Intro to Multithreading

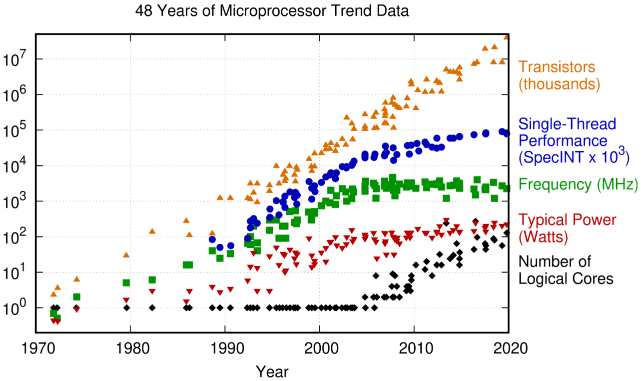
Nick Derr

Applied Math 205
Harvard University

September 22, 2020

The processor (CPU/central processing unit) is the collection of circuitry which executes the instructions making up a computer program.

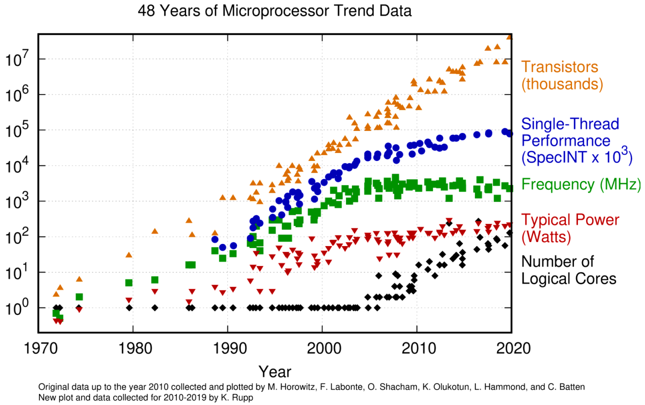
CPU clock speeds have stagnated since the mid-2000s. Hardware improvements have come in the form of more cores per CPU.



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

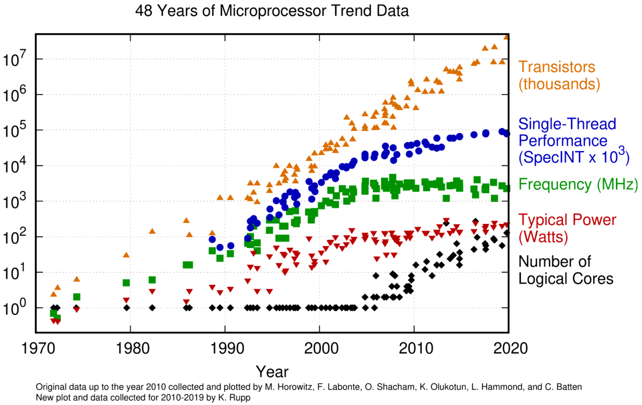
- [link to data]

CPU clock speeds have stagnaged since the mid-2000s. Hardware improvements have come in the form of more cores per CPU.



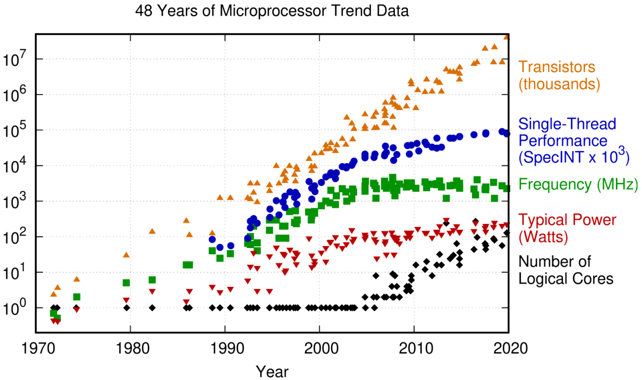
- [link to data]
- faster clock speeds $\Longrightarrow$ higher power

CPU clock speeds have stagnaged since the mid-2000s. Hardware improvements have come in the form of more cores per CPU.



- [link to data]
- faster clock speeds $\implies$ higher power
- more cores + slower clock speeds can yield **higher** compute power with **lower** power consumption

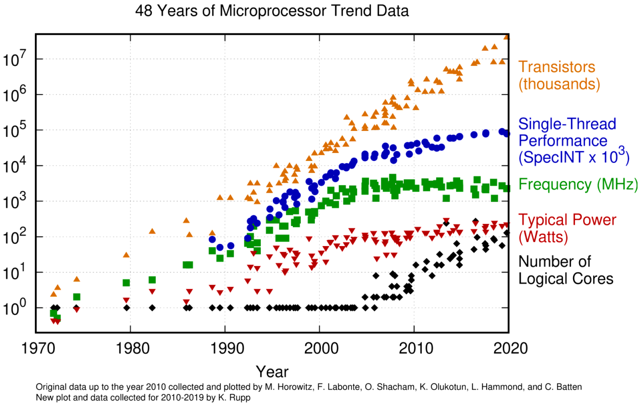
CPU clock speeds have stagnaged since the mid-2000s. Hardware improvements have come in the form of more cores per CPU.



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

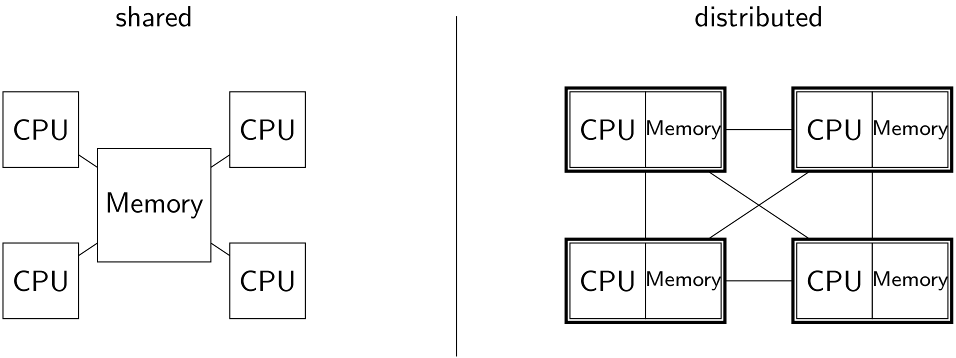
- [link to data]
- faster clock speeds $\implies$ higher power
- more cores + slower clock speeds can yield **higher** compute power with **lower** power consumption
- higher core counts are becoming more and more widely available

CPU clock speeds have stagnaged since the mid-2000s. Hardware improvements have come in the form of more cores per CPU.



