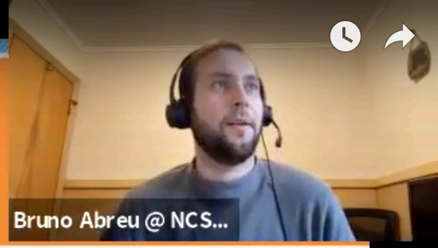
Outline



Part A: **Parallel Computing Overview** (30 min)

- Why parallelism?
- Task and Data parallelism
- Parallel computers
- Threads, processes and more

Part B: **How to Parallelize a Code** (60 min)

Hello, Parallel World!
- Parallel regions
- Compilation and execution

Task Parallelism
- Parallel sections
- Measuring performance

Data Parallelism
- Parallel loops
- Performance and scaling

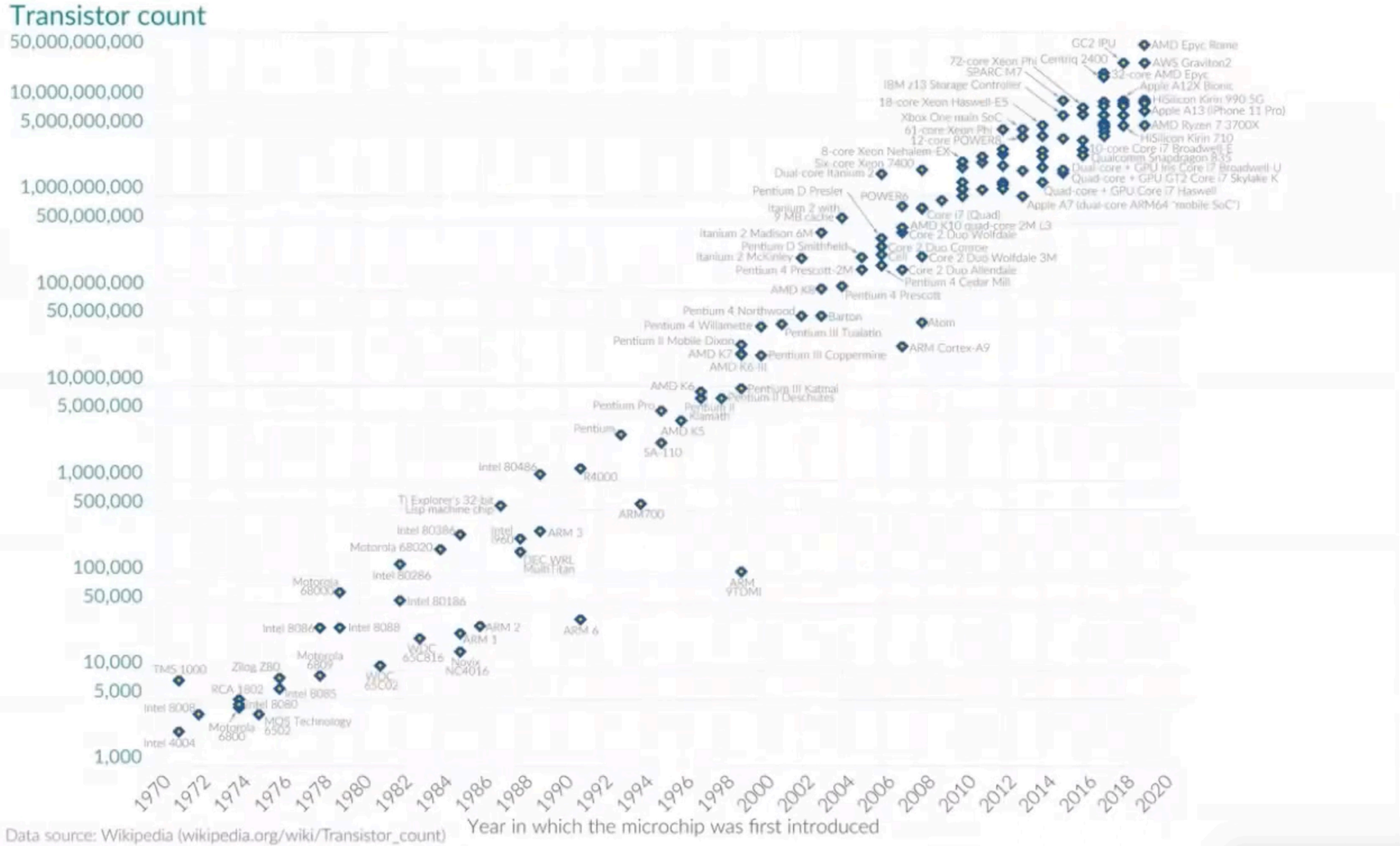
ILLINOIS NCSA

4:39 / 1:45:40 • Intro ›
Scroll for details
2021-11-17 10:30:29

Parallelism: why bother?

Moore's Law (1965)

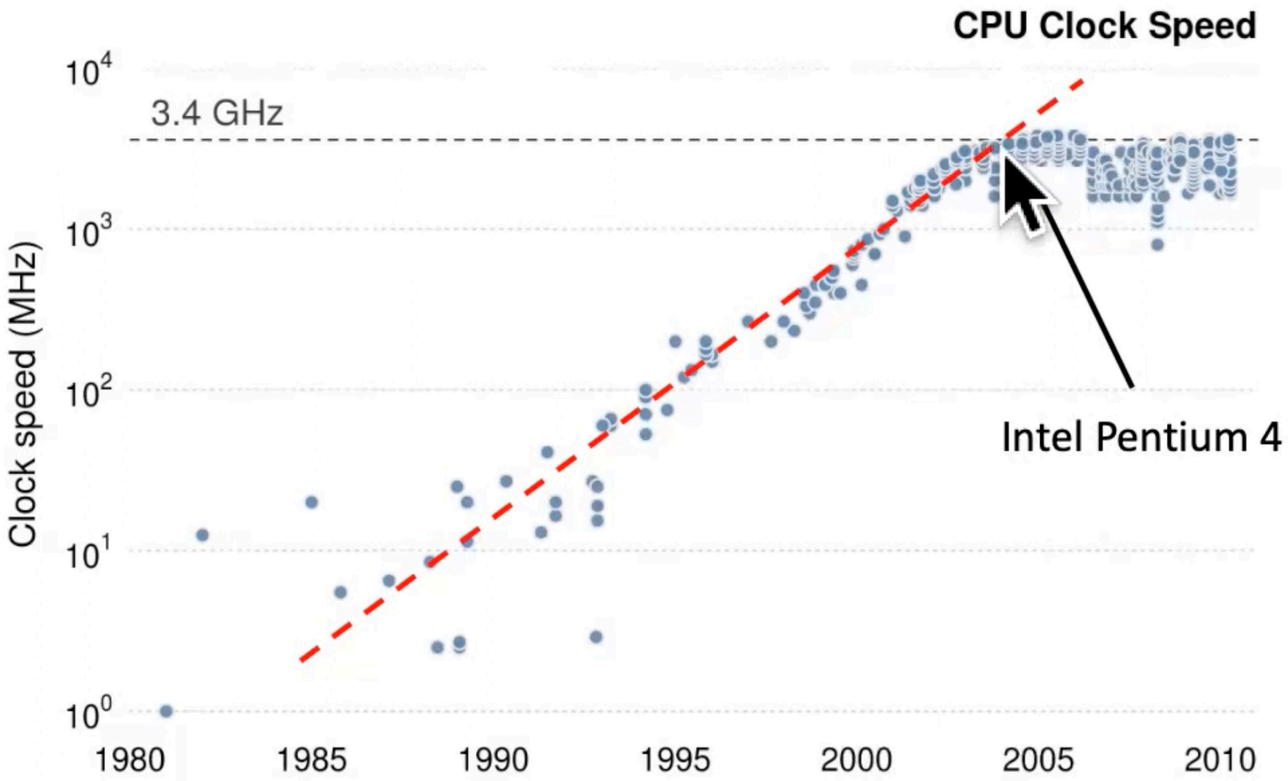
"The number of transistors in a dense integrated circuit doubles every two years"



https://ourworldindata.org/grapher/transistors-per-microprocessor

Parallelism: why bother?

Number of cycles a CPU executes per second



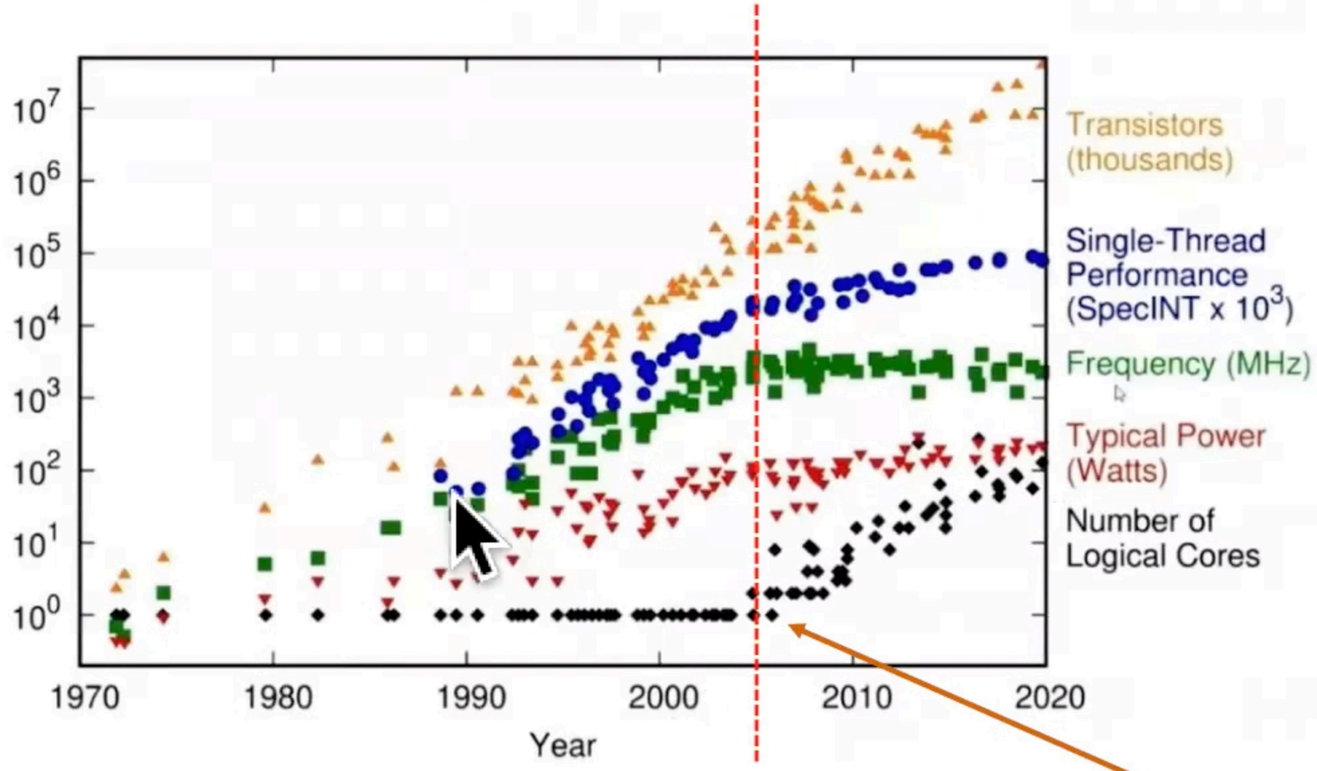
The speed at which a single CPU can process instructions has remained the same since the early 2000s!

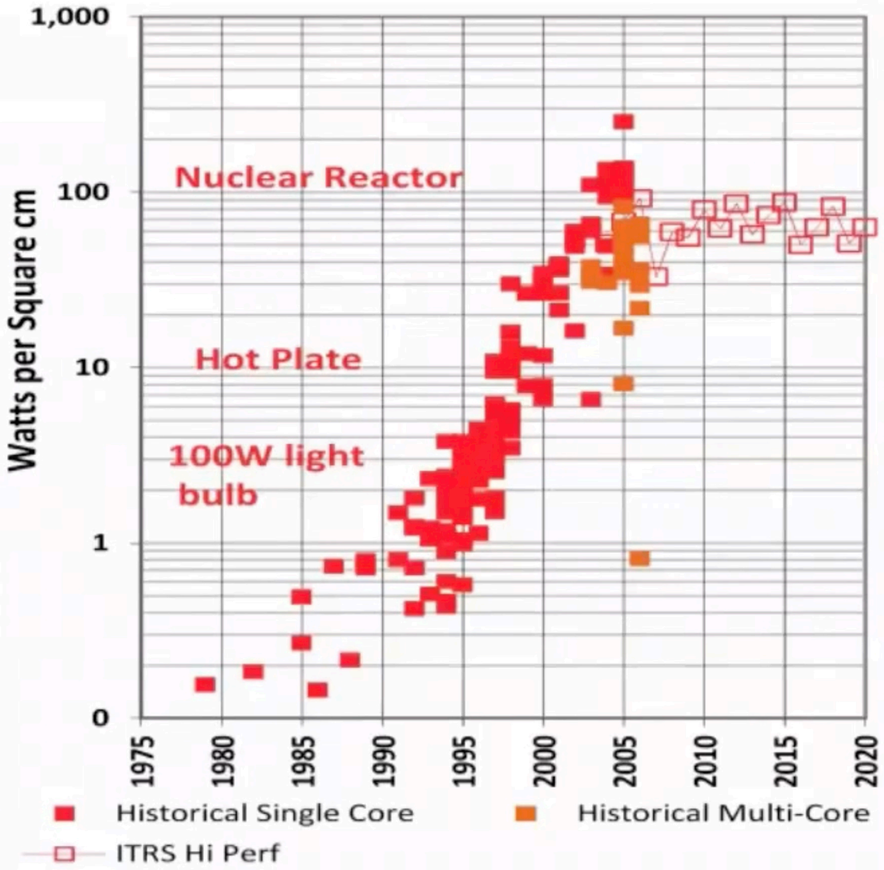
This one was sold ~ 1 year ago for $80

2021-11-17 10:25:52

Parallelism: the shift to multi-core architectures

48 years of Microprocessors Trend Data





https://zenodo.org/record/3947824#.YYrKHlNMH0p

Onset of multi-core architecture

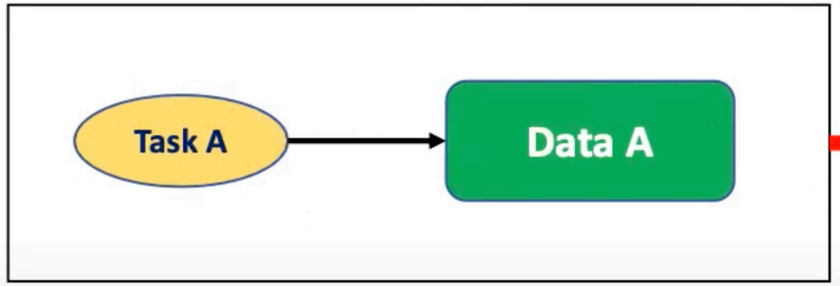
2021-11-17 10:27:10

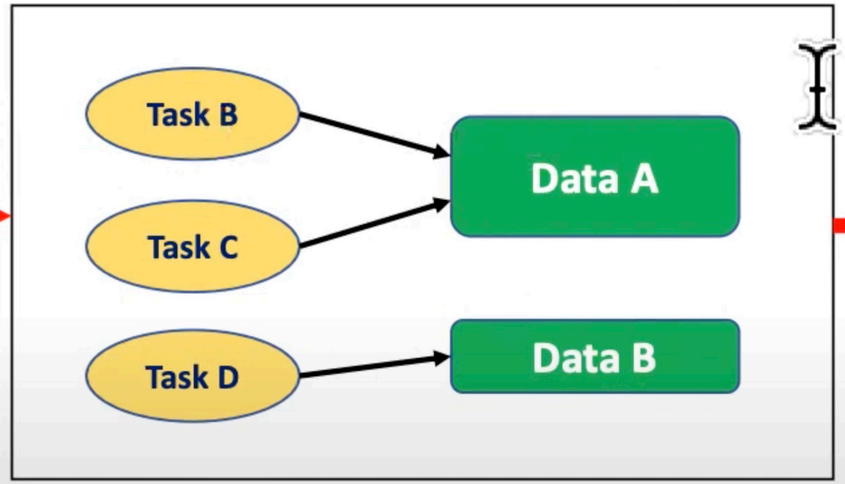
Computer Programs: Instructions blocks

A computer program is a sequence of **blocks of instructions**:

"Do this on that, and then..."

TASKS DATA





Instructions within a block *can* be executed in **parallel**

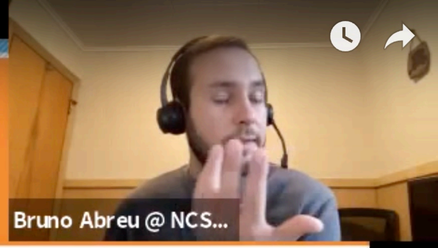
...

Different blocks are executed **sequentially**

ILLINOIS NCSA

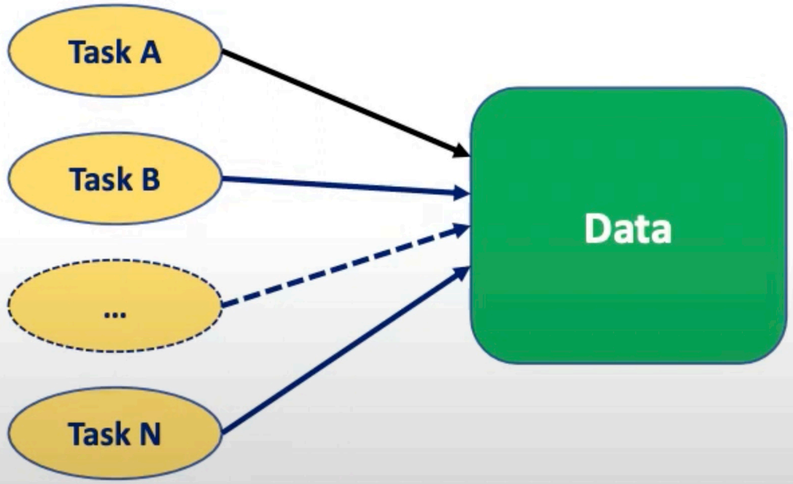
13:31 / 1:45:40 • How a Program Works ›
Scroll for details
2021-11-17 10:33:15

Computer Programs: Data & Task parallelism



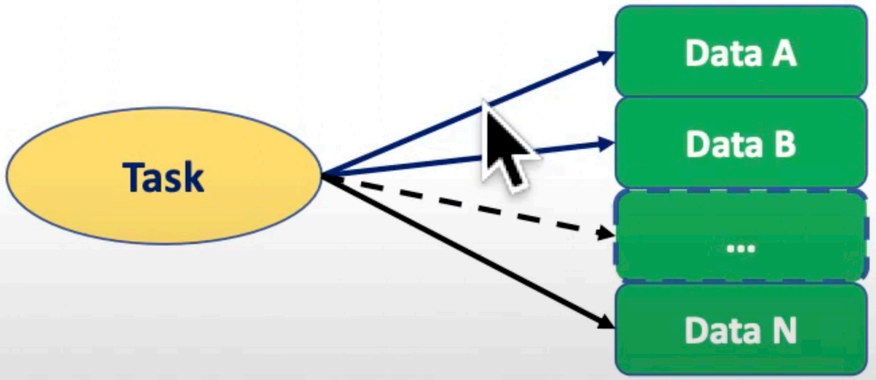
- An instruction is a pair (TASK, DATA)
- There are two "pure" ways of executing instructions in parallel:

Task Parallelism



Execute different tasks over the same data

Data Parallelism



Execute the same task over different chunks of data

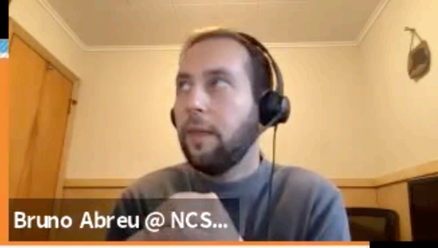
14:20 / 1:45:40 • Types of Parallelization ›
Scroll for details
2021-11-17 10:30:08

Performance: Terminology

Jargon	Description
Peak performance	Maximum speed at which the computer can operate
Clock rate (GHz)	Unit of measure of a processor's speed, reflecting the natural clock rate at which it can operate
Computer cycle	Shortest time in which a unit of work can be performed
Instructions per second	How quickly the computer can issue instructions (e.g. memory reads/writes, logical operations, floating point operations, integer operations, branch instructions)
FLOPS	Floating point operations per second (e.g. subtract, add, divide, multiply)
Speedup	Measures the benefit of parallelism, showing how a program scales as more processors are employed
Benchmark	Used to rate and compare performance of parallel computers (see Top500 Supercomputers)

