A Hands-on Introduction to Parallel and High Performance Computing

Jean Bartik Computing Symposium
2024

Nick Park
U.S. Department of Defense

Why parallel computing?

“The information technology sector […], manufacturing, financial and other services, science, engineering, education, defense and other government services, and entertainment—have grown dependent on continued growth in computing performance.”

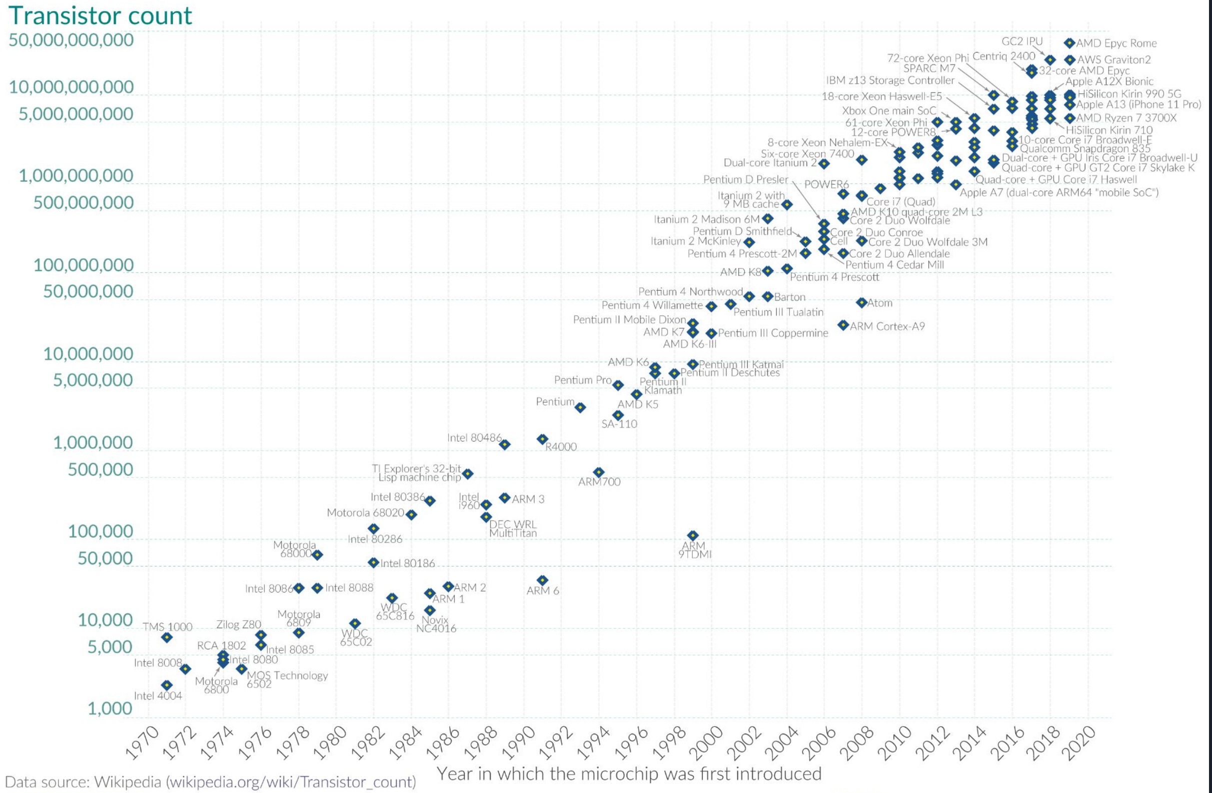
~ National Research Council

National Research Council. 2011. The Future of Computing Performance: Game Over or Next Level?. Washington, DC: The National Academies Press. https://doi.org/10.17226/12980.