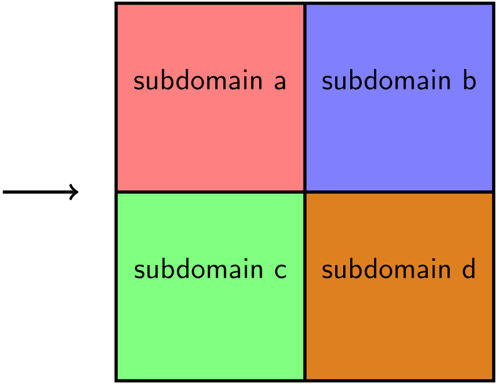
- [link to data]
- faster clock speeds $\implies$ higher power
- more cores + slower clock speeds can yield **higher** compute power with **lower** power consumption
- higher core counts are becoming more and more widely available
- important to understand how to adapt or write code to take advantage of parallel architecture

There are two main paradigms of code parallelization: shared memory and distributed memory.



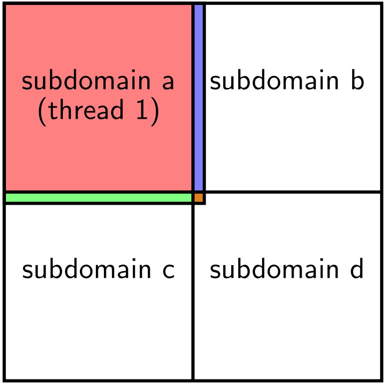
Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

One method of memory distribution is to divy up the **simulation domain** across the various threads.



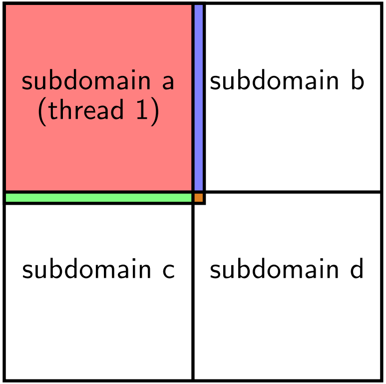
Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

- each thread contains data in its own subdomain, plus some buffer data



Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated

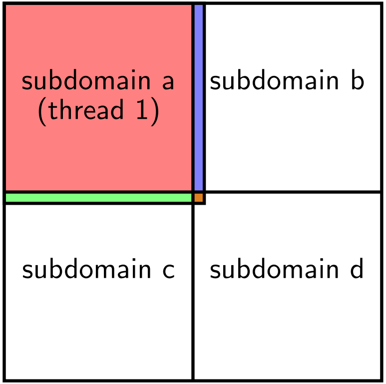


Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated
- buffer data is copied from neighboring subdomains

  $\Longrightarrow$ need **communication**

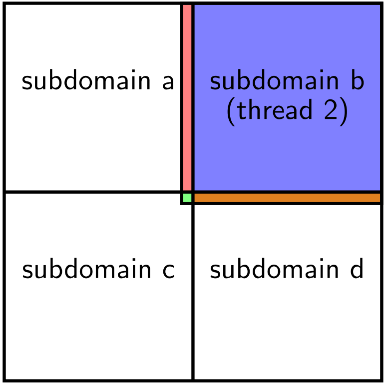
  b, c, d send to a



Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated
- buffer data is copied from neighboring subdomains
  $\Longrightarrow$ need **communication**

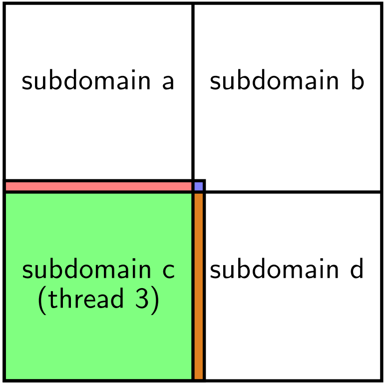
  a, c, d send to b



Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated
- buffer data is copied from neighboring subdomains
  $\Longrightarrow$ need **communication**

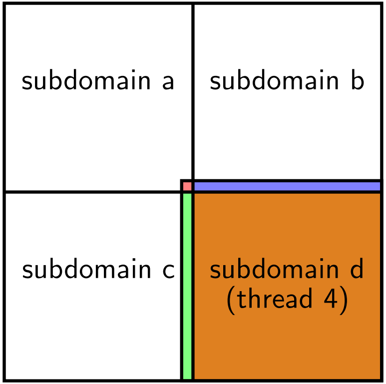
  a, b, d send to c



Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

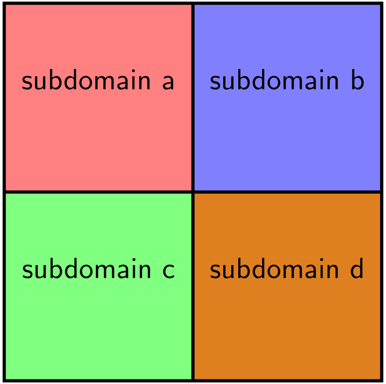
- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated
- buffer data is copied from neighboring subdomains
  $\implies$ need **communication**

  a, b, c send to d



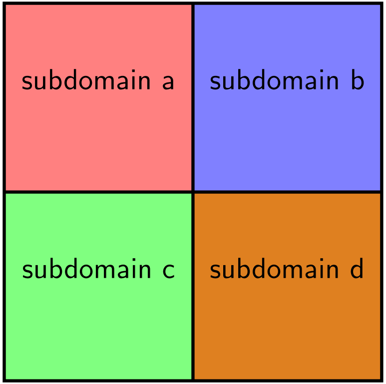
Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated
- buffer data is copied from neighboring subdomains
  $\implies$ need **communication**
- requires "message passing interface" (e.g. OpenMPI)



Programs parallelized with distributed memory run as several independent threads, pausing only to periodically communicate amongst each other.

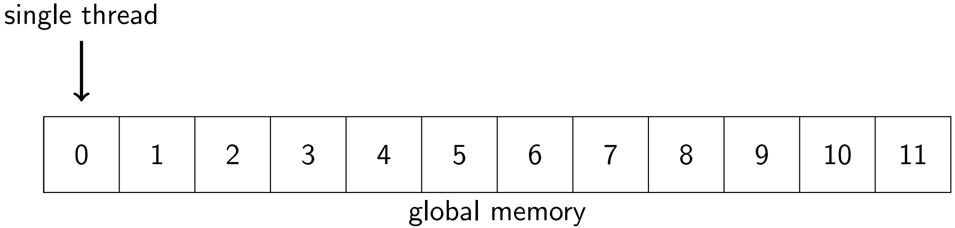
- each thread contains data in its own subdomain, plus some buffer data
- owned data is simulated
- buffer data is copied from neighboring subdomains
  $\Longrightarrow$ need **communication**
- requires "message passing interface" (e.g. OpenMPI)



Discuss: benefits? costs?