15:01 / 1:45:40 • Terminology ›
Scroll for details
2021-11-17 10:30:44

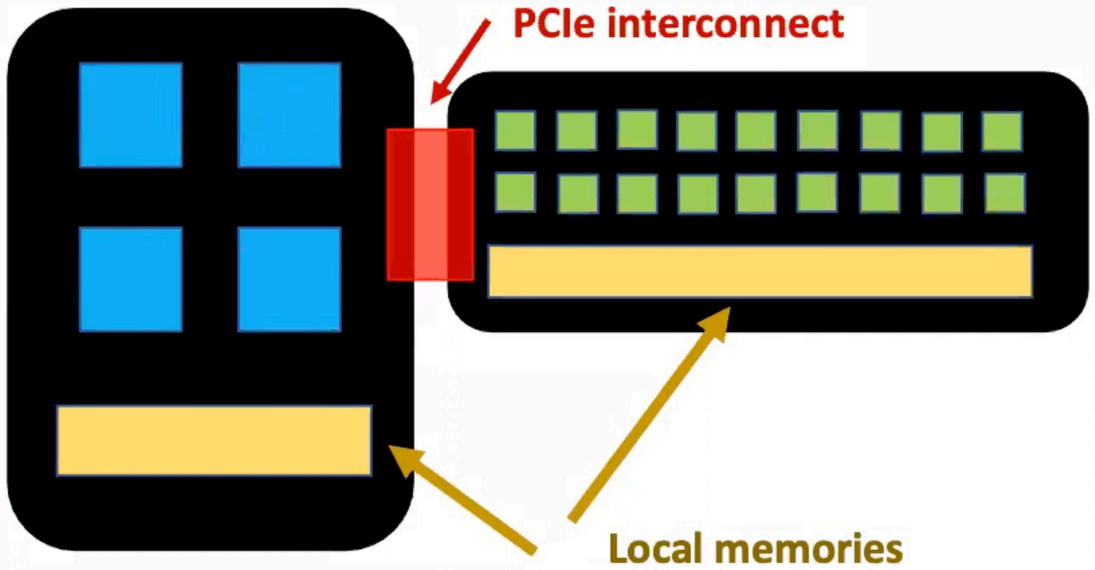
Parallel Computers: Processing Units

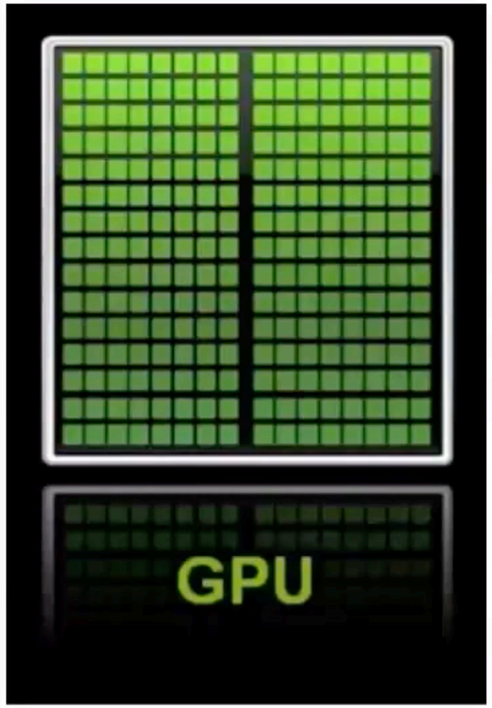


Moderate number of powerful processors

Large number of moderate processors

The usual setup these days…

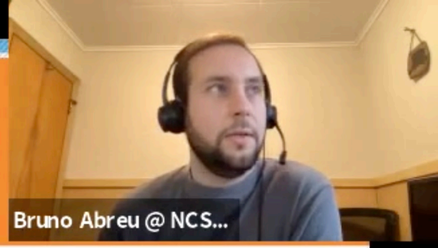




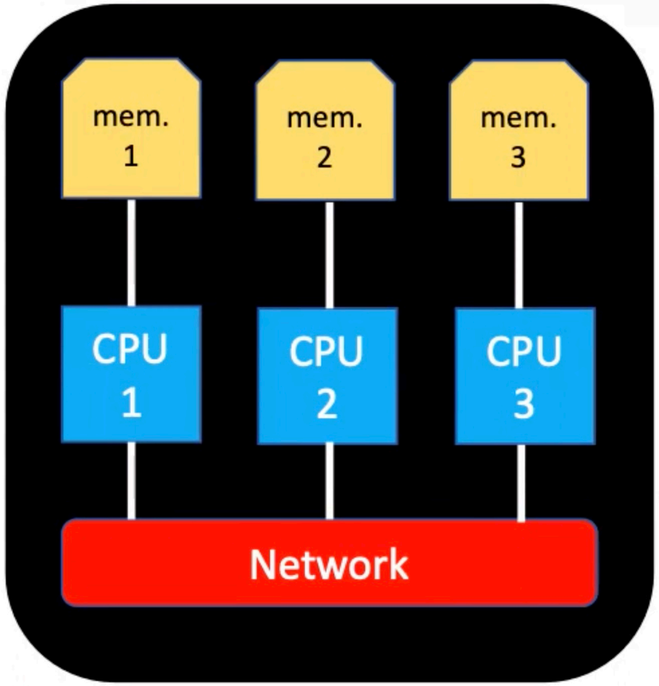
https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

2021-11-17 10:36:07

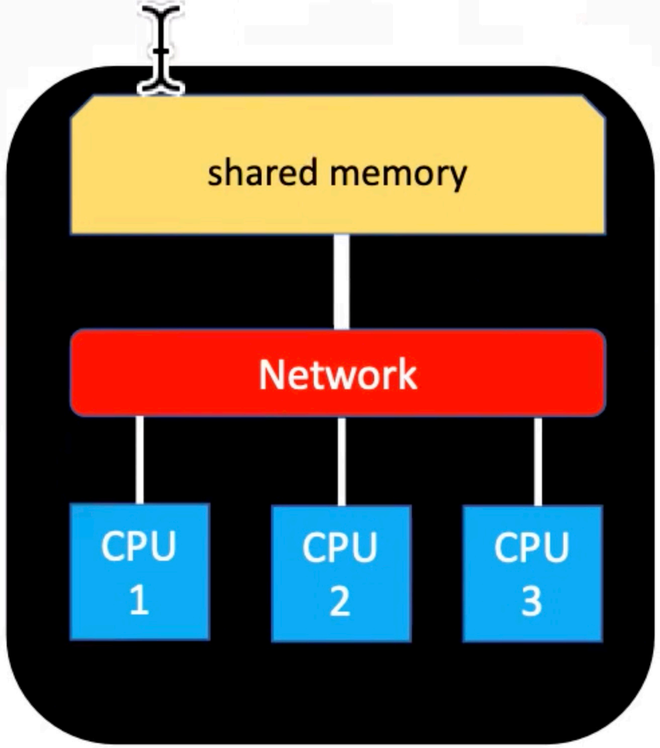
Parallel Computers: Memory



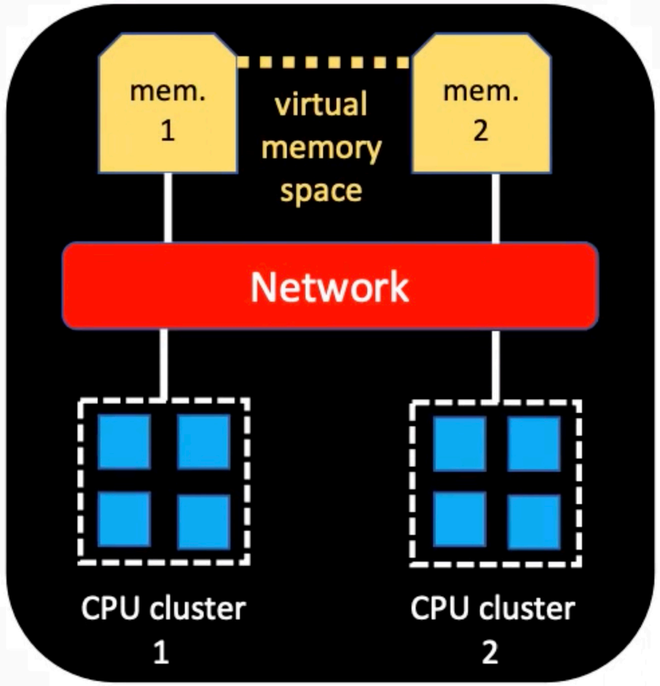
Distributed Memory



Shared Memory

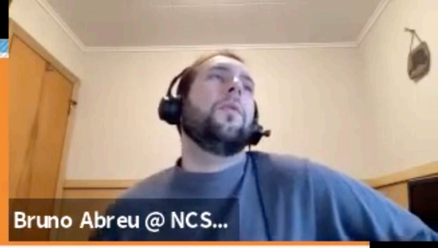


Distributed Shared Memory



2021-11-17 10:37:55

Parallel Computers: Flow of Control



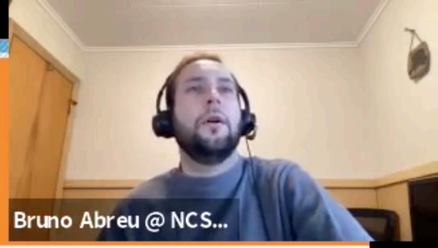
Flynn's Taxonomy

Describes how streams of **instructions** interact with streams of **data**

Instruction Streams	Data Streams: Single	Data Streams: Multiple
Single	SISD	SIMD
Multiple	MISD	MIMD

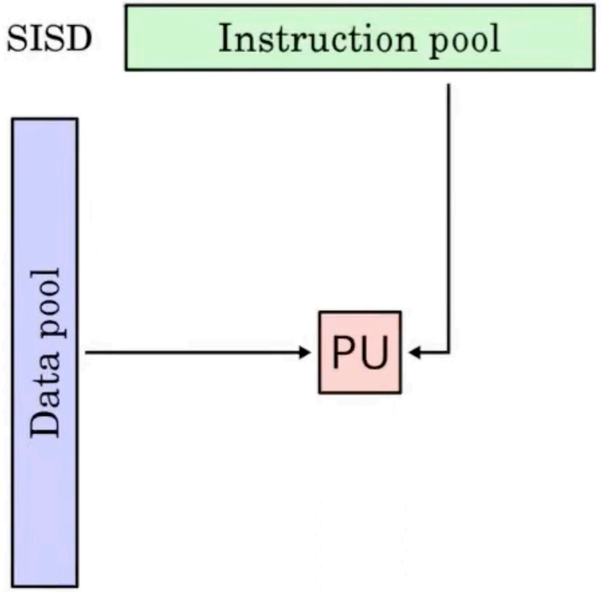
2021-11-17 10:40:13

Parallel Computers: Flow of Control



Flynn's Taxonomy

Describes how streams of **instructions** interact with streams of **data**



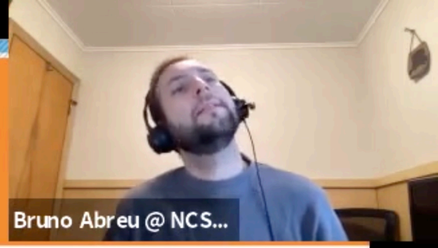
Instruction Streams \ Data Streams	Single	Multiple
Single	SISD	SIMD
Multiple	MISD	MIMD

SISD is naturally not parallelizable…

- Personal computers until ~2010

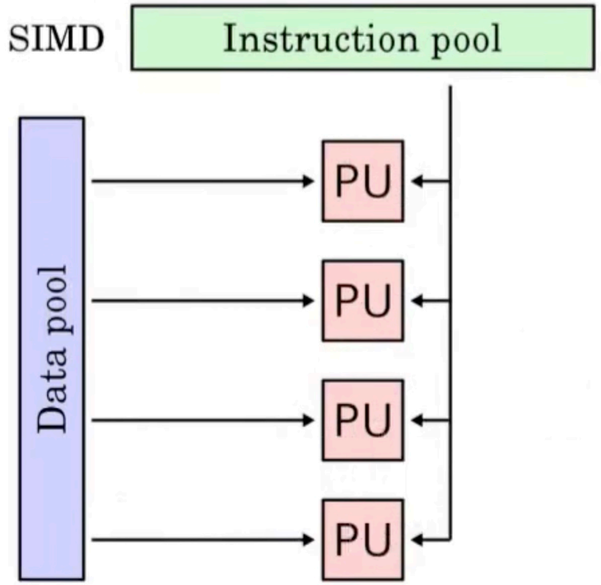
2021-11-17 10:40:35

Parallel Computers: Flow of Control



Flynn's Taxonomy

Describes how streams of **instructions** interact with streams of **data**



Instruction Streams	Data Streams: Single	Data Streams: Multiple
Single	SISD	SIMD
Multiple	MISD	MIMD

SIMD is parallelizable: it can **scale**

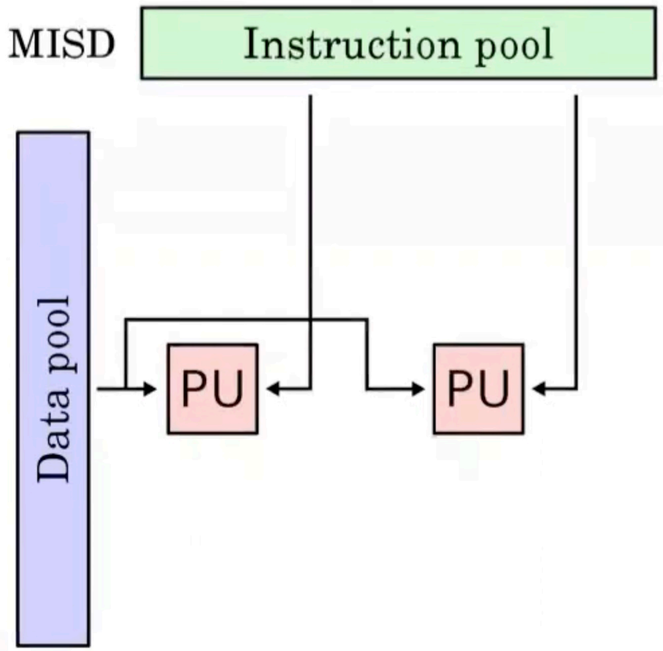
- Array (vector) processors: AVX (e.g. Intel Knights Landing, AMD Bulldozer)

2021-11-17 10:41:15

Parallel Computers: Flow of Control

Flynn's Taxonomy

Describes how streams of **instructions** interact with streams of **data**



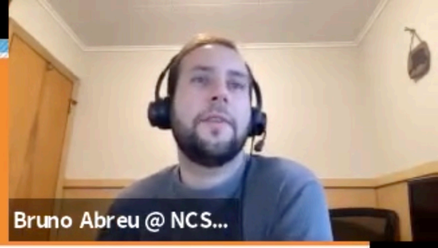
Instruction Streams \ Data Streams	Single	Multiple
Single	SISD	SIMD
Multiple	MISD	MIMD

MISD is parallelizable: it can **scale**, but it is not very common

- Generally used for fault tolerance (e.g. NASA Space Shuttle flight control computer)

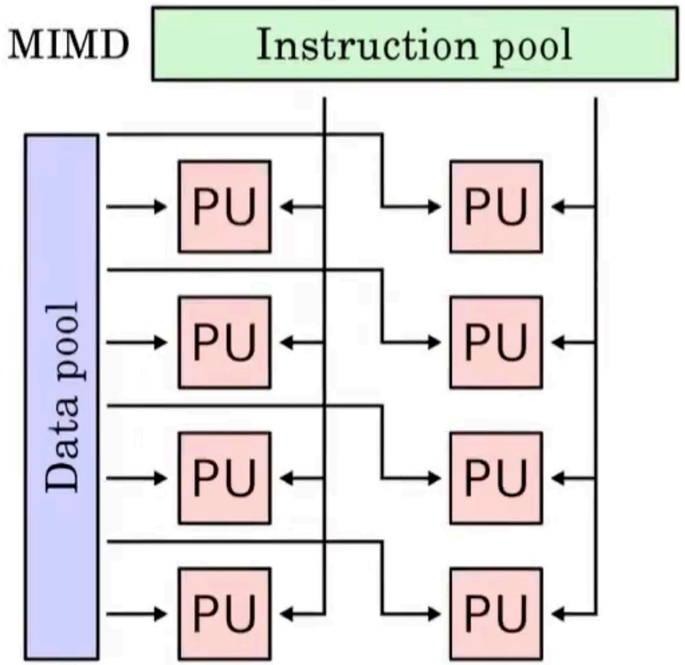
2021-11-17 10:41:49

Parallel Computers: Flow of Control



Flynn's Taxonomy

Describes how streams of **instructions** interact with streams of **data**



Instruction Streams	Data Streams: Single	Data Streams: Multiple
Single	SISD	SIMD
Multiple	MISD	MIMD

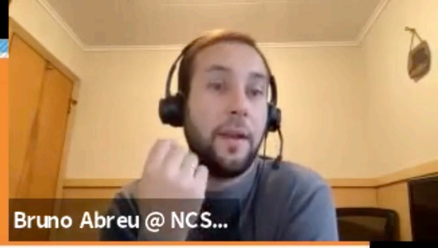
MIMD is highly parallelizable: it can **scale** either way

- Every supercomputer on Top500 since 2010

https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

2021-11-17 10:43:07

Parallel Computers: Interconnection Networks



Platform, wires and cables that define how multiple processors are connected to each other and to the memory units

Network Characteristics

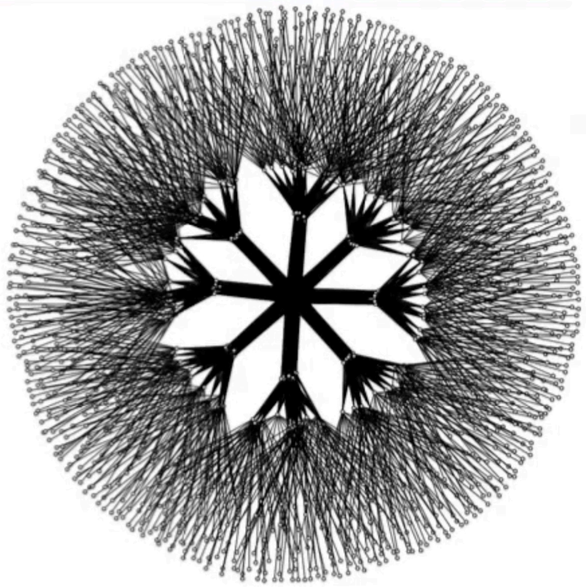
Diameter: maximum distance that data must travel for two processors to communicate

Bandwidth: amount of data that can be sent through the network within a certain time

Latency: delay on a network while a data packet is being stored and forwarded

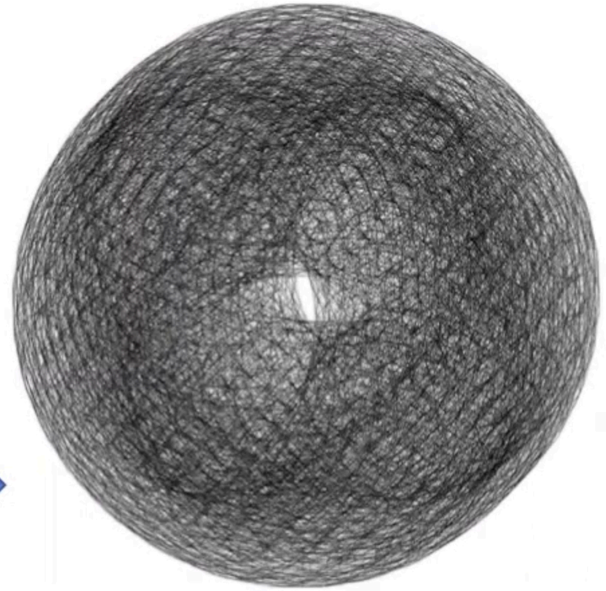
Degree: number of communicating wires coming out of each processor

Topology: configuration of the network that determines how processors are connected



Atlas @ Lawrence Livermore National Laboratory
1142 nodes
192 24-port InfiniBand SDR
3-staged folded Clos topology

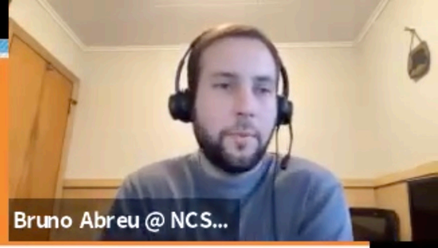
Jaguar Cray XT-5 partition @ Oak Ridge National Lab
18851 nodes
SeaStar 2 25x32x24 3D-Torus