Moore's Law: The number of transistors on microchips doubles every two years

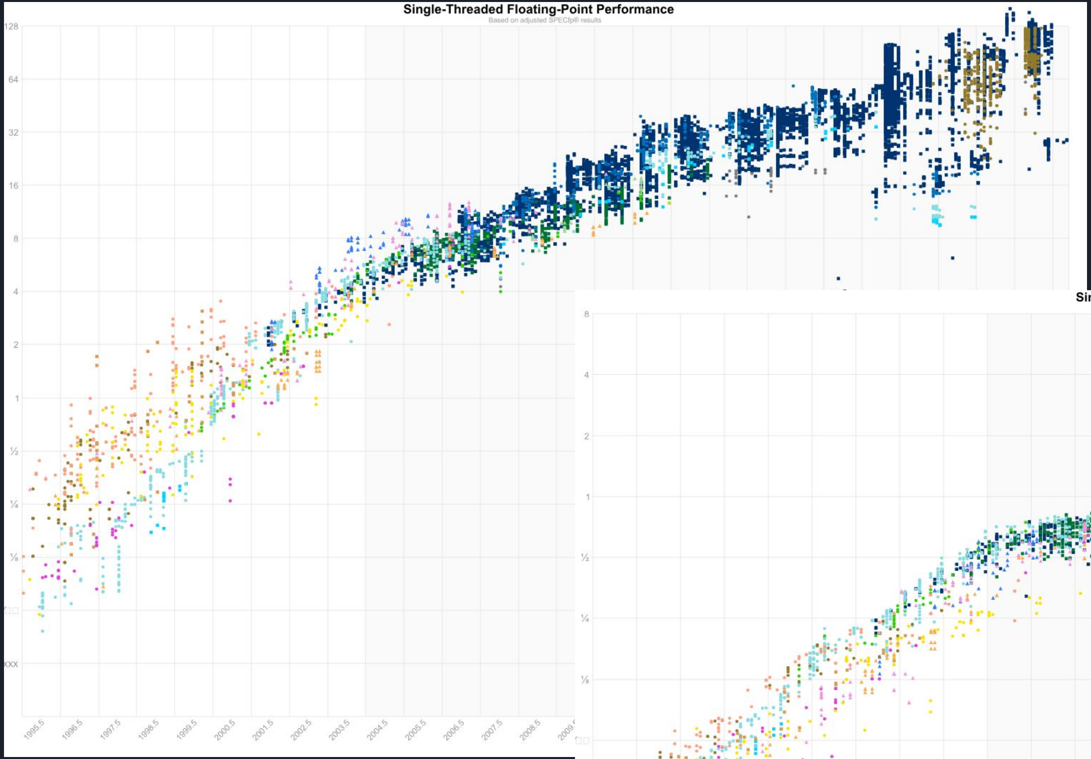
Our World in Data

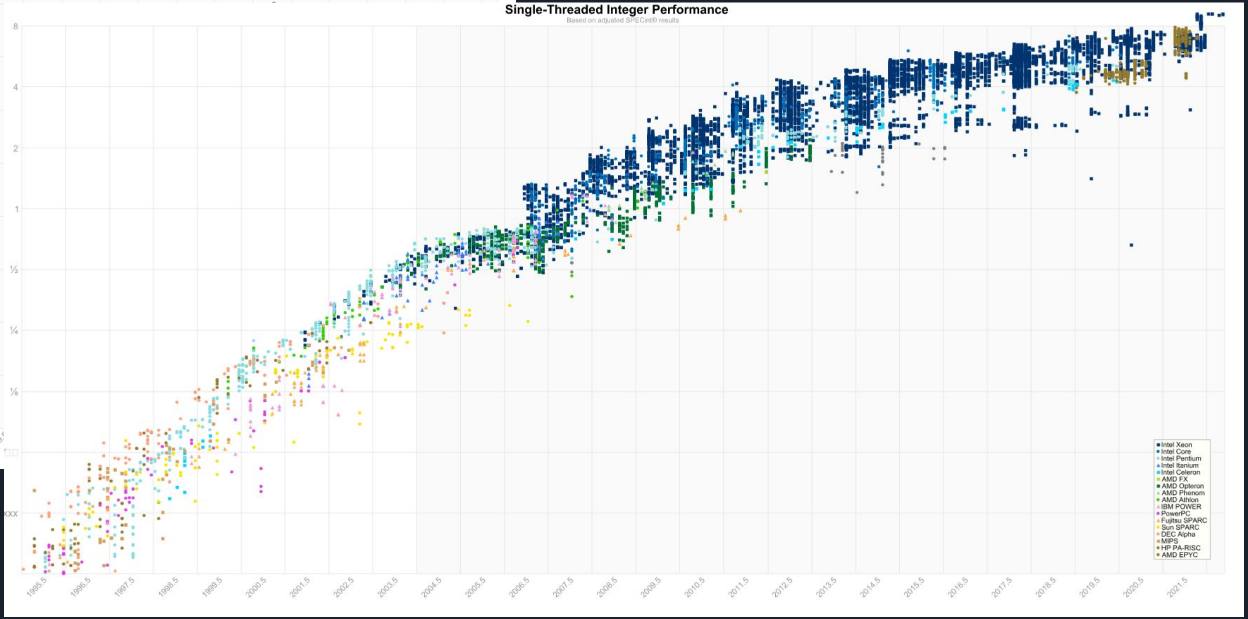
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



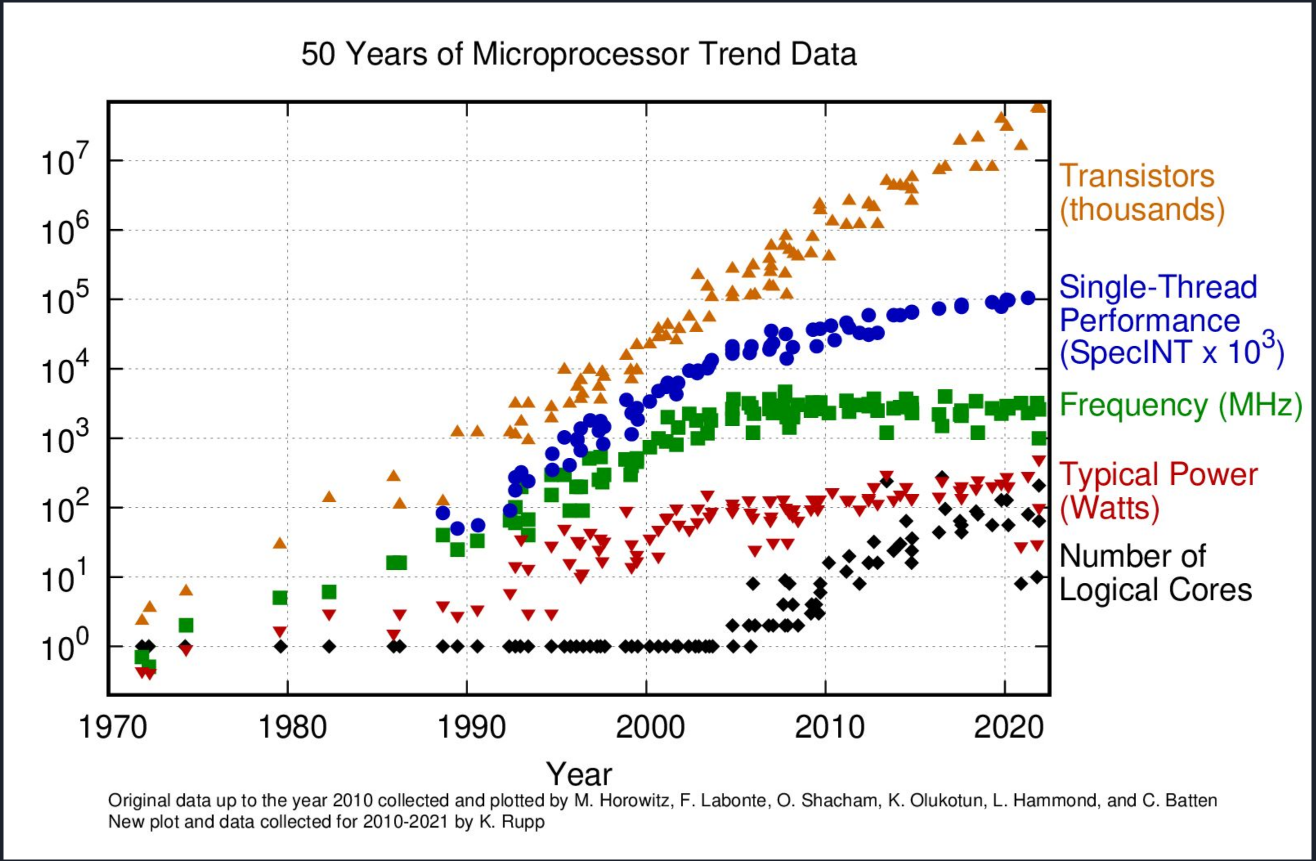
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.





"analyze-spec-benchmarks" by HenkPoley, based on work from Jeff Preshing, released under Public Domain


50 Years of Microprocessor Trend Data
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
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 2010-2021 by K. Rupp

“There is no known alternative to parallel systems for sustaining growth in computing performance.”

~ National Research Council

National Research Council. 2011. The Future of Computing Performance: Game Over or Next Level?. Washington, DC: The National Academies Press. https://doi.org/10.17226/12980.

Parallel Computing in Practice

What is parallel computing?

Parallel computing is a process in which many calculations are performed at the same time.

Parallelism is a collection of properties that allow for (parts of) a computation to be performed at the same time.

A question of scale

“If you were plowing a field, which would you rather use: two strong oxen or 1,024 chickens?”