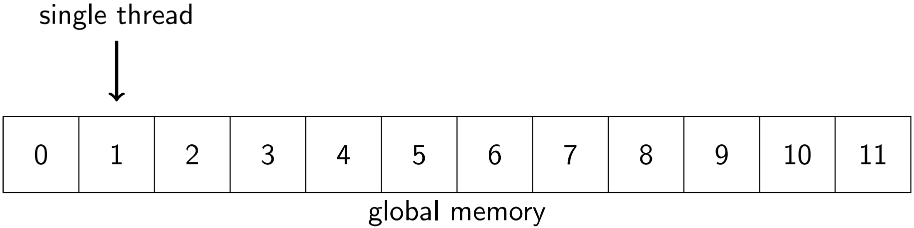
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



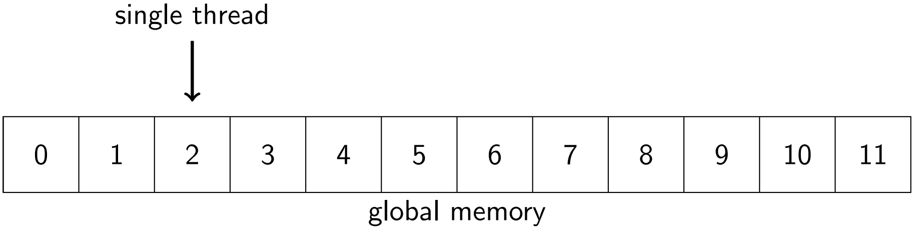
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



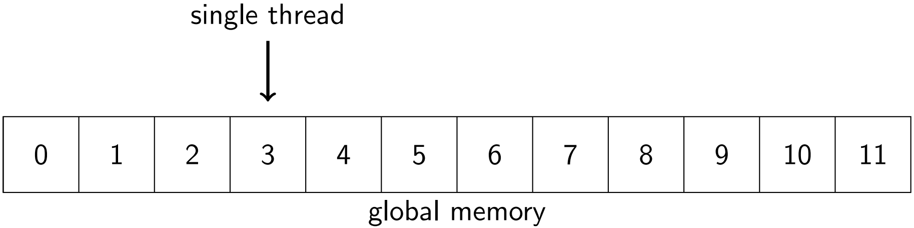
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



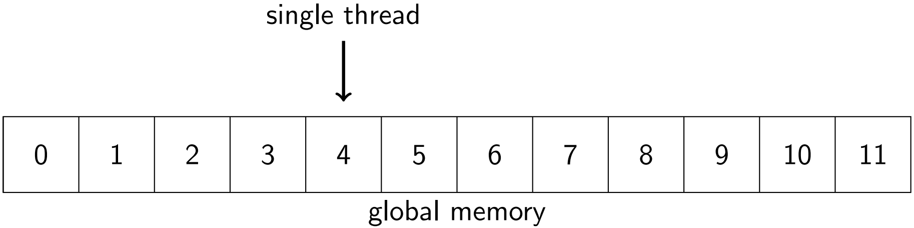
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



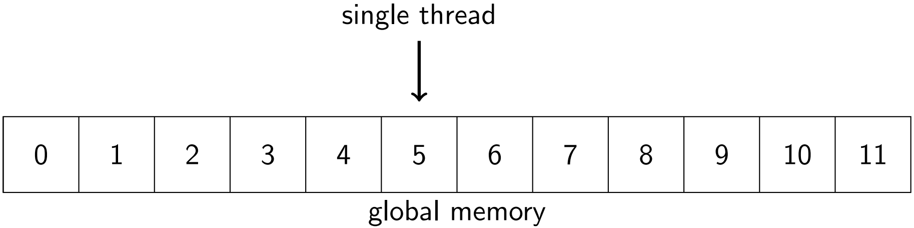
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



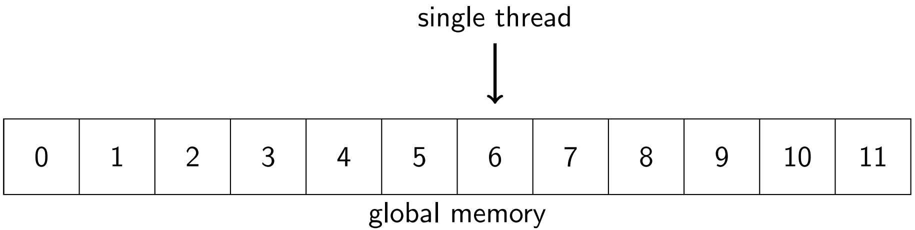
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



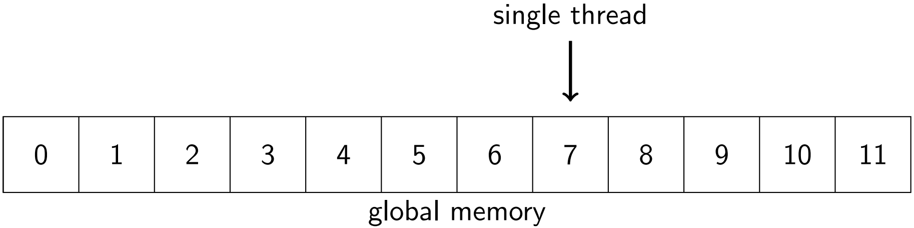
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



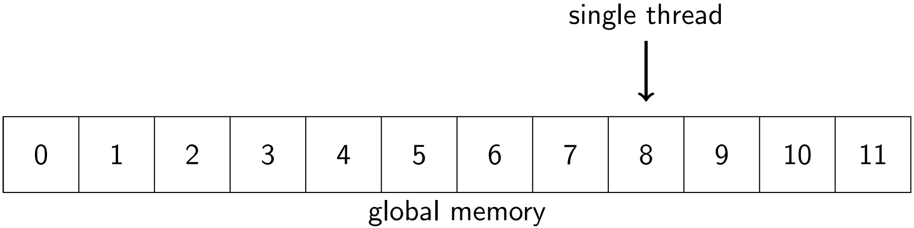
Programs using shared memory apply “divide-and-conquer” to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



