Dr. Xavier Buffat
Beams Department – Accelerator and Beam Physics
Hadron Synchrotron Collective effects
CERN, Switzerland, Geneva
November 2018
After this lecture you may:

- Know the basic principles of common computer architectures
  → Determine the type of computer architecture that best fits your problem
- Start writing code that uses efficiently your computer
  → Know the jargon to make your way through the doc on the web...
Content

- The Turing machine
- The Central Processing Unit (CPU)
  - Intrinsics, vectorisation
- Multiple CPUs
  - Multithreading
  - NUMA zones
  - Hyperthreading
- Amdhal's law and Moore's Law
- The Graphics Processing Unit (GPU)
- Computer clusters
  - The Message Passing Interface
- Volunteer and grid computing
- Summary
Content

- The Turing machine
- The Central Processing Unit (CPU)
  - Intrinsics, vectorisation
- Multiple CPUs
  - Multithreading
  - NUMA zones
  - Hyperthreading
- Amdhal's law and Moore's Law
- The Graphics Processing Unit (GPU)
- Computer clusters
  - The Message Passing Interface
- Volunteer and grid computing
- Summary
Content

I : Everyday computing

• The Turing machine
• The Central Processing Unit (CPU)
  – Intrinsics, vectorisation
• Multiple CPUs
  – Multithreading
  – NUMA zones
  – Hyperthreading
• Amdhal's law and Moore's Law

II : High performance computing

• The Graphics Processing Unit (GPU)
• Computer clusters
  – The Message Passing Interface
• Volunteer and grid computing
• Summary
What is a computer?
The Turing machine

• Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a computer
The Turing machine

- Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → **a computer**
The Turing machine

- Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a computer
The Turing machine

- Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a computer

State register

Tape:

```
0 0 0 0 5 K X 2 O 0 0 0 0
```
The Turing machine

- Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a computer

Program:

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
<th>Instruction 4</th>
<th>Instruction 5</th>
<th>Instruction 6</th>
<th>Instruction 7</th>
</tr>
</thead>
</table>

Tape:

| 0 | 0 | 0 | 0 | 5 | K | X | 2 | O | 0 | 0 | 0 | 0 |

Head

State register
Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a computer.
The Turing machine

- Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a computer

State register

Program:

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
Instruction 7

Tape:

0 0 0 0 5 K X 2 O 0 0 0 0
The Turing machine

Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a computer

<table>
<thead>
<tr>
<th>Program:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction 1</td>
</tr>
<tr>
<td>Instruction 2</td>
</tr>
<tr>
<td>Instruction 3</td>
</tr>
<tr>
<td>Instruction 4</td>
</tr>
<tr>
<td>Instruction 5</td>
</tr>
<tr>
<td>Instruction 6</td>
</tr>
<tr>
<td>Instruction 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tape:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

State register

Head
Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a **computer**
The Turing machine

- Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → a computer
The Turing machine

- Alan Turing came up with a mathematical model of a machine that could perform a given set of instructions, on a given data set → **a computer**

**Tape:**

| 0 | 0 | 0 | 0 | 5 | K | X | 2 | R | 0 | 0 | 0 |

**Program:**

- Instruction 1
- Instruction 2
- Instruction 3
- Instruction 4
- Instruction 5
- Instruction 6
- Instruction 7

**Head**

**State register**
Modern (or common) computing architecture

- One can build a mechanical implementation of the Turing machine (or even out of LEGO's...)
  - The number of operations per unit time by mechanical constraints...
- The breakthrough comes with semiconductors, with nanosecond response time (→ GHz)

![Diagram of CPU and RAM with text:](image)
Modern (or common) computing architecture

- One can build a mechanical implementation of the Turing machine (or even out of LEGOs...)
  - The number of operations per unit time by mechanical constraints...
- The breakthrough comes with semiconductors, with nanosecond response time (→ GHz)

```
| Instruction 1 |
| Instruction 2 |
| Instruction 3 |
| Instruction 4 |
| Instruction 5 |
| Instruction 6 |
| Instruction 7 |
```

- CPU
  - Register
- RAM
- Compiler
- Code (human readable)
Modern (or common) computing architecture

- One can build a mechanical implementation of the Turing machine (or even out of LEGOs...)
  - The number of operations per unit time by mechanical constraints...
- The breakthrough comes with semiconductors, with nanosecond response time (→ GHz)

![Diagram showing CPU, Register, RAM, Instruction 1 to 7, Compiler, and Code (human readable)]
Modern (or common) computing architecture

- One can build a mechanical implementation of the Turing machine (or even out of LEGO's...)
  - The number of operations per unit time by mechanical constraints...

- The breakthrough comes with semiconductors, with nanosecond response time (GHz)

![Diagram of CPU, Register, Compiler, Instruction set, Code (human readable), RAM](image-url)
Modern (or common) computing architecture

- One can build a mechanical implementation of the Turing machine (or even out of LEGOs...)
  - The number of operations per unit time by mechanical constraints...
- The breakthrough comes with semiconductors, with nanosecond response time (→ GHz)
The central processing unit

CPU

- List of built in instructions to performed on a given data type (int, float, double, …)
  - Data handling
  - Control flow
  - Basic arithmetic operations, but not only!
    → Square root, trigonometric functions,... can be implemented as a single instruction!
Intrinsics

- The choice of the right instruction in the 'catalogue' offered by a given CPU is the role of the compiler
  - In some (few) cases, you know better than the compiler!
  - You can impose the usage of a given instruction → Intrinsics

- An intrinsic applies to a given data type
  - 8-bits, ... , 64-bits, ...
  - For many applications its is interesting to perform the same operation on two or more numbers at once → Parallelism
Vectorisation

\[ x \rightarrow \sqrt{x} \]

x

Float 64-bits
(= double)
Vectorisation

\[ x_1 \rightarrow \sqrt{x_1} \]
\[ x_2 \rightarrow \sqrt{x_2} \]

\[ x \rightarrow \sqrt{x} \]

\[ x_1 \text{ Float 32-bits} \]
\[ x_2 \text{ Float 32-bits} \]

\[ x \text{ Float 64-bits (= double)} \]
Vectorisation

\[ x_1 \rightarrow \sqrt{x_1} \]
\[ x_2 \rightarrow \sqrt{x_2} \]

\[ x \rightarrow \sqrt{x} \]

\[ x_1 \rightarrow \sqrt{x_1} \]
\[ x_2 \rightarrow \sqrt{x_2} \]

\begin{tabular}{|c|c|}
\hline
x_1 & x_2 \\
\hline
Float 32-bits & Float 32-bits \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
x & \\
\hline
Float 64-bits (= double) \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
x_1 & x_2 \\
\hline
Float 64-bits & Float 64-bits \\
\hline
\end{tabular}
**Vectorisation**

- **Intrinsics in C (SSE2):**

  ```c
  extern __m128d _mm_sqrt_pd(__m128d v1);
  extern __m256d _mm256_sqrt_pd(__m256d v1);
  ```
Intrinsics in C (SSE2):

- `extern __m128d _mm_sqrt_pd(__m128d v1);`
- `extern __m256d _mm256_sqrt_pd(__m256d v1);`
Vectorisation

- In intrinsics in C (SSE2):
  - 4xfloats 32-bits or 2x64-bits
  - 8xfloats 32-bits or 4x64-bits

- In intrinsics in C (SSE2):
  - extern __m128d _mm_sqrt_pd(__m128d v1);
  - extern __m256d _mm256_sqrt_pd(__m256d v1);
Vectorisation

- Intrinsics in C (SSE2):
  - `extern __m128d _mm_sqrt_pd(__m128d v1);`
  - `extern __m256d _mm256_sqrt_pd(__m256d v1);`

- The vectorisation capabilities are particularly trendy nowadays:
  - Intel Sandy Bridge (128-bits operations, 2011)
  - Intel Haswell (256-bits operations, 2013)
  - Intel Xeon Phi (512-bits operations, 2016)
Vectorisation

- Intrinsics in C (SSE2):
  - `extern __m128d _mm_sqrt_pd(__m128d v1);`
  - `extern __m256d _mm256_sqrt_pd(__m256d v1);`

- The vectorisation capabilities are particularly trendy nowadays:
  - Intel Sandy Bridge (128-bits operations, 2011)
  - Intel Haswell (256-bits operations, 2013)
  - Intel Xeon Phi (512-bits operations, 2016) ➔ 8x64-bits operations at once!
Compiler options

- Intel and Microsoft's website offers documentation about the intrinsics.
- But *don't kid yourself*, there are people spending their lives writing compilers, they are usually better than you!
  → However you may help the compiler exploit the full capabilities of your hardware!
- In gcc the option `-ftree-vectorize` (also in `-O2`) enable vectorisation.
- The compiler tries to recognize vectorizable patterns in your loops (some compilers are better than others...improving every day).
Flynn's taxonomy

- Single instruction, single data
  - Most common / simple

Illustrations from https://en.wikipedia.org/wiki/Flynn's_taxonomy
Flynn's taxonomy

- Single instruction, single data
  - Most common / simple

- Single instruction, multiple data
  - Vectorisation
  - Graphics processing units (GPU)

Illustrations from https://en.wikipedia.org/wiki/Flynn's_taxonomy
Flynn's taxonomy

- **Single instruction, single data**
  - Most common / simple

- **Single instruction, multiple data**
  - Vectorisation
  - Graphics processing units (GPU)

- **Multiple instructions, single data**
  - ?

Illustrations from [https://en.wikipedia.org/wiki/Flynn's_taxonomy](https://en.wikipedia.org/wiki/Flynn's_taxonomy)
Flynn's taxonomy

- Single instruction, single data
  - Most common / simple

- Single instruction, multiple data
  - Vectorisation
  - Graphics processing units (GPU)

- Multiple instructions, single data
  - ?