Seymour Cray
Supercomputing pioneer and founder of Cray Research

Scale up vs. scale out

Scale up

- Increasing the compute/memory/storage capability of the compute node
- For example
  - 12 → 64 cores/socket
  - 1 → 4 sockets/node
  - 32 → 128 GiB RAM

Scale out

- Increasing the number of nodes in a cluster

Flavors of parallelism

● Instruction-level parallelism (ILP)	→	microarchitecture
● Thread-level parallelism (TLP)	→	threading
● Data parallelism (SIMD)	→	SIMD/vector programming
● Process parallelism	→	multiprocessing
○ SPMD	→	HPC (+ the above)
○ MIMD/MPMD	→	distributed/cloud computing

What is HPC?

High performance computing (HPC) is...

...an approach to building and using computer systems...

...that optimizes for maximum performance...

...over one or more application domains.

How does one compare...?

Cray-1 (1975)

LANL CM-5 (1993)

"Cray-1 at Science Museum, London" by 'Irid Escent' is licensed under CC BY-SA 2.0

"CM-5" by LANL used under license to the U.S. Government

How does one compare…?

Cray-1 (1975)

- 1 core @ 80 MHz
- 8 MB RAM
- 160 MFLOPS = 160×10^6 FLOPS
- 115 kW
- ~$8M

Ref: Computer History Museum, Wikipedia

LANL CM-5 (1993)

- 1,024 nodes
- 1,024 cores @ 32 MHz
- 32 GB RAM
- 59.7 GFLOPS = 59.7×10^9 FLOPS
- ~150 kW
- ~$47M

Ref: TOP500; "Littlebear," NCAR; "Supercomputers," Chris Vernon

How does one compare...?

Cray-1 (1975)

ORNL Frontier (2022)

How does one compare...?

Cray-1 (1975)

- 1 core @ 80 MHz
- 8 MB RAM
- 160 MFLOPS = 160×10^6 FLOPS
- 115 kW
- ~$8M

Ref: Computer History Museum, Wikipedia

ORNL Frontier (2022)

- 591,872 cores @ 2 GHz
- 36,992 GPUs
- 4.6 PB RAM
- 4.6 PB HBM
- 1.19 EFLOPS = 1.19×10^{18} FLOPS
- 22.7 MW
- ~$600M

Ref: TOP500, OLCF

How does one compare…?

LANL CM-5 (1993)

ORNL Frontier (2022)

How does one compare...?

LANL CM-5 (1993)

- 1,024 nodes
- 1,024 cores @ 32 MHz
- 32 GB RAM
- 59.7 GFLOPS = 59.7 × 10^9 FLOPS
- ~150 kW
- ~$47M

Ref: TOP500; “Littlebear,” NCAR; “Supercomputers,” Chris Vernon

ORNL Frontier (2022)

- 9,248 nodes
- 591,872 cores @ 2 GHz
- 36,992 GPUs
- 4.6 PB RAM
- 4.6 PB HBM
- 1.19 EFLOPS = 1.19 × 10^{18} FLOPS
- 22.7 MW
- ~$600M

Ref: TOP500, OLCF

HPL and the TOP500

Historically, the scientific and high performance computing communities have measured peak performance (Rmax) of an HPC with the High Performance LINPACK (HPL) benchmark, which solves a dense *N*-by-*N* system of linear equations, $Ax = b$.

HPL measures performance in FLOPS: floating point operations per second

The TOP500 list tracks, biannually, the top 500 HPC systems ranked by performance as measured with HPL.

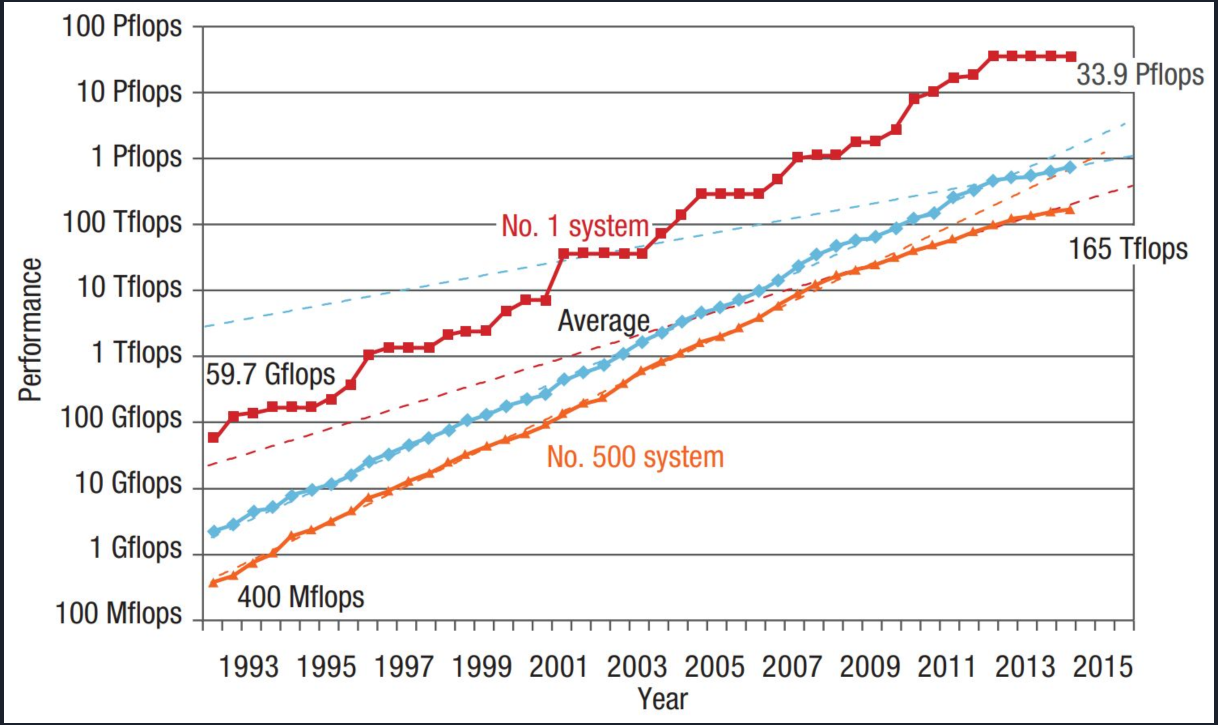
HPL and the TOP500

"The Linpack benchmark is one of those interesting phenomena—almost anyone who knows about it will deride its utility. They understand its limitations but it has mindshare because it's the one number we've all bought into over the years."