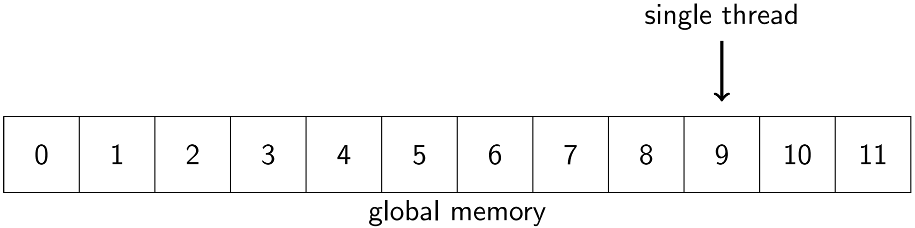
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



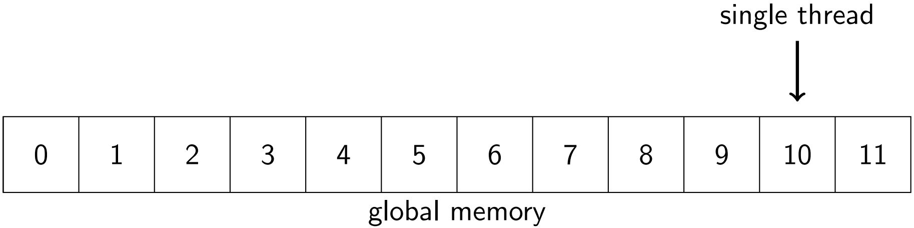
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



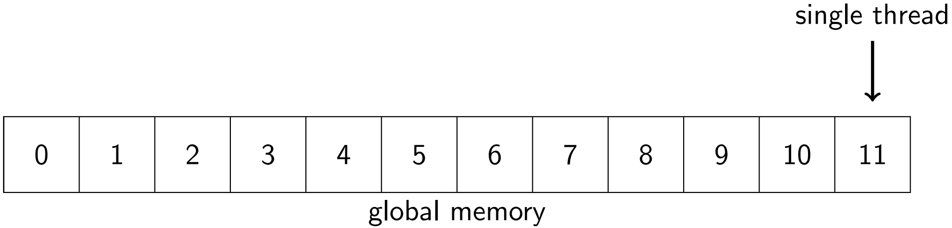
Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory



Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

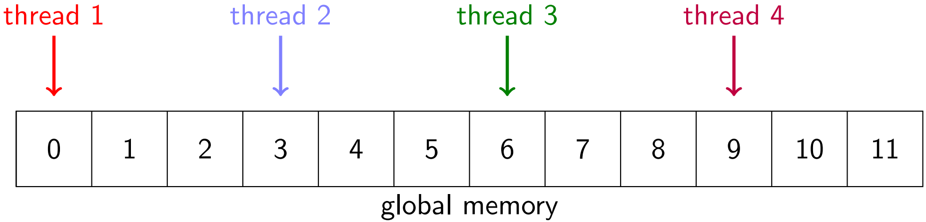
A single thread loops sequentially through each spot in memory



Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory

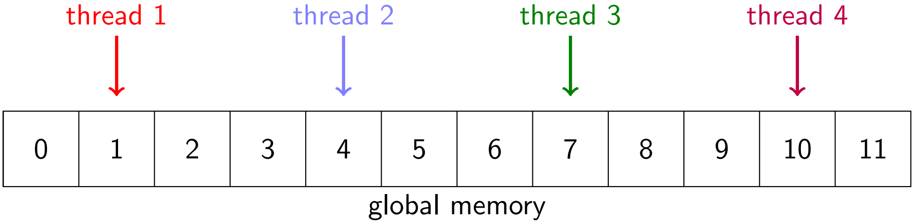
...but multiple threads can traverse different locations in memory at the same time.



Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory

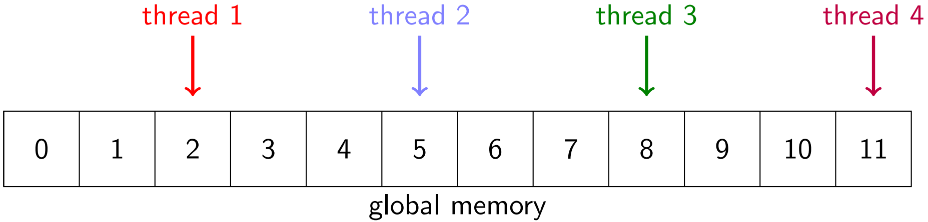
...but multiple threads can traverse different locations in memory at the same time.



Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory

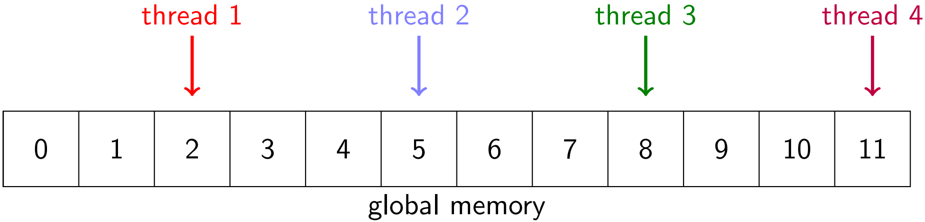
...but multiple threads can traverse different locations in memory at the same time.



Programs using shared memory apply "divide-and-conquer" to specific tasks, spawning multiple threads to divy up large tasks when necessary.

A single thread loops sequentially through each spot in memory

...but multiple threads can traverse different locations in memory at the same time.



Discuss: benefits? costs?

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$:

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$:

  savings $\approx$ cost (no benefit to more/fewer)

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$:

  savings $\approx$ cost (no benefit to more/fewer)
- $e_k \approx 0$:

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$:

  savings $\approx$ cost (no benefit to more/fewer)
- $e_k \approx 0$:

  savings $\ll$ cost (use fewer threads)

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$:

  savings $\approx$ cost (no benefit to more/fewer)
- $e_k \approx 0$:

  savings $\ll$ cost (use fewer threads)

How do costs relate to job size N?

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$:

  savings $\approx$ cost (no benefit to more/fewer)
- $e_k \approx 0$:

  savings $\ll$ cost (use fewer threads)

How do costs relate to job size N?

- overhead for k threads $\propto k$

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$:

  savings $\approx$ cost (no benefit to more/fewer)
- $e_k \approx 0$:

  savings $\ll$ cost (use fewer threads)

How do costs relate to job size N?

- overhead for k threads $\propto k$
- as k grows, $k > N$ (threading becomes bigger job than original goal)

Parallelizing decreases calculation time, but there is a point of diminishing returns due to, e.g., spawning overhead, memory bandwidth saturation

Let $T_k :=$ the time to perform a calculation across k threads. Efficiency on k threads:

$$e_k = \frac{T_1}{kT_k} < 1.$$

What do efficiency values imply?

