Parallel Programming for High-Performance Computing

Nicolas Maillard
Inf01182

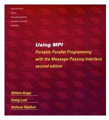
Let's talk!

Moodle

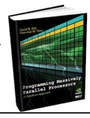
- Register in the course
  - http://moodle.inf.ufrgs.br/course/view.php?id=388
- Password: cuda1002
- Provides access to the schedule of the classes, hand-outs, links, upload engine for your (future) slides…

Basic Bibliography

- OpenMP
  - *Parallel Programming in OpenMP*. R. Chandra et al., Morgan Kaufmann, 2001.
  - *Using OpenMP - Portable Shared Memory Parallel Programming*, Barbara Chapman, Gabriele Jost and Ruud van der Pas



- Message Passing Interface
  - Gropp, William et al., *Using MPI*, MIT Press.
  - Gropp, William et al., *Using MPI-2*, MIT Press.
  - Snir, M. et al., Dongarra, J., *MPI: The Complete Reference*.
  - http://www.mpi-forum.org

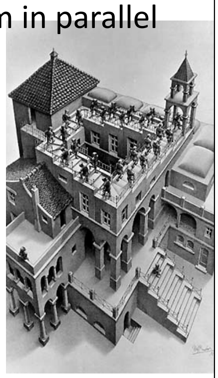


- CUDA
  - *Programming Massively Parallel Processors: A Hands-on Approach*. David B. Kirk and Wen-mei W. Hwu. Morgan Kaufmann 2010.
  - http://www.nvidia.com/object/cuda_home_new.html

Evaluation

1. Choose a classmate
2. Pick-up a problem, list algorithms to solve it.
3. Among the algorithms, choose the most parallel one.
4. Implement it.
5. Measure its performance.
6. Present 2-3-4-5 and discuss your choices/results.
   - You will have 20 minutes.

3 Reasons to program in parallel



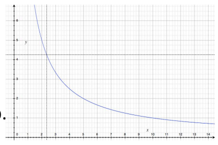
1. (You will practice your English.)
2. Parallelism is everywhere today.
   - From Multi-core chips to Cloud Computing.
3. If you want to be performant, you have to understand in depth what is going on.

Having an objective function in mind (performance), you will design/program better

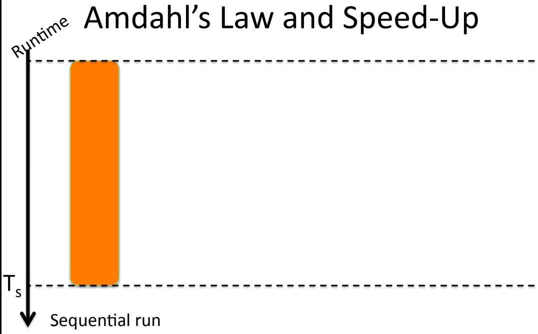
- How much faster does your code run?

Define "performance"

- Ideally, the parallel runtime T_p is iqual to the sequential runtime T_s divided by the number of "processors" p.
  - $T_p(p) = T_s / p$.
- Speed-up: $S(p) = T_s / T_p(p)$
  - Ideally S(p) is iqual to p.
  - In practice it is smaller than p.
- Efficiency: $e(p) = S(p) / p$
  - It is the percentage of usage of the p CPUs.
  - e(p) = 1 (100%) is ideal, in practice e(p) < 1.

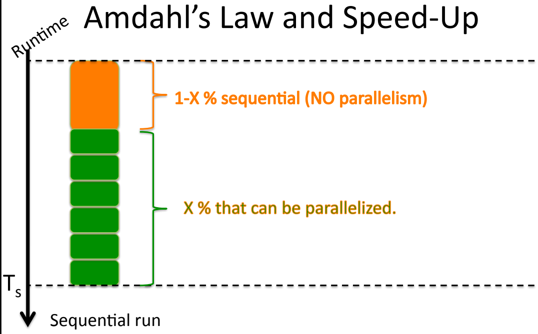


Amdahl's Law and Speed-Up



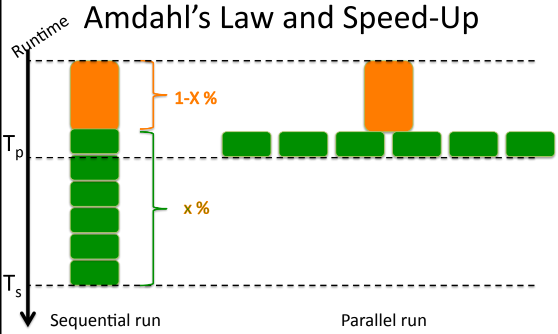
T_s

Amdahl's Law and Speed-Up



$T_s = (1-x)T_s + xT_s$

Amdahl's Law and Speed-Up

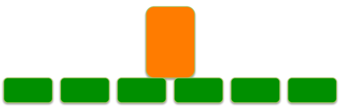


$T_s = (1-x)T_s + xT_s$

$T_p = (1-x)T_s + xT_s/p$

Amdahl's Law and Speed-Up

Speed-Up:



$S(p) = T_s / T_p$

$= 1 / (1-x+x/p)$

When p increases, the speed-up tends to 1/(1-x)

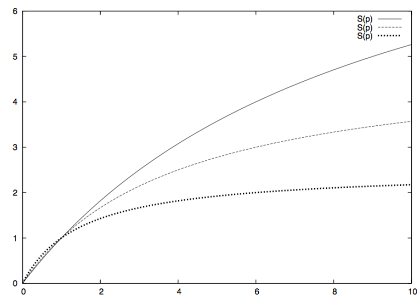
Sequential run

$T_s = xT_s + (1-x)T_s$

Parallel run

$T_p = xT_s + (1-x)T_s/p$

Speedup-up for x = 70%, 80%, and 90%

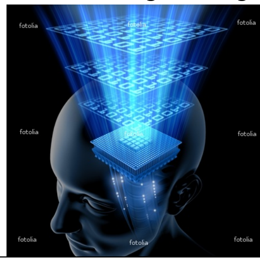


Amdahl's Law: moral

- Even a small non-parallel part of the program leads to a severe limitation in terms of speed-up.
  - Even with more CPUs, there is a limit in terms of acceleration.

How do you "think parallel"?

From "Computational thinking" to Parallel Programming.



How do you "think parallel"?

From "Computational thinking" to Parallel Programming.

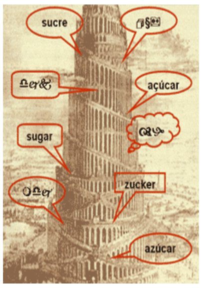
- You can send letters to friends to ask them for help.
- You can phone them to ask them for help.
- You can have a "To-do" list, pop tasks out of it when you are idle, ask for friends to do the same to help you...
  - Where do they put their results back?

What is required

- Define the "tasks" (what will remain sequential).
- Define the way the tasks interact:
  - How do they get their input?
  - How do they return their output?
  - How do they relate? Is there an order among the tasks?

2 options for Comp. Scientists

1. Invent a new language



2. Adapt the existing languages to enable parallelism.

Option 1 - Yet another language?

- (We will not do that in this course.)
- Nice... But is it worth the pain?
  - Who will use it?
  - Who will reprogram all the legacy code?

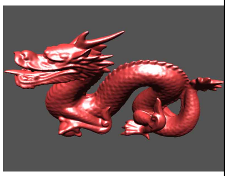
Option 2 - Adapt the existing languages...

4 approaches:

1. Let the compiler do the job!
   – Automatic parallelization,
   – Or annotate the sequential code.
2. Use OS resources
   – With shared memory – Threads
   – With networking (Message Passing) – SPMD.
3. Use the data to define the parallelism.
4. Provide a high-level abstraction – e.g. Object, First-Class function...
   – Ease the development;
   – How about performance?

Approach 1 – Use the compiler!

- Automatic Parallelization
  – It is very **hard** indeed!
- **Solution**: annotate the code to help the compiler extracting the parallelism.
  – OpenMP
  – Cilk
- Limitation: it works well (today) only for shared memory.

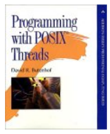


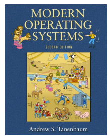
OpenMP basic example

Sequential code	Parallel Open-MP code
double res[10000];	double res[10000];
	#pragma omp parallel for
for (i=0 ; i<10000; i++) compute(res[i]);	for (i=0 ; i<10000; i++) compute(res[i]);

Approach 2 – use O.S. entities

- Extend existing sequential languages:
  – With thread libraries,
    - Define, create, sync threads.
    - Example: pthreads.
  – With message-passing libraries (SPMD)
    - Fork processes,
    - Open sockets,
    - Provide some quality of service guarantee...
    - Example: Message Passing Interface (MPI).





Example: MPI code

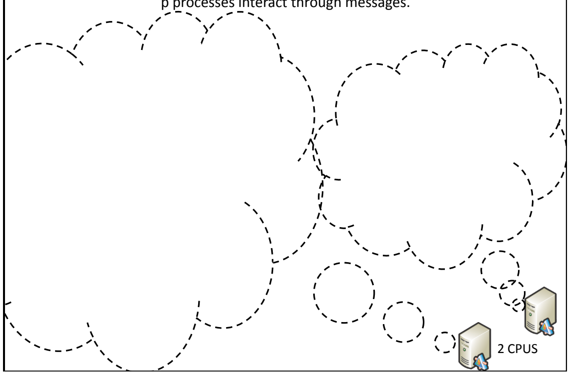
MPI defined datatype

Send/Receive

```
void  main() {
  int p, r, tag = 103;
  MPI_Status stat;
  double valor ;
  MPI_Init(&argc, &argv) ;
  MPI_Comm_rank(MPI_COMM_WORLD, &r);
  if (r==0) {
     printf("Processor 0 sends a message to 1\n");
     valor = 3.14 ;
     MPI_Send(&valor, 1, MPI_DOUBLE, 1, tag,
       MPI_COMM_WORLD);  }
  else {
     printf("Processor 1 receives a message from 0\n");
     MPI_Recv(&valor, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &stat);
     printf("O valor recebido vale %.2lf\n", valor);  }
}
```

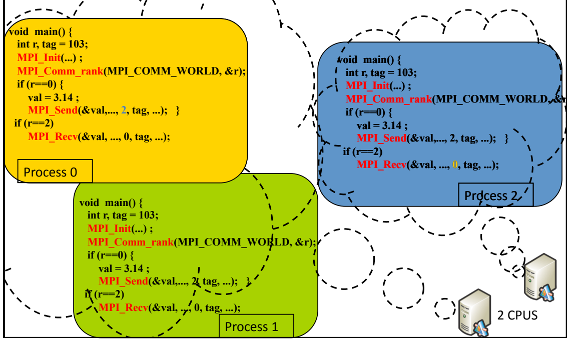
Programming with MPI

p processes interact through messages.



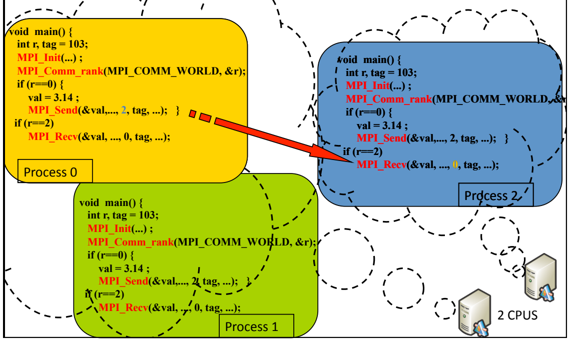
Programming with MPI

p processes interact through messages.



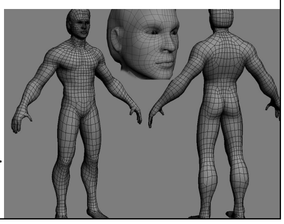
Programming with MPI

p processes interact through messages.



Approach 3 – Data Parallelism

- Simple idea:
  - Define data-structures (e.g arrays, trees...);
  - Decide which CPU stores each part of the data-structure;
  - Runs the computation in parallel on each distributed part.
    - Fine, if there are no complex communication.



Example: CUDA Code

```
void MatrixMul(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;  // Vectors that will be processed in parallel.
  dim3 dimGrid(1, 1);
  dim3 dimBlock(Width, Width);
  // Call a function to be run by each CUDA thread on its own
  // piece of data:
  MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd,
   Width);
}

__global__ void MatrixMulKernel(float* Md, float* Nd, float*
  Pd, int Width) {…;}
```

Approach 4 – High Level Abstractions

- Use virtual machines to abstract all the architecture details,
- Use Objects to hide the data and provide clean interfaces
  - E.g. RMI
- Use Web Services to publish and discover the services.
- You obtain a Cloud
  - (yet another modern avatar of an old story....)
- Very nice abstraction. How about performance?

Improving on what already exists

- Loop parallelism is very simple.
- Data parallelism also.
- SPMD programming is not that complicated.
- So what is wrong with that?

The Dark Side of the Parallel Programer

- When you design an algorithm, or when you program it for a sequential CPU, you abstract the architecture / the OS.
  - See Yale Patt.
- Is there no way to program in parallel, independently of the number of processors, the network or the shared memory,...?

Episode II

- Loop parallelism with OpenMP...

SOON TO COME!