- Multiple instructions, multiple data
  - Multithreading (shared memory parallelisation)
  - Cluster (distributed memory parallelisation)

Illustrations from [https://en.wikipedia.org/wiki/Flynn's_taxonomy](https://en.wikipedia.org/wiki/Flynn's_taxonomy)
Multithreading

Thread 0
- Instruction 1
- Instruction 2
- Instruction 3
- Instruction 4
- Instruction 5
- Instruction 6
- Instruction 7

CPU0
- Register

RAM
Multithreading

Thread 0
- Instruction 1
- Instruction 2
- Instruction 3
- Instruction 4
- Instruction 5
- Instruction 6
- Instruction 7

Thread 1
- Instruction' 1
- Instruction' 2
- Instruction' 3
- Instruction' 4
- Instruction' 5
- Instruction' 6
- Instruction' 7

CPU0
Register

CPU1
Register

RAM
Multithreading

Thread 0
- Instruction 1
- Instruction 2
- Instruction 3
- Instruction 4
- Instruction 5
- Instruction 6
- Instruction 7

CPU0
- Register

RAM

CPU1
- Register

Thread 1
- Instruction' 1
- Instruction' 2
- Instruction' 3
- Instruction' 4
- Instruction' 5
- Instruction' 6
- Instruction' 7

Thread 2
- Instruction'' 1
- Instruction'' 2
- Instruction'' 3
- Instruction'' 4
- Instruction'' 5
- Instruction'' 6
- Instruction'' 7

CPU2
- Register

CPU3
- Register

Thread 3
- Instruction''' 1
- Instruction''' 2
- Instruction''' 3
- Instruction''' 4
- Instruction''' 5
- Instruction''' 6
- Instruction''' 7
Multithreading: expectation vs reality
Multithreading: expectation vs reality
Multithreading: The limits

Thread 0
- Instruction 1
- Instruction 2
- Instruction 3
- Instruction 4
- Instruction 5
- Instruction 6
- Instruction 7

Thread 1
- Instruction' 1
- Instruction' 2
- Instruction' 3
- Instruction' 4
- Instruction' 5
- Instruction' 6
- Instruction' 7

Thread 2
- Instruction'' 1
- Instruction'' 2
- Instruction'' 3
- Instruction'' 4
- Instruction'' 5
- Instruction'' 6
- Instruction'' 7

Thread 3
- Instruction''' 1
- Instruction''' 2
- Instruction''' 3
- Instruction''' 4
- Instruction''' 5
- Instruction''' 6
- Instruction''' 7

CPU0
- Register

CPU1
- Register

CPU2
- Register

CPU3
- Register

RAM
Threads are almost separate processes, with the advantage that they share the same memory.

They are executed asynchronously.
Threads are almost separate processes, with the advantage that they share the same memory

They are executed asynchronously
Threads are almost separate processes, with the advantage that they share the same memory.

- They are executed asynchronously.
  - Concurrent memory access!
Threads are almost separate processes, with the advantage that they share the same memory.

- They are executed asynchronously
  - Concurrent memory access!
Multithreading: The limits

- Threads are almost separate processes, with the advantage that they share the same memory.
- They are executed asynchronously.
  - Concurrent memory access!
Threads are almost separate processes, with the advantage that they share the same memory.

- They are executed asynchronously
  - Concurrent memory access!
  - Thread synchronisation!
Talk is cheap, show me the code
- Linus Torvalds
#include<math.h>
#include<cstdio>
#include <sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0;i<nPart;++i){
        radius = (double)rand() / RAND_MAX;
        while(radius==0){
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }

    int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;

    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        for(int i = 0;i<nPart;++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }

    FILE* file = fopen("phaseSpace.csv","w");
    for(int i=0;i<nPart;++i){
        fprintf(file,"%E,%E\n",x[i],px[i]);
    }
    fclose(file);
}
Initialisation of a set of particles with random initial positions and transverse momentum
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters

```c
#include<math.h>
#include<cstdio>
#include <sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0;i<nPart;++i){
        radius = (double)rand() / RAND_MAX;
        while(radius==0){  
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }

    int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        for(int i = 0;i<nPart;++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/abs(x[i]);
            }
        }
    }

    FILE* file = fopen("phaseSpace.csv","w");
    for(int i=0;i<nPart;++i){  
        fprintf(file,"%E,%E
",x[i],px[i]);
    }
    fclose(file);
}
```
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)

\[
\begin{align*}
(x_{t+1})' &= \begin{pmatrix} \cos(\Phi) & \sin(\Phi) \\ -\sin(\Phi) & \cos(\Phi) \end{pmatrix} \cdot (x_t) \\
(x_{t+1}) &= \begin{pmatrix} x_{t+1} \\ x'_{t+1} \end{pmatrix}
\end{align*}
\]
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)

\[
\begin{pmatrix}
  x_{t+1}' \\
  x'_{t+1}
\end{pmatrix} =
\begin{pmatrix}
  1 & 0 \\
  f & 1
\end{pmatrix} \cdot
\begin{pmatrix}
  \cos(\Phi) & \sin(\Phi) \\
  -\sin(\Phi) & \cos(\Phi)
\end{pmatrix} \cdot
\begin{pmatrix}
  x_t \\
  x'_t
\end{pmatrix}
\]

Initialisation of a set of particles with random initial positions and transverse momentum

Initialisation of tracking parameters

Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file

```c
#include<math.h>
#include<cstdio>
#include <sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0;i<nPart;++i){
        radius = (double)rand() / RAND_MAX;
        while(radius==0){
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }
    int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        for(int i = 0;i<nPart;++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
    FILE* file = fopen("phaseSpace.csv","w");
    for(int i=0;i<nPart;++i){
        fprintf(file,"%E,%E
",x[i],px[i]);
    }
    fclose(file);
}
```
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file

```c
#include<math.h>
#include<stdio.h>
#include <sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0;i<nPart;++i){
        radius = (double)rand() / RAND_MAX;
        while(radius==0){
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }
    int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        for(int i = 0;i<nPart;++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
    FILE* file = fopen("phaseSpace.csv","w");
    for(int i=0;i<nPart;++i){
        fprintf(file,"%E,%E\n",x[i],px[i]);
    }
    fclose(file);
}
```
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file

```c
#include<math.h>
#include<stdio.h>
#include <sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0;i<nPart;++i){
        radius = (double)rand() / RAND_MAX;
        while(radius==0){
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }
    int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        for(int i = 0;i<nPart;++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
    FILE* file = fopen("phaseSpace.csv","w");
    for(int i=0;i<nPart;++i){
        fprintf(file,"%E,%E
",x[i],px[i]);
    }
    fclose(file);
}
```
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file

```c
#include<math.h>
#include<stdio.h>
#include <sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0;i<nPart;++i){
        radius = (double)rand() / RAND_MAX;
        while(radius==0){
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }
    FILE* file = fopen("phaseSpace.csv","w");
    for(int i=0;i<nPart;++i){
        fprintf(file,"%E,%E\n",x[i],px[i]);
    }
    fclose(file);
}
```
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file

```c
#include<math.h>
#include<stdio.h>
#include<sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0;i<nPart;++i){
        radius = (double)rand() / RAND_MAX;
        while(radius==0){
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }

    int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        for(int i = 0;i<nPart;++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }

    FILE* file = fopen("phaseSpace.csv","w");
    for(int i=0;i<nPart;++i){
        fprintf(file,"%E,%E\n",x[i],px[i]);
    }
    fclose(file);
}
```

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file
Example : non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here : beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one-turn matrix and a non-linear element (here: beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file

```c
#include<math.h>
#include<cstdio>
#include<sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0; i < nPart; ++i)
    {
        radius = (double)rand() / RAND_MAX;
        while(radius == 0)
        {
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }

    int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0; turn < nTurn; ++turn)
    {
        for(int i = 0; i < nPart; ++i)
        {
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres)
            {
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }

    FILE* file = fopen("phaseSpace.csv","w");
    for(int i = 0; i < nPart; ++i)
    {
        fprintf(file,":E,":E\n",x[i],px[i]);
    }
    fclose(file);
}
```
Example: non-linear dynamics in a collider

- Initialisation of a set of particles with random initial positions and transverse momentum
- Initialisation of tracking parameters
- Tracking through a one turn matrix and a non-linear element (here: beam-beam interaction)
- After tracking, writing the coordinates of all particles into a file

```c
#include<math.h>
#include<cstdio>
#include <sys/time.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    double radius;
    double angle;
    for(int i = 0;i<nPart;++i){
        radius = (double)rand() / RAND_MAX;
        while(radius==0){
            radius = rand();
        }
        radius = sqrt(-2.0*log(radius));
        angle = (double)2.0*M_PI*rand()/RAND_MAX;
        x[i] = radius*sin(angle);
        px[i] = radius*cos(angle);
    }
    int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        for(int i = 0;i<nPart;++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
    FILE* file = fopen("phaseSpace.csv","w");
    for(int i=0;i<nPart;++i){
        fprintf(file,"%E,%E\n",x[i],px[i]);
    }
    fclose(file);
}
```
Multithreading: The hard way

- In order to profit from the shared memory, the variables are now declared as global
- The rest of the initialisation remains identical

```c
#include<math.h>
#include<cstdio>
#include<stdlib.h>
#include <pthread.h>

const int nPart = 500000;
double x[nPart];
double px[nPart];

int nTurn = 100;
double sinPhix = sin(2.0*M_PI*0.31);
double cosPhix = cos(2.0*M_PI*0.31);
double scale = 1.0;
double thres = 1E-10;

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

double radius;
double angle;
for(int i = 0;i<nPart;++i){
    radius = (double)rand() / RAND_MAX;
    while(radius==0){
        radius = rand();
    }
    radius = sqrt(-2.0*log(radius));
    angle = (double)2.0*M_MPI*rand()/RAND_MAX;
    x[i] = radius*sin(angle);
    px[i] = radius*cos(angle);
}
...
... 

```c
pthread_t thread1;
pthread_t thread2;
pthread_t thread3;
pthread_t thread4;
int indices0[2] = {0,100000};
int indices1[2] = {400000,500000};
int indices2[2] = {100000,200000};
int indices3[2] = {200000,300000};
int indices4[2] = {300000,400000};

pthread_create(&thread1, NULL, track, indices1);
pthread_create(&thread2, NULL, track, indices2);
pthread_create(&thread3, NULL, track, indices3);
pthread_create(&thread4, NULL, track, indices4);
track(indices0);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
pthread_join(thread4, NULL);
...```
Multithreading : The hard way

- Preparing 4 POSIX threads, that will share the work load of the loop over the particles
  - The process running this code is 'thread 0'

```c
pthread_t thread1;
pthread_t thread2;
pthread_t thread3;
pthread_t thread4;

int indices0[2] = {0,100000};
int indices1[2] = {400000,500000};
int indices2[2] = {100000,200000};
int indices3[2] = {200000,300000};
int indices4[2] = {300000,400000};

pthread_create(&thread1, NULL, track, indices1);
pthread_create(&thread2, NULL, track, indices2);
pthread_create(&thread3, NULL, track, indices3);
pthread_create(&thread4, NULL, track, indices4);
track(indices0);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
pthread_join(thread4, NULL);
```
Multithreading: The hard way

- Preparing 4 POSIX threads, that will share the work load of the loop over the particles
  - The process running this code is 'thread 0'
- Creating the 4 threads, that will execute simultaneously (but independently) on the different CPUs, executing the function 'track' with different arguments
  - The thread 0 also, does its share

```c
pthread_t thread1;
pthread_t thread2;
pthread_t thread3;
pthread_t thread4;
int indices0[2] = {0,100000};
int indices1[2] = {400000,500000};
int indices2[2] = {100000,200000};
int indices3[2] = {200000,300000};
int indices4[2] = {300000,400000};
pthread_create(&thread1, NULL, track, indices1);
pthread_create(&thread2, NULL, track, indices2);
pthread_create(&thread3, NULL, track, indices3);
pthread_create(&thread4, NULL, track, indices4);
track(indices0);
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
pthread_join(thread4, NULL);
```
Multithreading: The hard way

- Preparing 4 POSIX threads, that will share the work load of the loop over the particles
  - The process running this code is 'thread 0'

- Creating the 4 threads, that will execute simultaneously (but independantly) on the different CPUs, executing the function 'track' with different arguments
  - The thread 0 also, does its share

- The join statement indicates the stops the execution of the thread 0, until the others are finished
  → Synchronisation

```c
pthread_t thread1;
pthread_t thread2;
pthread_t thread3;
pthread_t thread4;

int indices0[2] = {0,100000};
int indices1[2] = {400000,500000};
int indices2[2] = {100000,200000};
int indices3[2] = {200000,300000};
int indices4[2] = {300000,400000};

pthread_create(&thread1, NULL, track, indices1);
pthread_create(&thread2, NULL, track, indices2);
pthread_create(&thread3, NULL, track, indices3);
pthread_create(&thread4, NULL, track, indices4);
track(indices0);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
pthread_join(thread4, NULL);

...
Multithreading : The hard way

```c
void* track(void* indices_ptr){
    int* indices = (int*) indices_ptr;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        for(int i = indices[0];i<indices[1];++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
    ...
    int indices1[2] = {400000,500000};
Pthread_create(&thread1, NULL, track, indices1);
    ...
```
Multithreading: The hard way

The function code hasn't changed, except that the execution is performed only between the indices given in argument.