Thom Dunning, Jr.
Then-Director, National Center for Supercomputing Applications

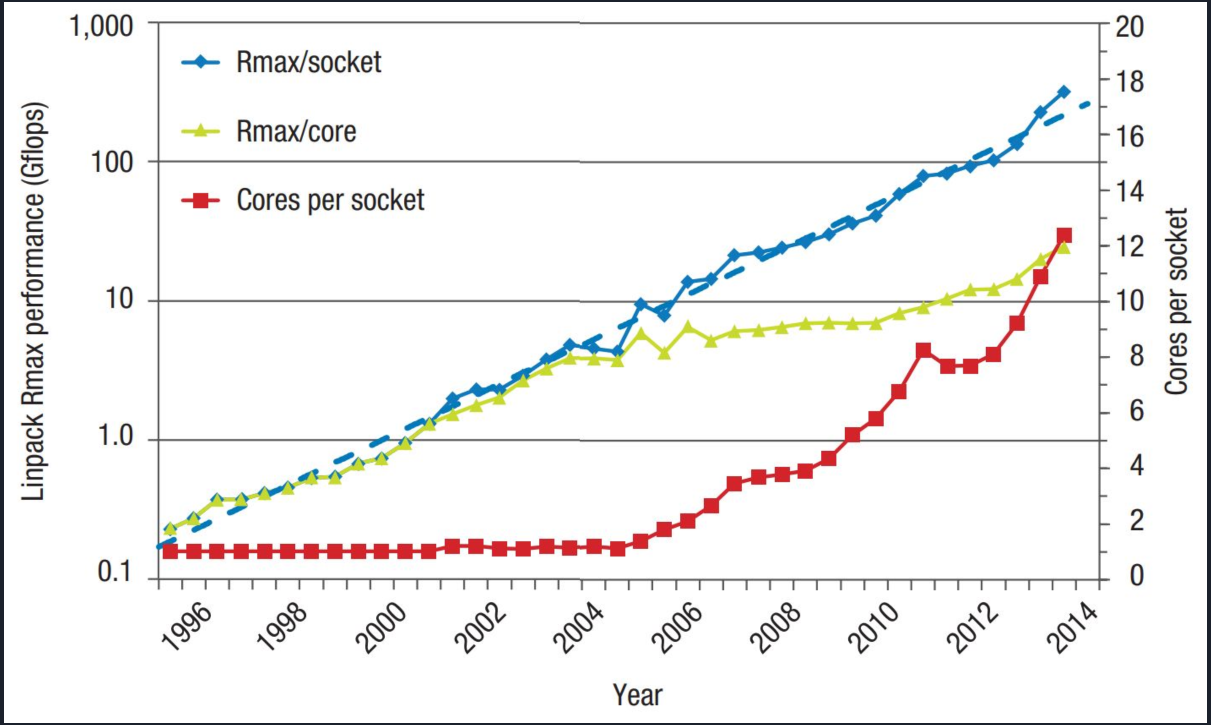
Jack Dongarra on TOP500: Past, Present, and Future

HPC performance over time by TOP500 (through 2015)



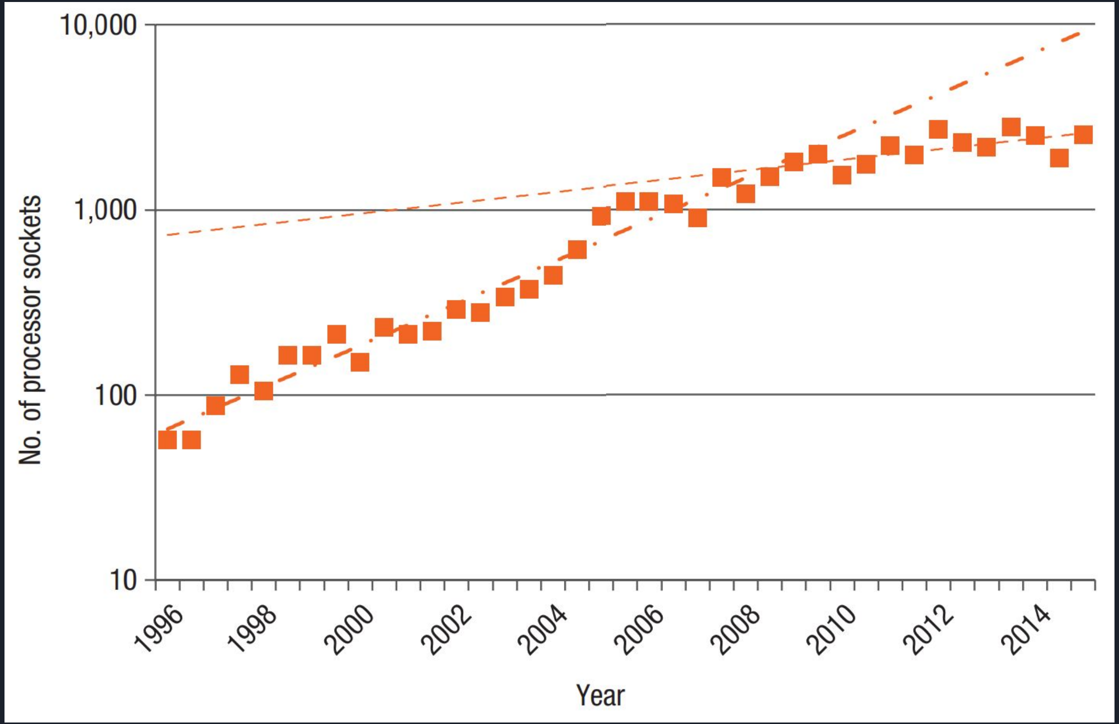
E. Strohmaier, H. Meuer, J. Dongarra and H. Simon, "The TOP500 List and Progress in High-Performance Computing" in Computer, vol. 48, no. 11, pp. 42-49, 2015.

Normalized peak performance of TOP500 systems



E. Strohmaier, H. Meuer, J. Dongarra and H. Simon, "The TOP500 List and Progress in High-Performance Computing" in Computer, vol. 48, no. 11, pp. 42-49, 2015.

Average # of processor sockets for new HPCs in the TOP500



E. Strohmaier, H. Meuer, J. Dongarra and H. Simon, "The TOP500 List and Progress in High-Performance Computing" in Computer, vol. 48, no. 11, pp. 42-49, 2015.

Scale up *then* out

Specialization matters

HPCs are larger, more expensive, more power hungry, and more complex than ever before