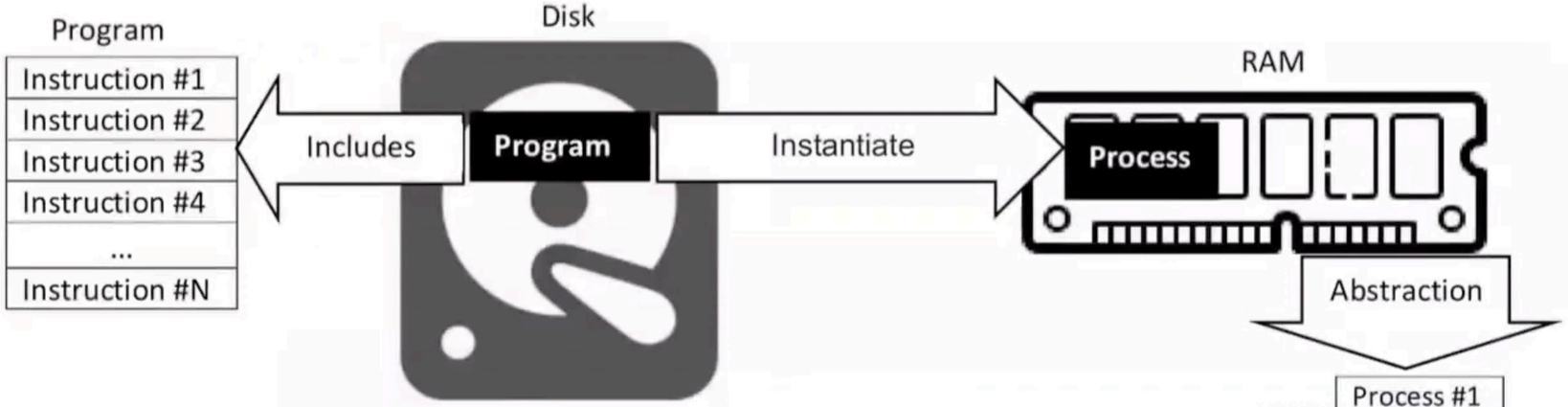


http://unixer.de/research/topologies/

2021-11-17 10:43:42

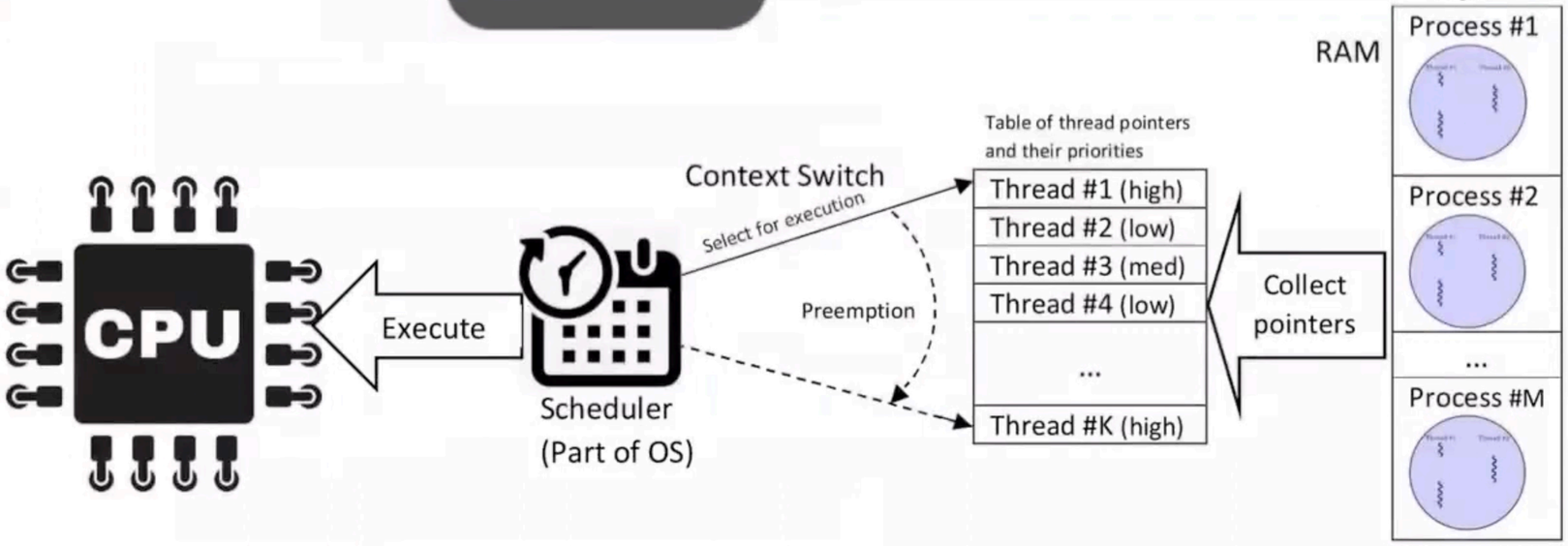
Processes, Threads and more

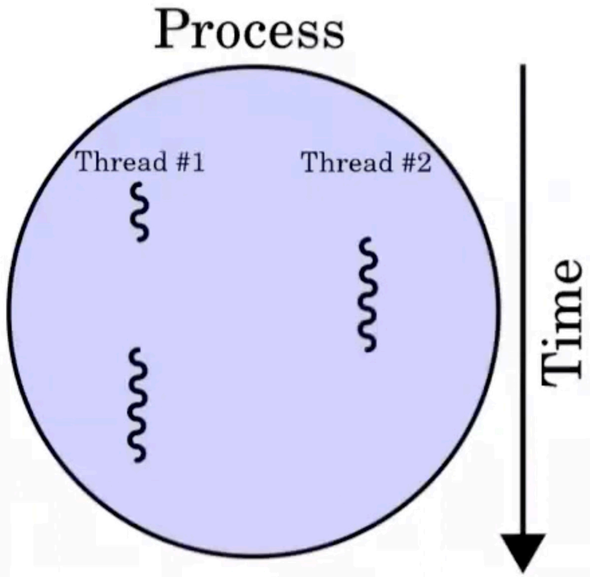




Thread: smallest sequence of instructions that the OS scheduler can manage

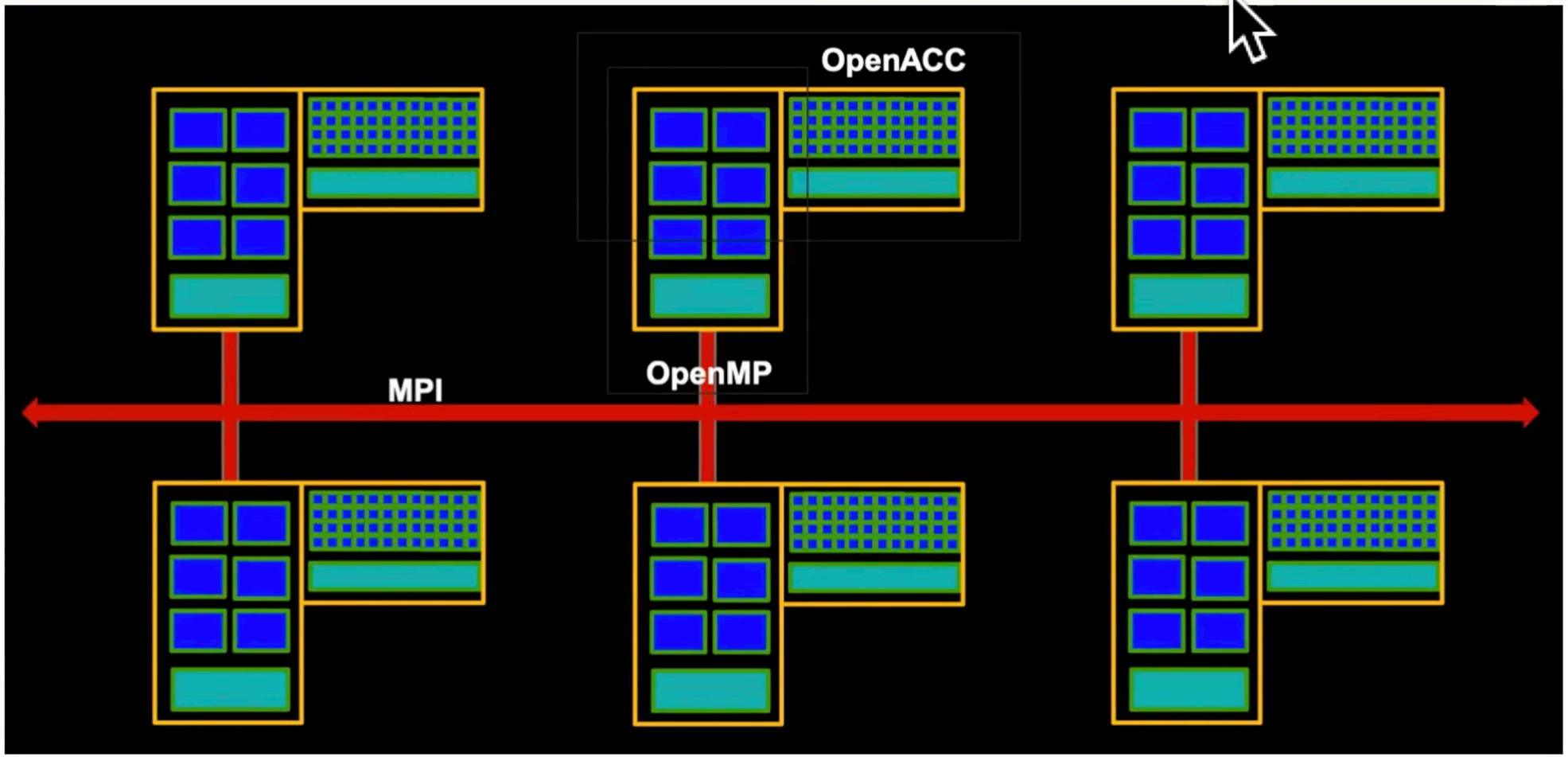
Process: composed of threads. Memory space allocated when program is loaded from disk.





2021-11-17 10:44:51

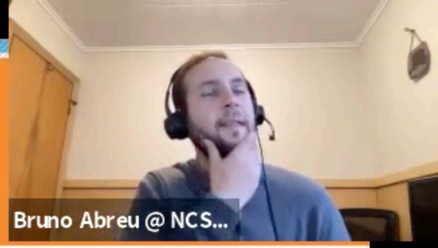
Parallel Computing Frameworks



Courtesy of John Urbanic @ PSC

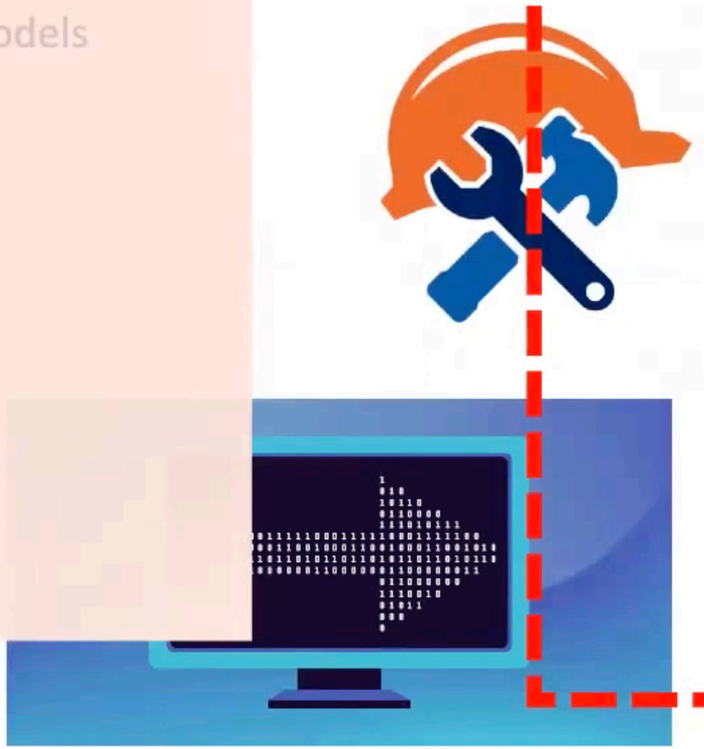
Bruno Abreu @ NCS...
ILLINOIS NCSA
2021-11-17 10:46:34

Outline



Part A: **Parallel Computing Overview** (30 min)

- Hardware design and architecture models
- Parallel programming models
- Domain and function decomposition
- Threads, processes and more



Part B: **How to Parallelize a Code** (60 min)

Hello, Parallel World!
- Parallel regions
- Compilation and execution

Task Parallelism
- Parallel sections
- Measuring performance

Data Parallelism
- Parallel loops
- Performance and scaling

2021-11-17 10:47:34

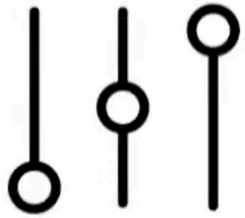
Why OpenMP?

OpenMP

"OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran."

https://www.openmp.org/

- Compiler directives
- Library routines
- Environment variables



Application runtime behavior

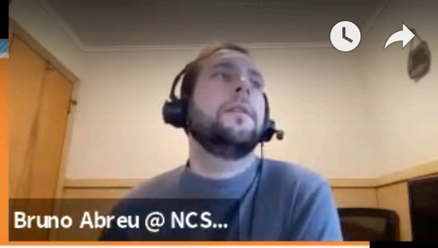
- Simple, robust, mature framework
- Incorporated into every major compiler
- Incremental parallelism
- Does not require message passing
- Allows for reasonable speedups with very little work

- Not fully scalable by itself
- Does not require redesigning or creating new applications within the parallel computing paradigm
- Performance usually limited by sequential blocks

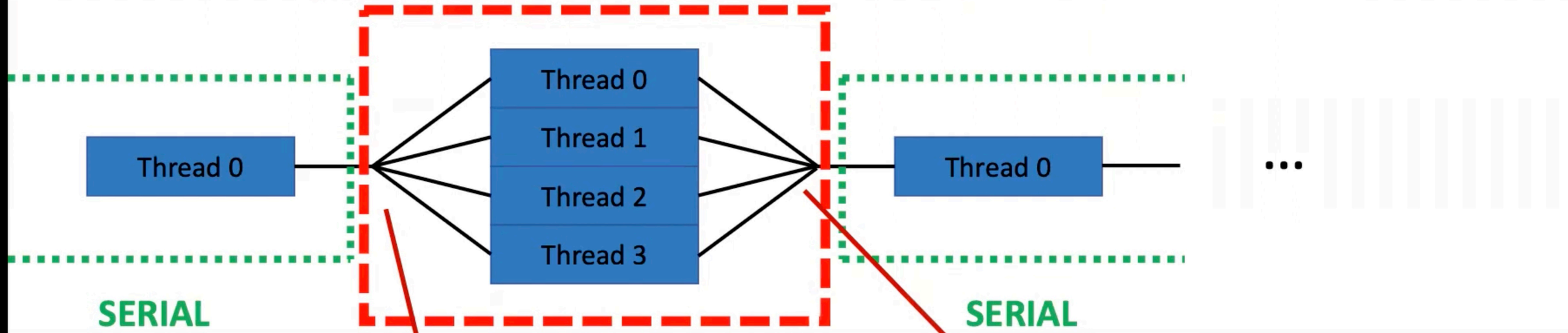
2021-11-17 10:50:57

Press esc to exit full screen

Parallel Regions



These are blocks of code where instructions can be executed independently by different threads



Other names for **Thread 0**:

- Main
- Parent
- Manager

A team of threads is created at the beginning of the parallel region, and we distribute instructions to each of them

As threads finish their instructions, they **wait at the end of the region until the entire team is done**. The team is then destroyed, and the next block is executed

ILLINOIS NCSA

35:42 / 1:45:40 • parallel regions ›
Scroll for details
2021-11-17 10:35:25

Parallel Regions with OpenMP: Syntax

C/C++

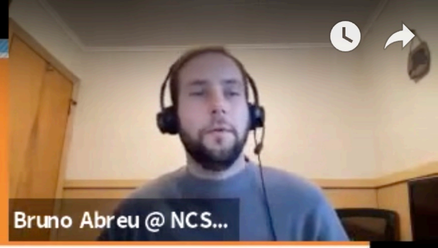
```
#include "omp.h"
.
.
... serial code ...
.
.
#pragma omp parallel
{
    ... instructions ...
}
.
.
... serial code ...
.
.
```

Fortran

```
use :: omp_lib
.
.
... serial code ...
.
.
!$omp parallel
   .
    ... instructions ...
   .
!$omp end parallel
.
.
... serial code ...
.
.
```


2021-11-17 10:55:45

Parallel Regions with OpenMP: Syntax



OpenMP library header

OpenMP **parallel** directive

PARALLEL BLOCKS

C/C++

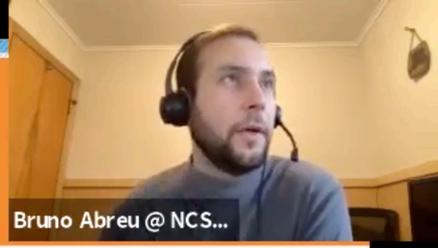
```
#include "omp.h"
.
.
... serial code ...
.
.
#pragma omp parallel
{
    ... instructions ...
}
.
... serial code ...
.
.
```

Fortran

```
use :: omp_lib
.
.
... serial code ...
.
.
!$omp parallel
    .
    ... instructions ...
    .
!$omp end parallel
.
.
... serial code ...
.
.
```


38:44 / 1:45:40 • parallel regions ›
Scroll for details
2021-11-17 10:4:27

Hello, World! : Serial code



Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/hello/serial/hello_serial.cpp

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string hello;
    hello = "Hello, Parallel World!";

    cout << hello << "\n";

    return 0;
}
```


2021-11-17 11:00:18

Hello, World! : Parallelized code

Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/hello/omp/hello_omp.cpp

```
#include <iostream>
#include <string>
#include "omp.h"

using namespace std;

int main()
{
    string hello;
    hello = "Hello, Parallel World!";
    #pragma omp parallel
    {
        cout << hello << endl;
    }
    return 0;
}
```

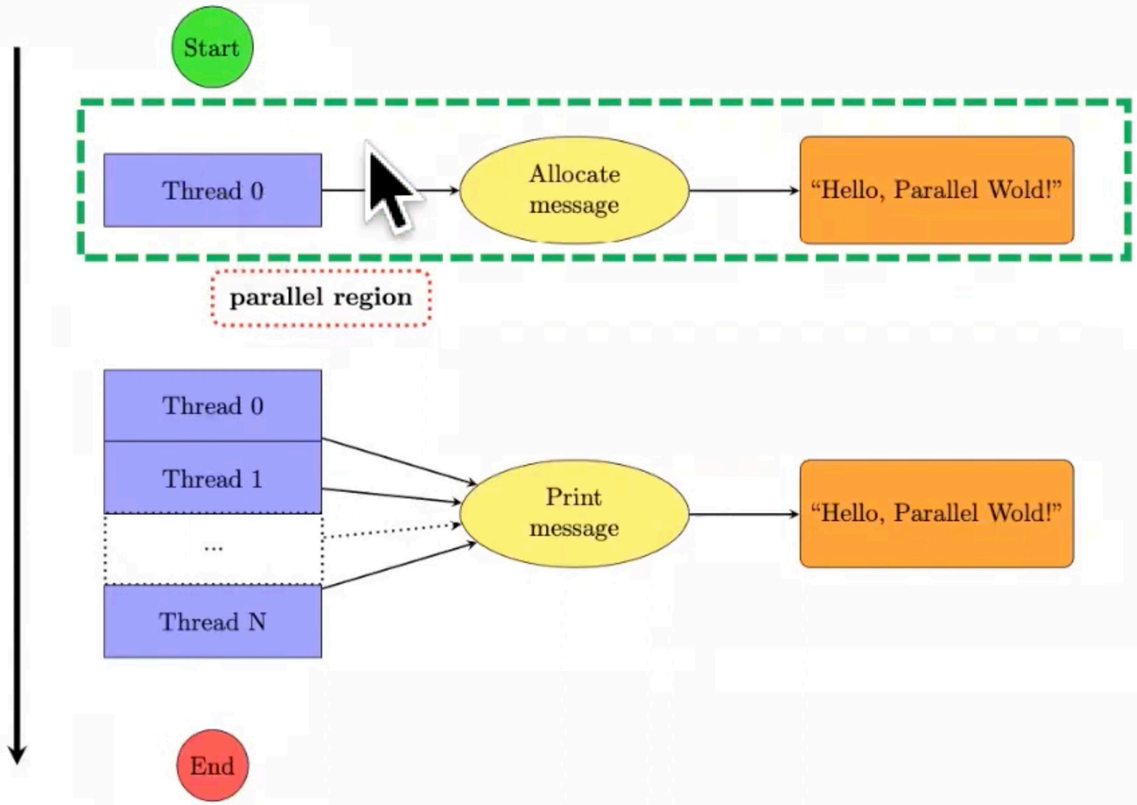
Open/close parallel region

2021-11-17 11:04:00

Hello, World! : Parallelized code

Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/hello/omp/hello_omp.cpp

```cpp
#include <iostream>
#include <string>
#include "omp.h"

using namespace std;

int main()
{
    string hello;
    hello = "Hello, Parallel World!";
    #pragma omp parallel
    {
        cout << hello << endl;
    }
    return 0;
}
```



2021-11-17 11:05:10

Compilation of parallel code (OpenMP)

The general syntax for compilation is:

```
compiler –[OpenMP flag] <SourceCode> -o <TargetBinary>
```

For the current example:

```
g++ -fopenmp hello_omp.cpp –o hello_omp.exe
```

Compiler	OpenMP flag
GCC	-fopenmp
Intel	-qopenmp
AOCC	-fopenmp

Programming Language	OpenMP Header
C/C++	#include "omp.h"
Fortran	use :: omp_lib

46:53 / 1:45:40 • compilation ›
Scroll for details
2021-11-17 11:30:37

ILLINOIS NCSA

Execution of parallel applications (OpenMP)

The number of threads that are invoked in a parallel region (the size of the team) is controlled by the **OMP_NUM_THREADS**

Before executing your application, use **export** to set it to the desired number:

```
export OMP_NUM_THREADS=N
```

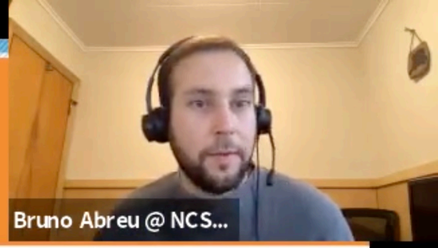
And then finally run the application:

```
./<TargetBinary>
```

ILLINOIS NCSA

```
[babreu@login02 ~]$ git clone https://github
oToPC
Cloning into 'IntroToPC'...
remote: Enumerating objects: 121, done.
remote: Counting objects: 100% (121/121), done.
remote: Compressing objects: 100% (97/97), done.
remote: Total 121 (delta 40), reused 71 (delta 22), pack-reused
Receiving objects: 100% (121/121), 22.34 KiB | 22.34 MiB/s, done.
Resolving deltas: 100% (40/40), done.
[babreu@login02 ~]$ ls
IntroToPC
[babreu@login02 ~]$ ls
IntroToPC
[babreu@login02 ~]$ cd IntroToPC/
[babreu@login02 IntroToPC]$ ls
hello  README.md  saxpy  stats
[babreu@login02 IntroToPC]$ cd hello/
[babreu@login02 hello]$ ls
omp  README.md  serial
[babreu@login02 hello]$ cd omp/
[babreu@login02 omp]$ ls
hello.jobscript  hello_omp.cpp  Makefile
[babreu@login02 omp]$ make
g++ -fopenmp ./hello_omp.cpp -o hello_omp.exe
----- COMPILATION DONE -----
[babreu@login02 omp]$ ls
hello.jobscript  hello_omp.cpp  hello_omp.exe  Makefile
[babreu@login02 omp]$
```

Hello, World! : Making sense of the output




```
Hello, Parallel World!Hello, Parallel World!
Hello, Parallel World!
Hello, Parallel World!
Hello, Parallel World!Hello, Parallel World!
Hello, Parallel World!

Hello, Parallel World!

```


2021-11-17 11:11:10

Hello, World! : Making sense of the output

```
.
.
.
hello = "Hello, Parallel World!"
#pragma omp parallel
{
    cout << hello << endl;
}
.
.
.
```

Fast Thread
Slow Thread
Fast Thread
Fast Thread

```
Hello, Parallel World!Hello, Parallel World!
Hello, Parallel World!
Hello, Parallel World!
Hello, Parallel World!Hello, Parallel World!
Hello, Parallel World!

Hello, Parallel World!

```

Hello, World! : Making sense of the output

```
.
.
.
hello = "Hello, Parallel World!"
#pragma omp parallel
{
    cout << hello << endl;
}
.
.
.
```

Fast Thread
Slow Thread
Fast Thread
Fast Thread

```
Hello, Parallel World!Hello, Parallel World!
Hello, Parallel World!
Hello, Parallel World!
Hello, Parallel World!Hello, Parallel World!
Hello, Parallel World!

Hello, Parallel World!
```


2021-11-17 11:12:46

Hello, World! : Making sense of the output

```
.
.
.
hello = "Hello, Parallel World!"
#pragma omp parallel
{
    cout << hello << endl;
}
.
.
.
```

Fast Thread
Slow Thread
Fast Thread
Fast Thread

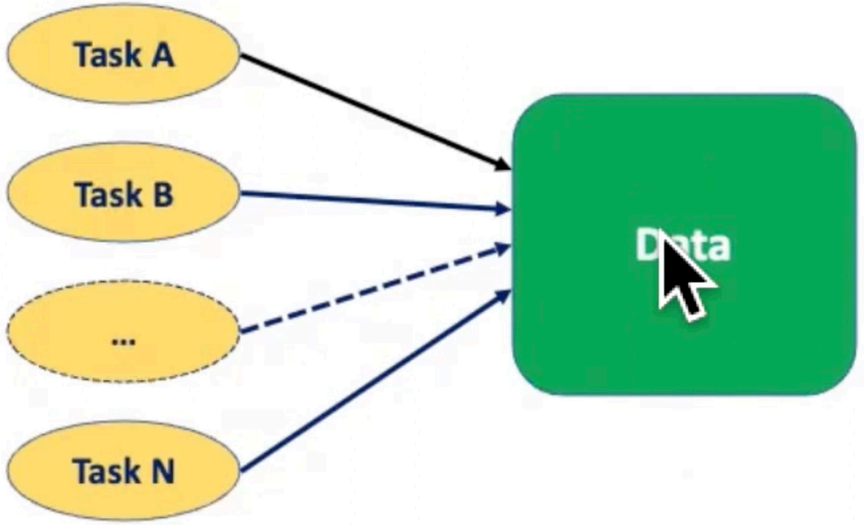
By default, threads execute their instructions **independently** and **asynchronously**!

```
Hello, Parallel World!Hello, Parallel World!
Hello, Parallel World!
Hello, Parallel World!
Hello, Parallel World!Hello, Parallel World!
Hello, Parallel World!