```c
void* track(void* indices_ptr)
{
    int* indices = (int*) indices_ptr;
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0; turn < nTurn; ++turn){
        for(int i = indices[0]; i < indices[1]; ++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;

            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
}

int indices1[2] = {400000,500000};
Pthread_create(&thread1, NULL, track, indices1);
...
Global variables, pointer to functions, type casting, complicated syntax, are you sure about this?
- Fuego, a rabbit that is unsure about this
Multithreading: The easy way

- OpenMP allows, with a single line of code (so-called a pragma) to tell the compiler to share the load of a loop's other multiple threads
  - The compiler needs to know which part of the memory is shared, which part should be allocated for each thread

```c
#include<math.h>
#include<cstdio>
#include<stdlib.h>

int main(int argc, char *argv[]){
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0; turn < nTurn; ++turn){
        #pragma omp parallel for default(firstprivate) shared(x,px) \
        schedule(guided,1000)
        for(int i = 0; i < nPart; ++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
    return 0;
}
```

→ Compiler option -fopenmp
Multithreading : The easy way

- OpenMP allows, with a single line of code (so-called a pragma) to tell the compiler to share the load of a loops other multiple threads
  - The compiler needs to know which part of the memory is shared, which part should be allocated for each thread

```c
#include<math.h>
#include<stdio.h>
#include<stdlib.h>

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

    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn){
        #pragma omp parallel for default(firstprivate) shared(x,px) \
        schedule(guided,1000)
        for(int i = 0;i<nPart;++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
}
```

→ Compiler option -fopenmp
Multithreading : The easy way

- OpenMP allows, with a single line of code (so-called a pragma) to tell the compiler to share the load of a loops other multiple threads
  - The compiler needs to know which part of the memory is shared, which part should be allocated for each thread

```c
#include<math.h>
#include<stdio.h>
#include<stdlib.h>

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

  double oldX = 0.0;
  double oldPx = 0.0;
  for(int turn = 0;turn<nTurn;++turn){
    //pragma omp parallel for default(firstprivate) shared(x,px) \ schedule(guided,1000)
    for(int i = 0;i<nPart;++i){
      oldX = x[i];
      oldPx = px[i];
      x[i] = cosPhix*oldX + sinPhix*oldPx;
      px[i] = -sinPhix*oldX + cosPhix*oldPx;
      if(abs(x[i]) > thres){
        px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
      }
    }
  }
}
```

By default each threads gets a copy of the variable

→ Compiler option -fopenmp
Multithreading: The easy way

- OpenMP allows, with a single line of code (so-called a pragma) to tell the compiler to share the load of a loop's other multiple threads.

  - The compiler needs to know which part of the memory is shared, which part should be allocated for each thread.

```c
#include<math.h>
#include<cstdio>
#include<stdlib.h>

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

    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0; turn < nTurn; ++turn){
        #pragma omp parallel for default(firstprivate) shared(x, px) \
            schedule(guided, 1000)
        for(int i = 0; i < nPart; ++i){
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix * oldX + sinPhix * oldPx;
            px[i] = -sinPhix * oldX + cosPhix * oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale * (1.0 - exp(-0.5 * x[i] * x[i])) / x[i];
            }
        }
    }
}
```

-- Compiler option -fopenmp
Multithreading : The easy way

- OpenMP allows, with a single line of code (so-called a pragma) to tell the compiler to share the load of a loops other multiple threads
  - The compiler needs to know which part of the memory is shared, which part should be allocated for each thread
  - A large variety of simple tools exists for various applications. You may gain factors in execution time!

```c
#include<math.h>
#include<cstdio>
#include<stdlib.h>

int main(int argc, char *argv[])
{
    double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0;turn<nTurn;++turn)
    {
        #pragma omp parallel for default(firstprivate) shared(x,px) \
            schedule(guided,1000)
        for(int i = 0;i<nPart;++i)
        {
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if(abs(x[i]) > thres){
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
}
```

→ Compiler option -fopenmp
Heterogenous multithreading

• Pthreads can be extremely powerfull for heterogenous tasks, but is clearly a bad choice for our example application

• Multithreading comes in particularly handy for example when programming software for control systems
Heterogenous multithreading

- Pthreads can be extremely powerful for heterogenous tasks, but is clearly a bad choice for our example application
- Multithreading comes in particularly handy for example when programming software for control systems
Heterogenous multithreading

- Pthreads can be extremely powerful for heterogenous tasks, but is clearly a bad choice for our example application
- Multithreading comes in particularly handy for example when programming software for control systems

- Sends instructions to move the orbit in IP1
Heterogenous multithreading

- Pthreads can be extremely powerful for heterogenous tasks, but is clearly a bad choice for our example application.
- Multithreading comes in particularly handy for example when programming software for control systems.

- Sends instructions to move the orbit in IP1.
- Sends instructions to move the orbit in IP5.
Heterogenous multithreading

- Pthreads can be extremely powerful for heterogenous tasks, but is clearly a bad choice for our example application.
- Multithreading comes in particularly handy for example when programming software for control systems.

- Sends instructions to move the orbit in IP1.
- Sends instructions to move the orbit in IP5.
- Receives online data from the two experiments.
Heterogenous multithreading

- Pthreads can be extremely powerful for heterogenous tasks, but is clearly a bad choice for our example application.
- Multithreading comes in particularly handy for example when programming software for control systems.

- Sends instructions to move the orbit in IP1.
- Sends instructions to move the orbit in IP5.
- Receives online data from the two experiments.
- Responsive GUI, to look at the data and possibly generate new tasks.
Heterogenous multithreading

- Pthreads can be extremely powerful for heterogenous tasks, but is clearly a bad choice for our example application
- Multithreading comes in particularly handy for example when programming software for control systems

- Sends instructions to move the orbit in IP1
- Sends instructions to move the orbit in IP5
- Receives online data from the two experiments
- Responsive GUI, to look at the data and possibly generate new tasks
  → At least 5 threads running simultaneously, with completely different tasks
Heterogenous multithreading

- Pthreads can be extremely powerful for heterogenous tasks, but is clearly a bad choice for our example application.
- Multithreading comes in particularly handy for example when programming software for control systems.

### Task Manager

<table>
<thead>
<tr>
<th>Task</th>
<th>Status</th>
<th>IP1</th>
<th>IP2</th>
<th>IP3</th>
<th>IP4</th>
<th>IP5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1</td>
<td>RUNNING</td>
<td>BUSY</td>
<td>BUSY</td>
<td>BUSY</td>
<td>BUSY</td>
<td>BUSY</td>
</tr>
<tr>
<td>CPU 2</td>
<td>Optimizing</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- Sends instructions to move the orbit in IP1.
- Sends instructions to move the orbit in IP5.
- Receives online data from the two experiments.
- Responsive GUI, to look at the data and possibly generate new tasks.
  - At least 5 threads running simultaneously, with completely different tasks.
- In principle there can be more threads than CPUs, the OS has to priorities.
Multithreading is the most versatile tool, since any type of tasks can be performed in parallel.
Multithreading is the most versatile tool, since any type of tasks can be performed in parallel

- Nevertheless, you can't cook the sauce before having cut the onions
Synchronisation, scheduling

• Multithreading is the most versatile tool, since any type of tasks can be performed in parallel
  – Nevertheless, you can't cook the sauce before having cut the onions
  → Like a good chef, prepare your recipe well and share the tasks accordingly!
• Multithreading is the most versatile tool, since any type of tasks can be performed in parallel
  
  – Nevertheless, you can't cook the sauce before having cut the onions
  → Like a good chef, prepare your recipe well and share the tasks accordingly!

• What could go wrong?
More on synchronization and concurrent memory access
More on synchronization and concurrent memory access

- Critical sections
More on synchronization and concurrent memory access

- Critical sections
- Reduction
More on synchronization and concurrent memory access

- Critical sections
- Reduction
- Atomic operations
More on synchronization and concurrent memory access

- Critical sections
- Reduction
- Atomic operations
- Locks
More on synchronization and concurrent memory access

- Critical sections
- Reduction
- Atomic operations
- Locks
- ...
Non-uniform memory access (NUMA)
In reality, there exists several levels of cache memory, shared between cores, sockets or nodes.