Optimization and Parallelization in Python

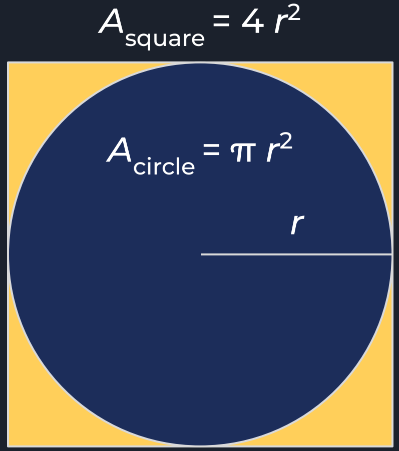
Approximating π

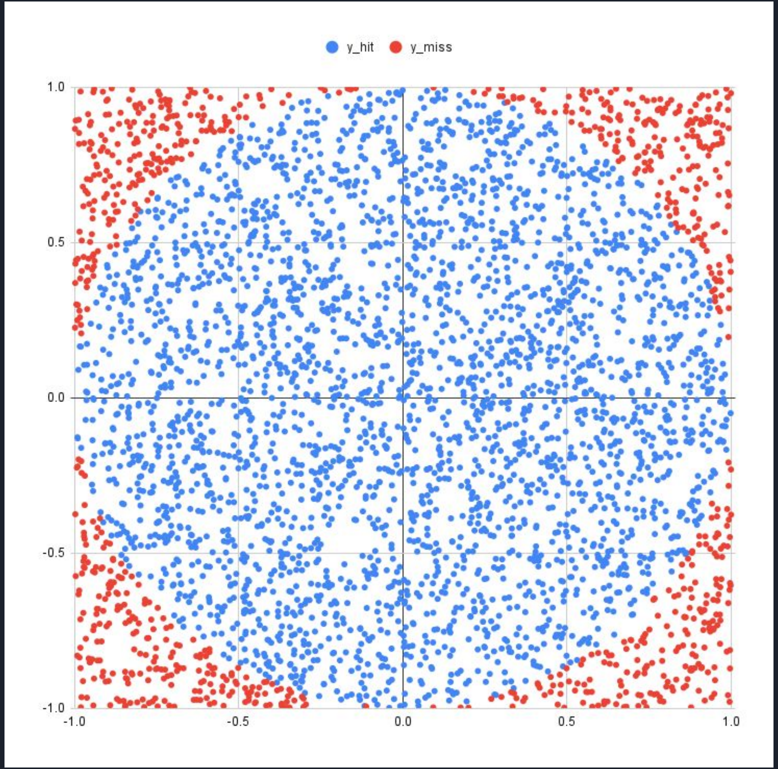
We will approximate π by Monte Carlo methods.

That is:

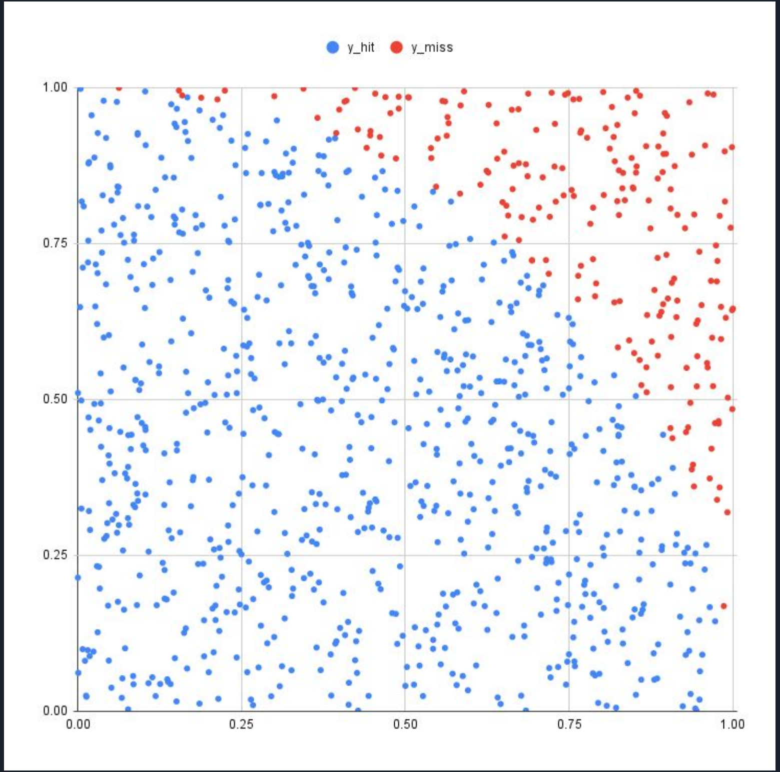
- We will randomly sample points inside a square with side length 2*r*.
- We will count how many of these samples fall in a circle with radius *r*, the "hits."

Then, π ≈ 4 · hits / total





● y_hit
● y_miss
1.0
0.5
0.0
-0.5
-1.0
-1.0
-0.5
0.0
0.5
1.0



● y_hit
● y_miss
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00

Approximating π with Python

Get the example code for the workshop

```
% git clone https://github.com/nspark/jbcs.git
% cd jbcs/python
```

Set up your Python environment

```
% python3 -m venv env
% source env/bin/activate
% pip install numpy numba
```

Approximating π with Python

Let's look at an initial, serial implementation and its performance

```
% python pi.py
pi ≈ 3.141306800
Elapsed: 11.556 seconds [Mode.SERIAL]
```

```python
def pi_serial(N):
    hits = 0
    for _ in range(N):
        x = random()
        y = random()
        if x*x + y*y < 1.0:
            hits += 1
    return 4.0 * hits / N
```

Approximating π with Python

What does `@jit` do here?

Numba is *compiling* the `pi_jit` function *at execution time*.

Hence, “just-in-time” (JIT) compilation.

```python
from numba import jit

@jit(nopython=True)
def pi_jit(N):
    hits = 0
    for _ in range(N):
        x = random()
        y = random()
        if x*x + y*y < 1.0:
            hits += 1
    return 4.0 * hits / N
```

Numba: Compiling Python code with `@jit`

Approximating π with Python

How does this JIT-compiled implementation perform?

```
% python pi.py -m JIT
pi ≈ 3.141609160
Elapsed: 0.720 seconds [Mode.JIT]
```

How long did the JIT compilation itself take?

```
% python pi.py -m JIT -v
Elapsed: 0.459 seconds [JIT: pi_jit]
pi ≈ 3.14172148
Elapsed: 0.723 seconds [Mode.JIT]
```

Approximating π with Python

But can @jit auto-parallelize?

It can!

Note the use of prange (vs. range), which specifies an *explicitly parallel loop* when used with @jit(parallel=True).

Note that hits is an *accumulated* value.

The summation into hits from the automatically parallelized contexts is automatically reduced by @jit.

Numba: Automatic parallelization with @jit

```
from numba import jit, prange

@jit(nopython=True, parallel=True)
def pi_parallel(N):
    hits = 0
    for _ in prange(N):
        x = random()
        y = random()
        if x*x + y*y < 1.0:
            hits += 1
    return 4.0 * hits / N
```

Approximating π with Python

But can `@jit` auto-parallelize?

It can!

Note the use of `prange` (vs. `range`), which specifies an *explicitly parallel loop* when used with `@jit(parallel=True)`.

Note that `hits` is an *accumulated* value.

