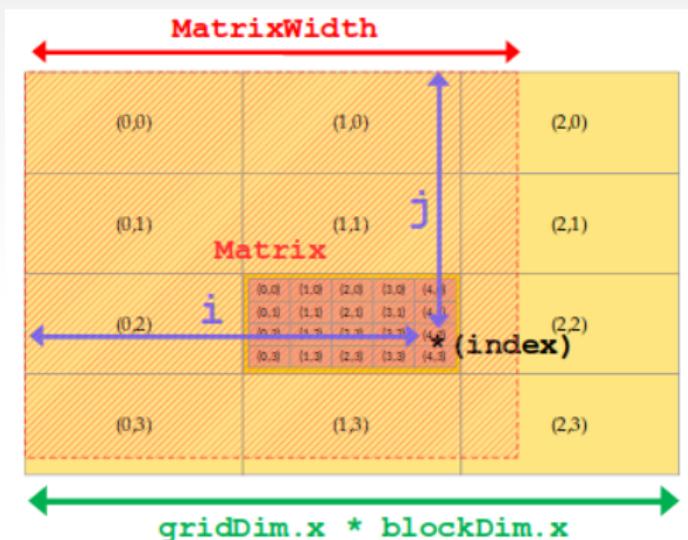
```
1 __global__ void gpuVectAdd(double *u, double *v, double *z, int N)
2 {
3     // define index
4     int i = blockIdx.x * blockDim.x + threadIdx.x;
5     // check that the thread is not out of the vector boundary
6     if (i >= N) return;
7     int index = i;
8     // write the operation for the sum of vectors
9     z[index] = u[index] + v[index];
10 }
11
12
13 int main(int argc, char *argv[]) {
14 ...
15     dim3 blockSize = 512;
16     dim3 gridSize((N-1)/blockSize.x + 1);
17
18     // define the execution configuration
19     gpuVectAdd<<<gridSize, blockSize>>>(u_dev, v_dev, z_dev, N);
20 ...
21 }
```



2D Thread Indexing



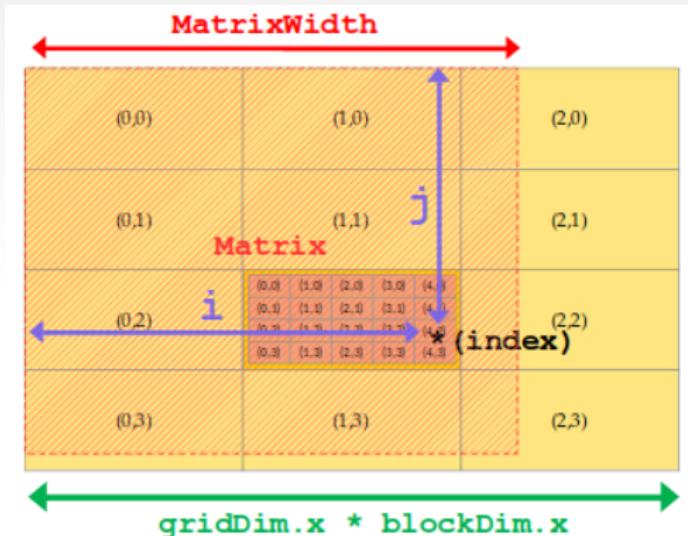
2D Thread Indexing



```
int i = blockIdx.x * blockDim.x + threadIdx.x;  
int j = blockIdx.y * blockDim.y + threadIdx.y;  
int index = j * MatrixWidth + i;
```



2D Thread Indexing



```
int i = blockIdx.x * blockDim.x + threadIdx.x;  
int j = blockIdx.y * blockDim.y + threadIdx.y;  
int index = j* gridDim.x * blockDim.x +i;
```



Matrix Add in 2D grids

```
1 __global__ void matrixAdd(int N, const float *A, const float *B, float *C) {
2     int i = blockIdx.x * blockDim.x + threadIdx.x;
3     int j = blockIdx.y * blockDim.y + threadIdx.y;
4     // matrix elements are organized in row major order in memory
5     int index = j * N + i;
6     if ( i < N && j < N )
7         C[index] = A[index] + B[index];
8 }
9
10 int main(int argc, char *argv[]) {
11     ...
12     // use 2D block threads
13     dim3 blockSize(32,32);
14     // use 2D grid blocks
15     dim3 gridSize( (N-1)/block.x + 1, (N-1)/block.y + 1 );
16     // add NxN matrices on GPU
17     matrixAdd <<< gridSize, blockSize >>> (N, A, B, C);
18     ...
19 }
```



Mapping Parallel partds into kernels

Remember that:

Threads execute the same kernel on different data

Steps for Vector Add

- ① map the loop into a kernel function
- ② map CUDA threads onto unique index
- ③ let threads retrieve, compute and store its own data
- ④ check out of border access

```
1 const int N = 1000;  
2 double a[N], b[N], c[N];  
3 // z = u + v  
4 for (i=0; i<N; i++)  
5     c[i] = a[i] + b[i];
```



Steps for Vector Add

```
1 __global__ void gpuVectAdd (int N, const double *a, const double
    *b, double *c)
2 {
3     // index is a unique identifier for each GPU thread
4     int index = blockIdx * blockDim.x + threadIdx.x ;
5     if (index < N)
6         c[index] = a[index] + b[index];
7 }
```