Hello, Parallel World!

```

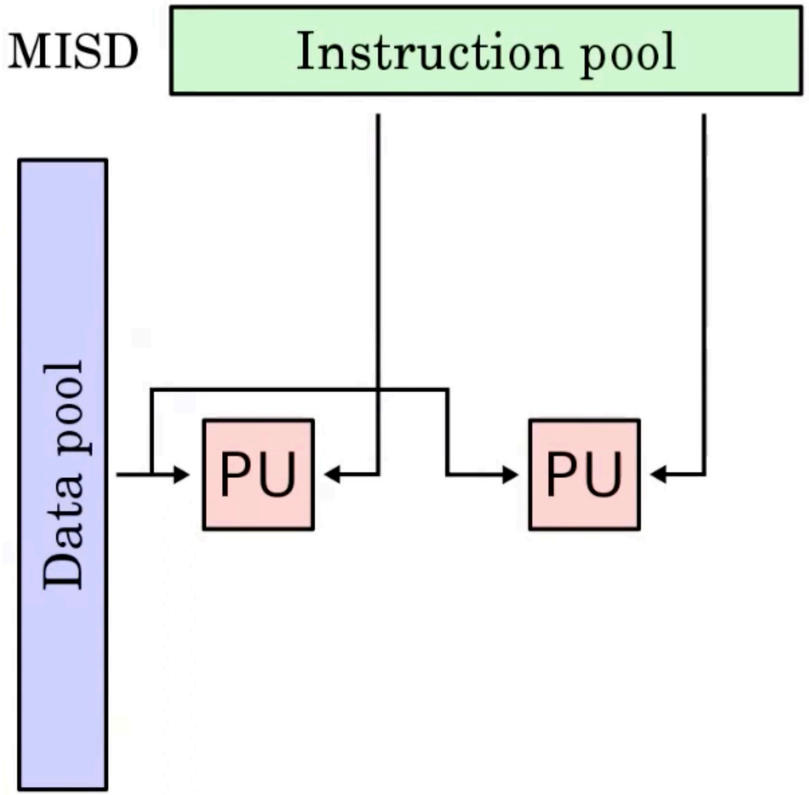

2021-11-17 11:12:57

Task Parallelism

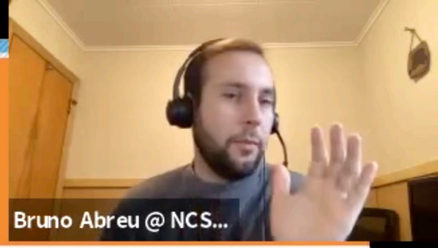
Task Parallelism



Execute different tasks over the same data



OpenMP parallel sections: Syntax



C/C++

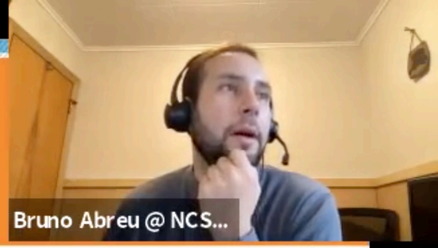
```
#include "omp.h"
.
.
... serial code ...
.
.
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            ... task A ...
        }
        #pragma omp section
        {
            ... task B ...
        }
    }
}
.
.
... serial code ...
.
.
```

Fortran

```
use :: omp_lib
.
.
... serial code ...
.
.
!$omp parallel
    !$omp sections
        !$omp section
            .
            ... task A ...
            .
        !$omp section
            .
            ... task B ...
            .
  !$omp end sections
!$omp end parallel
.
.
... serial code ...
.
.
```


2021-11-17 11:15:28

OpenMP parallel sections: Syntax



OpenMP library header

OpenMP parallel directives

OpenMP **sections**

C/C++

```c
#include "omp.h"
.
.
... serial code ...
.
.
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            ... task A ...
        }
        #pragma omp section
        {
            ... task B ...
        }
    }
}
.
.
... serial code ...
.
.
```

Fortran

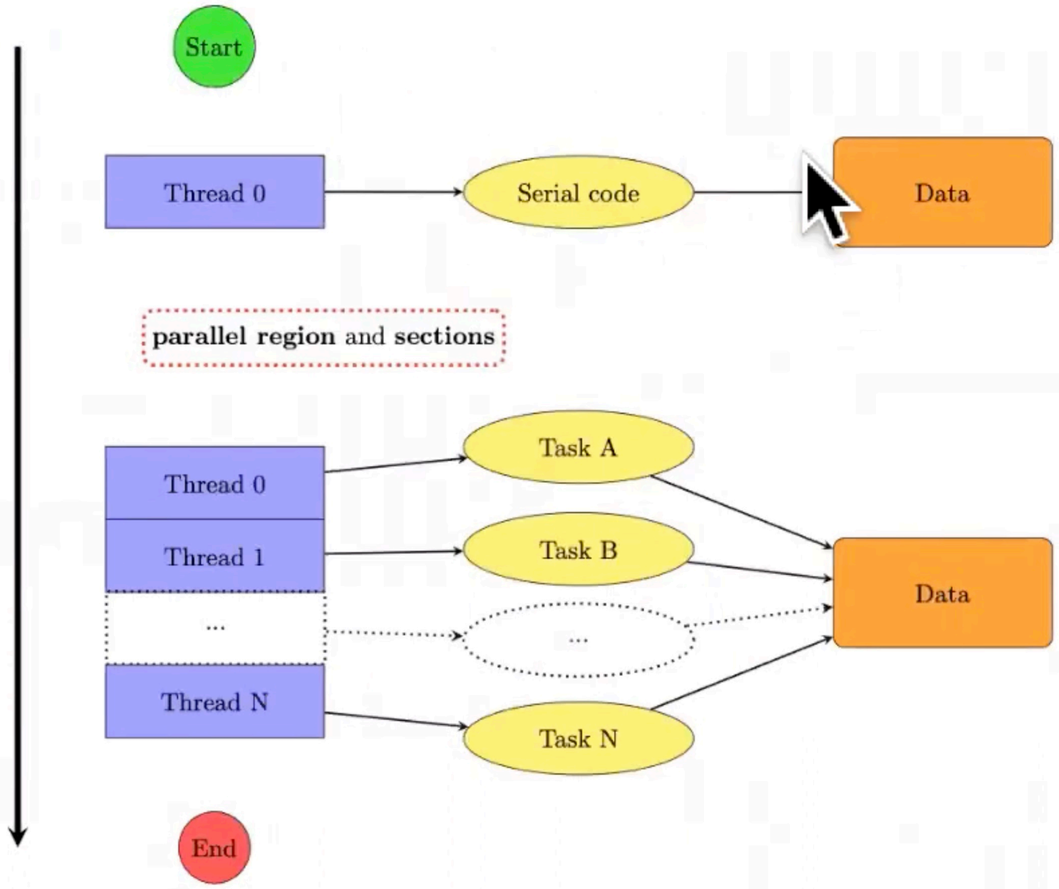
```fortran
use :: omp_lib
.
.
... serial code ...
.
.
!$omp parallel
    !$omp sections
        !$omp section
            .
            ... task A ...
            .
        !$omp section
            .
            ... task B ...
            .
  !$omp end sections
!$omp end parallel
.
.
... serial code ...
.
.
```


2021-11-17 11:15:49

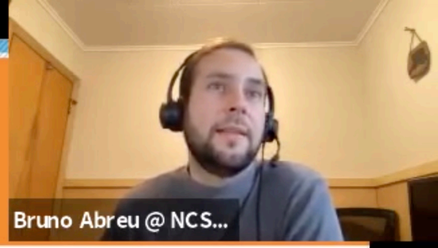
OpenMP parallel sections: workflow diagram

C/C++

```
#include "omp.h"
.
.
... serial code ...
.
.
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            ... task A ...
        }
        #pragma omp section
        {
            ... task B ...
        }
    }
}
.
.
... serial code ...
.
.
```

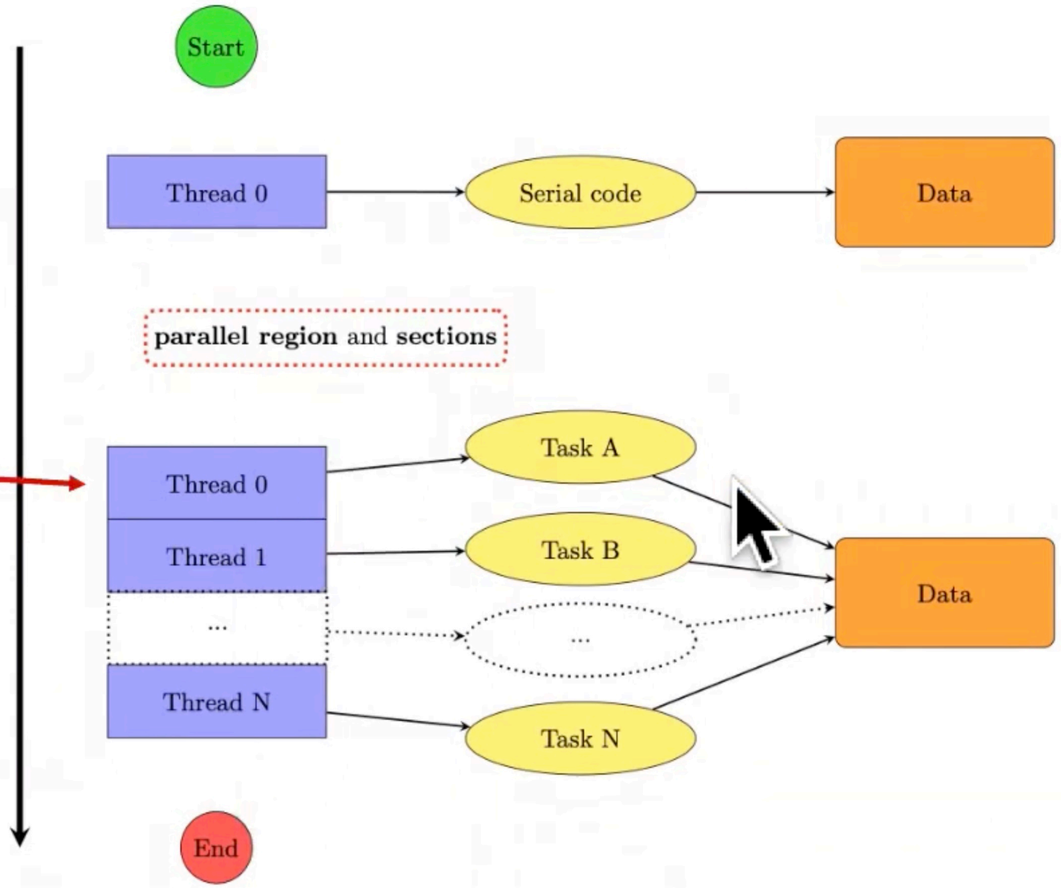


OpenMP parallel sections: workflow diagram

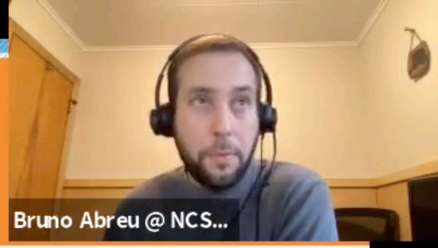


C/C++

```
#include "omp.h"
.
.
... serial code ...
.
.
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            ... task A ...
        }
        #pragma omp section
        {
            ... task B ...
        }
    }
}
.
.
... serial code ...
.
.
```



STATS: Serial code



Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/stats/serial/stats_serial.cpp

```
int main()
{
    // create a vector with random noise
    vector<double> vec;
    double elmnt, r;
    srand(time(NULL));
    for (int i = 0; i < ORD; i++)
    {
        r = rand() % RAND_MAX;
        elmnt = double(r) / (RAND_MAX);
        vec.push_back(elmnt);
    }

    // find vector min
    get_vec_min(vec);
    // find vec max
    get_vec_max(vec);
    // find vec avg
    get_vec_avg(vec);

    return 0;
}
```

Function implementation

```
void get_vec_min(const vector<double> &v)
{
    double min;
    min = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] < min)
            min = v[i];
    }
    cout << scientific;
    cout << "Min of vector: " << min << endl;
}
```


2021-11-17 11:18:54

STATS: Parallelized code

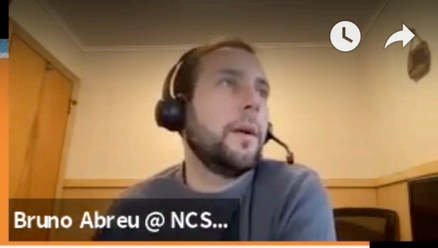
Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/stats/omp/stats_omp.cpp

```
int main()
{
    // create a vector with random noise
    vector<double> vec;
    double elmnt, r;
    srand(time(NULL));
    for (int i = 0; i < ORD; i++)
    {
        r = rand() % RAND_MAX;
        elmnt = double(r) / (RAND_MAX);
        vec.push_back(elmnt);
    }

    // variables for timing
    struct timeval start_time, stop_time, elapsed_time;

```

Variables to measure elapsed time

1:02:49 / 1:45:40 • Parallel Workflow ›
Scroll for details
2021-11-17 11:39

STATS: Parallelized code

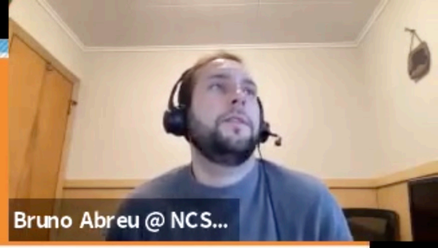


Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/stats/omp/stats_omp.cpp

```
gettimeofday(&start_time, NULL); // start clock
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            // find vector min
            get_vec_min(vec);
        }
        #pragma omp section
        {
            // find vec max
            get_vec_max(vec);
        }
        #pragma omp section
        {
            // find vec avg
            get_vec_avg(vec);
        }
    }
}
gettimeofday(&stop_time, NULL);                    // stop clock
```

Parallel Block

STATS: Parallelized code



Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/stats/omp/stats_omp.cpp

What about the implementation of each function?

```
void get_vec_min(const vector<double> &v)
{
    double min;
    min = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] < min)
            min = v[i];
    }
    cout << scientific;
    cout << "Min of vector: " << min << endl;
}
```

```
void get_vec_max(const vector<double> &v)
{
    double max;
    max = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] > max)
            max = v[i];
    }
    cout << scientific;
    cout << "Max of vector: " << max << endl;
}
```

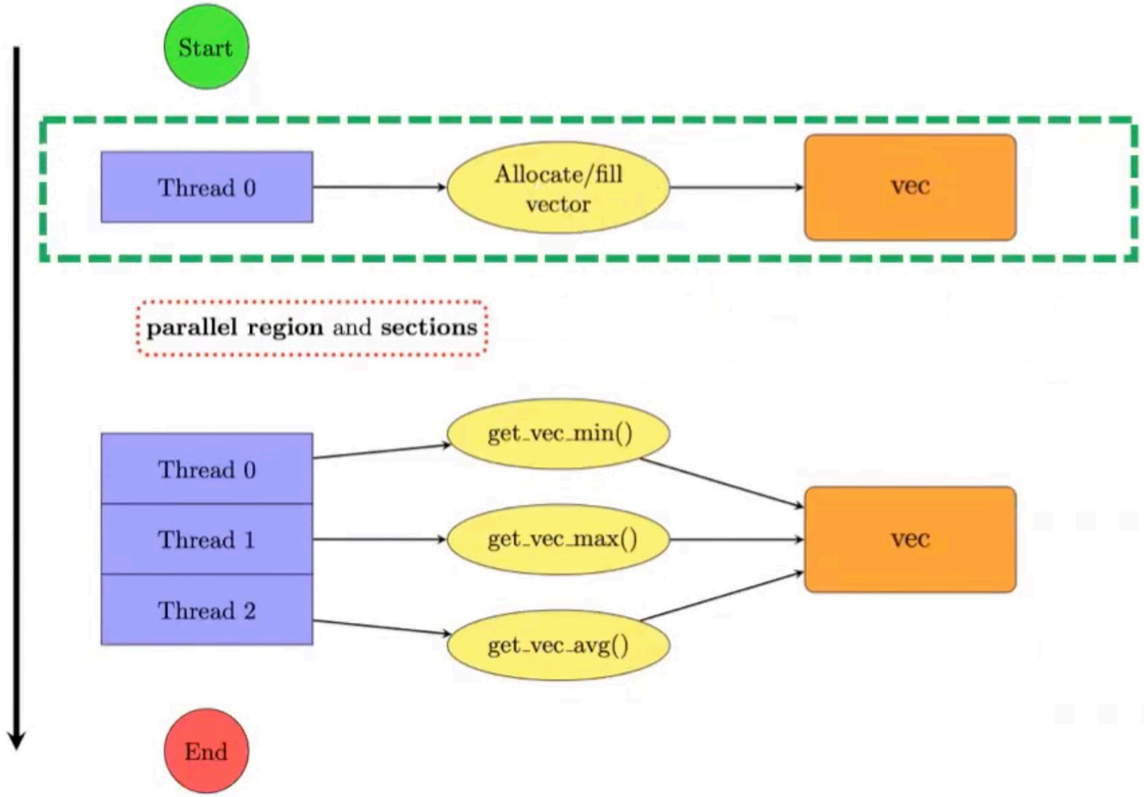
```
void get_vec_avg(const vector<double> &v)
{
    double sum;
    sum = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        sum += v[i];
    }
    cout << scientific;
    cout << "Avg of vector: " << sum / v.size() << endl;
}
```

They remain the same!

2021-11-17 11:22:19

STATS: Workflow diagram

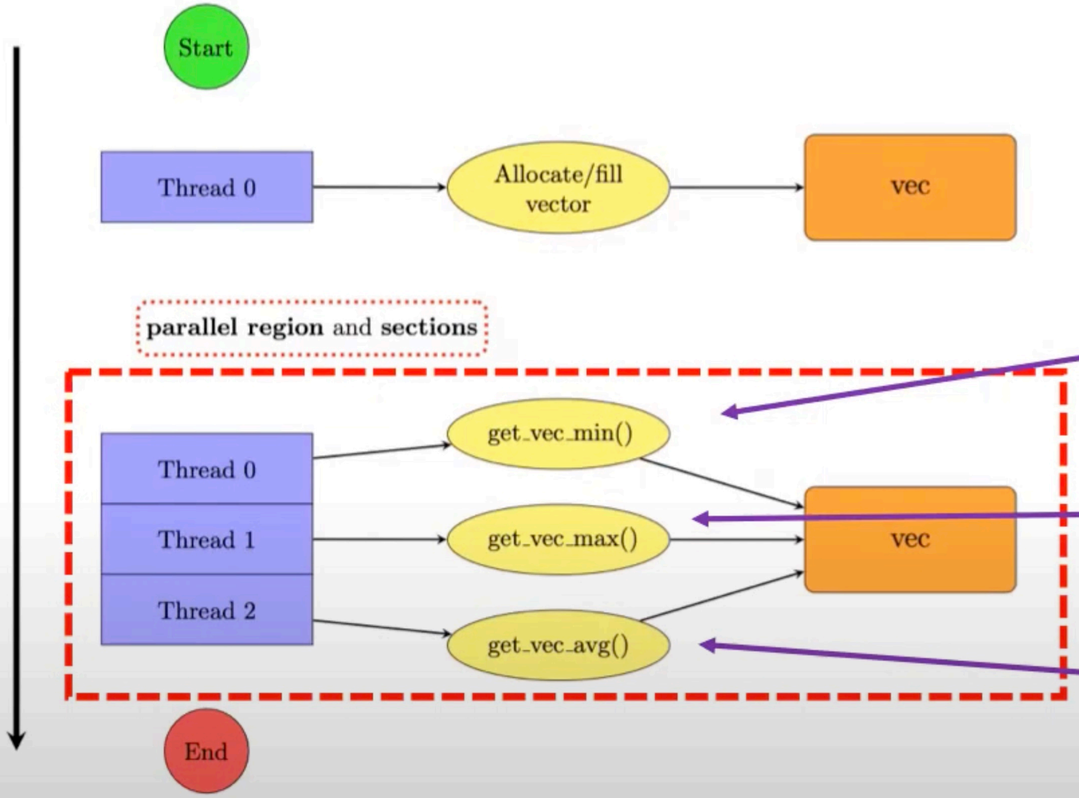
Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/stats/omp/stats_omp.cpp



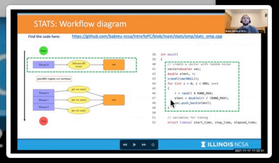

```
int main()
{
    // create a vector with random noise
    vector<double> vec;
    double elmnt, r;
    srand(time(NULL));
    for (int i = 0; i < ORD; i++)
    {
        r = rand() % RAND_MAX;
        elmnt = double(r) / (RAND_MAX);
        vec.push_back(elmnt);
    }

    // variables for timing
    struct timeval start_time, stop_time, elapsed_time;
```

STATS: Workflow diagram

Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/stats/omp/stats_omp.cpp



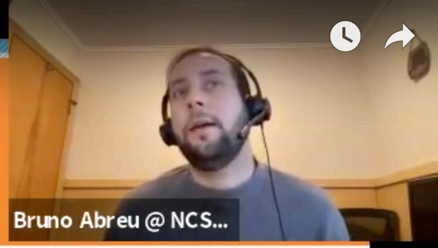

```
55      #pragma omp parallel
56      {
57          #pragma omp sections
58          {
59              #pragma omp section
60              {
61                  // find vector min
62                  get_vec_min(vec);
63              }
64              #pragma omp section
65              {
66                  // find vec max
67                  get_vec_max(vec);
68              }
69              #pragma omp section
70              {
71                  // find vec avg
72                  get_vec_avg(vec);
73              }
74          }
75      }
```

1:04:31 / 1:45:40 • Parallel Workflow ›
Scroll for details

Press esc to exit full screen

STATS: Compilation and execution



- **To compile:**