- The communication is slowest for data far from the CPU → Data locality!
Non-uniform memory access (NUMA)

- In reality, there exists several levels of cache memory, shared between cores, sockets or nodes.
  - The communication is slowest for data far from the CPU → Data locality!
- Imagine x, px are allocated by core0 of socket0, in its L3 cache, the sharing of the load with core7 of socket3 will be inefficient w.r.t. the sharing with core1 of the same socket → Equal sharing between CPUs is usually not the most efficient → Scheduling
Better not assign the cooking of the sauce to the waiter, since he is away from the hotplate...
Thread scheduling

```cpp
... double oldX = 0.0;
double oldPx = 0.0;
for(int turn = 0; turn < nTurn; ++turn) {
    #pragma omp parallel for default(firstprivate) shared(x,px) schedule(guided,1000)
    for(int i = 0; i < nPart; ++i) {
        oldX = x[i];
        oldPx = px[i];
        x[i] = cosPhix*oldX + sinPhix*oldPx;
        px[i] = -sinPhix*oldX + cosPhix*oldPx;
        if(x[i] > thres) {
            px[i] += scale*(1.0-exp(-0.5*x[i]))/x[i];
        }
    }
}
...```

- In OpenMP, you can indicate to the compiler to chunk the load in smaller pieces and assign the chunks as the threads perform their tasks
  - The CPU performing faster will be assigned more chunks
Performance

```
 top - 11:46:14 up 14 days, 21:06, 7 users, load average: 1.04, 0.60, 0.25
Tasks: 381 total, 2 running, 379 sleeping, 0 stopped, 0 zombie
 %Cpu0 : 0.3 us, 0.0 sy, 0.0 nt, 99.7 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu1 : 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu2 : 0.3 us, 0.0 sy, 0.0 nt, 98.0 ld 1.7 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu3 : 100.0 us, 0.0 sy, 0.0 nt, 0.0 ld, 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu4 : 2.3 us, 0.0 sy, 0.0 nt, 97.0 ld 0.7 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu5 : 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu6 : 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu7 : 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu8 : 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu9 : 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu10: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu11: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu12: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu13: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu14: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu15: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu16: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu17: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu18: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu19: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu20: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu21: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu22: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
 %Cpu23: 0.0 us, 0.0 sy, 0.0 nt, 100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
Kb Mem: 26412262 total, 742844 free, 10419636 used, 25296894 buff/cache
Kb Swap: 9765884 total, 9765884 free, 0 used, 25265972 avail Mem

 PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
21637 xbuffat 20 0 15028 9764 1864 R 100.0 0.0 3:38.36 phaseSpace_serl
```
### Performance

```
Top - 11:46:14 up 14 days, 21:08, 7 users, load average: 1.94, 0.68, 0.25
Tasks: 381 total, 2 running, 379 sleeping, 0 stopped, 0 zombie
%CPU : 0.3 us, 0.0 sy, 0.0 ni, 99.7 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU1: 0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU2: 0.3 us, 0.0 sy, 0.0 ni, 98.0 ld 1.7 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU3:100.0 us, 0.0 sy, 0.0 ni, 0.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU4: 2.3 us, 0.0 sy, 0.0 ni, 97.0 ld 0.7 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU5: 0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU6: 0.0 us, 0.0 sy, 0.0 ni, 99.9 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU7: 0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU8: 0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU9: 0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU10:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU11:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU12:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU13:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU14:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU15:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU16:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU17:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU18:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU19:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU20:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU21:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU22:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
%CPU23:0.0 us, 0.0 sy, 0.0 ni,100.0 ld 0.0 wa, 0.0 ht, 0.0 sl, 0.0 st
KIL Mem: 26412262+total, 742844 free, 10419636 used, 25296894+buff/cache
KIB Swap: 9765884 total, 9765884 free, 0 used, 25265972+avail Mem

PID USER PR NI VIRT RES SHR CPU %MEM TIME+ COMMAND
21637 xbuffat 20 0 15028 9764 1864 100.0 0.0 3:38.36 phaseSpace_serl
```
Performance

```
Top - 11:46:14 up 14 days, 21:06, 7 users, load average: 1.04, 0.60, 0.25
Tasks: 381 total, 2 running, 379 sleeping, 0 stopped, 0 zombie

Cpu0 :  0.3 us,  0.0 sy,  0.0 ni, 99.7 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu1 :  0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu2 :  0.0 us,  0.0 sy,  0.0 ni,  0.0 ld  1.7 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu3 :  100.0 us, 0.0 sy,  0.0 ni,  0.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu4 :   2.3 us,  0.0 sy,  0.0 ni,  0.0 ld  0.7 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu5 :  0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu6 :  0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu7 :  0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu8 :  0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu9 :  0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu10: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu11: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu12: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu13: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu14: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu15: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu16: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu17: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu18: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu19: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu20: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu21: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu22: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
Cpu23: 0.0 us,  0.0 sy,  0.0 ni,100.0 ld  0.0 wa,  0.0 hi,  0.0 si,  0.0 st

KiB Mem:  28412262 total,  742844 free, 10419636 used, 25296894+buff/cache
KiB Swap:  9765884 total,  9765884 free,  0 used, 25265972+avail Mem
```

```
PID USER PR NI VIRT RES SHR %CPU %MEM TIME+ COMMAND

21637 xbuffat 20  0 15028  9764 1864  100.0  0.0  3:38.36 phaseSpace_serl
```
Performance

- 1 process at 100% → one CPU at 100%
  - Serial code execution time : 277s
Performance

- 1 process at 100% → one CPU at 100%
  → Serial code execution time : 277s
• 1 process at 100% → one CPU at 100%
  → Serial code execution time : 277s
### Performance

- 1 process at 100% → one CPU at 100%
  - Serial code execution time: 277s
### Performance

- **1 process at 100% → one CPU at 100%**
  - Serial code execution time: 277s

- **1 Process at 2400% (i.e. 24 threads at 100%)**
  - 24 CPU at 100%
  - Multithreaded code execution time: 15s
Performance

- 1 process at 100% → one CPU at 100%
  → Serial code execution time : 277s

\[
\frac{277}{15} \approx 18
\]

- 1 Process at 2400% (i.e. 24 threads at 100%)
  → 24 CPU at 100%
- Multithreaded code execution time : 15s
18 faster with a 12 cores machine?

- Fuego, a rabbit that questions my sanity
From the operating system point of view, hyperthreaded cores appear as two separate CPUs.

The performance is not necessarily the one of two CPUs...
From the operating system point of view, hyperthreaded cores appear as two separate CPUs.

The performance is not necessarily the one of two CPUs...
From the operating system point of view, hyperthreaded cores appear as two separate CPUs.

The performance is not necessarily the one of two CPUs...
Hyperthreading and context switching

- Cores with hyperthreading are equipped with additional hardware, allowing to switch from one thread to another efficiently, thus avoiding idling of CPUs.
- When allocating more threads than CPUs, the operating system is also capable of switching threads (→ context switching), however this operation is slower.
Hyperthreading and context switching

- Cores with hyperthreading are equipped with additional hardware, allowing to switch from one thread to another efficiently, thus avoiding idling of CPUs.
- When allocating more threads than CPUs, the operating system is also capable of switching threads (→ context switching), however this operation is slower.
- Hyperthreading is particularly useful for your daily life, when you are checking emails, listening to music, reading newspapers, watching a Tutorial on Youtube, preparing slides for tomorrow's meeting and writing a Nobel price winning theory in LaTeX at the same time.
Hyperthreading and context switching

- Cores with hyperthreading are equipped with additional hardware, allowing to switch from one thread to another efficiently, thus avoiding idling of CPUs.

- When allocating more threads than CPUs, the operating system is also capable of switching threads (→ context switching), however this operation is slower.

- Hyperthreading is particularly useful for your daily life, when you are checking emails, listening to music, reading newspapers, watching a Tutorial on Youtube, preparing slides for tomorrow's meeting and writing a Nobel price winning theory in LaTeX at the same time.

- For scientific computing, it may be beneficial for codes where threads not synchronous, are non-uniformly loaded or requires heavy memory access.
Hyperthreading and context switching

- Cores with hyperthreading are equipped with additional hardware, allowing to switch from one thread to another efficiently, thus avoiding idling of CPUs.

- When allocating more threads than CPUs, the operating system is also capable of switching threads (→ context switching), however this operation is slower.

- Hyperthreading is particularly useful for your daily life, when you are checking emails, listening to music, reading newspapers, watching a Tutorial on Youtube, preparing slides for tomorrow's meeting and writing a Nobel price winning theory in LaTeX at the same time.

- For scientific computing, it may be beneficial for codes where threads not synchronous, are non-uniformly loaded or requires heavy memory access.