- $e_k \approx 1$:

  savings $\gg$ cost (use more threads)
- $e_k \approx 1/k$:

  savings $\approx$ cost (no benefit to more/fewer)
- $e_k \approx 0$:

  savings $\ll$ cost (use fewer threads)

How do costs relate to job size N?

- overhead for k threads $\propto k$
- as k grows, $k > N$ (threading becomes bigger job than original goal)
- as N grows, problem may become memory bandwidth-limited (implementation-dependent)

In C++, shared-memory parallelization can be implemented with the OpenMP API.

Anatomy of an OMP call:

```
// add two vectors of length N
static const int N=10;
double v1[N],v2[N],v3[N];

#pragma omp parallel for
for (int i=0; i<N; i++) {
    v3[i] = v1[i] + v2[i];
}
```

In C++, shared-memory parallelization can be implemented with the OpenMP API.

Anatomy of an OMP call:

```
// add two vectors of length N
static const int N=10;
double v1[N],v2[N],v3[N];

#pragma omp parallel for
for (int i=0; i<N; i++) {
    v3[i] = v1[i] + v2[i];
}
```

- Identifies preprocessor-OpenMP directive. Compiler will implement instructions if OMP is supported and appropriate compile flags are present

In C++, shared-memory parallelization can be implemented with the OpenMP API.

Anatomy of an OMP call:

```
// add two vectors of length N
static const int N=10;
double v1[N],v2[N],v3[N];

#pragma omp parallel for
for (int i=0; i<N; i++) {
    v3[i] = v1[i] + v2[i];
}
```

- Identifies preprocessor-OpenMP directive. Compiler will implement instructions if OMP is supported and appropriate compile flags are present
- Spawn multiple threads

In C++, shared-memory parallelization can be implemented with the OpenMP API.

Anatomy of an OMP call:

```
// add two vectors of length N
static const int N=10;
double v1[N],v2[N],v3[N];

#pragma omp parallel for
for (int i=0; i<N; i++) {
    v3[i] = v1[i] + v2[i];
}
```

- Identifies preprocessor-OpenMP directive. Compiler will implement instructions if OMP is supported and appropriate compile flags are present
- Spawn multiple threads
- Distribute loop iterations among spawned threads, i.e. thread 1: $i = [0, N/2]$, thread 2: $i = (N/2, N]$.

In C++, shared-memory parallelization can be implemented with the OpenMP API.

Anatomy of an OMP call:

```
// add two vectors of length N
static const int N=10;
double v1[N],v2[N],v3[N];

#pragma omp parallel for
for (int i=0; i<N; i++) {
    v3[i] = v1[i] + v2[i];
}
```

- Identifies preprocessor-OpenMP directive. Compiler will implement instructions if OMP is supported and appropriate compile flags are present
- Spawn multiple threads
- Distribute loop iterations among spawned threads, i.e. thread 1: $i = [0, N/2]$, thread 2: $i = (N/2, N]$.

Discuss: why is this for loop amenable to parallelization in this way?

In C++, shared-memory parallelization can be implemented with the OpenMP API.

Anatomy of an OMP call:

```
// add two vectors of length N
static const int N=10;
double v1[N],v2[N],v3[N];

#pragma omp parallel for
for (int i=0; i<N; i++) {
    v3[i] = v1[i] + v2[i];
}
```

- Identifies preprocessor-OpenMP directive. Compiler will implement instructions if OMP is supported and appropriate compile flags are present
- Spawn multiple threads
- Distribute loop iterations among spawned threads, i.e. thread 1: $i = [0, N/2]$, thread 2: $i = (N/2, N]$.

Discuss: why is this for loop amenable to parallelization in this way?
$\Longrightarrow$ no one iteration depends on another! “embarrassingly parallel”

Explicit precautions must be taken if multiple threads are writing to one location in memory or a value is being calculated across multiple threads.

```
// dot product (method 1)
double dot=0;
#pragma omp parallel for shared(dot)
for (int i=0; i<N; i++) {

#pragma omp atomic
    dot += v1[i] * v2[i];
}
```

Explicit precautions must be taken if multiple threads are writing to one location in memory or a value is being calculated across multiple threads.

```
// dot product (method 1)
double dot=0;
#pragma omp parallel for shared(dot)
for (int i=0; i<N; i++) {

#pragma omp atomic
    dot += v1[i] * v2[i];
}
```

- mark dot as a shared variable among all threads (this is actually the default behavior)

Explicit precautions must be taken if multiple threads are writing to one location in memory or a value is being calculated across multiple threads.

```
// dot product (method 1)
double dot=0;
#pragma omp parallel for shared(dot)
for (int i=0; i<N; i++) {

#pragma omp atomic
    dot += v1[i] * v2[i];
}
```

- mark dot as a shared variable among all threads (this is actually the default behavior)
- require that only one thread access dot in memory to update its value at one time

Explicit precautions must be taken if multiple threads are writing to one location in memory or a value is being calculated across multiple threads.

```
// dot product (method 1)
double dot=0;
#pragma omp parallel for shared(dot)
for (int i=0; i<N; i++) {

#pragma omp atomic
    dot += v1[i] * v2[i];
}
```

- mark dot as a shared variable among all threads (this is actually the default behavior)
- require that only one thread access dot in memory to update its value at one time

OMP is not smart enough to avoid multiple threads reading/writing to the same location at the same time ("race condition"). You have to help it!

Explicit precautions must be taken if multiple threads are writing to one location in memory or a value is being calculated across multiple threads.

A better way:

```
// dot product (method 2)
double dot = 0;
#pragma omp parallel for reduction(+:dot)
for (int i=0; i<N; i++) {
    dot += v1[i] * v2[i];
}
```

Explicit precautions must be taken if multiple threads are writing to one location in memory or a value is being calculated across multiple threads.

A better way:

```
// dot product (method 2)
double dot = 0;
#pragma omp parallel for reduction(+:dot)
for (int i=0; i<N; i++) {
    dot += v1[i] * v2[i];
}
```

- give each thread a private `dot` variable. At the end, set the global `dot` variable equal the sum of the private `dot`s.

Explicit precautions must be taken if multiple threads are writing to one location in memory or a value is being calculated across multiple threads.

A better way:

```
// dot product (method 2)
double dot = 0;
#pragma omp parallel for reduction(+:dot)
for (int i=0; i<N; i++) {
    dot += v1[i] * v2[i];
}
```

- give each thread a private dot variable. At the end, set the global dot variable equal the sum of the private dots.