```
make
```

- **To run:**

```
sbatch stats.jobscript
```

- **To check status:**

```
squeue -u $USER
```

Use `cat` to check on your output, copy it and paste in the chat!

The output file should be named like:

statsYourJobID.out

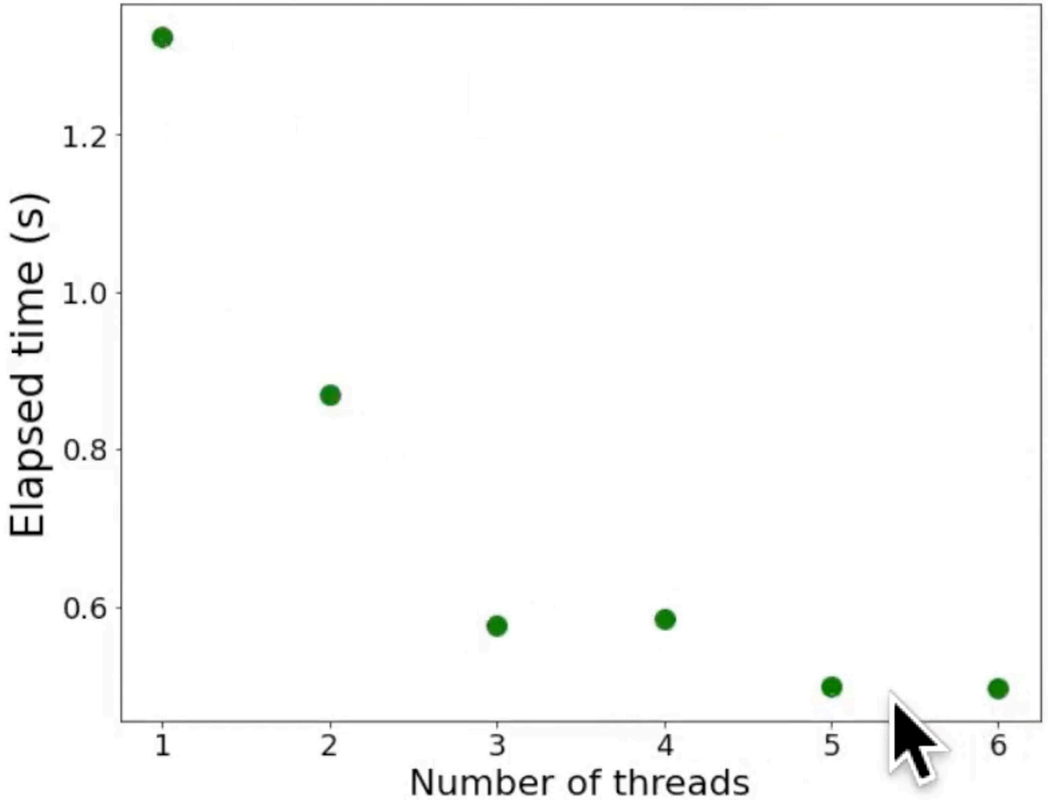
The tab button will help you find it!

1:04:59 / 1:45:40 • Parallel Workflow ›
Scroll for details
2021-11-17 11:19:48

STATS: Measuring performance

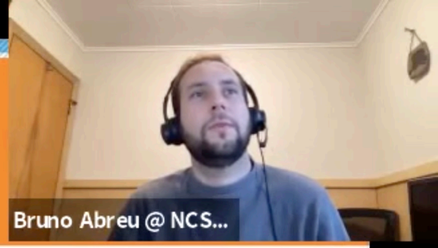
Raw time measurements are not very helpful…

Number of threads	Elapsed time (s)
1	1.32492
2	0.86940
3	0.57525
4	0.58531
5	0.49785
6	0.49656



2021-11-17 11:28:02

STATS: Measuring performance



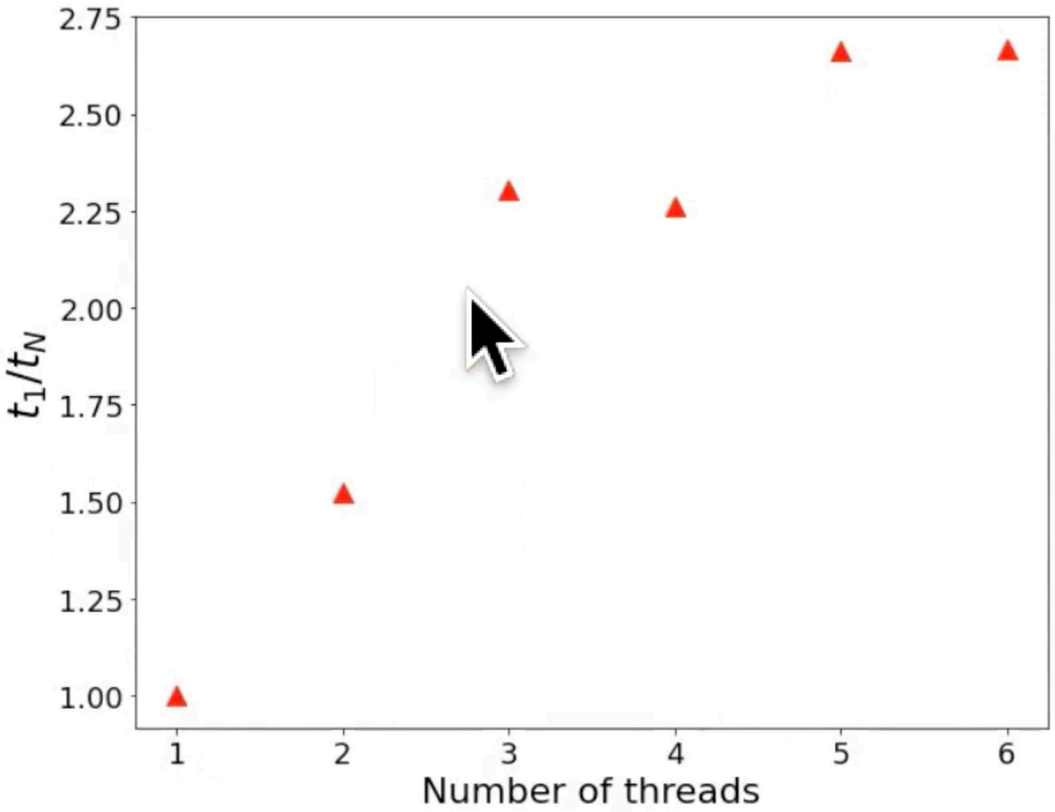
A better measure is how much faster the code runs in parallel when compared to when it runs serially:

$$\frac{t_1}{t_N}$$

one thread

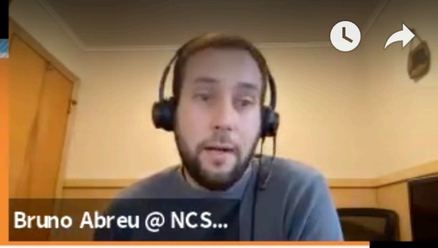
multiple threads

Number of threads	Elapsed time (s)	t_1 / t_N
1	1.32492	1.000
2	0.86940	1.524
3	0.57525	2.303
4	0.58531	2.264
5	0.49785	2.661
6	0.49656	2.668



2021-11-17 11:28:21

STATS: Measuring performance



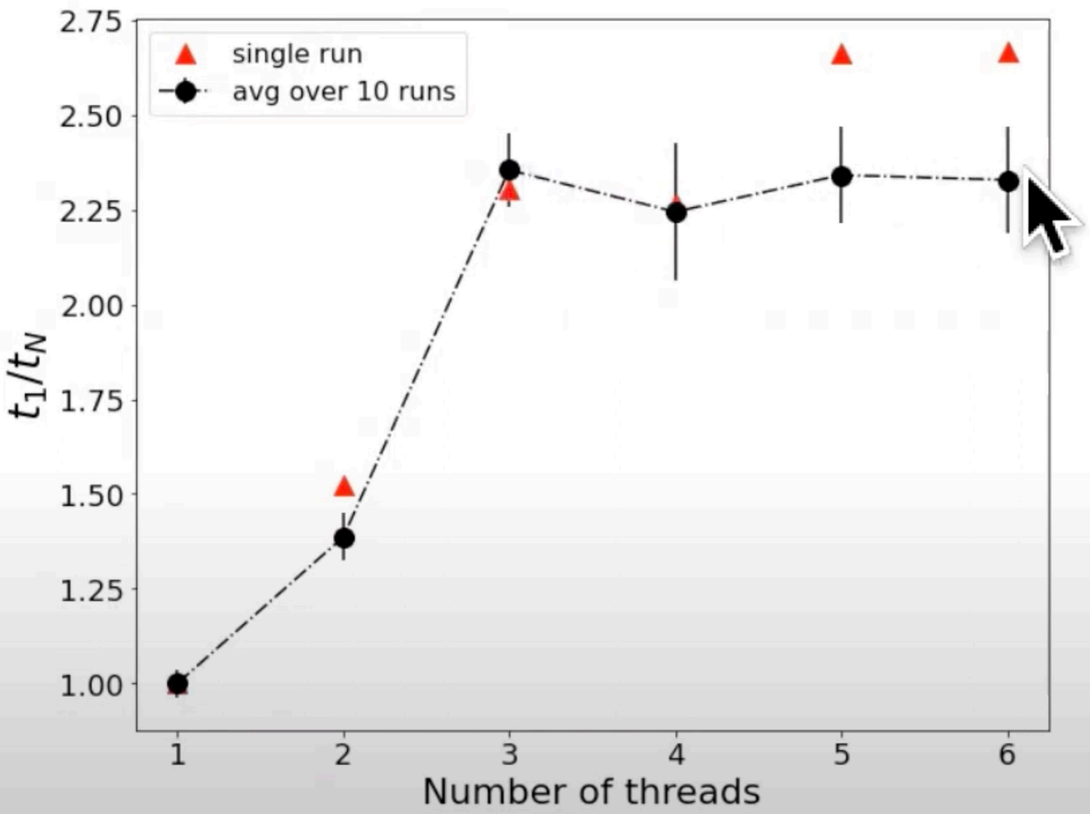
A better measure is how much faster the code runs in parallel when compared to when it runs serially:

$$\frac{t_1}{t_N}$$

one thread

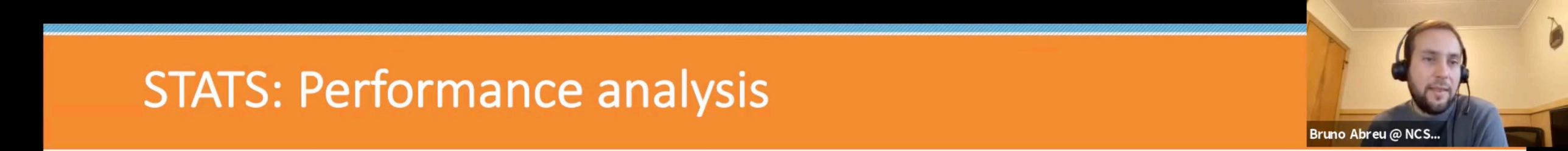
multiple threads

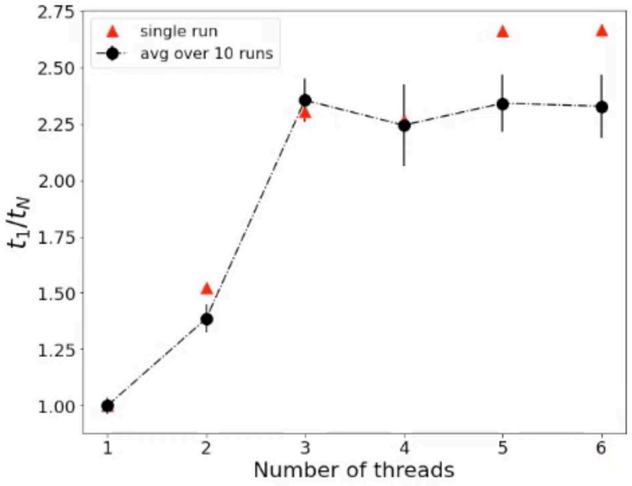
Number of threads	Elapsed time (s)	t_1 / t_N
1	1.32492	1.000
2	0.86940	1.524
3	0.57525	2.303
4	0.58531	2.264
5	0.49785	2.661
6	0.49656	2.668



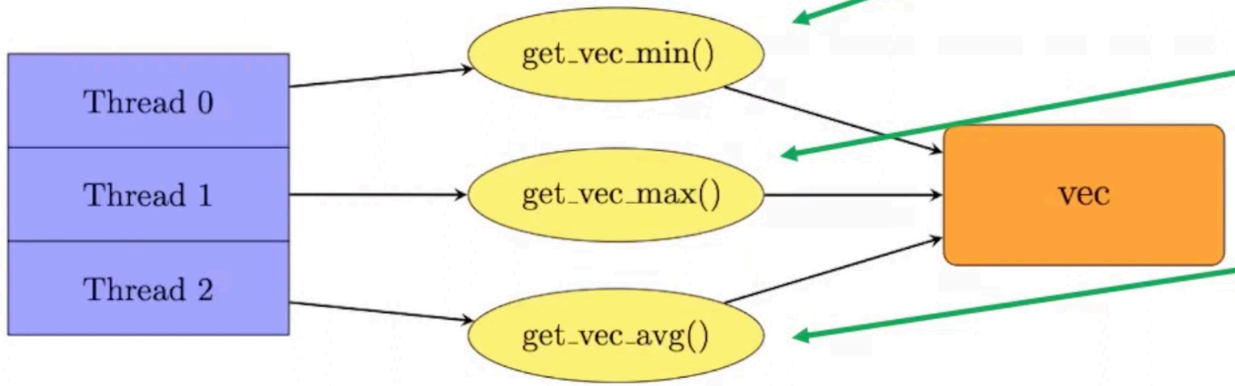
1:10:41 / 1:45:40 • Parallel Workflow ›
Scroll for details
2021-11-17 11:30:24

STATS: Performance analysis



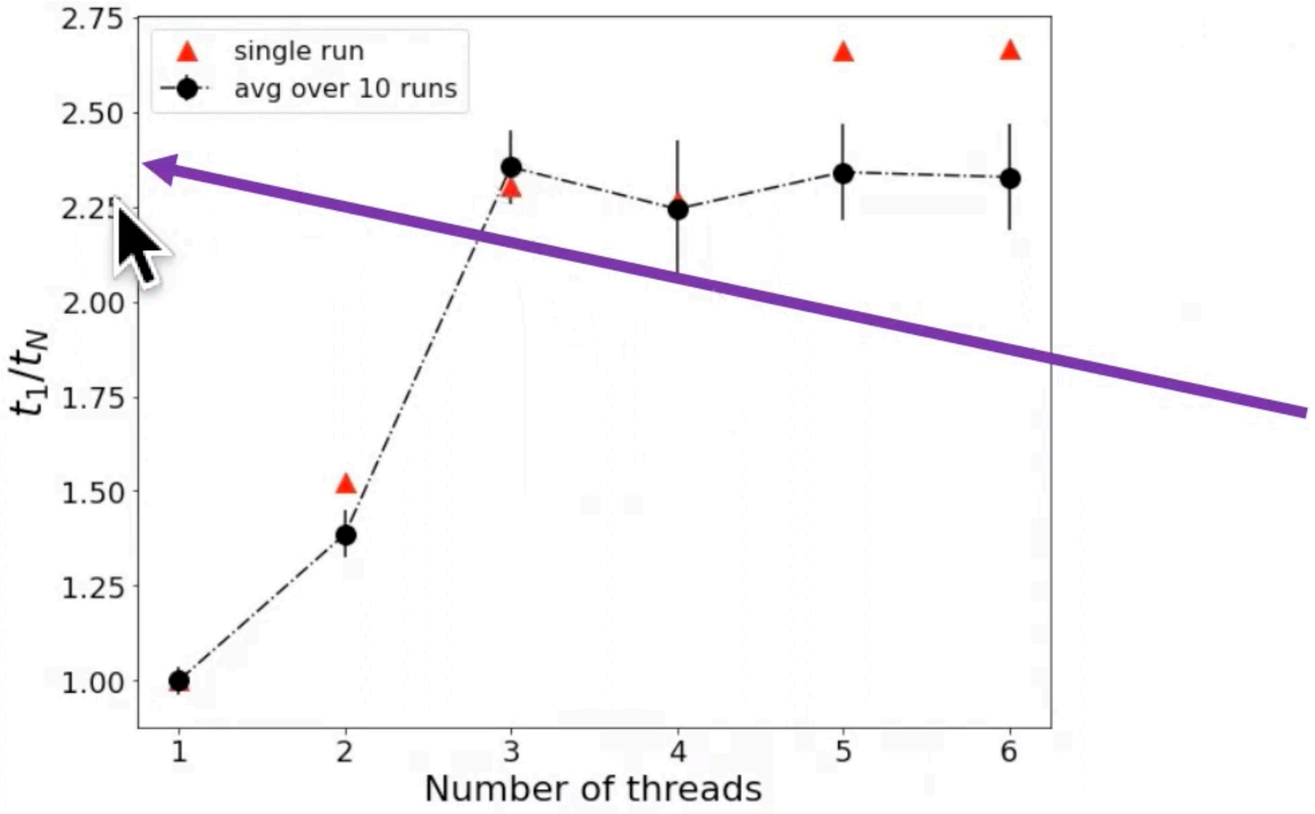


Additional threads are not assigned to any task and just sit idle!




```
55      #pragma omp parallel
56      {
57          #pragma omp sections
58          {
59              #pragma omp section
60              {
61                  // find vector min
62                  get_vec_min(vec);
63              }
64              #pragma omp section
65              {
66                  // find vec max
67                  get_vec_max(vec);
68              }
69              #pragma omp section
70              {
71                  // find vec avg
72                  get_vec_avg(vec);
73              }
74          }
75      }
```

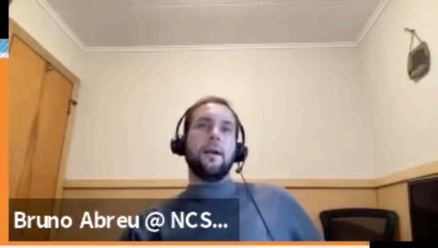
STATS: Performance analysis



Performance gain
is “only” about 2.4x

2021-11-17 11:30:53

STATS: Performance analysis




```
void get_vec_min(const vector<double> &v)
{
    double min;
    min = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] < min)
            min = v[i];
    }
    cout << scientific;
    cout << "Min of vector: " << min << endl;
}
```

0.423 s to run

```
void get_vec_max(const vector<double> &v)
{
    double max;
    max = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] > max)
            max = v[i];
    }
    cout << scientific;
    cout << "Max of vector: " << max << endl;
}
```

0.434 s to run

```
void get_vec_avg(const vector<double> &v)
{
    double sum;
    sum = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        sum += v[i];
    }
    cout << scientific;
    cout << "Avg of vector: " << sum / v.size() << endl;
}
```

0.588 s to run

2021-11-17 11:32:03

STATS: Performance analysis

```
void get_vec_min(const vector<double> &v)
{
    double min;
    min = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] < min)
            min = v[i];
    }
    cout << scientific;
    cout << "Min of vector: " << min << endl;
}
```

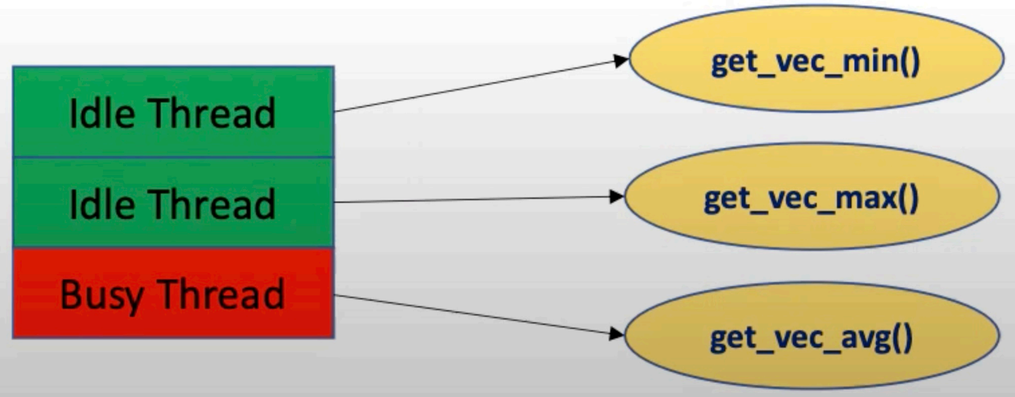
0.423 s to run

```
void get_vec_max(const vector<double> &v)
{
    double max;
    max = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        if (v[i] > max)
            max = v[i];
    }
    cout << scientific;
    cout << "Max of vector: " << max << endl;
}
```

0.434 s to run

```
void get_vec_avg(const vector<double> &v)
{
    double sum;
    sum = v[0];
    for (int i = 1; i < v.size(); i++)
    {
        sum += v[i];
    }
    cout << scientific;
    cout << "Avg of vector: " << sum / v.size() << endl;
}
```

0.588 s to run

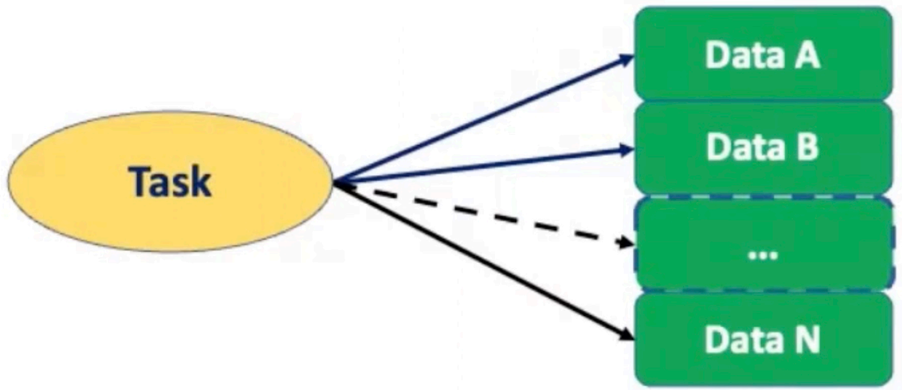


Total execution time is dominated by the thread with the largest workload!

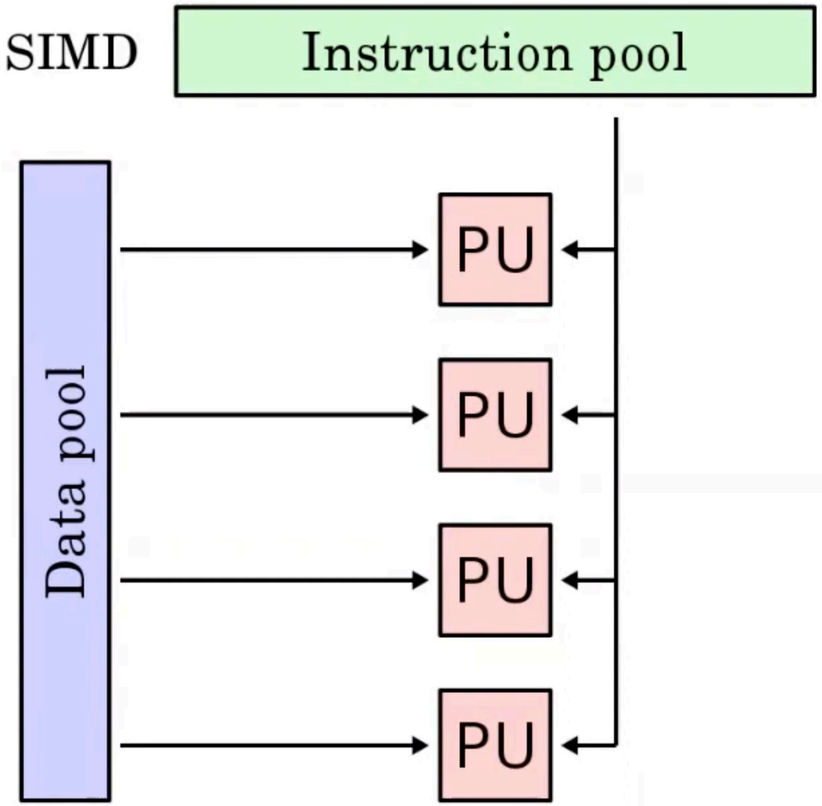
1:15:37 / 1:45:40 • Parallel Workflow ›
Scroll for details
2021-11-17 11:29:29

Data Parallelism

Data Parallelism

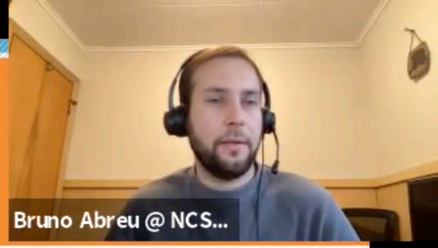


Execute the same task over different chunks of data



2021-11-17 11:35:29

OpenMP parallel loops: Syntax



OpenMP library header

OpenMP parallel directives

C/C++