Steps for Vector Add

- `--global--` identifies a Kernel.
- Kernel can be called only by the Host
- kernel can be called only by using execution configuration
- they must return `void`
- They are asynchronous call (they return immediately)
- explicit control is needed to check if a kernel completed its execution

```
1  __global__ void gpuVectAdd (int N, const double *a,
2                                const double *b, double *c)
3  {
4      // index is a unique identifier for each GPU thread
5      int index = blockIdx * blockDim.x + threadIdx.x ;
6      if (index < N)
7          c[index] = a[index] + b[index];
}
```



Memory Allocation ON GPU

```
cudaMalloc(void **bufferPtr, size_t n  
cudaFree(void * bufferPtr)
```

Notice the type of bufferPtr...

```
1 double *u_dev;  
2 cudaMalloc( (void**)&uDev , N*sizeof(double));
```

u_dev

is defined on the host memory
contains an address of the device memory
C semantics of passing parameters is *by copy*
we need to pass u_dev by reference the cast is to force cudaMalloc
to handle pointer to memory of any kind



Memory Allocation of Vector Add on GPU

```
1 double *u_dev, *v_dev, *z_dev;  
2  
3 cudaMalloc((void **)&u_dev, N * sizeof(double));  
4 cudaMalloc((void **)&v_dev, N * sizeof(double));  
5 cudaMalloc((void **)&z_dev, N * sizeof(double));
```



Memory Init ON GPU

*cudaMemset(void * devPtr, intval, size_tcount*



Memory Copy ON GPU

*cudaMemset(void * dst, void * stc, size_tsize, direction*

direction:

- H2D: cudaMemcpyHostToDevice
- D2H: cudaMemcpyDeviceToHost
- : on the same GPU: cudaMemcpyDeviceToDevice

Copy

The copy begins only after all previous kernel have finished.

The copy is **blocking**

*cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);*



CUDA Unified Virtual Addressing (UVA)

- CUDA 4.0 introduces the UVA
- the system keeps track of the buffer location
- the only transfer mode is: *cudaMemcpyDefault*

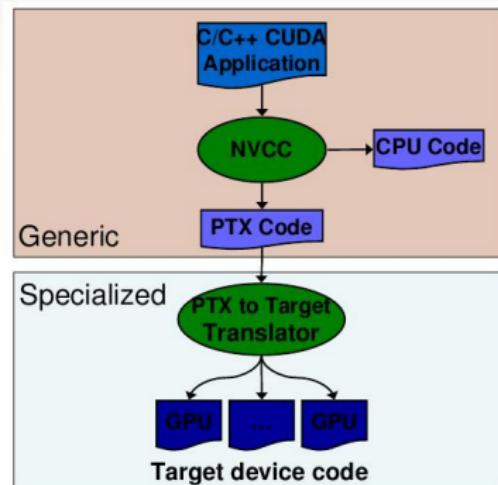


VectorAdd complete CUDA code

Click Here!



Cuda Compiling Process



- Two Steps (like for all compiling processes): Front-End (*Generic*) and a Back-End(*Special*).
- Compilation with a CUDA-aware compiler (e.g. *nvcc*)
- Front-End separates CPU and Device code.
- Device Code is provided by intermediate language (PTX)
- Back-End generates special code (*cubin* binary objects) for specific GPU architectures (both PTC and object code)



Compile with Targets (PTX)

compute capability and cubin architecture for K80

$nvcc - arch = compute_37 - code = sm_37$

other tools

- *deviceQuery* : show information on CUDA devices
- *nvidia – smi*: The Nvidia System Management Interface shows diagnostics on CUDA devices



Error Handling

cudaError_t

All CUDA API returns an error code of this type.

cudaSuccess means no error

Printing CUDA Errors

```
1 cudaError_t cerr = cudaMalloc(&d_a, size);  
2 if (cerr != cudaSuccess)  
3     fprintf(stderr, "%s\n", cudaGetErrorString(cerr));
```



Error Handling

- CUDA runtime has a function that returns the status of internal error variable (*cudaGetLastError*)
- the call resets the internal error to *cudaSuccess*
- to check error status, use *cudaDeviceSynchronize()*, but only at the end of a Kernel execution

```
1 // reset internal state
2 cudaError_t cerr = cudaGetLastError();
3 // launch kernel
4 kernelGPU<<<dimGrid,dimBlock>>>(...);
5 cudaDeviceSynchronize();
6 cerr = cudaGetLastError();
7 if (cerr != cudaSuccess)
8   fprintf(stderr, "%s\n", cudaGetErrorString(cerr));
```



Error Handling

useful macro

```
1 #define CUDA_CHECK(X) {\  
2     cudaError_t _m_cudaStat = X; \  
3     if(cudaSuccess != _m_cudaStat) {\  
4         fprintf(stderr, "\nCUDA_ERROR: %s in file %s line %d\n", \  
5             cudaGetErrorString(_m_cudaStat), __FILE__, __LINE__); \  
6         exit(1); \  
7     } } \  
8 ... \  
9 CUDA_CHECK( cudaMemcpy(d_buf, h_buf, bufferSize, \  
10             cudaMemcpyHostToDevice) );
```



Cuda Events

- Special variables used to *mark* code
- Can be used to:
 - get time
 - identify synchronization point between GPU and CPU execution flow (e.g: can block CPU until GPU ends a kernel)



Events for getting Time

```
1 cudaEvent_t start, stop;
2 cudaEventCreate(&start);
3 cudaEventCreate(&stop);
4 cudaEventRecord(start);
5 ...
6 kernel<<<grid, block>>>(...);
7 ...
8 cudaEventRecord(stop);
9 cudaEventSynchronize(stop);
10 float elapsed;
11 // execution time between events in ms
12 cudaEventElapsedTime(&elapsed, start, stop);
13 cudaEventDestroy(start);
14 cudaEventDestroy(stop);
```



Measures

Measures

Measuring Time is Ok, but for CUDA Applications it is important the evaluation of memory bandwidth. Memory Transfers require bus operations (PCIe, NVLinx) and it is limited by BUS bandwidth.