All roads lead to Rome, but some get there faster.

OMP calls allow for other specifications as well, including number of threads, private vs. shared variable defaults, scheduling, etc.

```
int nt=4;
double x,y;
#pragma omp parallel for num_threads(nt) default(none) shared(y) private(x) scheduling(static)
```

OMP calls allow for other specifications as well, including number of threads, private vs. shared variable defaults, scheduling, etc.

```
int nt=4;
double x,y;
#pragma omp parallel for num_threads(nt) default(none) shared(y) private(x) scheduling(static)
```

- specify number of threads to spawn
  - default: `OMP_NUM_THREADS` environment variable

OMP calls allow for other specifications as well, including number of threads, private vs. shared variable defaults, scheduling, etc.

```
int nt=4;
double x,y;
#pragma omp parallel for num_threads(nt) default(none) shared(y) private(x) scheduling(static)
```

- specify number of threads to spawn
  - default: OMP_NUM_THREADS environment variable
- specify default private/shared status of variables
  - shared (default)
  - none: need to explicitly specify any variable you use as private or shared

OMP calls allow for other specifications as well, including number of threads, private vs. shared variable defaults, scheduling, etc.

```
int nt=4;
double x,y;
#pragma omp parallel for num_threads(nt) default(none) shared(y) private(x) scheduling(static)
```

- specify number of threads to spawn
  - default: OMP_NUM_THREADS environment variable
- specify default private/shared status of variables
  - shared (default)
  - none: need to explicitly specify any variable you use as private or shared
- declare shared variables

OMP calls allow for other specifications as well, including number of threads, private vs. shared variable defaults, scheduling, etc.

```
int nt=4;
double x,y;
#pragma omp parallel for num_threads(nt) default(none) shared(y) private(x) scheduling(static)
```

- specify number of threads to spawn
  - default: `OMP_NUM_THREADS` environment variable
- specify default private/shared status of variables
  - shared (default)
  - none: need to explicitly specify any variable you use as private or shared
- declare shared variables
- declare private variables

OMP calls allow for other specifications as well, including number of threads, private vs. shared variable defaults, scheduling, etc.

```
int nt=4;
double x,y;
#pragma omp parallel for num_threads(nt) default(none) shared(y) private(x) scheduling(static)
```

- specify number of threads to spawn
  - default: `OMP_NUM_THREADS` environment variable
- specify default private/shared status of variables
  - shared (default)
  - none: need to explicitly specify any variable you use as private or shared
- declare shared variables
- declare private variables
- set thread-spawning routine
  - static: assign set number of iterations to each thread at outset (low overhead: **use if each iterate requires roughly equal work**)
  - dynamic: assign iterates to threads as needed (high overhead: **use if work per iterate can vary widely**)

OMP calls allow for other specifications as well, including number of threads, private vs. shared variable defaults, scheduling, etc.

```
int nt=4;
double x,y;
#pragma omp parallel for num_threads(nt) default(none) shared(y) private(x) scheduling(static)
```

- specify number of threads to spawn
  - default: OMP_NUM_THREADS environment variable
- specify default private/shared status of variables
  - shared (default)
  - none: need to explicitly specify any variable you use as private or shared
- declare shared variables
- declare private variables
- set thread-spawning routine
  - static: assign set number of iterations to each thread at outset (low overhead: **use if each iterate requires roughly equal work**)
  - dynamic: assign iterates to threads as needed (high overhead: **use if work per iterate can vary widely**)

Let's look at OMP in action.

We can also divide up embarassingly parallel for loops using Python using `joblib`, a wrapper for the `multiprocessing` module.

Anatomy of an joblib call:

```
import joblib as jl

# function for sum of ith components
def sum(i):
  global v1
  global v2
  return v1[i] + v2[i]

# generator of functs and args
fgen = (jl.delayed(sum)(i)
                for i in range(len(v1)))

# parallel sum
v3 = jl.Parallel(n_jobs=NUM_THREADS)(fgen)
```

We can also divide up embarassingly parallel for loops using Python using `joblib`, a wrapper for the `multiprocessing` module.

Anatomy of an joblib call:

```
import joblib as jl

# function for sum of ith components
def sum(i):
  global v1
  global v2
  return v1[i] + v2[i]

# generator of functs and args
fgen = (jl.delayed(sum)(i)
                for i in range(len(v1)))

# parallel sum
v3 = jl.Parallel(n_jobs=NUM_THREADS)(fgen)
```

- require the joblib module

We can also divide up embarassingly parallel for loops using Python using `joblib`, a wrapper for the `multiprocessing` module.

Anatomy of an joblib call:

```
import joblib as jl

# function for sum of ith components
def sum(i):
  global v1
  global v2
  return v1[i] + v2[i]

# generator of functs and args
fgen = (jl.delayed(sum)(i)
                for i in range(len(v1)))

# parallel sum
v3 = jl.Parallel(n_jobs=NUM_THREADS)(fgen)
```

- require the joblib module
- function to produce requested value for each item in loop

We can also divide up embarassingly parallel for loops using Python using `joblib`, a wrapper for the `multiprocessing` module.

Anatomy of an joblib call:

```
import joblib as jl

# function for sum of ith components
def sum(i):
  global v1
  global v2
  return v1[i] + v2[i]

# generator of functs and args
fgen = (jl.delayed(sum)(i)
                for i in range(len(v1)))

# parallel sum
v3 = jl.Parallel(n_jobs=NUM_THREADS)(fgen)
```

- require the joblib module
- function to produce requested value for each item in loop
- generator to produce tuple with pointer to function and args for each item in loop

We can also divide up embarassingly parallel for loops using Python using `joblib`, a wrapper for the `multiprocessing` module.

Anatomy of an joblib call:

```
import joblib as jl

# function for sum of ith components
def sum(i):
  global v1
  global v2
  return v1[i] + v2[i]

# generator of functs and args
fgen = (jl.delayed(sum)(i)
                for i in range(len(v1)))

# parallel sum
v3 = jl.Parallel(n_jobs=NUM_THREADS)(fgen)
```

- require the joblib module
- function to produce requested value for each item in loop
- generator to produce tuple with pointer to function and args for each item in loop
- Parallel object takes in args (e.g. n_jobs) and generator, returns array with result of passed function at each spot in loop

For loops—and therefore the spawning of parallel threads—take place at a higher level in Python, so there is a much higher overhead cost.

OMP: compiler $\rightarrow$ machine code

For loops—and therefore the spawning of parallel threads—take place at a higher level in Python, so there is a much higher overhead cost.