```
#include "omp.h"
.
.
... serial code ...
.
.
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
    {
        ... task ...
    }
}
.
.
... serial code ...
.
.
```

Fortran

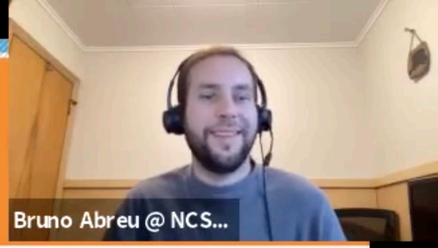
```
use :: omp_lib
.
.
... serial code ...
.
.
!$omp parallel
    !$omp do
    do i = 1, N
        .
        ... task ...
        .
    enddo
    !$omp end do
!$omp end parallel
.
.
... serial code ...
.
.
```

OpenMP **do/for**

Loop implementation must follow!

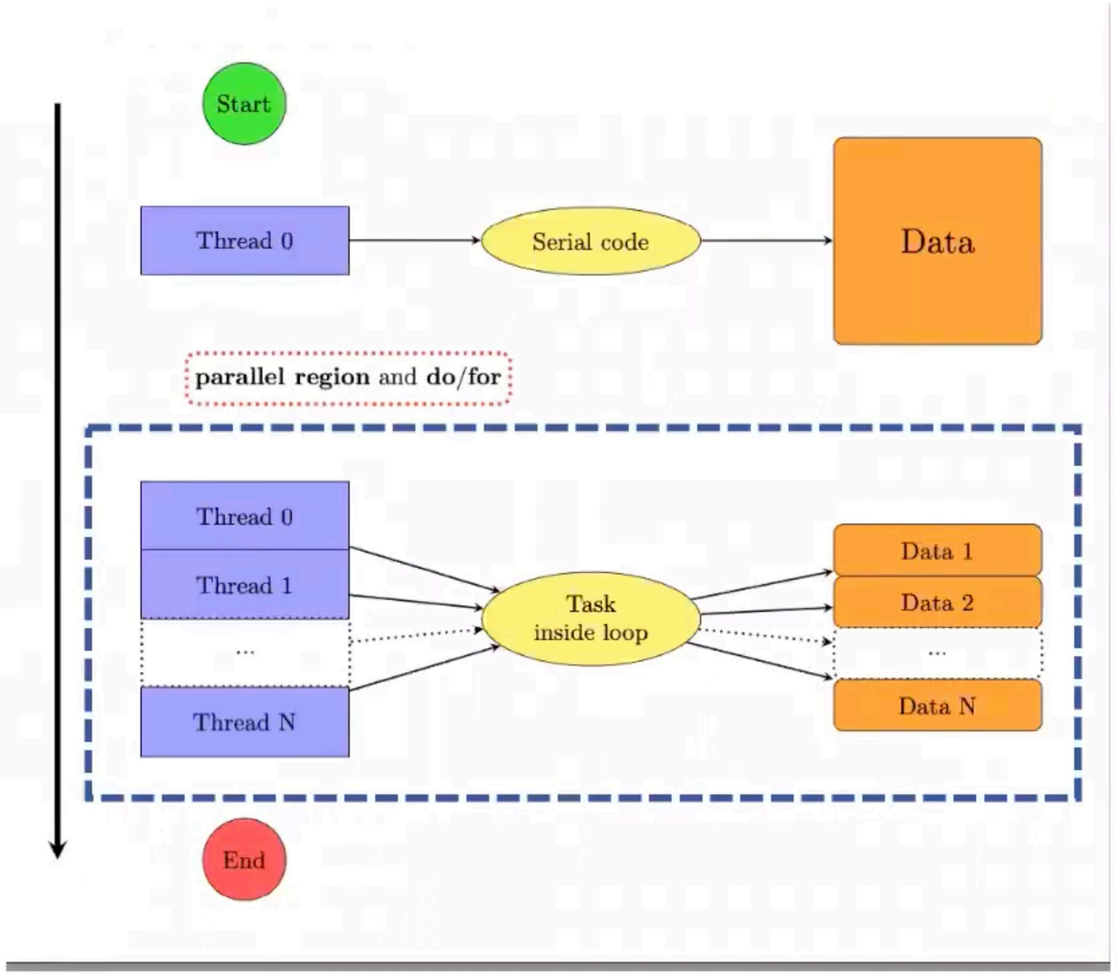
2021-11-17 11:36:32

OpenMP parallel loops: workflow diagram

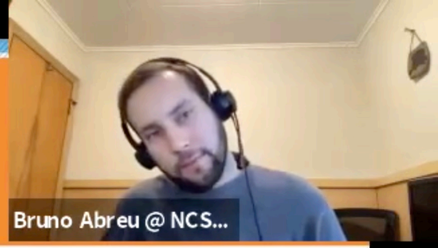


C/C++

```
#include "omp.h"
.
.
... serial code ...
.
.
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
    {
        ... task ...
    }
}
.
.
... serial code ...
.
.
```



SAXPY: Serial code



Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/saxpy/serial/saxpy_serial.cpp

SAXPY: Single-precision **A*X Plus Y**

$$y[i] \rightarrow a \cdot x[i] + y[i]$$

Scalar

Vectors

2021-11-17 11:37:19

SAXPY: Serial code

Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/saxpy/serial/saxpy_serial.cpp

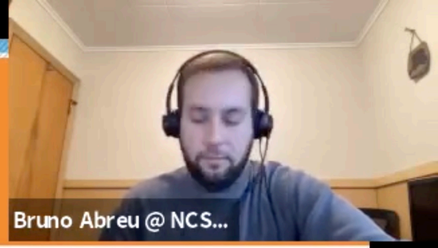
```
int main()
{
    // instantiate vectors and attribute values
    vector<float> x, y;
    float r, elmnt;
    srand(time(NULL));
    for (int i = 0; i < ORD; i++)
    {
        r = rand() % RAND_MAX;
        elmnt = float(r) / (RAND_MAX);
        x.push_back(elmnt); // x is noise
        y.push_back(1.0);   // y is just ones
    }

    float a = 2.0; // scalar multiplier (y -> a*x + y)

    // now call saxpy
    saxpy(a, x, y);

    return 0;
}
```

Function implementation

```

void saxpy(float a, const vector<float> &x, vector<float> &y)
{
    for (int i = 0; i < x.size(); i++)
    {
        y[i] = a * x[i] + y[i];
    }
}
```


1:20:47 / 1:45:40 • Parallel Workflow ›
Scroll for details
2021-11-17 11:30:39

SAXPY: Parallelized code

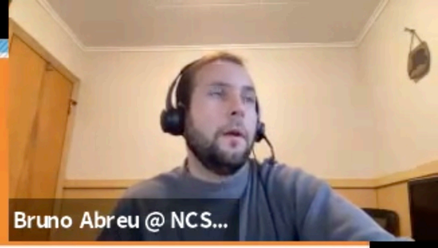


Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/saxpy/omp/saxpy_omp.cpp

```
    // stopwatch variables
    struct timeval start_time, stop_time, elapsed_time;

    // now call saxpy
    gettimeofday(&start_time, NULL);
    #pragma omp parallel
    {
        saxpy(a, x, y);
    }
    gettimeofday(&stop_time, NULL);

    timersub(&stop_time, &start_time, &elapsed_time);
    cout << "Elapsed time (s): " << elapsed_time.tv_sec + elapsed_time.tv_usec / 1000000.0 << endl;

    return 0;
}
```


2021-11-17 11:41:21

SAXPY: Parallelized code

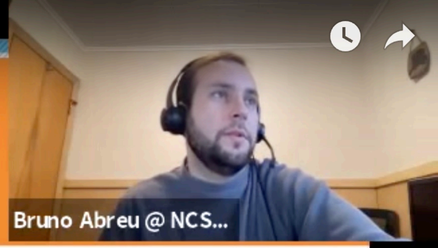


Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/saxpy/omp/saxpy_omp.cpp

```
    // stopwatch variables
    struct timeval start_time, stop_time, elapsed_time;

    // now call saxpy
    gettimeofday(&start_time, NULL);
    #pragma omp parallel
    {
        saxpy(a, x, y);
    }
    gettimeofday(&stop_time, NULL);

    timersub(&stop_time, &start_time, &elapsed_time);
    cout << "Elapsed time (s): " << elapsed_time.tv_sec + elapsed_time.tv_usec / 1000000.0 << endl;

    return 0;
}
```

Measure elapsed time

SAXPY: Parallelized code



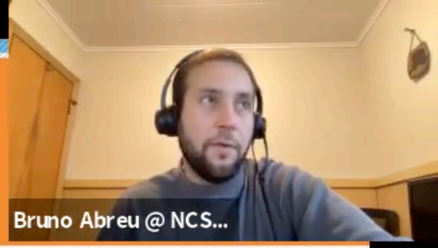
Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/saxpy/omp/saxpy_omp.cpp

```
    // stopwatch variables
    struct timeval start_time, stop_time, elapsed_time;

    // now call saxpy
    gettimeofday(&start_time, NULL);
    #pragma omp parallel
    {
        saxpy(a, x, y);
    }
    gettimeofday(&stop_time, NULL);

    timersub(&stop_time, &start_time, &elapsed_time);
    cout << "Elapsed time (s): " << elapsed_time.tv_sec + elapsed_time.tv_usec / 1000000.0 << endl;

    return 0;
}
```

Enclose function call with OpenMP **parallel** directive

1:22:46 / 1:45:40 • Parallel Workflow ›
Scroll for details

SAXPY: Parallelized code



Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/saxpy/omp/saxpy_omp.cpp

```
    // stopwatch variables
    struct timeval start_time, stop_time, elapsed_time;

    // now call saxpy
    gettimeofday(&start_time, NULL);
    #pragma omp parallel
    {
        saxpy(a, x, y);
    }
    gettimeofday(&stop_time, NULL);

    timersub(&stop_time, &start_time, &elapsed_time);
    cout << "Elapsed time (s): " << elapsed_time.tv_sec + elapsed_time.tv_usec / 1000000.0 << endl;

    return 0;
}
```

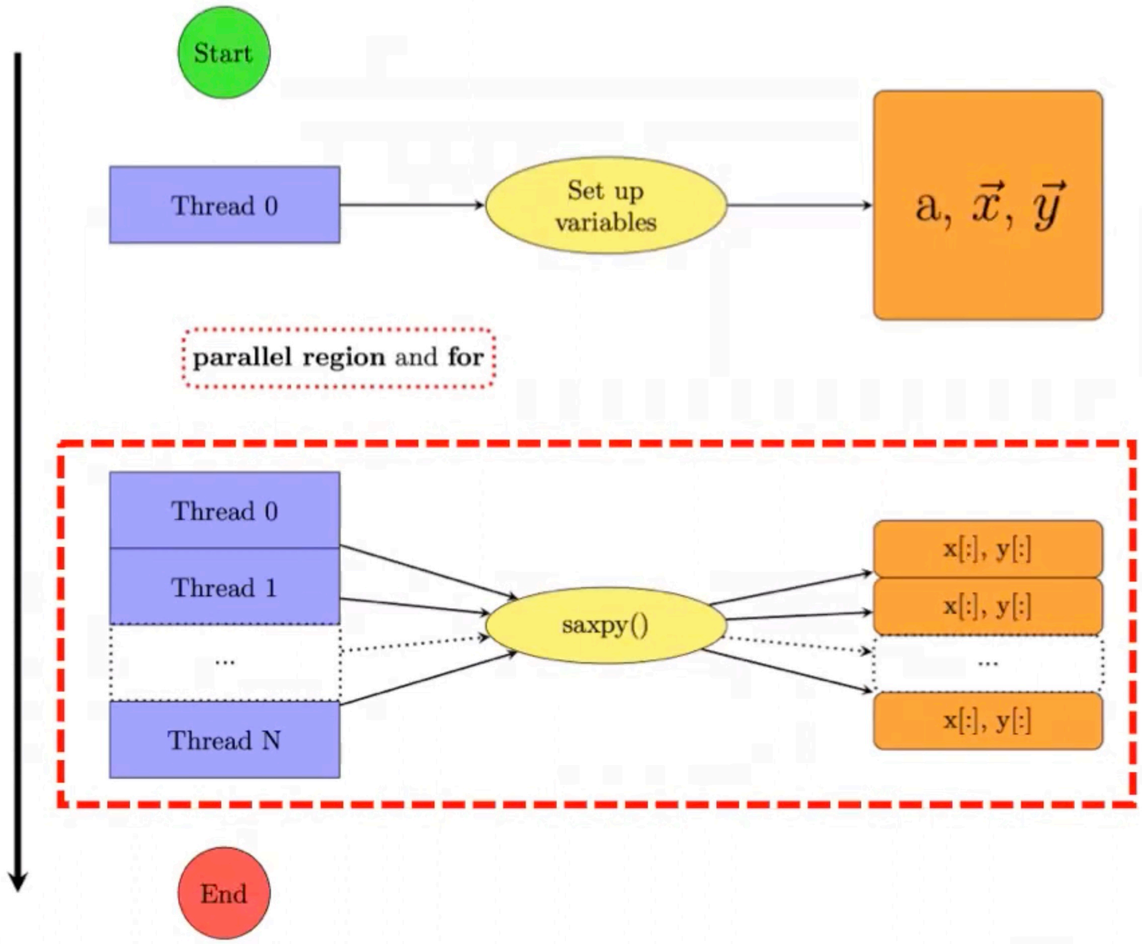
Insert OpenMP **for** directive at function level, right before loop

```
void saxpy(float a, const vector<float> &x, vector<float> &y)
{
    #pragma omp for
    for (int i = 0; i < x.size(); i++)
    {
        y[i] = a * x[i] + y[i];
    }
}
```


2021-11-17 11:41:51

SAXPY: Parallelized code workflow diagram

Find the code here: https://github.com/babreu-ncsa/IntroToPC/blob/main/saxpy/omp/saxpy_omp.cpp



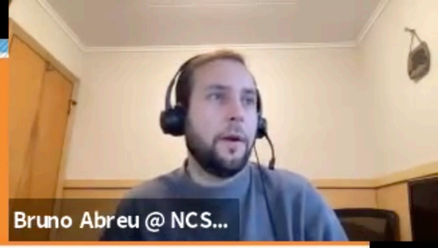

```
56      // now call saxpy
57      gettimeofday(&start_time, NULL);
58      #pragma omp parallel
59      {
60          saxpy(a, x, y);
61      }
62      gettimeofday(&stop_time, NULL);
63
```

```
70  void saxpy(float a, const vector<float> &x, vector<float> &y)
71  {
72      #pragma omp for
73      for (int i = 0; i < x.size(); i++)
74      {
75          y[i] = a * x[i] + y[i];
76      }
77  }
```

ILLINOIS NCSA

2021-11-17 11:42:37

SAXPY: Parallelized loop



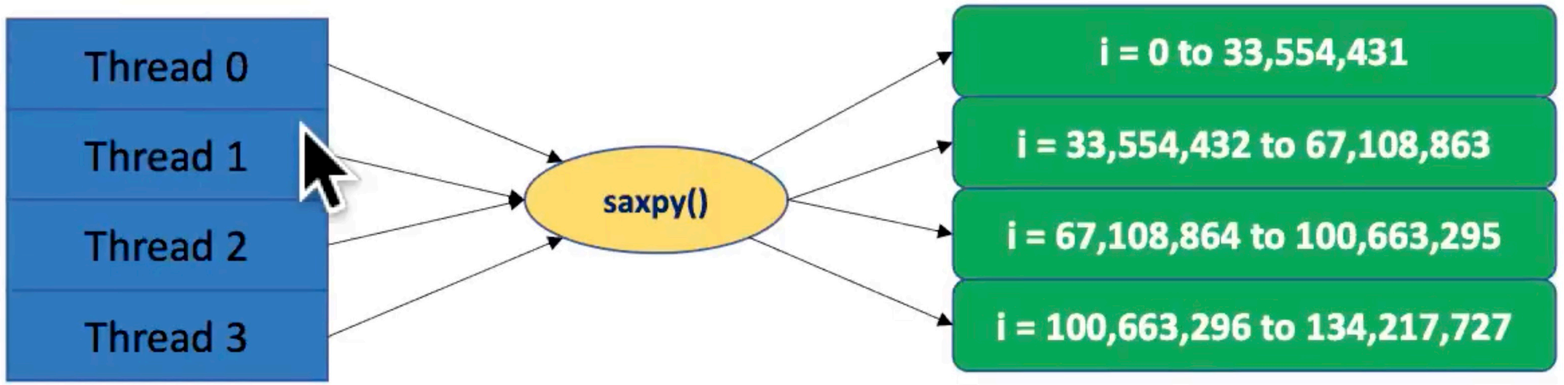

```
70  void saxpy(float a, const vector<float> &x, vector<float> &y)
71  {
72      #pragma omp for
73      for (int i = 0; i < x.size(); i++)
74      {
75          y[i] = a * x[i] + y[i];
76      }
77  }
```

2^{27} = 134,217,728 iterations