```c
double oldX = 0.0;
double oldPx = 0.0;
for(int turn = 0;turn < nTurn;++turn){
    #pragma omp parallel for default(firstprivate) shared(x,px) \
        schedule(guided,1000)
    for(int i = 0;i < nPart;++i){
        oldX = x[i];
        oldPx = px[i];
        x[i] = cosPhix*oldX + sinPhix*oldPx;
        px[i] = -sinPhix*oldX + cosPhix*oldPx;
        if(abs(x[i]) > thres){
            px[i] += scale*(-0.5*x[i]*x[i])/x[i];
        }
    }
}
```

- In my tracking example, the function `exp` is prone to acceleration with hyperthreading.
Multithreading
Multithreading

Hyperthreading
Serial, multi- and hyper-threaded codes

- Modern computers are based on a set of instructions to be performed by a processing unit (CPU) on a given data

- Most computers today allow for multithreading, i.e. multiple CPUs working in parallel on a shared memory
  - Due to technological limitations, the access to the shared memory is not uniformly efficient → The guide line is **maximise data locality**
  - Multithreading is easy (even in Fortran!), especially if you let the compiler do the job
    - It usually becomes extremely easy in high level languages such as Python or Matlab, it can be even automatic when using built-in functions or libraries (one of the reason to avoid for loops in such languages)
  - Multithreading can provide speedup linear with the number of real CPU in your machine, possibly more thanks to hyperthreading
  - The speed up can be limited by causality and memory access
    → Memory bound problem usually don't profit of such a scheme

- Hyperthreading is a technology allowing to maximise the efficiency of a single processing unit by switching fast between different threads busy with other tasks such as memory transfer
  - Even though they are not physically separate CPUs, they appear as such to the operating system (and to you)
About Moore's law

- From the point of view of number of transistors, Moore's law is still ~alive (for how long?)

(Original data collected and plotted by M. Horowitz, F. Laberte, O. Shacham, K. Olukolun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore)
About Moore's law

- From the point of view of number of transistors, Moore's law is still ~alive (for how long?)

- The performance increase nowadays come with additional cores (rather than clock speed for example)

Original data collected and plotted by M. Horowitz, F. Labarte, O. Shacham, K. Olikotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
Amdahl's law

Non parallelisable code

\[ S = \frac{\alpha + 1 - \alpha}{\alpha + \frac{1-\alpha}{N_p}} \]

\[ = \frac{1}{\alpha + \frac{1-\alpha}{N_p}} \]
Amdahl's law

For a fixed problem size, the speed up saturates for large number of cores

- Usually high performance codes are needed to increase the problem size rather than speed up what is achievable with a single CPU

- Comparing the speed up you obtained to Amdahl's law (fitting alpha) is a good way to assess the quality of your scheme

\[
S = \frac{\alpha + \frac{1 - \alpha}{N_p}}{\alpha + \frac{1 - \alpha}{N_p}} = \frac{1}{\alpha + \frac{1 - \alpha}{N_p}}
\]
About Moore's Law

- The computing power of the most powerful machine in the world does keep increasing!

Source: http://www.top500.org/
About Moore's Law

- The computing power of the most powerful machine in the world does keep increasing!

Source: http://www.top500.org/

Tomorrow we harness that power!
## Content

<table>
<thead>
<tr>
<th>I : Everyday computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The Turing machine</td>
</tr>
<tr>
<td>• The Central Processing Unit (CPU)</td>
</tr>
<tr>
<td>- Intrinsics, vectorisation</td>
</tr>
<tr>
<td>• Multiple CPUs</td>
</tr>
<tr>
<td>- Multithreading</td>
</tr>
<tr>
<td>- NUMA zones</td>
</tr>
<tr>
<td>- Hyperthreading</td>
</tr>
<tr>
<td>• Amdhal's law and Moore's Law</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>II : High performance computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The Graphics Processing Unit (GPU)</td>
</tr>
<tr>
<td>• Computer clusters</td>
</tr>
<tr>
<td>- The Message Passing Interface</td>
</tr>
<tr>
<td>• Volunteer and grid computing</td>
</tr>
<tr>
<td>• Summary</td>
</tr>
</tbody>
</table>
Flynn's taxonomy

- Single instruction, single data
  - Most common / simple

- Single instruction, multiple data
  - Vectorisation
  - Graphics processing units (GPU)

- Multiple instructions, single data
  - ?

- Multiple instructions, multiple data
  - Multithreading (shared memory parallelisation)
  - Cluster (distributed memory parallelisation)

Illustrations from https://en.wikipedia.org/wiki/Flynn's_taxonomy
The Graphical Processing Unit (GPU)

- Image processing usually require a single (simple) operation to be perform on large data set (e.g. 3D projection on 2D screen)
  → Computer games, special effects (VFX, CGI...)
  → Large market, therefore lots of money for development!
The Graphical Processing Unit (GPU)

- Image processing usually require a single (simple) operation to be perform on large data set (e.g. 3D projection on 2D screen)
  - Computer games, special effects (VFX, CGI...)
  - Large market, therefore lots of money for development!
The Graphical Processing Unit (GPU)

- Image processing usually require a single (simple) operation to be perform on large data set (e.g. 3D projection on 2D screen)
  - Computer games, special effects (VFX, CGI...)
  - Large market, therefore lots of money for development!

CPU (hidden behind the cooling fan)
The Graphical Processing Unit (GPU)

- Image processing usually require a single (simple) operation to be perform on large data set (e.g. 3D projection on 2D screen)
  - Computer games, special effects (VFX, CGI...)
  - Large market, therefore lots of money for development!
The Graphical Processing Unit (GPU)

- Image processing usually require a single (simple) operation to be perform on large data set (e.g. 3D projection on 2D screen)
  → Computer games, special effects (VFX, CGI...)
  → Large market, therefore lots of money for development!
The Graphical Processing Unit (GPU)

- Image processing usually require a single (simple) operation to be perform on large data set (e.g. 3D projection on 2D screen)
  - Computer games, special effects (VFX, CGI...)
  - Large market, therefore lots of money for development!
- These developments can be very useful for scientific applications as well
  - Watch out, most scientific application require double precision, whereas image processing is usually done in single precision
  - Compute capability >1.3 indicate double precision capacity, but not necessarily good performance with double precision!
    - Dedicated cards exists → some people talk about General Purpose GPU (GPGPU)
The Graphical Processing Unit (GPU)

- Image processing usually require a single (simple) operation to be perform on large data set (e.g. 3D projection on 2D screen)
  - Computer games, special effects (VFX, CGI...)
  - Large market, therefore lots of money for development!
- These developments can be very useful for scientific applications as well
  - Watch out, most scientific application require double precision, whereas image processing is usually done in single precision
  - Compute capability >1.3 indicate double precision capacity, but not necessarily good performance with double precision!
  - Dedicated cards exists → some people talk about General Purpose GPU (GPGPU)

![CPU (hidden behind the cooling fan)](image1.png)

![Memory](image2.png)

![Cheap GPU (~100CHF)](image3.png)

![Not so cheap GPUs...](image4.png)
The Graphical Processing Unit (GPU)

- Image processing usually require a single (simple) operation to be performed on large data set (e.g. 3D projection on 2D screen)
  - Computer games, special effects (VFX, CGI...)
  - Large market, therefore lots of money for development!
- These developments can be very useful for scientific applications as well
  - Watch out, most scientific applications require double precision, whereas image processing is usually done in single precision
  - Compute capability >1.3 indicate double precision capacity, but not necessarily good performance with double precision!
  - Dedicated cards exist → some people talk about General Purpose GPU (GPGPU)
• GPUs clearly overtake CPUs in terms of number of floating point operation per second, but...
Taking vectorisation to the extreme

The GPU is an extension of vectorisation to the extreme, allowing amazing speed up for some applications, useless for others...

→ The handling of large arrays takes time, the performance of a GPU has to rely on an efficient scheduling → multithreading and fast context switching!

By essence, the GPU requires a different memory architecture w.r.t. the CPU

→ The two are independent and should be handled properly!
The GPU

Host (Computer with a CPU)

GPU
The GPU

Host
(Computer with a CPU)

Kernel0

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
Instruction 7

GPU
The computing unit is now called a Streaming multiprocessor
- Optimised towards efficient switching between threads (~hyperthreading)
- The basic unit is called Warp in NVIDIA (Wave front in AMD)
  - Each Warp can process 32 threads simultaneously (64 in AMD)!
The threads are organised in blocks on a grid

- The threads within a block have access to a shared memory
- The execution of threads in a block can be synchronised
The GPU

- The threads are organised in blocks on a grid
  - The threads within a block have access to a shared memory
  - The execution of threads in a block can be synchronised
The host can hold several kernels (with its corresponding grid)

- Only one kernel can execute at a time on the GPU.
The GPU

Host (Computer with a CPU)

Kernel0
- Instruction 1
- Instruction 2
- Instruction 3
- Instruction 4
- Instruction 5
- Instruction 6
- Instruction 7

Kernel1
- Instruction 1
- Instruction 2
- Instruction 3
- Instruction 4
- Instruction 5
- Instruction 6
- Instruction 7

GPU

SMP
- Warp0
- Warp1
- Warp2
- Warp3
- Warp4
- Warp5

Grid0
- Thread block (0,0)
- Thread block (0,1)
- Thread block (1,0)
- Thread block (1,1)
- Thread block (2,0)
- Thread block (2,1)

Grid1
- Thread block (0,0)
- Thread block (0,1)
- Thread block (1,0)
- Thread block (1,1)
- Thread block (2,0)
- Thread block (2,1)
Changing your mindset

- Think data parallelism, not execution parallelism
- Warps are most efficient if they are full → Each block has to be large enough (>32 / 64 , but limited to 512 - 1024)
- As for CPUs, the more local the memory, the more efficient
- Since the threads execute synchronously, avoid divergence in the execution (at least within the same block)
Example: The hard way

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>

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

    // Initialisation
    // Computing
    // Output

}
```

Nothing special to include, however the compiler has to be compatible (gcc → nvcc)
Example : The hard way

Initialisation

```c
const int nPart = 500000;
double x[nPart];
double px[nPart];
double radius;
double angle;
for(int i = 0; i < nPart; ++i){
    radius = (double)rand() / RAND_MAX;
    while(radius == 0){
        radius = rand();
    }
    radius = sqrt(-2.0*log(radius));
    angle = (double)2.0*M_PI*rand() / RAND_MAX;
    x[i] = radius*sin(angle);
    px[i] = radius*cos(angle);
}
```

Copy the data from the host memory to the GPU memory

```c
double* x_cuda;
double* px_cuda;
cudaMalloc(&x_cuda, nPart*sizeof(double));
cudaMalloc(&px_cuda, nPart*sizeof(double));
cudMemcp(x_cuda, x, nPart*sizeof(double), cudaMemcpyHostToDevice);
cudMemcp(px_cuda, px, nPart*sizeof(double), cudaMemcpyHostToDevice);
...```
#include<math.h>
#include<cstdio>
#include<cstdlib.h>

__global__
void track(double* x, double* px, int nPart, double sinPhix, double cosPhix, double scale, double thres) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    double oldX = 0.0;
    double oldPx = 0.0;

    oldX = x[i];
    oldPx = px[i];
    x[i] = cosPhix*oldX + sinPhix*oldPx;
    px[i] = -sinPhix*oldX + cosPhix*oldPx;
    if (abs(x[i]) > thres) {
        px[i] += scale*(1.0-exp(-0.5*x[i]*x[i])/x[i]);
    }
}

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

    int threadsperblock = 512;
    int blockspergrid = ceil(nPart / threadsperblock);
    for(int turn = 0; turn<nTurn;++turn){
        track<<<blockspergrid,threadsperblock>>>(x_cuda,px_cuda,nPart,sinPhix,cosPhix,scale,thres);
    }
}

Output
Example: The hard way

```c
cudaMemcpy(x, x_cuda, nPart*sizeof(double), cudaMemcpyDeviceToHost);
cudaMemcpy(px, px_cuda, nPart*sizeof(double), cudaMemcpyDeviceToHost);
cudaFree(x_cuda);
cudaFree(px_cuda);