OMP: compiler $\rightarrow$ machine code

joblib: interpreter $\rightarrow$ system processes $\rightarrow$ precompiled libraries (machine code)

For loops—and therefore the spawning of parallel threads—take place at a higher level in Python, so there is a much higher overhead cost.

OMP: compiler $\to$ machine code

joblib: interpreter $\to$ system processes $\to$ precompiled libraries (machine code)

Discuss: what properties make a job more (or less) appropriate for joblib?

For loops—and therefore the spawning of parallel threads—take place at a higher level in Python, so there is a much higher overhead cost.

OMP: compiler $\rightarrow$ machine code

joblib: interpreter $\rightarrow$ system processes $\rightarrow$ precompiled libraries (machine code)

Discuss: what properties make a job more (or less) appropriate for joblib?

Often, modules such as numpy already have precompiled code which is parallelized
$\implies$ you don't have to do it yourself! Write **vectorized code**.

Vectorization refers to code with "invisible" for loops.

For two numpy arrays `a` and `b`, we can do the same calculation with an explicit for loop or vectorized numpy functions that call precompiled C++-code.

Vectorization refers to code with "invisible" for loops.

For two numpy arrays `a` and `b`, we can do the same calculation with an explicit for loop or vectorized numpy functions that call precompiled C++-code.

- sum

```
for i in range(len(a)):
  c[i] = a[i] + b[i]
```

```
c = a + b
```

Vectorization refers to code with "invisible" for loops.

For two numpy arrays `a` and `b`, we can do the same calculation with an explicit for loop or vectorized numpy functions that call precompiled C++-code.

- sum

```
for i in range(len(a)):
    c[i] = a[i] + b[i]
```

```
c = a + b
```

- dot product

```
d = 0
for i in range(len(a)):
    d = d + a[i] * b[i]
```

```
d = a.dot(b)
```

Vectorization refers to code with "invisible" for loops.

For two numpy arrays `a` and `b`, we can do the same calculation with an explicit for loop or vectorized numpy functions that call precompiled C++-code.

- sum

```
for i in range(len(a)):
    c[i] = a[i] + b[i]
```

```
c = a + b
```

- dot product

```
d = 0
for i in range(len(a)):
    d = d + a[i] * b[i]
```

```
d = a.dot(b)
```

- elementwise multiply

```
for i in range(len(a)):
    c[i] = a[i] * b[i]
```

```
c = a * b
```

Vectorized code is faster first and foremost because of precompiled library functions, but these libraries are often parallelized with OpenMP as well.

$\boxed{\texttt{M.dot(v)}}$

(numpy code)

Vectorized code is faster first and foremost because of precompiled library functions, but these libraries are often parallelized with OpenMP as well.

$\boxed{\texttt{M.dot(v)}}$ $\longrightarrow$

(numpy code) (interpreter call)

Vectorized code is faster first and foremost because of precompiled library functions, but these libraries are often parallelized with OpenMP as well.

`M.dot(v)` ⟶

```
#pragma omp parallel for
for (int r=0;r<N;r++) {
    for (int c=0;c<N;c++) {
        out[r] += M[r][c] * v[c];
    }
}
```

(numpy code) (interpreter call) (machine code equivalent to above, compiled)

Vectorized code is faster first and foremost because of precompiled library functions, but these libraries are often parallelized with OpenMP as well.

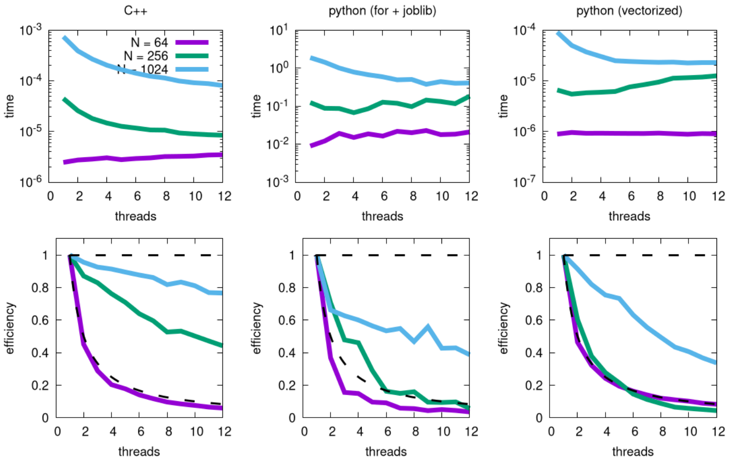
`M.dot(v)` ⟶

```
#pragma omp parallel for
for (int r=0;r<N;r++) {
    for (int c=0;c<N;c++) {
        out[r] += M[r][c] * v[c];
    }
}
```

(numpy code) (interpreter call) (machine code equivalent to above, compiled)

Easiest way to set thread number for vectorized `numpy`? `OMP_NUM_THREADS` environment variable!

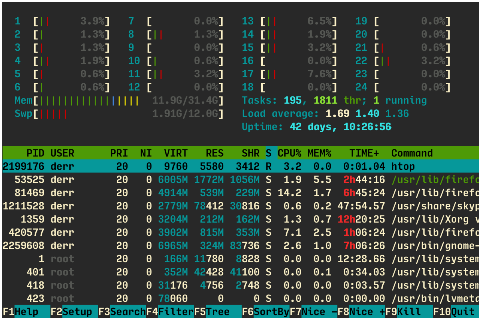
We can compare the parallel results across C++ (OMP), Python (joblib), and Python (vectorized) for a matrix multiplication



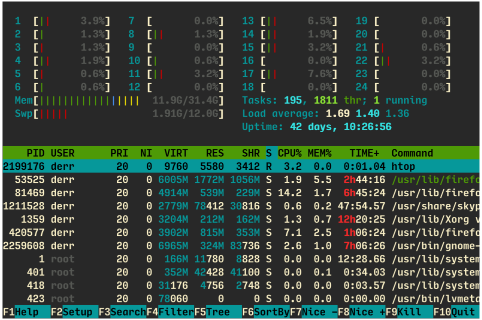
“Hyperthreading” refers to a CPU managing two sets of tasks per core.

I have 12 cores, `htop` reports 24: use them all?

answer: maybe



"Hyperthreading" refers to a CPU managing two sets of tasks per core.