The summation into `hits` from the automatically parallelized contexts is automatically reduced by `@jit`.

```
% python pi.py -m PARALLEL -v
Elapsed: 0.865 seconds
[JIT: pi_parallel]
pi ≈ 3.14147568
Elapsed: 0.140 seconds
[Mode.PARALLEL]
```

Numba: Automatic parallelization with `@jit`

Approximating π with Python

Now, for some explicit parallelization using executors:

```python
def pi_pool(N, Executor, inner_fn):
    n_workers = cpu_count()
    with Executor(max_workers=n_workers) as pool:
        hits = sum(
            pool.map(
                inner_fn,
                zip(repeat(N), repeat(n_workers), range(n_workers))))
    return 4.0 * hits / N
```

`concurrent.futures` — Launching parallel tasks — Python Documentation

Approximating π with Python

```python
def pi_pool_inner(args):
    (N, W, i) = args
    lo = (N *  i      ) // W
    hi = (N * (i + 1)) // W
    hits = 0
    for _ in range(lo, hi):
        x = random()
        y = random()
        if x*x + y*y < 1.0:
            hits += 1
    return hits
```

```python
from concurrent.futures import \
  ThreadPoolExecutor

pi = pi_pool(N,
             ThreadPoolExecutor,
             pi_pool_inner)
```

`concurrent.futures` — Launching parallel tasks — Python Documentation

Approximating π with Python

How does it perform?

```
% python pi.py -m THREADPOOL
pi ≈ 3.141644160
Elapsed: 46.935 seconds [Mode.THREADPOOL]
```

Recall:

```
% python pi.py
pi ≈ 3.141306800
Elapsed: 11.556 seconds [Mode.SERIAL]
```

What happened?

Approximating π with Python

CPython implementation detail: In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use `multiprocessing` or `concurrent.futures.ProcessPoolExecutor`. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

`threading` — Thread-based parallelism — Python Documentation

Approximating π with Python

```
from concurrent.futures import \
  ThreadPoolExecutor

pi = pi_pool(N,
             ThreadPoolExecutor,
             pi_pool_inner)

% python pi.py -m THREADPOOL
pi ≈ 3.141644160
Elapsed: 46.935 seconds
[Mode.THREADPOOL]
```

```
from concurrent.futures import \
  ProcessPoolExecutor

pi = pi_pool(N,
             ProcessPoolExecutor,
             pi_pool_inner)

% python pi.py -m PROCESSPOOL
pi ≈ 3.141371160
Elapsed: 2.571 seconds
[Mode.PROCESSPOOL]
```

Approximating π with Python

Since JIT-optimized code was so much faster than native Python code, let's try using `@jit` on the callable invoked by the executor.

```
@jit(nopython=True)
def pi_pool_inner_jit(args):
    ...
```

```
% python pi.py -m THREADPOOLJIT
pi ≈ 3.141826960
Elapsed: 1.131 seconds
[Mode.THREADPOOLJIT]
```

```
% python pi.py -m PROCESSPOOLJIT
<no result>
Elapsed: 0.083 seconds
[Mode.PROCESSPOOLJIT]
```

Approximating π with Python

The JIT-optimized `pi_pool_inner_jit` is not directly callable by the `ProcessPoolExecutor`.

However, a “trampoline”-like function can serve as a workaround.

```
def pi_pool_inner_tramp(args):
    return pi_pool_inner_jit(args)

pi = pi_pool(N,
             ProcessPoolExecutor,
             pi_pool_inner_tramp)
```

```
% python pi.py -m PROCESSPOOLJIT
pi ≈ 3.141560720
Elapsed: 0.219 seconds
[Mode.PROCESSPOOLJIT]
```

Approximating π with Python

```
% python pi.py -m ALL
Elapsed: 11.556 seconds [Mode.SERIAL]
Elapsed:  0.720 seconds [Mode.JIT]             → ~16X improvement (over serial)
Elapsed:  0.140 seconds [Mode.PARALLEL]        → ~5X improvement (over JIT)
                                               → ~82X improvement (over serial)

Elapsed: 46.935 seconds [Mode.THREADPOOL]      → ~4X slowdown ☹️
Elapsed:  2.571 seconds [Mode.PROCESSPOOL]     → ~4.5X improvement (over serial)

Elapsed:  1.131 seconds [Mode.THREADPOOLJIT]   → ~10X improvement (over serial)
Elapsed:  0.219 seconds [Mode.PROCESSPOOLJIT]  → ~12X improvement (with JIT)
                                               → ~53X improvement (over serial)
```

Approximating π with Python

```
% python pi.py -m ALL
Elapsed: 11.556 seconds [Mode.SERIAL]
Elapsed:  0.720 seconds [Mode.JIT]
Elapsed:  0.140 seconds [Mode.PARALLEL]


Elapsed: 46.935 seconds [Mode.THREADPOOL]
Elapsed:  2.571 seconds [Mode.PROCESSPOOL]

Elapsed:  1.131 seconds [Mode.THREADPOOLJIT]
Elapsed:  0.219 seconds [Mode.PROCESSPOOLJIT]
```

In this example, auto-parallelization performed best.

This will not always be the case.

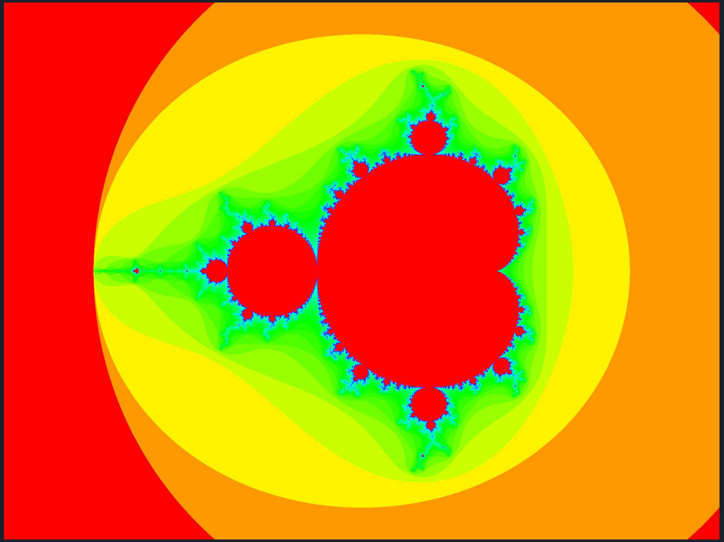
BUT, the biggest *single* improvement came from JIT compilation and optimization.

Hands-on: the Mandelbrot set

Goal
Apply Python optimization and parallelization techniques to compute a visualization of the Mandelbrot set.

You will start with:

- a serial implementation
- timing and I/O boilerplate code



On parallel scaling and efficiency

Parallel scaling

<u>Amdahl's Law</u>

$$S = \frac{1}{(1-p) + \frac{p}{N}}$$

assumes fixed problem size
("strong scaling")

<u>Gustafson's Law</u>

$$S = 1 + (N-1) \times p$$

assumes problem size scales with *N*
("weak scaling")

where:

- *p* is the fraction of the program to be parallelized
- *N* is the number of processors

Parallel efficiency

When using *N* parallel resources, it is good to measure how efficiently these resources are used.

$$E(N) = \frac{S(N)}{N} = \frac{t_{\text{serial}}}{t_{\text{parallel}} \times N}$$

Running the π approximation on a 6-core CPU[1]:

```
0.720 seconds [JIT]
0.140 seconds [PARALLEL]
0.219 seconds [PROCESSPOOLJIT]
```

Yields:

- ~73% efficiency `[PARALLEL]`
- ~58% efficiency `[PROCESSPOOLJIT]`

1. Running on a 6-core Intel Xeon CPU with 2 HyperThreads per core.

Parallelism in C with OpenMP

OpenMP

OpenMP is a `pragma`-based language extension parallelism in C, C++, and Fortran. Here, we will use C.