FILE* file = fopen("phaseSpace.csv", "w");
for(int i=0;i<nPart;++i){
    fprintf(file,"%E,%E\n",x[i],px[i]);
}
fclose(file);
```

Need to copy back the memory to the host
Example: The hard way

- 277s on a single CPU
- 2.1s on the (9 years old) GPU
  → 132x faster!
Example: the easy ways

With OpenACC, some simple instructions lets the compiler do the jobs

→ Similar performance can be achieved for several applications, without effort!

Several libraries exists to speed up your code without effort (also in high level languages)

double oldX = 0.0;
double oldPx = 0.0;
#pragma acc parallel
for(int turn = 0; turn < nTurn; ++turn){
    #pragma acc loop
    for(int i = 0; i < nPart; ++i){
        oldX = x[i];
        oldPx = px[i];
        x[i] = cosPhix*oldX + sinPhix*oldPx;
        px[i] = -sinPhix*oldX + cosPhix*oldPx;
        if(abs(x[i]) > thres){
            px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
        }
    }
}

→ Compiler option -fopenacc
Applications

- Do's (key word: large weakly interdependent data sets)
  - Processing of large data sets (optics correction algorithms (for large machines), fast online data processing, machine learning, ...)
  - Solution of differential equation (time domain electromagnetic / mechanical / thermal simulation, beam evolution, ...)
  - Linear algebra (matrix manipulations, frequency domain electromagnetic / mechanical simulations, beam stability, ...)
  - Monte Carlo (particle – matter interactions, detectors, vacuum, ...)
  - Beam physics (multiparticle tracking, ...)
  - ...

---

28.05.2018 154
Applications

- **Do's** (key word: *large weakly interdependent data sets*)
  - Processing of large data sets (optics correction algorithms (for large machines), fast online data processing, machine learning, ...)
  - Solution of differential equation (time domain electromagnetic / mechanical / thermal simulation, beam evolution, ...)
  - Linear algebra (matrix manipulations, frequency domain electromagnetic / mechanical simulations, beam stability, ...)
  - Monte Carlo (particle – matter interactions, detectors, vacuum, ...)
  - Beam physics (multiparticle tracking, ...)
  - ...

- **Maybe's** (key word: *large strongly interdependent data sets*)
  - Particle-in-cell
  - collective beam instabilities
Applications

- **Do's (key word: large weakly interdependent data sets)**
  - Processing of large data sets (optics correction algorithms (for large machines), fast online data processing, machine learning, ...)
  - Solution of differential equation (time domain electromagnetic / mechanical / thermal simulation, beam evolution, ...)
  - Linear algebra (matrix manipulations, frequency domain electromagnetic / mechanical simulations, beam stability, ...)
  - Monte Carlo (particle – matter interactions, detectors, vacuum, ...)
  - Beam physics (multiparticle tracking, ...)
  - ...

- **Maybe's (key word: large strongly interdependent data sets)**
  - Particle-in-cell
  - collective beam instabilities

- **Don'ts (key word: small data sets)**
  - Processing of small data sets
  - Single particle tracking, optics calculations
  - Heterogeneous tasks
Applications

- **Do's** (key word: *large weakly interdependent data sets*)
  - Processing of large data sets (optics correction algorithms (for large machines), fast online data processing, machine learning, ...)
  - Solution of differential equation (time domain electromagnetic / mechanical / thermal simulation, beam evolution, ...)
  - Linear algebra (matrix manipulations, frequency domain electromagnetic / mechanical simulations, beam stability, ...)
  - Monte Carlo (particle – matter interactions, detectors, vacuum, ...)
  - Beam physics (multiparticle tracking, ...)
  - ...

- **Maybe's** (key word: *large strongly interdependent data sets*)
  - Particle-in-cell
  - collective beam instabilities

- **Don'ts** (key word: *small data sets*)
  - Processing of small data sets
  - Single particle tracking, optics calculations
  - Heterogeneous tasks

- Since 2012, about ~10 IPAC papers/year are dedicated to GPU acceleration (www.jacow.org)
What should I buy?

- Keep in mind when comparing the performance of different hardware:
  - GPU's currently offer much more core than CPUs, but are not as flexible (Single Instruction, multiple data)
  - For the same price, you usually get less CPU's, but are fully flexible (multiple instructions, multiple data)
What should I buy?

- Keep in mind when comparing the performance of different hardware:
  - GPU's currently offer much more core than CPUs, but are not as flexible (Single Instruction, multiple data)
  - For the same price, you usually get less CPU's, but are fully flexible (multiple instructions, multiple data)
  - Memory bound problems can't be accelerated with multiple CPUs, thanks to the large memory bandwidth, GPU are usually more suited.
What should I buy?

Keep in mind when comparing the performance of different hardware:
- GPU's currently offer much more core than CPUs, but are not as flexible (Single Instruction, multiple data)
- For the same price, you usually get less CPU's, but are fully flexible (multiple instructions, multiple data)
- Memory bound problems can't be accelerated with multiple CPUs, thanks to the large memory bandwidth, GPU are usually more suited

→ Always keep an eye on new technologies... (See e.g. Intel Xeon Phi, a single CPU with up to 72 cores, with vectorization of 8 doubles)
About Moore's Law

The computing power of the most powerful machine in the world does keep increasing!

→ Computer clusters

Source: http://www.top500.org/
The computer cluster

- A set of computers (now called nodes)

Illustration: MichiganTech
The computer cluster

- A set of computers (now called nodes)
- The nodes are connected together with a fast link (→ ~Gb/s)

Illustration: MichiganTech
The computer cluster

- A set of computers (now called nodes)
- The nodes are connected together with a fast link (→ ~Gb/s)
- Only one master node is equipped with an interface → the set of computers acts as a single entity: a Cluster

Illustration: MichiganTech
The computer cluster

- A set of computers (now called nodes)
- The nodes are connected together with a fast link (→ ~Gb/s)
- Only one master node is equipped with an interface → the set of computers acts as a single entity: a Cluster
- Various processes of a program can be run across computers

Illustration: MichiganTech
The computer cluster

- A set of computers (now called nodes)
- The nodes are connected together with a fast link (→ ~Gb/s)
- Only one master node is equipped with an interface → the set of computers acts as a single entity: a Cluster
- Various processes of a program can be run across computers
- The memory (RAM) can no longer be shared → Distributed memory

Illustration: MichiganTech
The computer cluster

- A set of computers (now called nodes)
- The nodes are connected together with a fast link (→ ~Gb/s)
- Only one master node is equipped with an interface → the set of computers acts as a single entity: a Cluster
- Various processes of a program can be run across computers
- The memory (RAM) can no longer be shared → Distributed memory

Illustration: MichiganTech

How can we (programmers) handle this?
Largest clusters

- Piz Daint (Swiss National Supercomputing Centre)
- 361,760 cores
- ~20 Pflop/s
- ~2 MW
- Largest supercomputer in Europe, 6th in the world
Largest clusters

- Piz Daint (Swiss National Supercomputing Centre)
  - 361,760 cores
  - ~20 Pflop/s
  - ~2 MW
  - Largest supercomputer in Europe, 6th in the world

- Summit (Oak ridge National Lab. USA)
  - 2,282,544 cores
  - ~120 Pflop/s
  - ~8 MW
  - Largest supercomputer in the world
Largest clusters

- Piz Daint (Swiss National Supercomputing Centre)
  - 361,760 cores
  - ~20 Pflop/s
  - ~2 MW
  - Largest supercomputer in Europe, 6th in the world

- Summit (Oak ridge National Lab. USA)
  - 2,282,544 cores
  - ~120 Pflop/s
  - ~8 MW
  - Largest supercomputer in the world

Source: [http://www.top500.org/](http://www.top500.org/) (June 2018)
Most playful cluster
Most playful cluster

- Boss, I need 200 Playstation 3 for my research
Most playful cluster

- Boss, I need 200 Playstation 3 for my research
- Sure, no problem!
Most playful cluster

- Boss, I need 200 Playstation 3 for my research
- Sure, no problem!

… said no boss ever?
- Boss, I need 200 Playstation 3 for my research
- Sure, no problem!
… said no boss ever?
The message passing interface

- MPI allows for transfer of memory between nodes
- Each node can be made of several CPUs and GPUs
The message passing interface

- A single code is compiled with a dedicated compiler (mpicc)
- The code is executed by a given number of processes specified by the running command
  - mpirun -np 800 executable_name
The message passing interface

- A single code is compiled with a dedicated compiler (mpicc)
- The code is executed by a given number of processes specified by the running command
  - `mpirun -np 800 executable_name`
- Example on CERN's HPC-BATCH
The message passing interface

- A single code is compiled with a dedicated compiler (mpicc)
- The code is executed by a given number of processes specified by the running command
  - `mpirun -np 800 executable_name`
- Example on CERN's HPC-BATCH
The message passing interface

- A single code is compiled with a dedicated compiler (mpicc)
- The code is executed by a given number of processes specified by the running command
  - `mpirun -np 800 executable_name`
- Example on CERN's HPC-BATCH
The message passing interface

- A single code is compiled with a dedicated compiler (mpicc)
- The code is executed by a given number of processes specified by the running command
  - `mpirun -np 800 executable_name`
- Example on CERN's HPC-BATCH

- MPI defines how these processes can exchange data through the fast link
- Several implementations are available: OpenMPI, IntelMPI, MPICH, MVAPICH2,...
Example: The hard way

```c
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[]){
    Initialisation
    Computing
    Output
}
```
Example: The hard way

```
int main(int argc, char *argv[]){
    int TAGREADY = 0;
    int TAGX = 1;
    int TAGPX = 2;
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    MPI_Init(&argc, &argv);
    int myRank;
    int activeProcs;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &activeProcs);
    int chunkSize = ceil(nPart/activeProcs);
    if(myRank == 0){
        double radius;
        double angle;
        for(int i = 0;i<nPart;++i){
            radius = (double)rand() / RAND_MAX;
            while(radius==0){
                radius = rand();
            }
            radius = sqrt(-2.0*log(radius));
            angle = (double)2.0*M_PI*rand() / RAND_MAX;
            x[i] = radius*sin(angle);
            px[i] = radius*cos(angle);
        }
        for(int iProc=1;iProc<activeProcs;++iProc){
            MPI_Send(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc,TAGX, MPI_COMM_WORLD);
            MPI_Send(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc,TAGPX, MPI_COMM_WORLD);
        }
    }
    else{
        MPI_Status status;
        MPI_Recv(x,chunkSize, MPI_DOUBLE, 0,TAGX, MPI_COMM_WORLD, &status);
        MPI_Recv(px,chunkSize, MPI_DOUBLE, 0,TAGPX, MPI_COMM_WORLD, &status);
    }
}
```
Example: The hard way

```c
int main(int argc, char *argv[])
{
  int TAGREADY = 0;
  int TAGX = 1;
  int TAGPX = 2;
  const int nPart = 500000;
  double x[nPart];
  double px[nPart];

  MPI_Init(&argc, &argv);
  int myRank;
  int activeProcs;
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &activeProcs);
  int chunkSize = ceil(nPart / activeProcs);

  if(myRank == 0){
    double radius;
    double angle;
    for(int i = 0; i < nPart; ++i){
      radius = (double)rand() / RAND_MAX;
      while(radius == 0){
          radius = rand();
      }
      radius = sqrt(-2.0*log(radius));
      angle = (double)2.0*M_PI*rand() / RAND_MAX;
      x[i] = radius*sin(angle);
      px[i] = radius*cos(angle);
    }

    for(int iProc=1; iProc<activeProcs; ++iProc){
      MPI_Send(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGX, MPI_COMM_WORLD);
      MPI_Send(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGPX, MPI_COMM_WORLD);
    }
  } else{
    MPI_Status status;
    MPI_Recv(x,chunkSize, MPI_DOUBLE, 0, TAGX, MPI_COMM_WORLD, &status);
    MPI_Recv(px,chunkSize, MPI_DOUBLE, 0, TAGPX, MPI_COMM_WORLD, &status);
  }
  ..
}
```
Example: The hard way

```c
int main(int argc, char *argv[]){
    int TAGREADY = 0;
    int TAGX = 1;
    int TAGPX = 2;
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    MPI_Init(&argc, &argv);

    int myRank;
    int activeProcs;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &activeProcs);
    int chunkSize = ceil(nPart / activeProcs);

    if (myRank == 0){
        double radius;
        double angle;
        for (int i = 0; i < nPart; ++i){
            radius = (double)rand() / RAND_MAX;
            while (radius == 0){
                radius = rand();
            }
            radius = sqrt(-2.0*log(radius));
            angle = (double)2.0*M_PI*rand() / RAND_MAX;
            x[i] = radius*sin(angle);
            px[i] = radius*cos(angle);
        }

        for (int iProc = 1; iProc < activeProcs; ++iProc){
            MPI_Send(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGX, MPI_COMM_WORLD);
            MPI_Send(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGPX, MPI_COMM_WORLD);
        }
    } else{
        MPI_Status status;
        MPI_Recv(x, chunkSize, MPI_DOUBLE, 0, TAGX, MPI_COMM_WORLD, &status);
        MPI_Recv(px, chunkSize, MPI_DOUBLE, 0, TAGPX, MPI_COMM_WORLD, &status);
    }
}
```

Each process is started with a different `rank`
Example: The hard way