Best Practice

The best is minimize the number of transfers ... But You may need to transfer lot of data (e.g. if device memory do not fit required data).



Measures

CUDA SDK tool to measure bandwidth

```
1 ./bandwidthTest --mode=range --start=<B> --end=<B> --increment=<B>
```

Mflops

Count the number of fp operations; Divide for 1M (MFlops)
Measure the time (in seconds)
divide MFlops/time(s)



Example

Matrix dot Matrix on HOST and DEVICE

Measure performances

Try with different grid and block sizes

Limits !!!

There is a limit on the max number of allowed threads per block
(depends on compute capability)



Select proper sizes

- Respect limits for threads per block
- select block grid to cover all elements to process
- select block size in order to avoid race conditions among threads
- use builtin vars (blockIdx etc.) to identify data parts that threads have to process



Resources

- CUDA kernel requires resources to run
- when you assign blocks to SM, registers are assigned to each thread block, depending on execution configuration
- Resources assigned to threads are freed at the end of the block execution
- threads have private registers (no overload for context switching among warps)



Example

Setup

32768 regs per SM; Kernel grud with 32x8 thread blocks; the kernel needs 30 registers

How many Thread Blocks can be hosted on a single SM

?



Example

Setup

32768 regs per SM; Kernel grid with 32x8 thread blocks; the kernel needs 30 registers

How many Thread Blocks can be hosted on a single SM

each block requires: 30x32x8 regs (7680)

$32768/7680 = 4$ Block + a little bit ...

you can host ONLY 4 blocks (on an architecture ... fermi ... that has 8 blocks)!

What if you need 33 regs

?



Example

Setup

32768 regs per SM; Kernel grid with 32x8 thread blocks; the kernel needs 33 registers

How many Thread Blocks can be hosted on a single SM

each block requires: 33x32x8 regs (8448)

$32768/8448 = 3$ Block + a little bit ...

you can host ONLY 3 blocks (on an architecture ... fermi ... that has 8 blocks)!

We have a 25% loss in potential parallelism



Best Allocation for MxM

choose best *TILE_WIDTH* (block size)

on Fermi architecture each SM can handle up to 1536 threads

TILE_WIDTH = 8

$8 \times 8 = 64$ threads - $\therefore 1536/64 = 24$ blocks in a SM

... consider cc too: in cc 2.x there is a limit of max 8 resident blocks 8 (24 ARE TOO MUCH)

Real number of threads per SM is $64 \times 8 = 512$ on a maximum of 1536 (33% occupancy)

TILE_WIDTH = 16

$16 \times 16 = 256$ threads - $\therefore 1536/256 = 6$ blocks in a SM Real number of threads per SM is $6 \times 256 = 1536$ on a maximum of 1536 (100% occupancy)



Best Allocation for MxM

choose best *TILE_WIDTH* (block size)

on Fermi architecture each SM can handle up to 1536 threads

TILE_WIDTH = 32

$32 \times 32 = 1024$ threads - $1536/1024 = 1.5$ blocks in a SM 1024 threads per SM on a maximum of 1536 (66% occupancy)

The Winner is *TILE_WIDTH* = 16



Changing Compute Capability

choose best *TILE_WIDTH* (block size)

on Kepler architecture each SM can handle up to 2048 threads
(and a maximum of 16 resident blocks per SM)



Changing Compute Capability

choose best *TILE_WIDTH* (block size)

on Kepler architecture each SM can handle up to 2048 threads
(and a maximum of 16 resident blocks per SM)

TILE_WIDTH = 8

$8 \times 8 = 64$ threads - $\frac{2048}{64} = 32$ (16) blocks in a SM

$64 \times 16 = 1024$ threads x SM on a maximum of 2048 (50% occupancy)



Changing Compute Capability

choose best *TILE_WIDTH* (block size)

on Kepler architecture each SM can handle up to 2048 threads
(and a maximum of 16 resident blocks per SM)

TILE_WIDTH = 16

$16 \times 16 = 256$ threads - $\lceil 2048 / 256 \rceil = 8$ blocks in a SM $8 \times 256 = 2048$
threads x SM on a maximum of 2048 (100% occupancy)



Changing Compute Capability

choose best *TILE_WIDTH* (block size)

on Kepler architecture each SM can handle up to 2048 threads
(and a maximum of 16 resident blocks per SM)

TILE_WIDTH = 32

$32 \times 32 = 1024$ threads - $\lceil 2048/1024 \rceil = 2$ blocks in a SM
 $2 \times 1024 = 2048$ threads x SM on a maximum of 2048 (100% occupancy)

The Winners are *TILE_WIDTH* = 16, 32



MxM Errors and Performances

Add Errors and Performance Management on MxM by using different kernel parameters

CUDA GetLastError

```
1 mycudaerror=cudaGetLastError() ;  
2 <Call KERNEL>  
3 cudaDeviceSynchronize() ;  
4 mycudaerror=cudaGetLastError() ;  
5 if(mycudaerror != cudaSuccess)  
6   fprintf(stderr,"%s\n", cudaGetErrorString(mycudaerror)) ;
```

Events

Puts events in the right places, recording start and stop events; use *cudaEventSynchronize* to sync HOST and DEV; measure time with *cudaEventElapsedTime* and destroy events (*cudaEventDestroy*)