A `pragma` directive tells the C compiler to behave in an “implementation-defined” manner.

```
#pragma omp parallel
{ … }
```

OpenMP is best known for its loop-based parallelism.

```
#pragma omp parallel for
for (int i; …; i++) { … }
```

π — now in C!

```
% cd jbcs/openmp
% mkdir build && cd build

% cmake .. && make
  –or–
% gcc -fopenmp -O3 -o pi ../pi.c

% ./pi
Elapsed: 1.076 seconds [Serial]
pi ≈ 3.141678240
```

```c
double pi_serial(const size_t N) {
  size_t hits = 0;

  for (size_t i = 0; i < N; i++) {
    double x = drand48();
    double y = drand48();
    if (x * x + y * y < 1.0)
      hits++;
  }
  return 4.0 * hits / N;
}
```

π — now in C with OpenMP!

The parallel loop construct in OpenMP will create a parallel pool of threads and the distribute iteration space over the threads.

```
% ./pi
Elapsed: 1.076 seconds [Serial]
pi ≈ 3.141678240
Elapsed: 5.129 seconds [OpenMP #1a]
pi ≈ 0.651781120
```

```c
double pi_omp1a(const size_t N) {
  size_t hits = 0;
#pragma omp parallel for
  for (size_t i = 0; i < N; i++) {
    double x = drand48();
    double y = drand48();
    if (x * x + y * y < 1.0)
      hits++;
  }
  return 4.0 * hits / N;
}
```

OpenMP Specifications, OpenMP 4.5 API C/C++ Syntax Reference Guide

π — now in C with OpenMP!

This code has *three* problems.

1. The `drand48` function is not required to be thread safe.
2. The increment of `hits` is a *data race* (or *race condition*) when updated in parallel.
3. It's slower.

```c
double pi_omp1a(const size_t N) {
  size_t hits = 0;
#pragma omp parallel for
  for (size_t i = 0; i < N; i++) {
    double x = drand48();
    double y = drand48();
    if (x * x + y * y < 1.0)
      hits++;
  }
  return 4.0 * hits / N;
}
```

pthreads and drand48 concurrency performance — Stack Overflow
drand48 — POSIX.1-2008

π — now in C with OpenMP!

This code has *three* problems.

1. The `drand48` function is not required to be thread safe.
2. The increment of `hits` is a *data race* (or *race condition*) when updated in parallel.
3. It's slower.

While 1 and 2 mean the same thing.

1. The `drand48` function is provided by the C standard library.
2. The increment of `hits` is implemented by the application.

[pthreads and drand48 concurrency performance — Stack Overflow]

Data races

A race condition occurs when—in the absence of any synchronizing operations—two or more threads concurrently access the same location in memory, and at least one access is a write.

For example:

```
  int count = 0;
#pragma omp parallel
  count++;
  printf(“count = %d\n”, count);
```

Yields:

```
% ./count
count = 5
% ./count
count = 2
% ./count
count = 7
```

Resolving data races: atomics

The OpenMP `atomic` directive is *one* synchronization mechanism.

```
% ./pi
Elapsed: 1.076 seconds [Serial]
pi ≈ 3.141678240
Elapsed: 5.129 seconds [OpenMP #1a]
pi ≈ 0.651781120
Elapsed: 6.054 seconds [OpenMP #1b]
pi ≈ 3.149456840
```

So… correct 👍 but slow(er) 👎

```
double pi_omp1b(const size_t N) {
  size_t hits = 0;
#pragma omp parallel for
  for (size_t i = 0; i < N; i++) {
    double x = drand48();
    double y = drand48();
    if (x * x + y * y < 1.0)
#pragma omp atomic
      hits++;
  }
  return 4.0 * hits / N;
}
```

Resolving data races: atomics

The OpenMP `atomic` directive is *one* synchronization mechanism.

```
% ./pi
Elapsed: 1.076 seconds [Serial]
pi ≈ 3.141678240
Elapsed: 5.129 seconds [OpenMP #1a]
pi ≈ 0.651781120
Elapsed: 6.054 seconds [OpenMP #1b]
pi ≈ 3.149456840
```

So… correct 👍 but slow(er) 👎

The slowdown occurs because atomic memory operations are not “free” with respect to performance.

Used properly, they can eliminate a data race, but they have a small performance penalty.

Since `hits` is updated frequently, we incur that small penalty many times.

What if `hits` could be updated less frequently?

Resolving data races: reductions

The `reduction` clause on the `parallel for` directive will collectively combine the changes to `hits` across the parallel threads when the directive closes.

This results in *many* fewer memory accesses to the original `hits`.

Reductions are not a universal solution to data races. However, they are well-suited to this case.

```
double pi_omp1c(const size_t N) {
  size_t hits = 0;
#pragma omp parallel for \
  reduction(+:hits)
  for (size_t i = 0; i < N; i++) {
    double x = drand48();
    double y = drand48();
    if (x * x + y * y < 1.0)
      hits++;
  }
  return 4.0 * hits / N;
}
```

Resolving data races: reductions

The `reduction` clause on the `parallel for` directive will collectively combine the changes to `hits` across the parallel threads when the directive closes.

This results in *many* fewer memory accesses to the original `hits`.

Reductions are not a universal solution to data races. However, they are well-suited to this case.

```
% ./pi
Elapsed: 1.076 seconds [Serial]
pi ≈ 3.141678240
Elapsed: 5.129 seconds [OpenMP #1a]
pi ≈ 0.651781120
Elapsed: 6.054 seconds [OpenMP #1b]
pi ≈ 3.149456840
Elapsed: 5.048 seconds [OpenMP #1c]
pi ≈ 3.138292680
```

Caches and sharing

Our initial code had three problems:

1. The `drand48` function is not required to be thread safe.
2. The increment of `hits` is a *data race* (or *race condition*) when updated in parallel.
3. It's slower.

We resolved #2 and (mostly) solved the correctness issues using the `atomic` directive[1] or `reduction` clause.

Problems #1 and #3 are related; however, thread *safety* does not mean thread *performant*.

1. The **`critical`** directive could be used instead of **`atomic`**, but it is much more expensive. For those familiar with Pthreads, it is equivalent to using a Pthreads mutex.

Caches and sharing

The `drand48` function uses global state to generate pseudo-random numbers.

Even if `drand48` was thread-safe, when invoked concurrently by multiple threads, this internal state will bounce between the caches of the cores running the threads.

We want to create thread-private state and generate random values without excessive data sharing and cache migration.

The GNU C Library (glibc) has an extension to `drand48`, `drand48_r`, that operates on user-specified buffers.