```
export OMP_NUM_THREADS=N
```

Each loop gets a chunk of size $\frac{2^{27}}{N}$ to work on

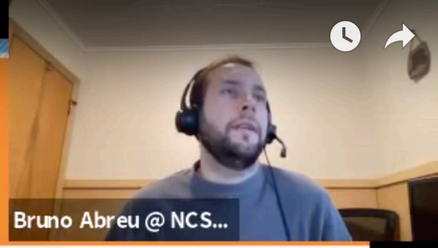
N = 4:



OpenMP does **static scheduling** by default

2021-11-17 11:43:26

SAXPY: Performance and scaling



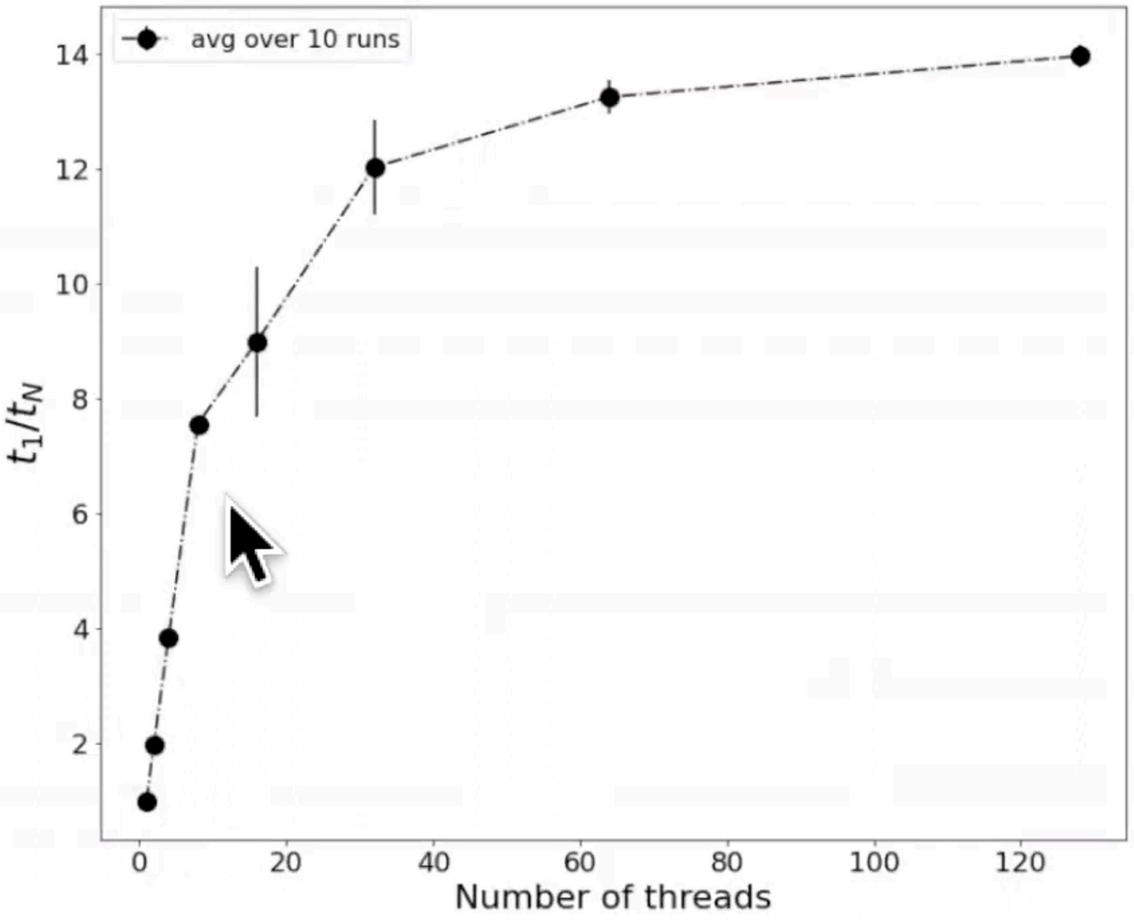
Here are some results with 10 runs for each N

Number of threads	Average elapsed time (s)	t_1 / t_N
1	0.7533	1.000
2	0.3810	1.977
4	0.1960	3.844
8	0.0995	7.569
16	0.0837	8.997
32	0.0626	12.035
64	0.0568	13.250
128	0.0539	13.972

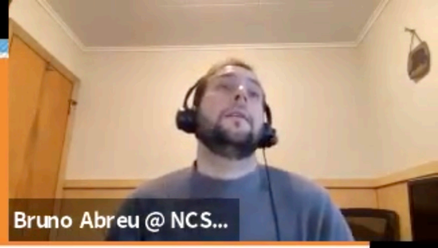
1:30:02 / 1:45:40 • Parallel Workflow ›
Scroll for details
2021-11-17 11:45

SAXPY: Performance and scaling

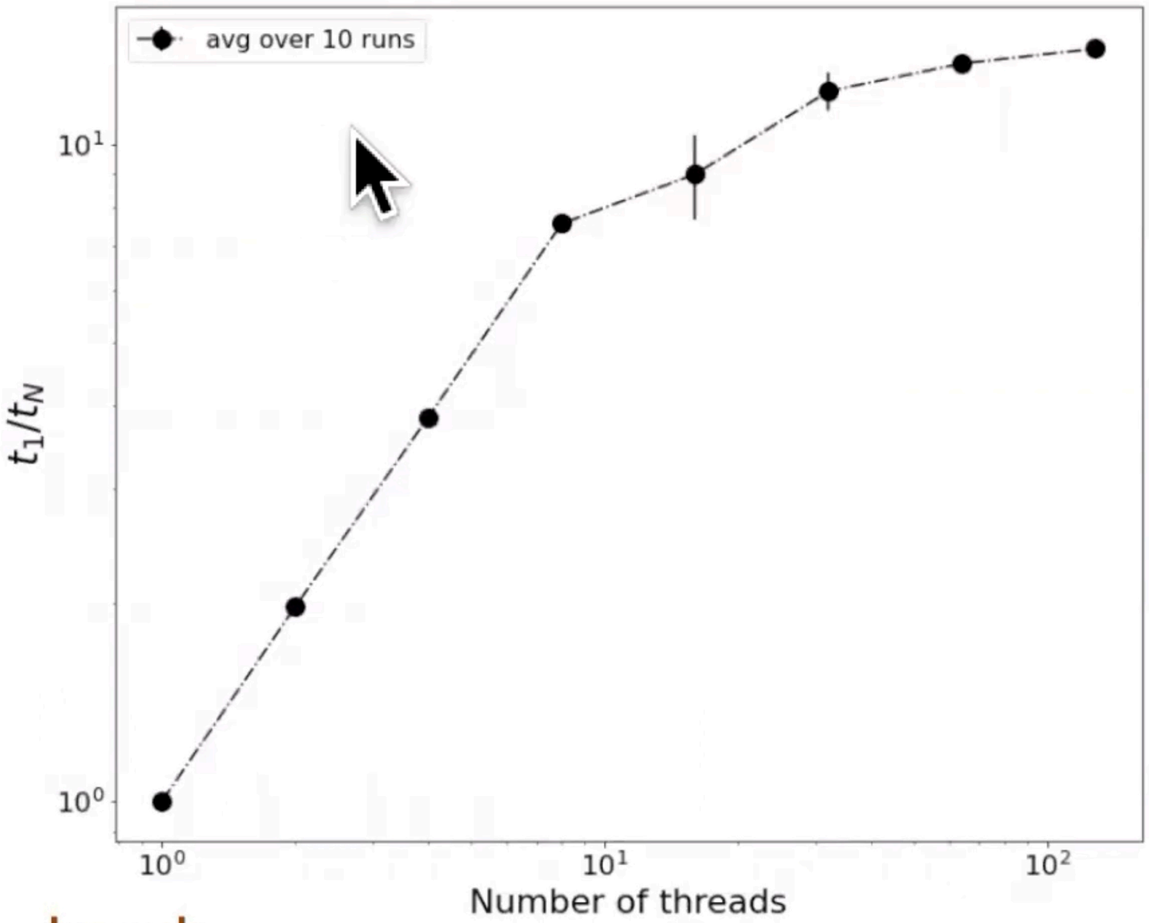
Number of threads	Average elapsed time (s)	t_1 / t_N
1	0.7533	1.000
2	0.3810	1.977
4	0.1960	3.844
8	0.0995	7.569
16	0.0837	8.997
32	0.0626	12.035
64	0.0568	13.260
128	0.0539	13.972



SAXPY: Performance and scaling



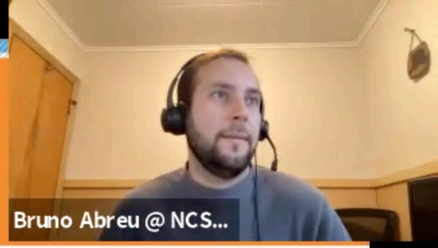
Number of threads	Average elapsed time (s)	t_1 / t_N
1	0.7533	1.000
2	0.3810	1.977
4	0.1960	3.844
8	0.0995	7.569
16	0.0837	8.997
32	0.0626	12.035
64	0.0568	13.260
128	0.0539	13.972



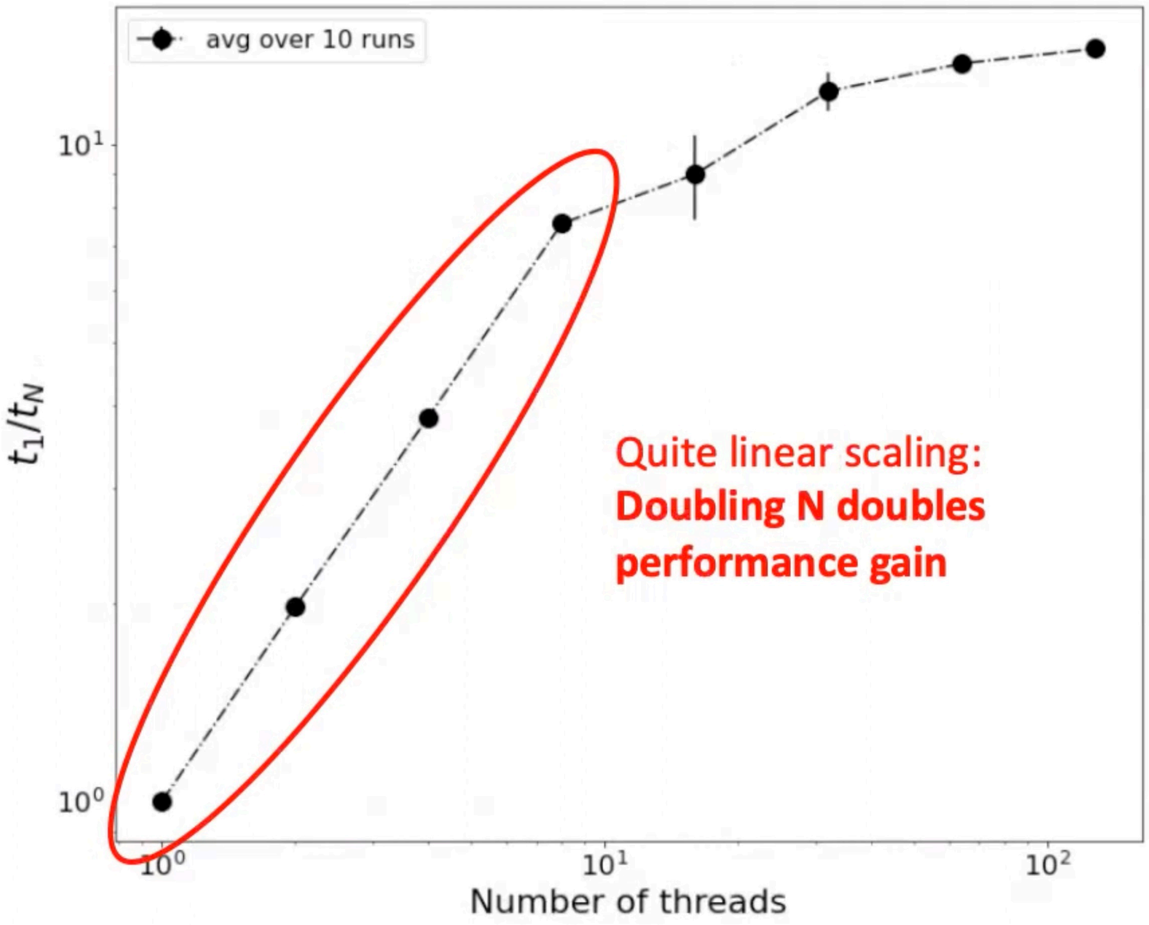
log scale

2021-11-17 11:49:29

SAXPY: Performance and scaling

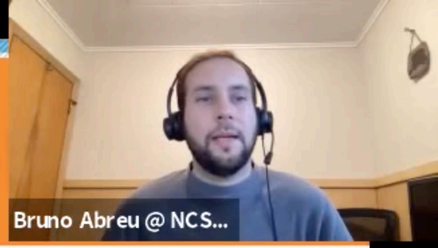


Number of threads	Average elapsed time (s)	t_1 / t_N
1	0.7533	1.000
2	0.3810	1.977
4	0.1960	3.84
8	0.0995	7.569
16	0.0837	8.997
32	0.0626	12.035
64	0.0568	13.260
128	0.0539	13.972

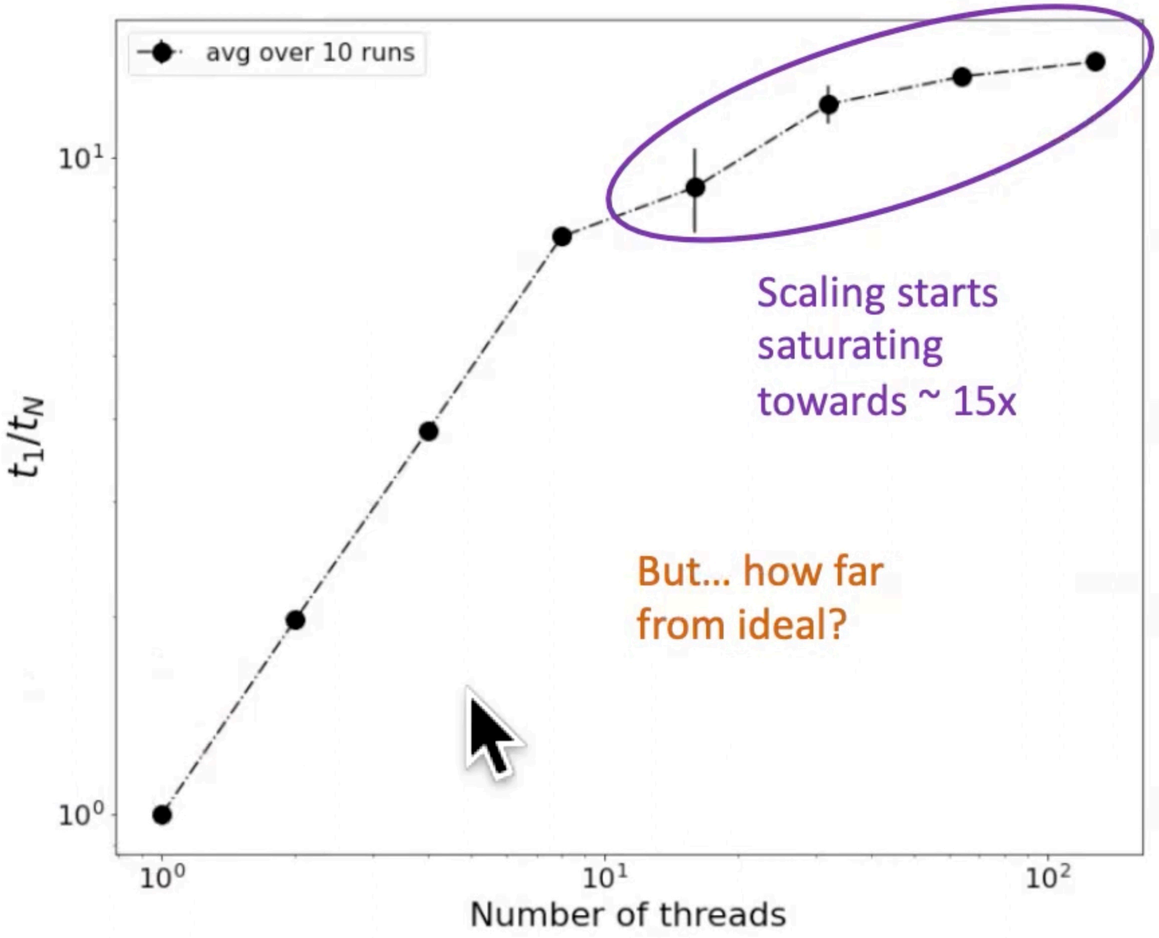


2021-11-17 11:49:51

SAXPY: Performance and scaling

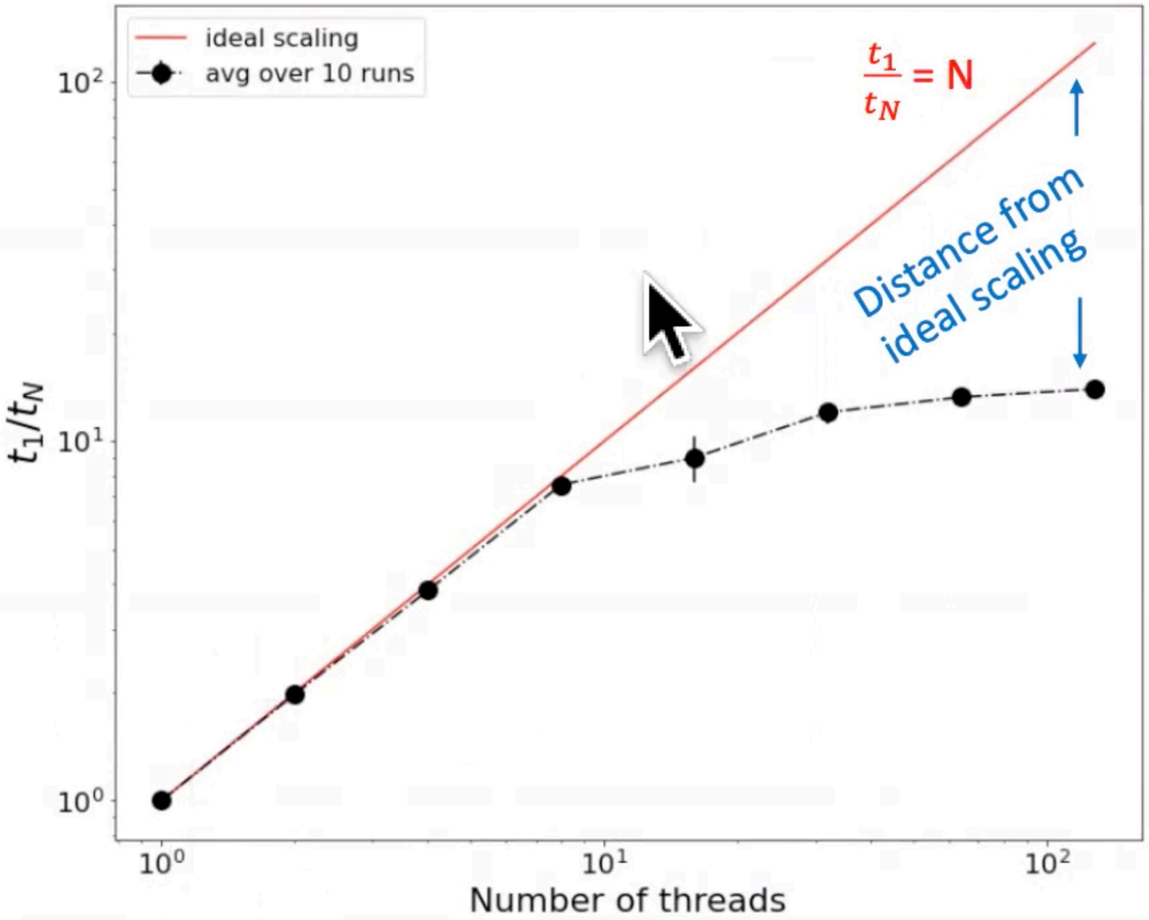


Number of threads	Average elapsed time (s)	t_1 / t_N
1	0.7533	1.000
2	0.3810	1.977
4	0.1960	3.844
8	0.0995	7.569
16	0.0837	8.997
32	0.0626	12.035
64	0.0568	13.260
128	0.0539	13.972



SAXPY: Performance and scaling

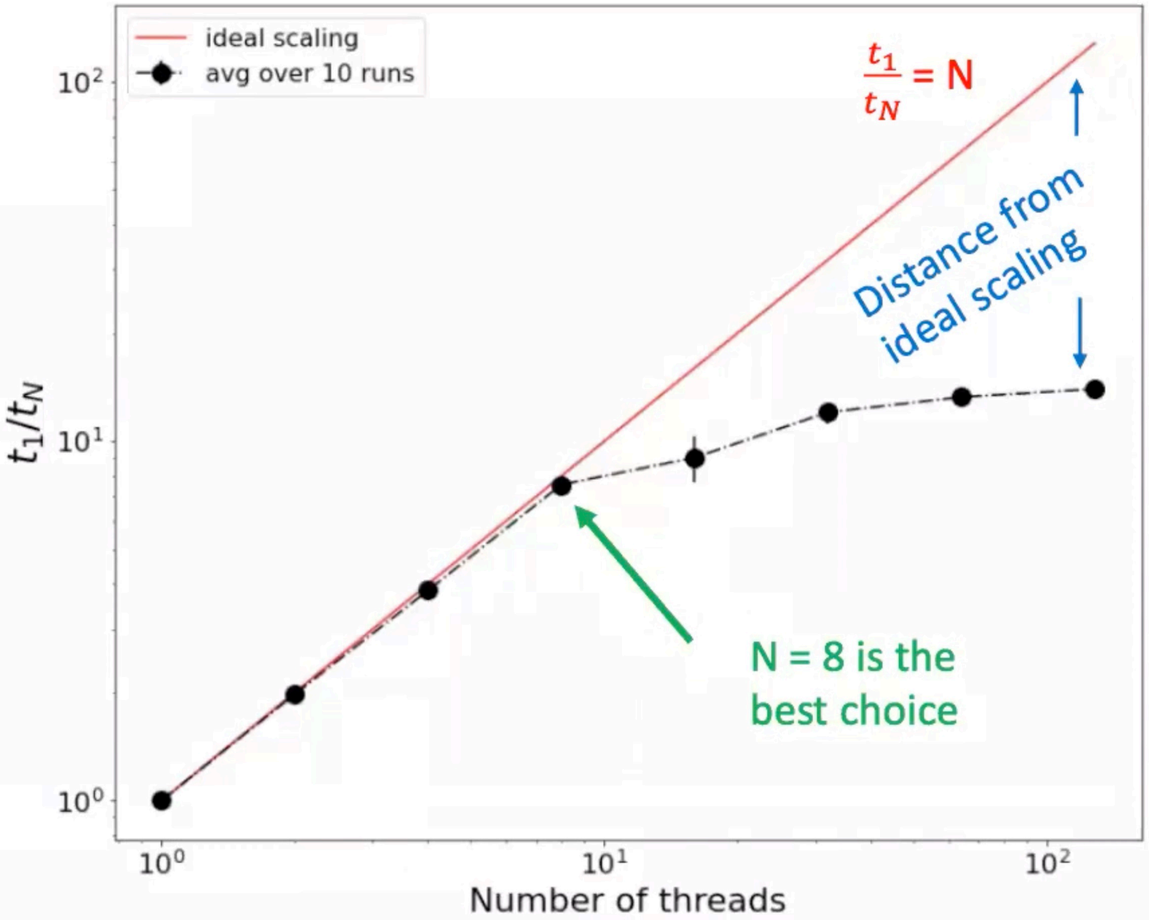
Number of threads	Average elapsed time (s)	t_1 / t_N
1	0.7533	1.000
2	0.3810	1.977
4	0.1960	3.844
8	0.0995	7.569
16	0.0837	8.997
32	0.0626	12.035
64	0.0568	13.260
128	0.0539	13.972



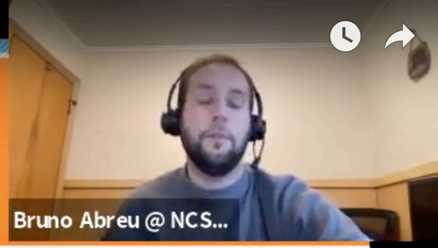
2021-11-17 11:50:33

SAXPY: Performance and scaling

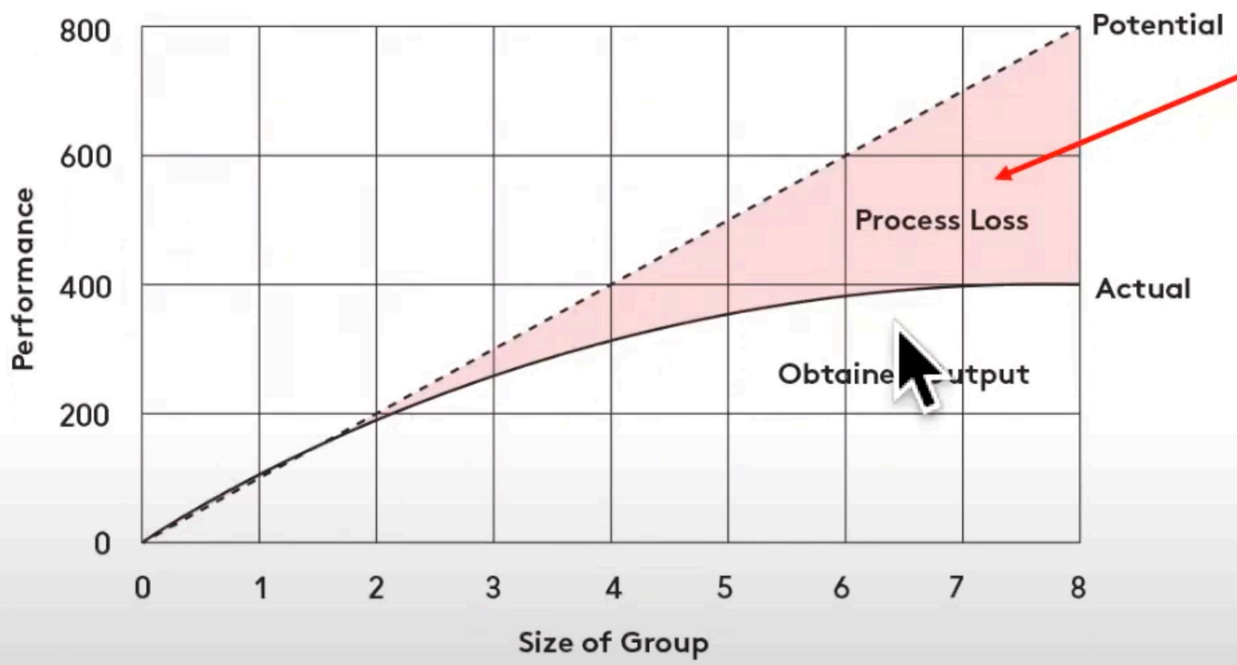
Number of threads	Average elapsed time (s)	t_1 / t_N
1	0.7533	1.000
2	0.3810	1.977
4	0.1960	3.844
8	0.0995	7.569
16	0.0837	8.997
32	0.0626	12.035
64	0.0568	13.260
128	0.0539	13.972



SAXPY: Performance analysis



Threads are not much different from people...



This is called the **Ringelmann effect**

Losses generally come from:

- More complex management requirements
- Increased communication requirements
- Task conflicts

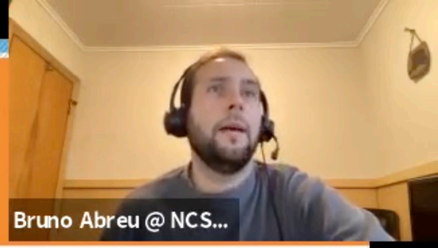
In parallel computing language, this is called **parallel overhead**

- Creating, distributing workloads and destroying a larger number of threads simply takes longer!

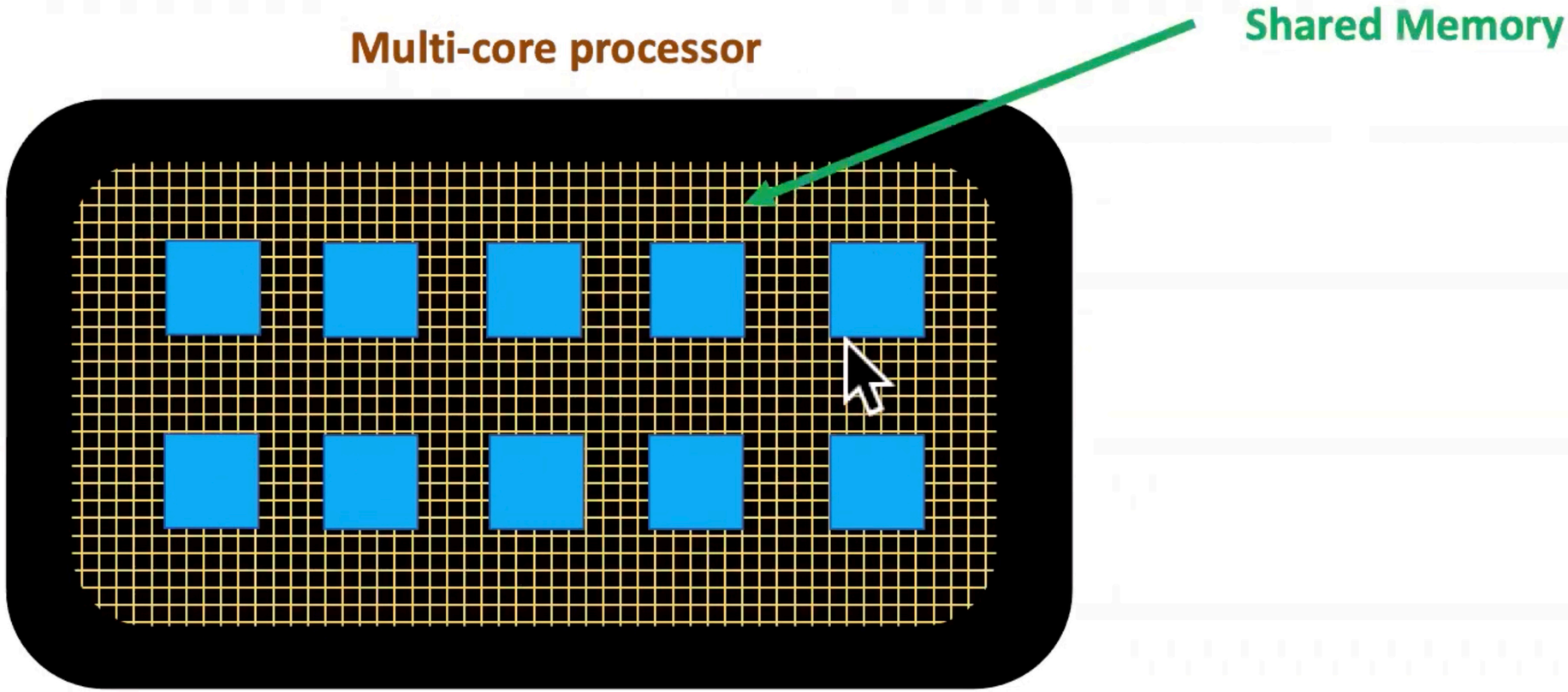