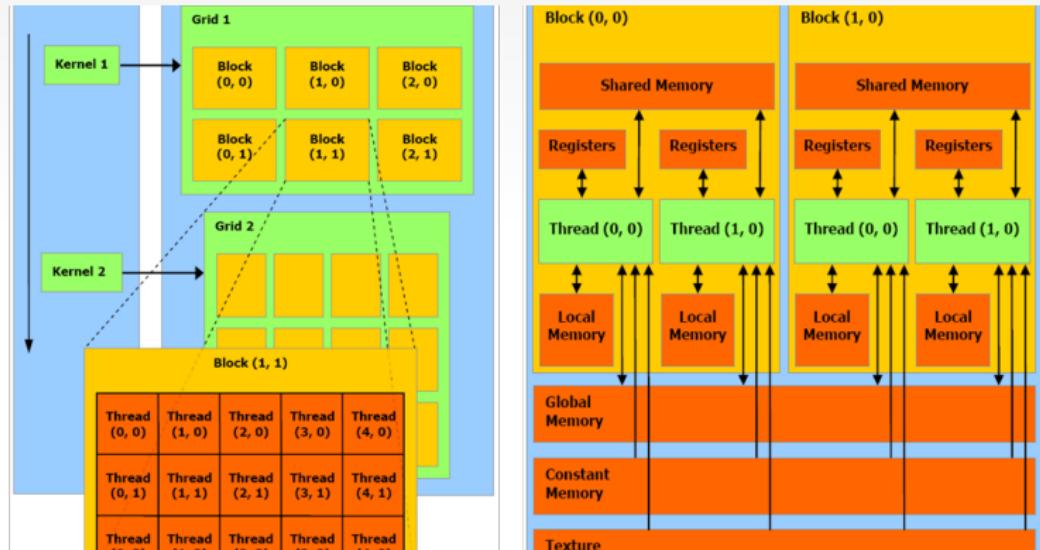


Example

Matrix dot Matrix on HOST and DEVICE
Here!



GPGPU Memory



Memory Hierarchy

Resources assigned to Blocks:

- Registers
- **Shared Memory**

Memory available on GPU (these have persistent storage duration)

- Global
- **Constant**
- **Texture**



Global Memory

- the largest memory on device (similar to CPU RAM)
- kernel executions do not reset this memory
- r/w from all threads
- Very High Bandwidth (up to 760 GB/s ... and ever increasing ...)
- Very High Latency (about 400-800 clock cycles)



Manage Global Memory

```
1 __device__ type variable_name; // static
2
3 // dynamic (malloc) allocation
4 type *pointer_to_variable;
5 cudaMalloc((void **) &pointer_to_variable, size);
6 cudaFree(pointer_to_variable);
```



Caches for Global Memory

- From Fermi TESLA have L1 and L2 Caches *for each block*
- L2 is shared among SM (Fermi has 768K, Kepler 1536K, Pascal 4M etc.)
- 25% less latency than Global Memory
- all accesses to Global Memory pass through L2 cache (H2D and D2H transfers too)
- L1 is private to each SM
- L1 size: 16 to 48K configurable (L1+Shared Memory = 64 KB on fermi, 32 KB on Kepler/Pascal)

Setting Cache

```
1 cudaFuncSetCacheConfig(kernel1, cudaFuncCachePreferL1); // 48KB L1 / 16KB ShMem  
2 cudaFuncSetCacheConfig(kernel2, cudaFuncCachePreferShared); // 16KB L1 / 48KB ShMem
```



Load And Stores on Caches

Two Load Configuration

- With Cache (default): L1- \rightarrow L2- \rightarrow Global Memory. The cache line length is 128 Byte
- Without cache (compile with $Xptxas -dlcm = cg$): L1 is disabled, L2 is active and all transfers use it . The L2 cache line is 32-byte

Store

When a Store occurs, L1 is invalidated and L2 is updated



Strides and Offsets

Two kinds of Memory access for transfers

Stride-based transfer jumps stride by stride

Offset-based transfer displaces by offset

Stride copy

```
1 __global__ void strideCopy (float *odata, float* idata, int stride) {  
2     int xid = (blockIdx.x*blockDim.x + threadIdx.x) * stride;  
3     odata[xid] = idata[xid];  
4 }
```

Offset copy

```
1 __global__ void offsetCopy(float *odata, float* idata, int offset) {  
2     int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;  
3     odata[xid] = idata[xid];  
4 }
```



Exercise

Evaluate Bandwidth for different Strides (1,2,8,16,32) and different Offsets (0,1,8,16,32)



Accesses to Global Memory

- Accesses are per-warps
- threads in a warp compute the address to access
- load/store units evaluate position of memory segments
- load/store units perform the transfer



Same Examples

32 consecutive 4-byte word (128 bytes)

Caching Load	Non-caching Load
addresses fall within 1 cache line	addresses fall within 4 cache segments
128 bytes are moved across the bus	128 bytes are moved across the bus
bus utilization: 100%	bus utilization: 100%
 Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448	 Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448

32 permuted 4-byte word (128 bytes) aligned to a segment

Caching Load	Non-caching Load
addresses fall within 1 cache line	addresses fall within 4 cache segments
128 bytes are moved across the bus	128 bytes are moved across the bus
bus utilization: 100%	bus utilization: 100%
 Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448	 Memory addresses: 0 32 64 96 128 160 192 224 256 288 320 352 384 416 448