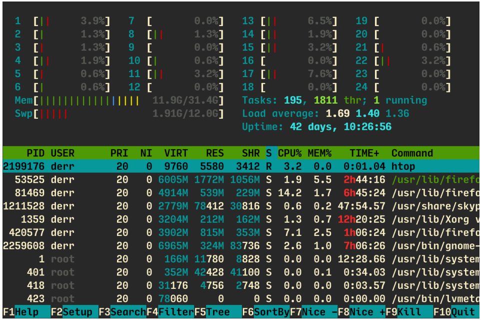


I have 12 cores, `htop` reports 24: use them all?

answer: maybe

if loop iteration is constant block of computation (frequently the case in PDE solvers), using k threads with $k >$ the number of physical cores will be highly inefficient

"Hyperthreading" refers to a CPU managing two sets of tasks per core.



I have 12 cores, `htop` reports 24: use them all?

answer: maybe

if loop iteration is constant block of computation (frequently the case in PDE solvers), using k threads with $k >$ the number of physical cores will be highly inefficient

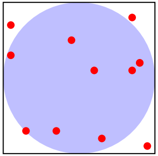
if loop iterations have dead time (can happen with very short iteration tasks), hyperthreading may allow for faster simulation time (with lower efficiency than adding more physical cores)

Summary

- Shared vs. distributed memory: use the paradigm appropriate for your problem
- Compiled languages
  - want a speed up? find main for loop, drop in `#pragma omp parallel for`
  - threads will do what you tell them to do: **be careful!** Use OMP options
  - compiler flexibility (multithreaded code works on non-OMP compilers)
- Interpreted languages
  - parallelization often include by default in calls to precompiled library functions
  - eschew for loops: **vectorize, vectorize, vectorize**
  - help your CPU out in determining number of threads vs. cores

Group activity: calculating π stochastically

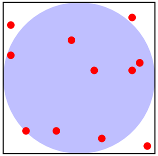
Consider a square of area 4 inscribed with a circle of radius 1:



If the red dots are randomly placed in the square, what fraction will land in the circle?

Group activity: calculating π stochastically

Consider a square of area 4 inscribed with a circle of radius 1:



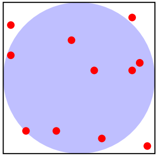
If the red dots are randomly placed in the square, what fraction will land in the circle?

$$f = \frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi}{4} \implies \boxed{\pi = 4f.}$$

Idea: generate random points, calculate approximation to π based on fraction in circle.

Group activity: calculating π stochastically

Consider a square of area 4 inscribed with a circle of radius 1:



If the red dots are randomly placed in the square, what fraction will land in the circle?

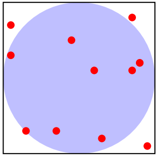
$$f = \frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi}{4} \implies \boxed{\pi = 4f.}$$

Idea: generate random points, calculate approximation to π based on fraction in circle.

...but how to generate an estimate for the accuracy of our approximation?

Group activity: calculating π stochastically

Consider a square of area 4 inscribed with a circle of radius 1:



If the red dots are randomly placed in the square, what fraction will land in the circle?

$$f = \frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi}{4} \implies \boxed{\pi = 4f.}$$

Idea: generate random points, calculate approximation to π based on fraction in circle.

...but how to generate an estimate for the accuracy of our approximation?

$\rightarrow$ generate N values p_i, $0 <= i < N$, from N batches of k points:

$$\boxed{\text{approx value } \bar{\pi} = \text{mean}(p_i), \qquad \text{standard error } \sigma_\pi = \frac{\text{std}(p_i)}{\sqrt{N}}}$$

Problem specification

- Using C++ and OMP, Python and joblib, or vectorized numpy, write a program that takes in integers N, k and t, with
  - N: number of π approximations
  - k: number of points generated for each π approximation
  - t: number of threads of parallelization

  and returns the compute time T and standard error σ_π.

Problem specification

- Using C++ and OMP, Python and joblib, or vectorized numpy, write a program that takes in integers N, k and t, with
  - N: number of π approximations
  - k: number of points generated for each π approximation
  - t: number of threads of parallelization

  and returns the compute time T and standard error σ_π.
- Generate T and σ_π for the values $N \in \{100, 1000, 10000\}$, $k \in \{100, 1000, 10000\}$, and $t \in \{1, 2, 3, 4\}$.

Problem specification

- Using C++ and OMP, Python and joblib, or vectorized numpy, write a program that takes in integers N, k and t, with
  - N: number of π approximations
  - k: number of points generated for each π approximation
  - t: number of threads of parallelization

  and returns the compute time T and standard error σ_π.
- Generate T and σ_π for the values $N \in \{100, 1000, 10000\}$, $k \in \{100, 1000, 10000\}$, and $t \in \{1, 2, 3, 4\}$.
- Answer the following questions:

Problem specification

- Using C++ and OMP, Python and joblib, or vectorized numpy, write a program that takes in integers N, k and t, with
  - N: number of π approximations
  - k: number of points generated for each π approximation
  - t: number of threads of parallelization

  and returns the compute time T and standard error σ_π.
- Generate T and σ_π for the values $N \in \{100, 1000, 10000\}$, $k \in \{100, 1000, 10000\}$, and $t \in \{1, 2, 3, 4\}$.
- Answer the following questions:
  1. What determines the scaling of σ_π? (what axes give you a log-log plot with a straight line?)

Problem specification

- Using C++ and OMP, Python and joblib, or vectorized numpy, write a program that takes in integers N, k and t, with
  - N: number of π approximations
  - k: number of points generated for each π approximation
  - t: number of threads of parallelization

  and returns the compute time T and standard error σ_π.
- Generate T and σ_π for the values $N \in \{100, 1000, 10000\}$, $k \in \{100, 1000, 10000\}$, and $t \in \{1, 2, 3, 4\}$.
- Answer the following questions:
  1. What determines the scaling of σ_π? (what axes give you a log-log plot with a straight line?)
  2. For constant $n_{\text{pts}} = Nk = 1000^2$, how does T scale with k? With N? (make some plots and explain what they mean)

Problem specification

- Using C++ and OMP, Python and joblib, or vectorized numpy, write a program that takes in integers N, k and t, with
  - N: number of π approximations
  - k: number of points generated for each π approximation
  - t: number of threads of parallelization

  and returns the compute time T and standard error σ_π.
- Generate T and σ_π for the values $N \in \{100, 1000, 10000\}$, $k \in \{100, 1000, 10000\}$, and $t \in \{1, 2, 3, 4\}$.
- Answer the following questions:
  1. What determines the scaling of σ_π? (what axes give you a log-log plot with a straight line?)
  2. For constant $n_{\text{pts}} = Nk = 1000^2$, how does T scale with k? With N? (make some plots and explain what they mean)
  3. Given the above, for what types of values of N and k is multithreading most appropriate? Do your results reflect this?