BUT...

1:34:01 / 1:45:40 • Parallel Workflow ›
Scroll for details
2021-11-17 11:44

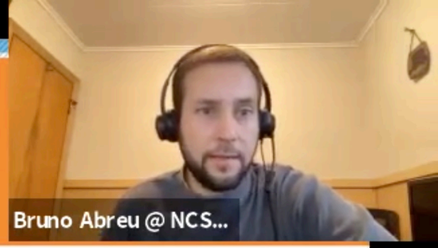
SAXPY: Performance analysis



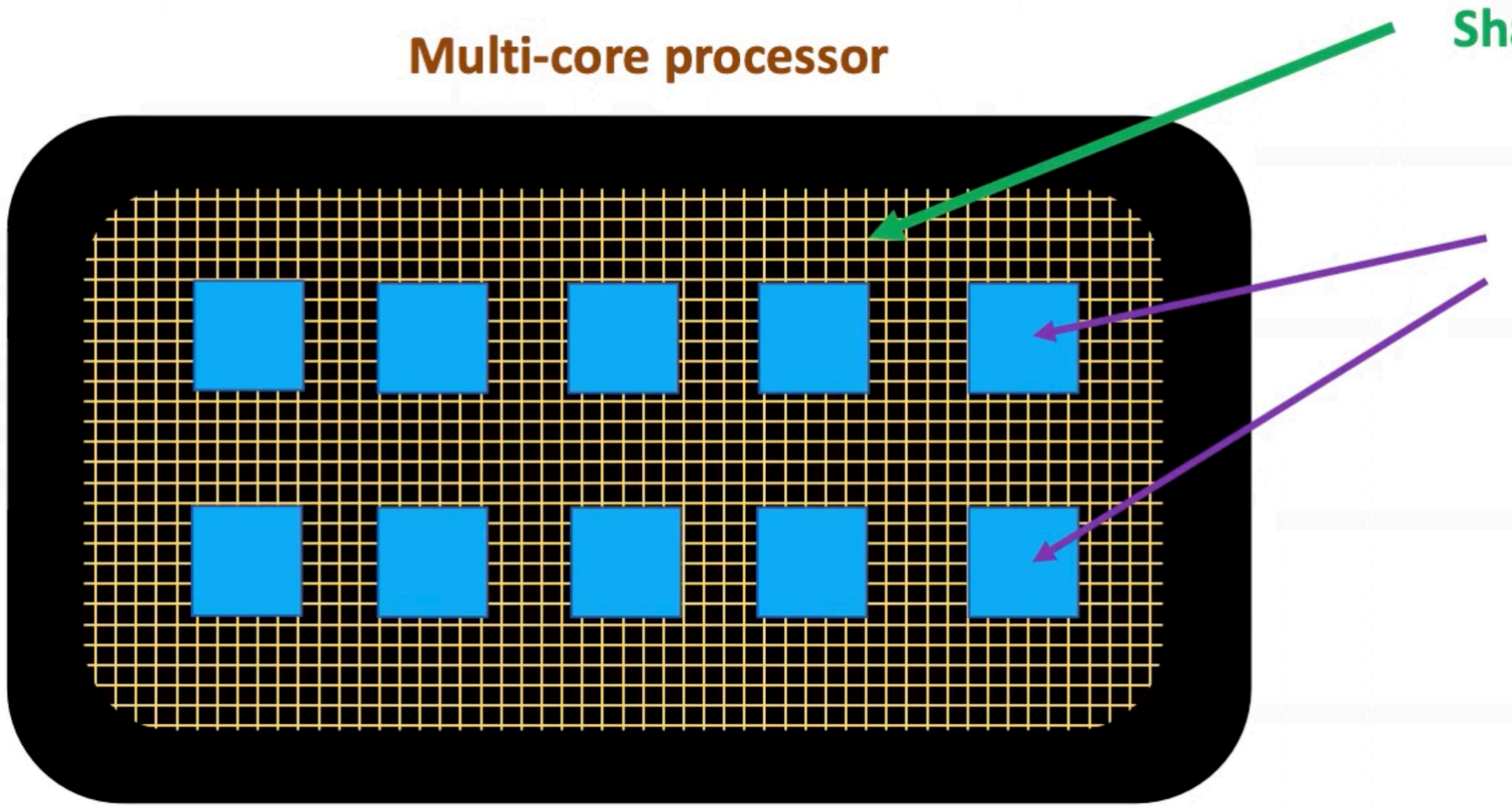
Parallel overhead is not that much of a problem here…



SAXPY: Performance analysis



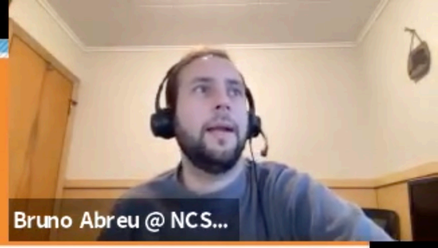
Parallel overhead is not that much of a problem here…



Each **thread** is usually associated to a certain **CPU** (thread affinity)

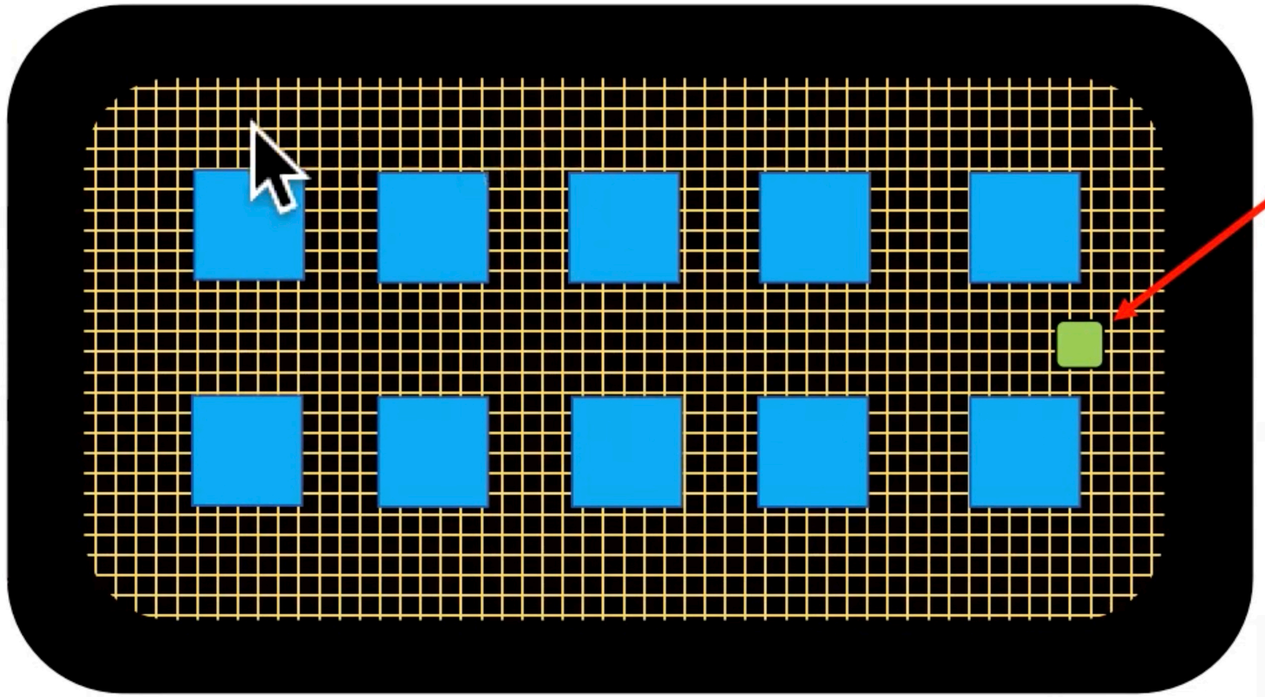
2021-11-17 11:54:24

SAXPY: Performance analysis



We start the code by allocating two vectors, **x** and **y**, with 2^{27} entries each…

Multi-core processor

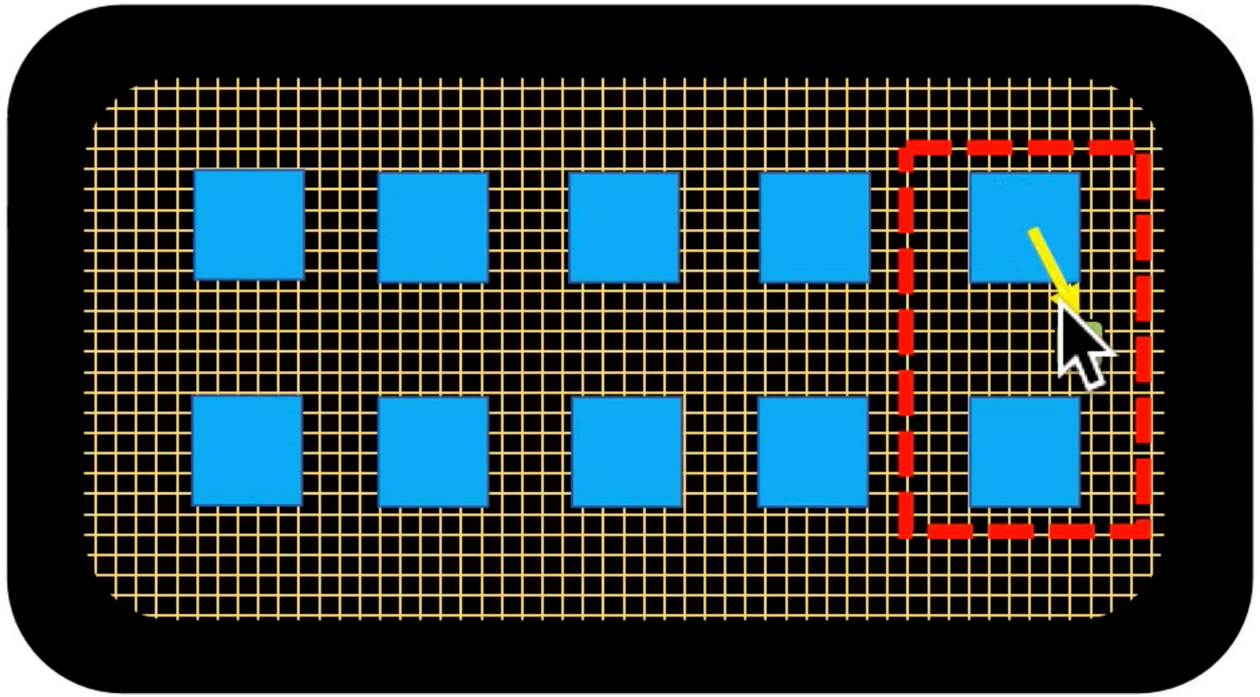


Which is only about 1 GB out a total of 256 GB!

2021-11-17 11:54:45

SAXPY: Performance analysis

Multi-core processor

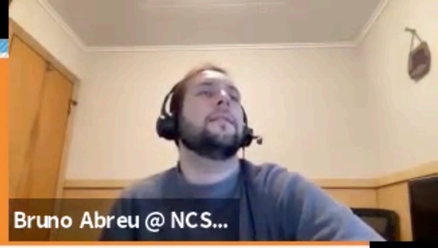


Maximum distance that a thread needs to travel to access memory address

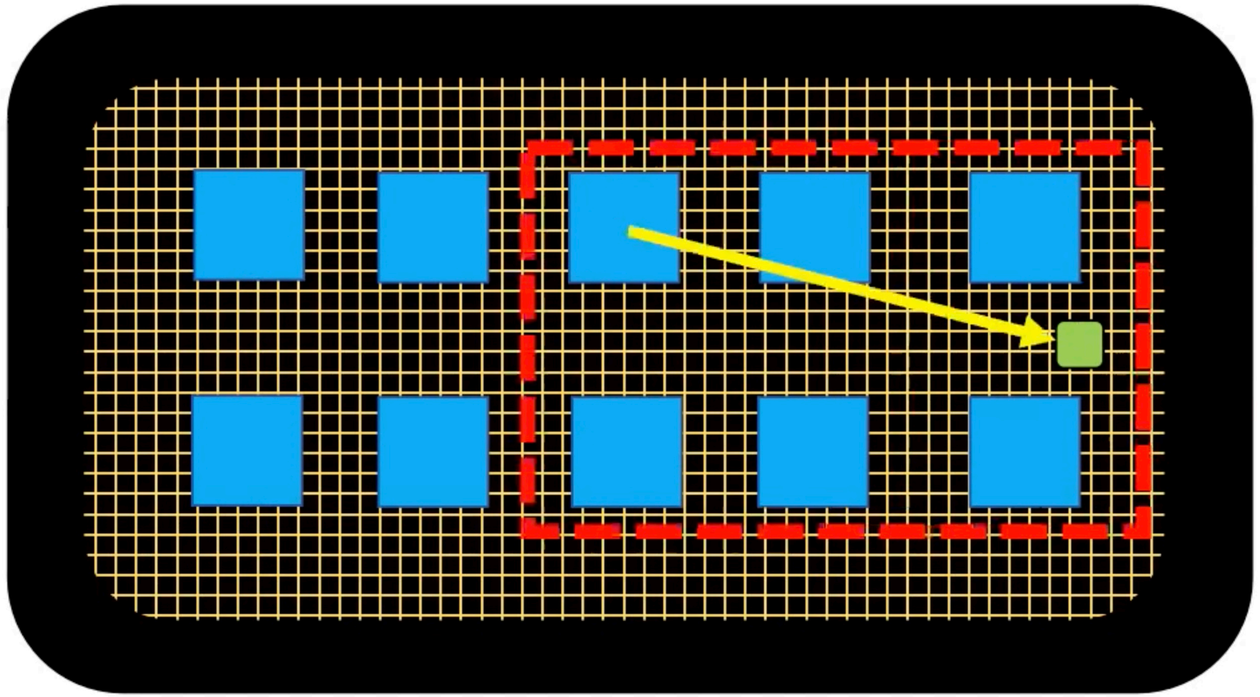
N = 2 Threads

2021-11-17 11:55:15

SAXPY: Performance analysis



Multi-core processor

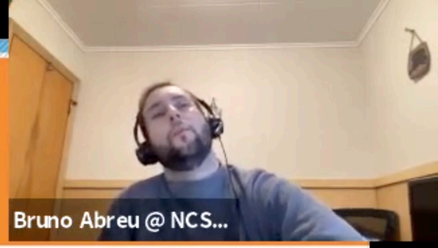


Maximum distance that a thread needs to travel to access memory address

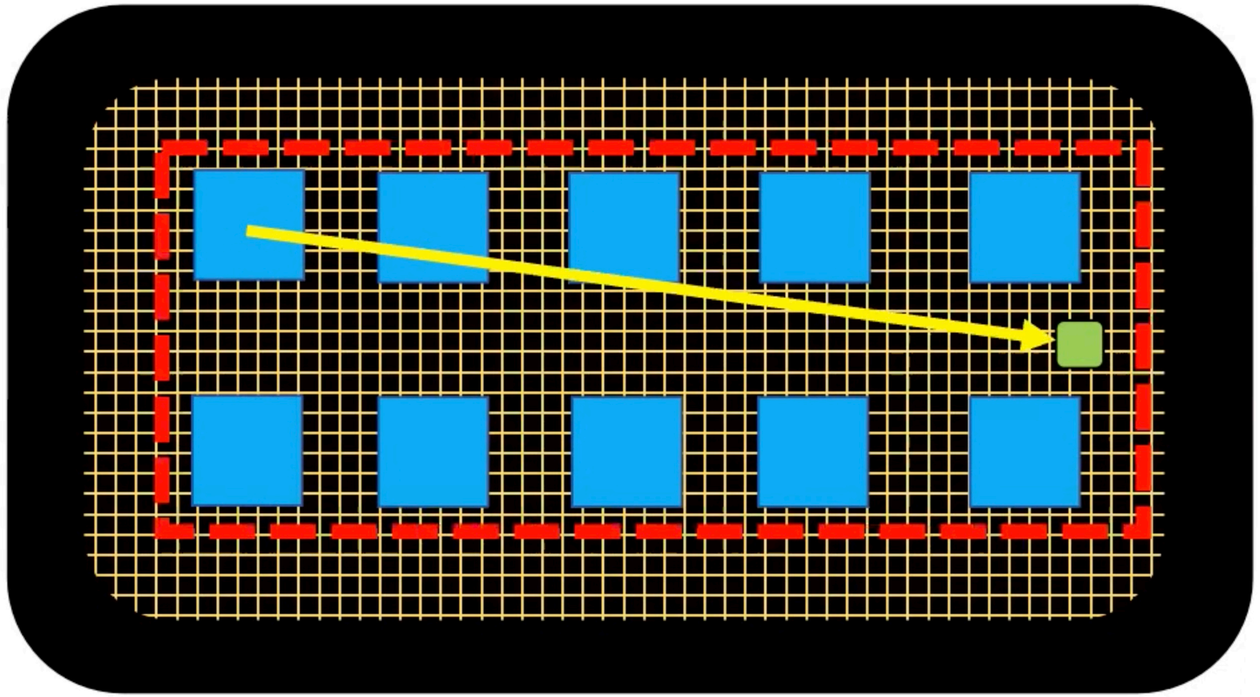
N = 6 Threads

2021-11-17 11:55:22

SAXPY: Performance analysis



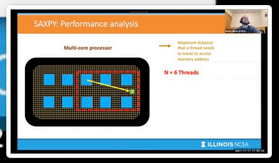
Multi-core processor



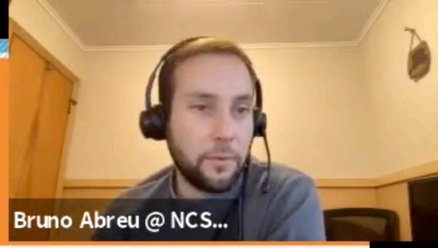
Maximum distance that a thread needs to travel to access memory address

N = 10 Threads

Ideal scaling requires problem scaling as well!



Summary



Parallel computing is essential for modern scientific research computing

Computers today execute multiple instructions on multiple pieces of data (MIMD)

OpenMP allows for incremental parallelization with very simple directives

Using Task Parallelism, we improved the performance of the STATS application by 2.4x

Using Data Parallelism, we improved the performance of the SAXPY application by 8x

2021-11-17 11:56:21

Take-Home Message

If your serial code takes **one week** to run… **168h**

- **Take this course** **2h**
- **Work on additional exercises** **2h**
- **Parallelize your code, test it** **4h**
- **Run your code faster (~8x)** **21h**

$$\text{ROI} = \frac{168\text{h} - 21\text{h}}{2\text{h} + 2\text{h} + 4\text{h}} = 18.4$$

For **one single** run!

1:39:41 / 1:45:40 • Parallel Workflow ›
Scroll for details
2021-11-17 11:50:24

Take-home message

If your serial code takes **one week** to run… **168h**

- **Take this course** **2h**
- **Work on additional exercises** **2h**
- **Parallelize your code, test it** **4h**
- **Run your code faster (~8x)** **21h**

$$\text{ROI} = \frac{168\text{h} - 21\text{h}}{2\text{h} + 2\text{h} + 4\text{h}} = 18.4$$

For **one single** run!

It is worth giving it a try, and we are here to help you!

2021-11-17 11:58:59