Same Examples

32 consecutive 4-byte word (128 bytes) not-aligned to a segment

Caching Load	Non-caching Load
addresses fall within 2 cache lines	addresses fall within at most 5 segments
256 bytes are moved across the bus	256 bytes are moved across the bus
bus utilization: 50%	bus utilization: at least 80%
 addresses from a warp	 addresses from a warp

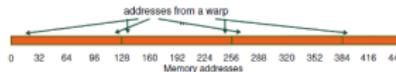
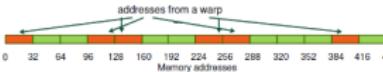
all threads in a warp request the same 4bytes

Caching Load	Non-caching Load
addresses fall within a single cache line	addresses fall within a single segment
128 bytes are moved across the bus	32 bytes are moved over the bus
bus utilization: 3.125%	bus utilization: 12.5%
 addresses from a warp	 addresses from a warp



Same Examples

32 non-contiguous 4-byte word (128 bytes)

Caching Load	Non-caching Load
addresses fall within N different cache lines	addresses fall within N different segments
$N \times 128$ bytes are moved across the bus	$N \times 32$ bytes are moved across the bus
bus utilization: $128 / (N \times 128)$	bus utilization: $128 / (N \times 32)$
	

Data Alignment in Global Memory

- The Goal is to have aligned access (*coalesced*)
- *cudaMalloc* aligns the first element -*↳* linear arrays
- *cudaMallocPitch* is good for 2D buffers (elements are padded so each row is aligned); it returns an int (*pitch*) that can be used to access to row elements (with a stride)



Data Alignment in Global Memory

```
1 // on host
2 int width = 64, height = 64;
3 float *devPtr;
4 int pitch;
5 cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);
6
7 // on device
8 __global__ myKernel(float *devPtr, int pitch, int width, int height)
9 {
10    for (int r = 0; r < height; r++) {
11        float *row = devPtr + r * pitch;
12        for (int c = 0; c < width; c++)
13            float element = row[c];
14    }
15    ...
16 }
```



Shared Memory

- small
- shared in the block (r/w)
- reset at the end of a kernel execution
- very low latency (2 clocks px cycle)
- Throughput: 32 bit / 2 cycles
- 32, 48 Kbyte (configurable)
- synchronization is needed



Shared Memory Allocation

```
1 // statically inside the kernel
2 __global__ myKernelOnGPU (...) {
3     ...
4     __shared__ type shmem[MEMSZ];
5     ...
6 }
7 // dynamic allocation
8 // dynamically sized
9 extern __shared__ type *dynshmem;
10 __global__ myKernelOnGPU (...) {
11     ...
12     dynshmem[i] = ... ;
13     ...
14 }
15 void myHostFunction() {
16     ...
17     myKernelOnGPU<<<gs,bs,MEMSZ>>>();
18 }
```



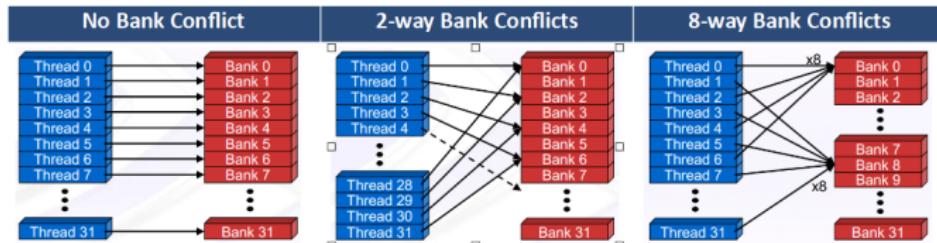
Threads (in a Block) Synchronization

- `--syncthreads()`: like a barrier
- if you use it in a branch, you have to be sure that all threads will choose the *same* branch.



Shared Memory Banks

- 32 banks (32 bit words map a bank)
- data are distributed cycling by 4-bytes in successive banks
- access is *per-warp*
- **Multicast**: if N threads in the warp access the same element, the access is executed with only one transfer
- **Broadcast**: if ALL threads in the warp access the same element, the access is executed with only one transfer
- **Bank Conflict**: if 2 or more threads request *different* data in the *same* bank, accesses are serialized . (Accesses on the same column of a *naive allocated* matrix generates a bank conflict)



Solving Bank Conflict Problem in 2D arrays

- a shared memory of 32x32 floats has each element on a single bank (4 bytes)
- read/write on the same column has the worst type of conflict
- The solution is allocating 33 elements with padding

```
1 __shared__ float tile[TILE_DIM][TILE_DIM+1];
```



Constant Memory

- read-only (global)
- very efficient when all threads in a warp request the same memory address
- 64KB
- Throughput: 32 bit per warp every 2 cycles



Global vs Constant Memory

Suppose you have 320 warps per SM with all threads requesting the same data

Global Memory

All warps request the same data; The first access moves the data in L2; if other data pass in L2, the requested variable can be lost; other threads will request the data with an high probability

Constant Memory

Data is copied in constant memory. In Constant Cache there are less requests, so we will probably not loose the data in the cache due to other reads (this results in a low traffic on the bus too).



Constant Memory Allocation

```
1 __constant__ type variable_name; // static
2
3 cudaMemcpyToSymbol(const_mem, &host_src, sizeof(type),
4                     cudaMemcpyHostToDevice);
5 // NO dynamic allocation
```



Texture Memory

- Basic rendering functionalities
- it is a global memory fetched across a dedicated texture-cache
- read-only memory accessed by *texture fetch* API
- address resolution is performed by dedicated HW
- specialized for out-of-bound address resolution, Floating Point interpolation, type conversion or bit operations.



How To Manage Texture Memory

CPU

- ① allocate global memory on device (standard, pitched or `cudaArray`) with `cudaMalloc(&M, memsize)`
- ② Create a texture reference:
 $texture < datatype, dim > MtextureRef$. `datatype` cannot be a double, `dim` can be 1,2,3.
- ③ create a channel descriptor:
`cudaChannelFormatDescMdesc =`
`cudaCreateChannelDesc < datatype > ();`
- ④ bind the texture reference to memory:
`cudaBindTexture(0, MtextureRef, M, Mdesc);`
- ⑤ when finished, ,unbind:
`cudaUnbindTexture(MtextureRef);`



How To Manage Texture Memory

GPU

- ① Access data from CUDA kernels by using the texture reference
- ② for linear memory: `tex1Dfetch(MtextureRef, address)`
- ③ for pitched linear textures and `cudaArray`: `tex1d()`, `tex2D()`,
`tex3D()`



Texture Memory

Example

```
1 __global__ void shiftCopy(int N, int shift, float *odata, float *idata)
2 {
3     int xid = blockIdx.x * blockDim.x + threadIdx.x;
4     odata[xid] = idata[xid+shift];
5 }
6
7 texture<float, 1> texRef; // TEXTURE creation
8
9 __global__ void textureShiftCopy(int N, int shift, float *odata)
10 {
11     int xid = blockIdx.x * blockDim.x + threadIdx.x;
12     odata[xid] = tex1Dfetch(texRef, xid+shift); // TEXTURE FETCHING
13 }
14
15 ...
16
17 ShiftCopy<<<nBlocks, NUM_THREADS>>>(N, shift, d_out, d_inp);
18
19 cudaChannelFormatDesc d_a_desc = cudaCreateChannelDesc<float>(); // CREATE DESC
20 cudaBindTexture(0, texRef, d_a, d_a_desc); // BIND TEXTURE MEMORY
21 textureShiftCopy<<<nBlocks, NUM_THREADS>>>(N, shift, d_out);
```



Texture Memory from Kepler and RO cache

From Kepler constant memory loads from global memory can pass through texture cache without explicit binding, and without limits on the max number of textures

```
1 __global__ void kernel_copy (float *odata, float *idata) {
2     int index = blockIdx.x * blockDim.x + threadIdx.x;
3     odata[index] = __ldg(idata[index]);
4 }
5
6 __global__ void kernel_copy (float *odata, const __restrict__ float *idata) {
7     int index = blockIdx.x * blockDim.x + threadIdx.x;
8     odata[index] = idata[index];
9 }
```



Registers

- used to store scalars frequently accessed by each thread
(Fermi has 63 register per thread/32 KB, Kepler and Pascal 255 / 64KB, ...)
- the less registers a kernel needs, the more blocks can be in a SM!
- you can limit number of registers at compile time
 $(--maxregcount)$
- number of active blocks per kernel can be forced

```
1 __global__ void
2 __launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor)
3 my_kernel ...
```



Local Memory

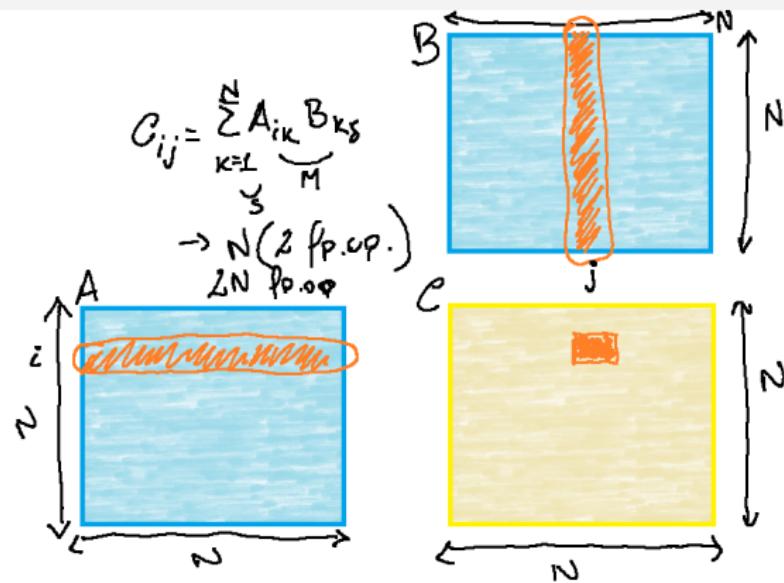
- Is not a "physical" memory
- automatic variables are placed here by compilers
- large structures or arrays that would consume too much registers are allocated here
- if a kernel needs more registers than available, can use this memory as swap
- usually mapped onto global memory

Getting infos about memories and registers

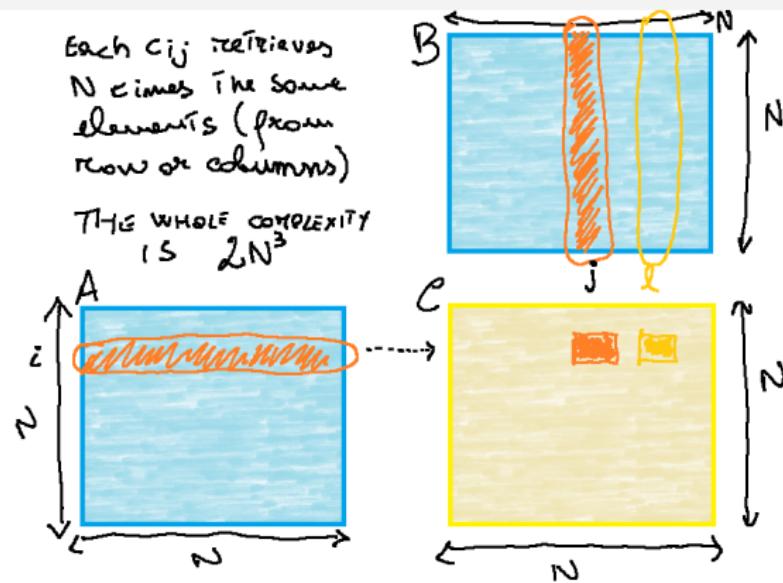
```
1 nvcc -arch=sm_XX -ptxas-options=-v kernel.cu
```



Matrix dot Matrix and Memories



Matrix dot Matrix and Memories



Matrix Dot Matrix and Memories

Shared Memory

We can avoid repetitions in fetching the same element by using the *shared memory*:

Threads can retrieve one data element in parallel and store it into the shared memory. All threads can access all the elements sharing a full row or column

But...

The shared memory is **small** (16/48 KByte)



Matrix dot Matrix with Shared Memory

Use Blocks

A solution consists in using $Nb \times Nb$ blocks. Threads blocks participate in computing elements of sub-matrices.

The final result is obtained as combination of results in all sub-matrices.



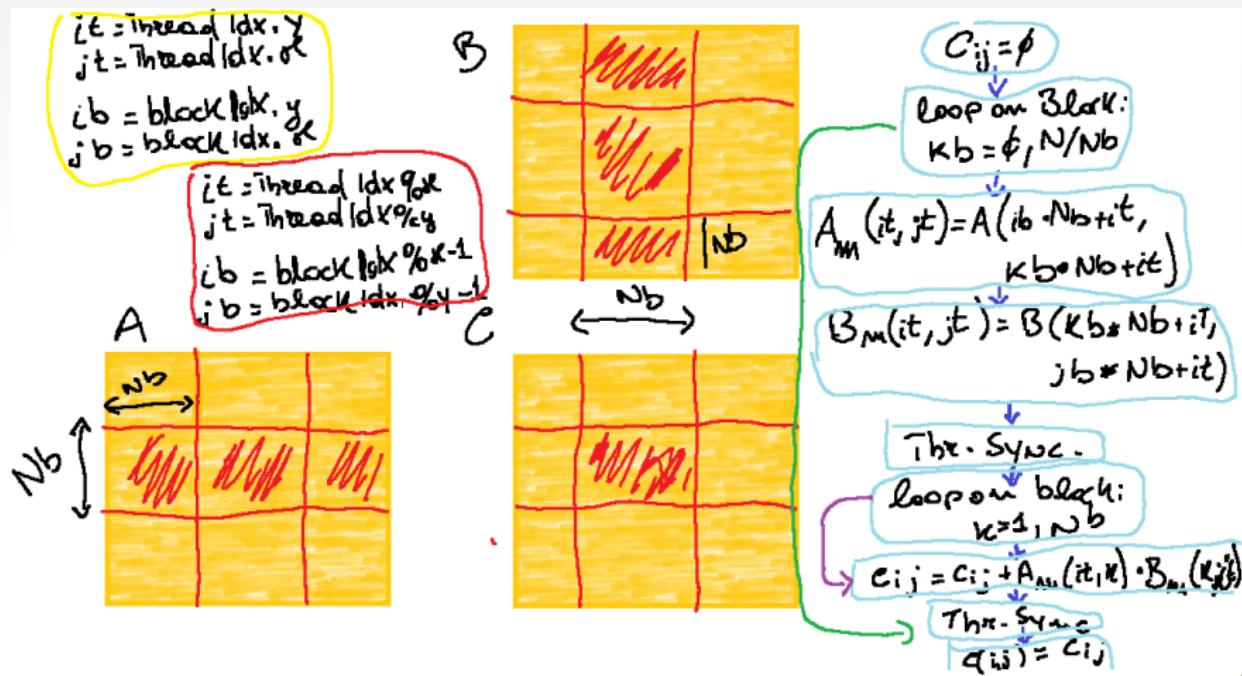
Matrix dot Matrix with Shared Memory

It is divided into two phases:

- ① Threads load a block ($Nb \times Nb$) of both A and B from Global memory and they save them in shared memory
- ② elements in the same sub-block are accumulated used local variables in registers and stored in global memory
- ③ Threads are synchronized after a load of sub-block of matrices and after the evaluation of partial sub-blocks products. This in order to grant that the next load of other sub-block do not overwrite elements not yet used in current block evaluation.



Matrix dot Matrix with Shared Memory



Matrix dot Matrix with Shared Memory

Click Here!



Matrix dot Matrix with Shared Texture Memory

Click Here!



Blocking or Non Blocking

Blocking (Synchronous)

- returns to host when execution on device ends
- all memory transfer \geq 64KB
- all memory allocation on device
- allocation of page locked memory on host

Non Blocking (Asynchronous)

- return control to host immediately
- kernel launches
- memory transfer \geq 64KB
- memory init on device (cudaMemset)
- memory copies from device to device
- explicit async memory transfers