```
int main(int argc, char *argv[]){
    int TAGREADY = 0;
    int TAGX = 1;
    int TAGPX = 2;
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];

    MPI_Init(&argc, &argv);
    int myRank;
    int activeProcs;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &activeProcs);
    int chunkSize = ceil(nPart/activeProcs);

    if(myRank == 0){
        double radius;
        double angle;
        for(int i = 0;i<nPart;++i){
            radius = (double)rand() / RAND_MAX;
            while(radius==0){
                radius = rand();
            }
            radius = sqrt(-2.0*log(radius));
            angle = (double)2.0*M_PI*rand() / RAND_MAX;
            x[i] = radius*sin(angle);
            px[i] = radius*cos(angle);
        }

        for(int iProc=1;iProc<activeProcs;++iProc){
            MPI_Send(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc,TAGX, MPI_COMM_WORLD);
            MPI_Send(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc,TAGPX, MPI_COMM_WORLD);
        } else{
            MPI_Status status;
            MPI_Recv(x,chunkSize, MPI_DOUBLE, 0,TAGX, MPI_COMM_WORLD, &status);
            MPI_Recv(px,chunkSize, MPI_DOUBLE, 0,TAGPX, MPI_COMM_WORLD, &status);
        }
    ..
```
Example: The hard way

```c
int main(int argc, char *argv[]){
    int TAGREADY = 0;
    int TAGX = 1;
    int TAGPX = 2;
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    MPI_Init(&argc, &argv);
    int myRank;
    int activeProcs;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &activeProcs);
    int chunkSize = ceil(nPart / activeProcs);
    if(myRank == 0){
        double radius;
        double angle;
        for(int i = 0; i < nPart; ++i){
            radius = (double)rand() / RAND_MAX;
            while(radius == 0){
                radius = rand();
            }
            radius = sqrt(-2.0*log(radius));
            angle = (double)2.0*M_PI*rand() / RAND_MAX;
            x[i] = radius*sin(angle);
            px[i] = radius*cos(angle);
        }
        for(int iProc = 1; iProc < activeProcs; ++iProc){
            MPI_Send(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGX, MPI_COMM_WORLD);
            MPI_Send(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGPX, MPI_COMM_WORLD);
        }
    } else{
        MPI_Status status;
        MPI_Recv(x, chunkSize, MPI_DOUBLE, 0, TAGX, MPI_COMM_WORLD, &status);
        MPI_Recv(px, chunkSize, MPI_DOUBLE, 0, TAGPX, MPI_COMM_WORLD, &status);
    }
}
```

Initialisation

Each process is started with a different rank

We'll share the load between processes
Example: The hard way

```c
int main(int argc, char *argv){
    int TAGREADY = 0;
    int TAGX = 1;
    int TAGPX = 2;
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    MPI_Init(&argc, &argv);
    int myRank;
    int activeProcs;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &activeProcs);
    int chunkSize = ceil(nPart / activeProcs);
    if(myRank == 0){
        double radius;
        double angle;
        for(int i = 0;i<nPart;++i){
            radius = (double)rand() / RAND_MAX;
            while(radius==0){
                radius = rand();
            }
            radius = sqrt(-2.0*log(radius));
            angle = (double)2.0*M_PI*rand() / RAND_MAX;
            x[i] = radius*sin(angle);
            px[i] = radius*cos(angle);
        }
        for(int iProc=1;iProc<activeProcs;++iProc){
            MPI_Send(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc,TAGX, MPI_COMM_WORLD);
            MPI_Send(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc,TAGPX, MPI_COMM_WORLD);
        }
    } else{
        MPI_Status status;
        MPI_Recv(x,chunkSize, MPI_DOUBLE, 0,TAGX, MPI_COMM_WORLD, &status);
        MPI_Recv(px,chunkSize, MPI_DOUBLE, 0,TAGPX, MPI_COMM_WORLD, &status);
    }
    ..
}
```

Initialisation

- Each process is started with a different rank
- We'll share the load between processes

Here only the process with rank 0 does the initialization
Example: The hard way

```c
int main(int argc, char *argv[]){
    int TAGREADY = 0;
    int TAGX = 1;
    int TAGPX = 2;
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
   (MPI_Init(&argc, &argv);
    int myRank;
    int activeProcs;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &activeProcs);
    int chunkSize = ceil(nPart/activeProcs);
    if(myRank == 0){
        double radius;
        double angle;
        for(int i = 0;i<nPart;++i){
            radius = (double)rand() / RAND_MAX;
            while(radius == 0){
                radius = rand();
            }
            radius = sqrt(-2.0*log(radius));
            angle = (double)2.0*M_PI*rand() / RAND_MAX;
            x[i] = radius*sin(angle);
            px[i] = radius*cos(angle);
        }
        for(int iProc=1;iProc<activeProcs;++iProc){
            MPI_Send(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGX, MPI_COMM_WORLD);
            MPI_Send(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGPX, MPI_COMM_WORLD);
        }
    } else{
        MPI_Status status;
        MPI_Recv(x,chunkSize, MPI_DOUBLE, 0,TAGX, MPI_COMM_WORLD, &status);
        MPI_Recv(px,chunkSize, MPI_DOUBLE, 0,TAGPX, MPI_COMM_WORLD, &status);
    }
}
```

Initialisation

Each process is started with a different rank
We'll share the load between processes

Here only the process with rank 0 does the initialization
Then it sends the corresponding data (chunk of the full set of particles coordinates) to other processes
Example: The hard way

```c
int main(int argc, char *argv[]){
    int TAGREADY = 0;
    int TAGX = 1;
    int TAGPX = 2;
    const int nPart = 500000;
    double x[nPart];
    double px[nPart];
    MPI_Init(&argc, &argv);
    int myRank;
    int activeProcs;
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &activeProcs);
    int chunkSize = ceil(nPart / activeProcs);
    if (myRank == 0){
        double radius;
        double angle;
        for (int i = 0; i < nPart; ++i){
            radius = (double)rand() / RAND_MAX;
            while (radius == 0){
                radius = rand();
            }
            radius = sqrt(-2.0*log(radius));
            angle = (double)2.0*M_PI*rand() / RAND_MAX;
            x[i] = radius*sin(angle);
            px[i] = radius*cos(angle);
        }
        for (int iProc=1; iProc<activeProcs; ++iProc){
            MPI_Send(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGX, MPI_COMM_WORLD);
            MPI_Send(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, iProc, TAGPX, MPI_COMM_WORLD);
        }
    } else{
        MPI_Status status;
        MPI_Recv(x,chunkSize, MPI_DOUBLE, 0, TAGX, MPI_COMM_WORLD, &status);  // In the mean time, the other processes wait to receive the data
        MPI_Recv(px,chunkSize, MPI_DOUBLE, 0, TAGPX, MPI_COMM_WORLD, &status);
    }
}
```

---

**Initialisation**

- Each process is started with a different `rank`
- We'll share the load between processes

---

**Here only the process with rank 0 does the initialization**

Then it sends the corresponding data (chunk of the full set of particles coordinates) to other processes

---

In the mean time, the other processes wait to receive the data
Example: The hard way

Computing

... int nTurn = 10000;
    double sinPhix = sin(2.0*M_PI*0.31);
    double cosPhix = cos(2.0*M_PI*0.31);
    double scale = 1.0;
    double thres = 1E-10;

double oldX = 0.0;
    double oldPx = 0.0;
    for(int turn = 0; turn < nTurn; ++turn) {
        for(int i = 0; i < chunkSize; ++i) {
            oldX = x[i];
            oldPx = px[i];
            x[i] = cosPhix*oldX + sinPhix*oldPx;
            px[i] = -sinPhix*oldX + cosPhix*oldPx;
            if (abs(x[i]) > thres) {
                px[i] += scale*(1.0-exp(-0.5*x[i]*x[i]))/x[i];
            }
        }
    }
...
Example: The hard way

Output

...  
if(myRank==0){
   int nRecv = 1;
   while(nRecv<activeProcs){
      MPI_Status status;
      MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, TAGREADY, MPI_COMM_WORLD, &status);
      int iProc = status.MPI_SOURCE;
      MPI_Recv(x[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE, TAGX, MPI_COMM_WORLD, &status);
      MPI_Recv(px[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE, TAGPX, MPI_COMM_WORLD, &status);
      ++nRecv;
   }
} else{
   MPI_Send(NULL, 0, MPI_BYTE, 0, TAGREADY, MPI_COMM_WORLD);
   MPI_Send(x,chunkSize, MPI_DOUBLE, 0, TAGX, MPI_COMM_WORLD);
   MPI_Send(px,chunkSize, MPI_DOUBLE, 0, TAGPX, MPI_COMM_WORLD);
}  
MPI_Finalize();  
...
Example: The hard way

```c
...
if(myRank==0){
    int nRecv = 1;
    while(nRecv<activeProcs){
        MPI_Status status;
        MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,TAGREADY, MPI_COMM_WORLD,&status);
        int iProc = status.MPI_SOURCE;
        MPI_Recv(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE,TAGX, MPI_COMM_WORLD,&status);
        MPI_Recv(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE,TAGPX, MPI_COMM_WORLD,&status);
        ++nRecv;
    }
} else{
    MPI_Send(NULL, 0, MPI_BYTE, 0,TAGREADY, MPI_COMM_WORLD);
    MPI_Send(x,chunkSize, MPI_DOUBLE, 0,TAGX, MPI_COMM_WORLD);
    MPI_Send(px,chunkSize, MPI_DOUBLE, 0,TAGPX, MPI_COMM_WORLD);
}
MPI_Finalize();
...
```

Output

The process with rank 0 collects back the data from other processes
Example: The hard way

Output

```
... if(myRank==0){
    int nRecv = 1;
    while(nRecv<activeProcs){
        MPI_Status status;
        MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,TAGREADY, MPI_COMM_WORLD,&status);
        int iProc = status.MPI_SOURCE;
        MPI_Recv(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE,TAGX, MPI_COMM_WORLD,&status);
        MPI_Recv(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE,TAGPX, MPI_COMM_WORLD,&status);
        ++nRecv;
    }
} else{
    MPI_Send(NULL, 0, MPI_BYTE, 0,TAGREADY, MPI_COMM_WORLD);
    MPI_Send(x,chunkSize, MPI_DOUBLE, 0,TAGX, MPI_COMM_WORLD);
    MPI_Send(px,chunkSize, MPIDOUBLE, 0,TAGPX, MPI_COMM_WORLD);
}
MPI_Finalize();
...```

The process with rank 0 collects back the data from other processes.