Putting it all together

Each OpenMP thread should seed its RNG state uniquely, otherwise each thread generates the same stream of values.

We create a shared base `seed` value and make it unique across threads by adding the thread id via `omp_get_thread_num()`, which requires including the OpenMP header.

```
#include <omp.h>

double pi_omp2(const size_t N) {
  size_t hits = 0;
  const size_t seed = random();
#pragma omp parallel reduction(+:hits)
  {
    struct drand48_data rngbuf;
    srand48_r(seed + omp_get_thread_num(), &rngbuf);
#pragma omp for
    for (size_t i = 0; i < N; i++) {
      double x, y;
      drand48_r(&rngbuf, &x);
      drand48_r(&rngbuf, &y);
      if (x * x + y * y < 1.0)
        hits++;
    }
  }
  return 4.0 * hits / N;
}
```

Putting it all together

```
% ./pi
Elapsed: 1.076 seconds [Serial]
pi ≈ 3.141678240
Elapsed: 5.129 seconds [OpenMP #1a]
pi ≈ 0.651781120
Elapsed: 6.054 seconds [OpenMP #1b]
pi ≈ 3.149456840
Elapsed: 5.048 seconds [OpenMP #1c]
pi ≈ 3.138292680
Elapsed: 0.191 seconds [OpenMP #2]
pi ≈ 3.141501120
```

```c
#include <omp.h>

double pi_omp2(const size_t N) {
  size_t hits = 0;
  const size_t seed = random();
#pragma omp parallel reduction(+:hits)
  {
    struct drand48_data rngbuf;
    srand48_r(seed + omp_get_thread_num(), &rngbuf);
#pragma omp for
    for (size_t i = 0; i < N; i++) {
      double x, y;
      drand48_r(&rngbuf, &x);
      drand48_r(&rngbuf, &y);
      if (x * x + y * y < 1.0)
        hits++;
    }
  }
  return 4.0 * hits / N;
}
```

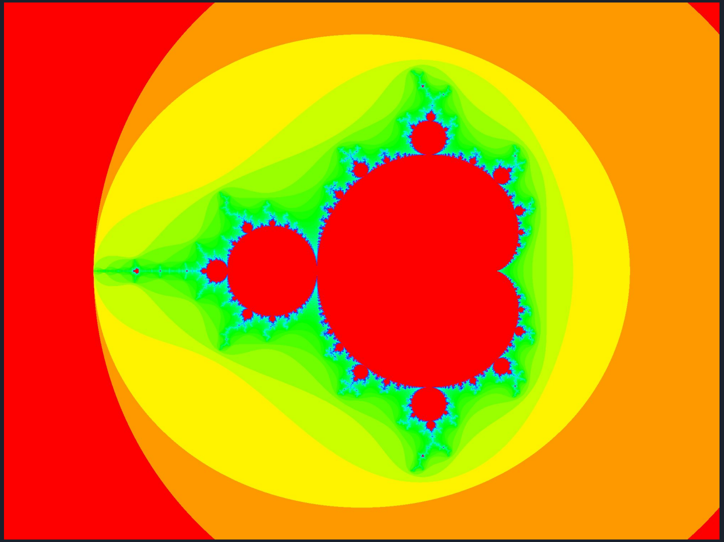
Hands-on: Mandelbrot + OpenMP

Goal

Apply parallelization using OpenMP to compute a visualization of the Mandelbrot set.

As before, you will start with:

- a serial implementation
- timing and I/O boilerplate code



OpenMP Specifications, OpenMP 4.5 API C/C++ Syntax Reference Guide

Hands-on: Mandelbrot + OpenMP

Plan

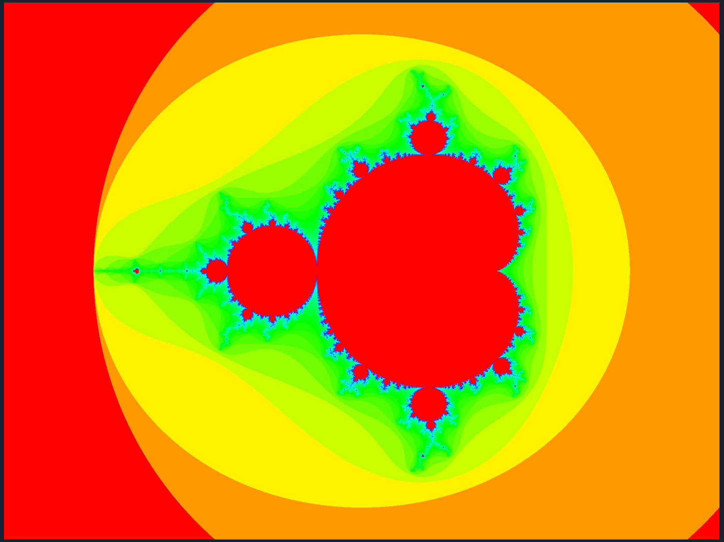
Start with the `parallel for` directive.

Then, look at the `collapse` and `schedule` clauses.

Try setting and varying the OMP_NUM_THREADS environment variable.

Reference

https://www.openmp.org/specifications/



OpenMP Specifications, OpenMP 4.5 API C/C++ Syntax Reference Guide

Wrapping up

Hands-on reflection

Interestingly, for the π calculation:

- The JIT-optimized Python outperformed the compiled, serial C application.
- The auto-parallelized Python outperformed the OpenMP implementation.

For the Mandelbrot generator…

- Which language had the better serial performance?
- Which implementation gave the best parallel performance?
- What techniques did you employ to improve parallel performance?
- What speedup and efficiency did you measure?

Start with fast serial code.

Then parallelize.

What was not covered

- SIMD/vector programming
- Distributed-memory HPC programming
  - SPMD: single program, multiple data
  - Message Passing Interface (MPI)
  - OpenSHMEM
- GPU programming
  - CUDA
  - HIP/ROCm
  - OpenCL
- “Non-HPC” distributed computing
  - Cloud
  - Map/Reduce

Thank you!

Any questions?

Further reading

On “scalability”

McSherry, F., Isard, M., & Murray, D. G. (2015). Scalability! but at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV).* https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

> *We offer a new metric for big data platforms, COST, or the Configuration that Outperforms a Single Thread. The COST of a given platform for a given problem is the hardware configuration required before the platform outperforms a competent single-threaded implementation. COST weighs a system’s scalability against the overheads introduced by the system, and indicates the actual performance gains of the system, without rewarding systems that bring substantial but parallelizable overheads.*

Python

- “Introduction to the Infamous Python GIL,” Intel Granulate
  https://granulate.io/blog/introduction-to-the-infamous-python-gil/
- “Bypassing the GIL for Parallel Processing in Python,” Real Python
  https://realpython.com/python-parallel-processing/
- “What Is the Python Global Interpreter Lock (GIL)?” Real Python
  https://realpython.com/python-gil/