Async functions allow for CPUs and Device overlapping

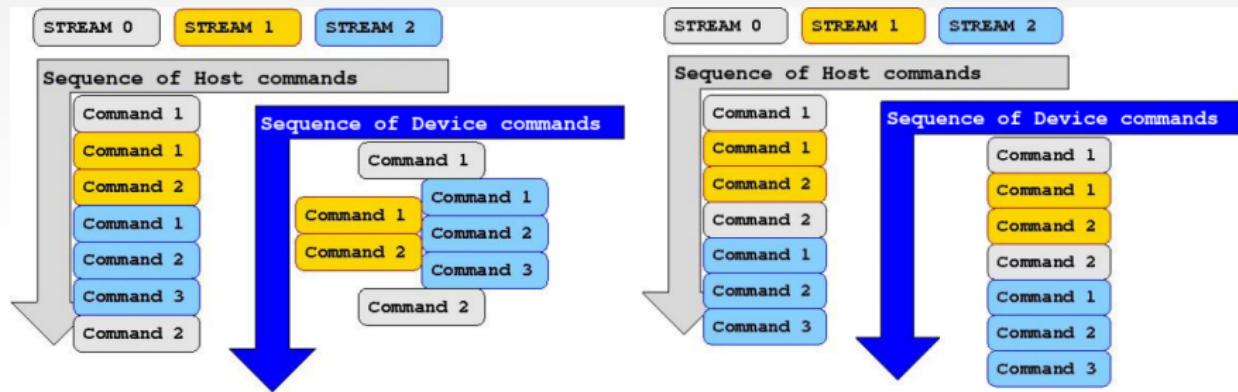


CUDA Streams

- GPU operations on CUDA use execution queues, called streams
- Operations pushed in a stream will be executed only after all other operations in the same stream are completed (FIFO queue behavior)
- Operations assigned to different streams can be executed in any order
- CUDA runtime provides a default stream (stream 0)
- operations assigned to the default stream are executed only after all preceding operations assigned to other streams are completed
- any further operation assigned to stream different from default, will start only after all operations on the default stream are completed
- operations assigned to default stream introduce implicit synchronization barriers among other streams.



CUDA Streams



Synchronization

- Explicit Synchronization
 - `cudaDeviceSynchronize()`: blocks host code until all operations on device are completed
 - `cudaStreamSynchronize(stream)`: blocks host code until all operations on a stream are completed
 - `cudaStreamWaitEvent(stream, event)`: blocks all operations assigned to a stream until event is reached
- Implicit Synchronization
 - All operations assigned to the default stream
 - Page-locked memory allocations
 - Memory allocations on device
 - Settings operations on device
 - ...



CUDA Streams Management

- `cudaStreamCreate()`
- `cudaStreamSynchronize()`
- `cudaStreamDestroy()`
- concurrent execution of more than one kernel per GPU
- concurrent asynchronous data transfers (H2D and D2H)
- concurrent execution on device/host and data transfers from host and device



Kernels Concurrent Execution

```
1 cudaStreamCreate(stream1)
2 cudaStreamCreate(stream2)
3
4 // concurrent execution of the same kernel
5 Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp_1, out_1)
6 Kernel_1<<<blocks, threads, SharedMem, stream2>>>(inp_2, out_2)
7
8 // concurrent execution of different kernels
9 Kernel_1<<<blocks, threads, SharedMem, stream1>>>(inp, out_1)
10 Kernel_2<<<blocks, threads, SharedMem, stream2>>>(inp, out_2)
11
12 cudaStreamDestroy(stream1)
13 cudaStreamDestroy(stream2)
```



Async Data Transfers

- *cudaMallocHost()*
- *cudaFreeHost()*
- *cudaHostRegister()*
- *cudaHostUnregister()*
- *cudaMemcpyAsync()*
- Transfers must be queued into a stream different from the default to be async
- Transfers use page-locked memory increasing bandwidth



Async Data Transfers

```
1 cudaStreamCreate(stream_a)
2 cudaStreamCreate(stream_b)
3
4 cudaMemcpyAsync(d_buffer_a, h_buffer_a, buffer_a_size, cudaMemcpyHostToDevice, stream_a)
5 cudaMemcpyAsync(h_buffer_b, d_buffer_b, buffer_b_size, cudaMemcpyDeviceToHost, stream_b)
6
7 cudaFreeHost(h_buffer_a)
8 cudaFreeHost(h_buffer_b)
```

```
9 // concurrent and asynchronous data transfer H2D and D2H
```

```
10 cudaMemcpyAsync(d_buffer_a, h_buffer_a, buffer_a_size, cudaMemcpyHostToDevice, stream_a)
11 cudaMemcpyAsync(h_buffer_b, d_buffer_b, buffer_b_size, cudaMemcpyDeviceToHost, stream_b)
```

```
12
```

```
13 cudaStreamDestroy(stream_a)
14 cudaStreamDestroy(stream_b)
```

```
15
```

```
16 cudaFreeHost(h_buffer_a)
17 cudaFreeHost(h_buffer_b)
```



Async Data Transfers

```
1 cudaStream_t stream[4];
2 for (int i=0; i<4; ++i) cudaStreamCreate(&stream[i]);
3 float* hPtr; cudaMallocHost((void**)&hPtr, 4 * size);
4 for (int i=0; i<4; ++i) {
5     cudaMemcpyAsync(d_inp + i*size, hPtr + i*size,
6                     size, cudaMemcpyHostToDevice, stream[i]);
7     MyKernel<<<100, 512, 0, stream[i]>>>(d_out+i*size, d_inp+i*size, size);
8     cudaMemcpyAsync(hPtr + i*size, d_out + i*size,
9                     size, cudaMemcpyDeviceToHost, stream[i]);
10 }
11 cudaDeviceSynchronize();
12 for (int i=0; i<4; ++i) cudaStreamDestroy(&stream[i]);
```

Sequential Version



Asynchronous Versions



Any Question ?



¹image from: <https://pigswithcrayons.com/illustration/dd-players-strategy-guide-illustrations/>