When finished, the other processes send their data to the process with rank 0.
Example: The hard way

Output

... if (myRank == 0) {
    int nRecv = 1;
    while (nRecv < activeProcs) {
        MPI_Status status;
        int iProc = status.MPI_SOURCE;
        MPI_Recv(&x[iProc * chunkSize], chunkSize, MPI_DOUBLE,
                  status.MPI_SOURCE, TAGX, MPI_COMM_WORLD, &status);
        MPI_Recv(&px[iProc * chunkSize], chunkSize, MPI_DOUBLE,
                  status.MPI_SOURCE, TAGPX, MPI_COMM_WORLD, &status);
        ++nRecv;
    }
} else {
    MPI_Send(NULL, 0, MPI_BYTE, 0, TAGREADY, MPI_COMM_WORLD);
    MPI_Send(x, chunkSize, MPI_DOUBLE, 0, TAGX, MPI_COMM_WORLD);
    MPI_Send(px, chunkSize, MPI_DOUBLE, 0, TAGPX, MPI_COMM_WORLD);
}
MPI_Finalize();
...

The process with rank 0 collects back the data from other processes

When finished, the other processes send their data to the process with rank 0
Example: The hard way

Output

... if(myRank==0){
    int nRecv = 1;
    while(nRecv<activeProcs){
        MPI_Status status;
        MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, TAGREADY, MPI_COMM_WORLD, &status);
        int iProc = status.MPI_SOURCE;
        MPI_Recv(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE, TAGX, MPI_COMM_WORLD, &status);
        MPI_Recv(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE, TAGPX, MPI_COMM_WORLD, &status);
        ++nRecv;
    }
} else{
    MPI_Send(NULL, 0, MPI_BYTE, 0, TAGREADY, MPI_COMM_WORLD);
    MPI_Send(x, chunkSize, MPI_DOUBLE, 0, TAGX, MPI_COMM_WORLD);
    MPI_Send(px, chunkSize, MPI_DOUBLE, 0, TAGPX, MPI_COMM_WORLD);
}
MPI_Finalize();
...

The process with rank 0 collects back the data from other processes.

When finished, the other processes send their data to the process with rank 0.
Example: The hard way

Output

```c
...
if(myRank==0){
    int nRecv = 1;
    while(nRecv<activeProcs){
        MPI_Status status;
        MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,TAGREADY,
                 MPI_COMM_WORLD,&status);
        int iProc = status.MPI_SOURCE;
        MPI_Recv(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE,
                 status.MPI_SOURCE,TAGX, MPI_COMM_WORLD,&status);
        MPI_Recv(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE,
                 status.MPI_SOURCE,TAGPX, MPI_COMM_WORLD,&status);
        ++nRecv;
    }
} else{
    MPI_Send(NULL, 0, MPI_BYTE, 0, TAGREADY, MPI_COMM_WORLD);
    MPI_Send(x,chunkSize, MPI_DOUBLE, 0, TAGX, MPI_COMM_WORLD);
    MPI_Send(px,chunkSize, MPI_DOUBLE, 0, TAGPX, MPI_COMM_WORLD);
}
MPI_Finalize();
...
```

The process with rank 0 collects back the data from other processes

When finished, the other processes send their data to the process with rank 0.
Example: The hard way

```
... if(myRank==0){
  int nRecv = 1;
  while(nRecv<activeProcs){
    MPI_Status status;
    MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,TAGREADY,
              MPI_COMM_WORLD,&status);
    int iProc = status.MPI_SOURCE;
    MPI_Recv(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE,
              status.MPI_SOURCE,TAGX, MPI_COMM_WORLD,&status);
    MPI_Recv(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE,
              status.MPI_SOURCE,TAGPX, MPI_COMM_WORLD,&status);
    ++nRecv;
  }
} else{
  MPI_Send(NULL, 0, MPI_BYTE, 0,TAGREADY, MPI_COMM_WORLD);
  MPI_Send(x,chunkSize, MPIDOUBLE, 0,TAGX, MPI_COMM_WORLD);
  MPI_Send(px,chunkSize, MPI_DOUBLE, 0,TAGPX, MPI_COMM_WORLD);
}
MPI_Finalize();
...
```

- It is your responsibility to make sure that the data send finds a receiver
  → Avoid deadlocks!

Output

The process with rank 0 collects back the data from other processes

When finished, the other processes send their data to the process with rank 0
**Example: The hard way**

```c
if(myRank==0){
  int nRecv = 1;
  while(nRecv<activeProcs){
    MPI_Status status;
    MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,TAGREADY, MPI_COMM_WORLD,&status);
    int iProc = status.MPI_SOURCE;
    MPI_Recv(&x[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE,TAGX, MPI_COMM_WORLD,&status);
    MPI_Recv(&px[iProc*chunkSize], chunkSize, MPI_DOUBLE, status.MPI_SOURCE,TAGPX, MPI_COMM_WORLD,&status);
    ++nRecv;
  }
} else{
  MPI_Send(NULL, 0, MPI_BYTE, 0,TAGREADY, MPI_COMM_WORLD);
  MPI_Send(x,chunkSize, MPI_DOUBLE, 0,TAGX, MPI_COMM_WORLD);
  MPI_Send(px,chunkSize, MPI_DOUBLE, 0,TAGPX, MPI_COMM_WORLD);
}
MPI_Finalize();
```

- It is your responsibility to make sure that the data send finds a receiver
  → Avoid deadlocks!
- The data type as well as the tag (user defined int) avoids miss-deliveries...

```
int TAGREADY = 0;
int TAGX = 1;
int TAGPX = 2;
```
Example: The easy way

Not that the road is difficult, is that what is difficult is the way.

- Søren Kierkegaard
Operating system

(Among the top 500)

UNIX

Linux
Operating system

In a world without walls and fences, who needs windows and gates?
- Dino Esposito
Operating system

In a world without walls and fences, who needs windows and gates?
- Dino Esposito

If you want to harness this power, you'll probably need to go out of your comfort zone
- Fuego, Chief advisory rabbit
Batch processing

- Computer clusters are usually shared resources
- A scheduler assigns jobs to resources, based on job descriptions given by the user
  - In order to efficiently allocate time for the various jobs, the scheduler usually asks for quite some details about your jobs → Not for Voodoo programmers!
Batch processing

- Computer clusters are usually shared resources
- A scheduler assigns jobs to resources, based on job descriptions given by the user
  - In order to efficiently allocate time for the various jobs, the scheduler usually asks for quite some details about your jobs → Not for Voodoo programmers!

Example with Slurm:

```bash
#!/bin/bash
#SBATCH --nodes 20
#SBATCH --tasks-per-node 40
#SBATCH --cpus-per-task 1
#SBATCH --time 00:10:00
#SBATCH --mail-user xbuffat@cern.ch

module load mpi/mvapich2/2.3rc2

srun executable
```
Batch processing

- Computer clusters are usually shared resources
- A scheduler assigns jobs to resources, based on job descriptions given by the user
  - In order to efficiently allocate time for the various jobs, the scheduler usually asks for quite some details about your jobs → Not for Voodoo programmers!

Example with Slurm:

```
#!/bin/bash

#SBATCH --nodes 20
#SBATCH --tasks-per-node 40
#SBATCH --cpus-per-task 1
#SBATCH --time 00:10:00
#SBATCH --mail-user xbuffat@cern.ch

module load mpi/mvapich2/2.3rc2

srun executable
```

Total of 800 processes without multithreading
Batch processing

- Computer clusters are usually shared resources
- A scheduler assigns jobs to resources, based on job descriptions given by the user
  - In order to efficiently allocate time for the various jobs, the scheduler usually asks for quite some details about your jobs → Not for Voodoo programmers!

Example with Slurm:

```bash
#!/bin/bash
#SBATCH --nodes 20
#SBATCH --tasks-per-node 40
#SBATCH --cpus-per-task 1
#SBATCH --time 00:10:00
#SBATCH --mail-user xbuffat@cern.ch

module load mpi/mvapich2/2.3rc2

srun executable
```

Total of 800 processes without multithreading
Maximum execution time
Applications

- MPI is used for every application that requires more than CPUs or GPUs than a single node can provide
  - Affordable shared memory system (multithreading) are limited to tens to hundreds of CPUs
  - Motherboards typically come with few GPU slots (fashionable for bitcoin mining)
- MPI can be using in combination with shared memory parallelism (e.g. OpenMP) and multiple GPUs
- Large memory transfer between node can become a limitation
  → Data locality is again a key for efficient parallelisation
- In accelerators:
  - Plasma simulations (particle sources)
  - Collective beam instabilities (space charge, electron clouds, beam-beam interactions, wake fields)
  - Safety (fire propagation simulation)
  - ...
Volunteer computing


VOLUNTEER COMPUTING

- Heterogeneous resources
- Non-dedicated resources
- Geographically dispersed
- Internet services
- No need for cooling systems
Volunteer computing


- Significant overhead w.r.t. dedicated machines (portability, compatibility, tails, ...)

Heterogeneous resources
Non-dedicated resources
Geographically dispersed
Internet services
No need for cooling systems
Volunteer computing


- Significant overhead w.r.t. dedicated machines (portability, compatibility, tails, ...)
- The connection through the Internet is too slow for MPI
  - A system (commonly BOINC) distributes **fully independent** jobs to idling computers and collects output files
  - Appropriate only for fully independent tasks
Volunteer computing

- The largest volunteer computing project, SETI@home reached hundreds of Tflop/s

- This performance used to exceed those of computer clusters, now it is ~3 orders of magnitude less than the fastest cluster
Volunteer computing

- The largest volunteer computing project, SETI@home reached hundreds of Tflop/s
- This performance used to exceed those of computer clusters, now it is ~3 orders of magnitude less than the fastest cluster

- LHC@home contributed significantly to the understanding of non-linear dynamics in the LHC and HL-LHC
- This project is still running, volunteer! http://lhcathome.web.cern.ch/
Volunteer computing

- The largest volunteer computing project, SETI@home reached hundreds of Tflop/s
- This performance used to exceed those of computer clusters, now it is ~3 orders of magnitude less than the fastest cluster

→ Nowadays the development, operational and maintenance costs of a volunteer computing project often exceeds the price of a computer clusters

- LHC@home contributed significantly to the understanding of non-linear dynamics in the LHC and HL-LHC
- This project is still running, volunteer! http://lhcathome.web.cern.ch/
Grid computing

- When a single cluster is not enough (computing, storage or even budget wise)!
  - The load is shared among distributed (but dedicated) systems using slow connections through the Internet
- The jobs sent through to the different systems are independent
  - Within each cluster, the job can be accelerated using multiple CPUs and GPUs
Summary

• Computing techniques evolve as the technology advances
  – Nowadays the goes towards several computing units working in parallel
    → Vectorisation, multithreading, hyperthreading, GPU acceleration

• Programming languages also adapt to these technologies, with various levels of complexity and efficiency
  → In C: Compiler auto-vectorisation, OpenMP, OpenACC, CUDA
  → Several task can be accelerated by a novice, just use the tools made available to you (compiler options, pragmas, accelerated libraries)

• Most of our problems can be solved with a single computer, but some applications rely on larger infrastructures
  – Computer clusters
  – Computing grids

• The choice of technology needed depends
  – On the computing requirements
  – On the memory requirements
  – Communications requirement (data locality!)
... just avoid that: