Performance Tuning Is Hard And That Is Why You Should Do It

Ruud van der Pas

Senior Principal Software Engineer

Oracle Linux and Virtualization Engineering

Multicore World 2024

Christchurch, New Zealand, February 12-16, 2024

Agenda

The Talk 15:55 - 16:26

- Why Talking about Performance?
- The Impact of AI
- Deep in the Dungeon
- Repeating Myself

Q and (some) A 16:27 - 16:30

Summary - For the Impatient

Benefits of improving the application performance:

- ***Save money***
- ***Reduce product development time and/or quality of the product***
- ***Much better understanding of your resource requirements***
- ***Educate your developers to write efficient code from day 1***

Get your application performance expert today!

Why Talking about Performance?

Myths - Greatest Hits

"Compilers are so smart by now, they will fix my inefficient code"

"Compilers have no clue, so I need to do all the low level tuning"

"It is cheaper to just buy a faster processor"

"We will do the performance tuning when we're done with the code"

"AI can do the performance tuning for us"

"That is all fine and good, but it is too expensive to do"

"Sure. Let's do some basic calculus."

Basic Calculus

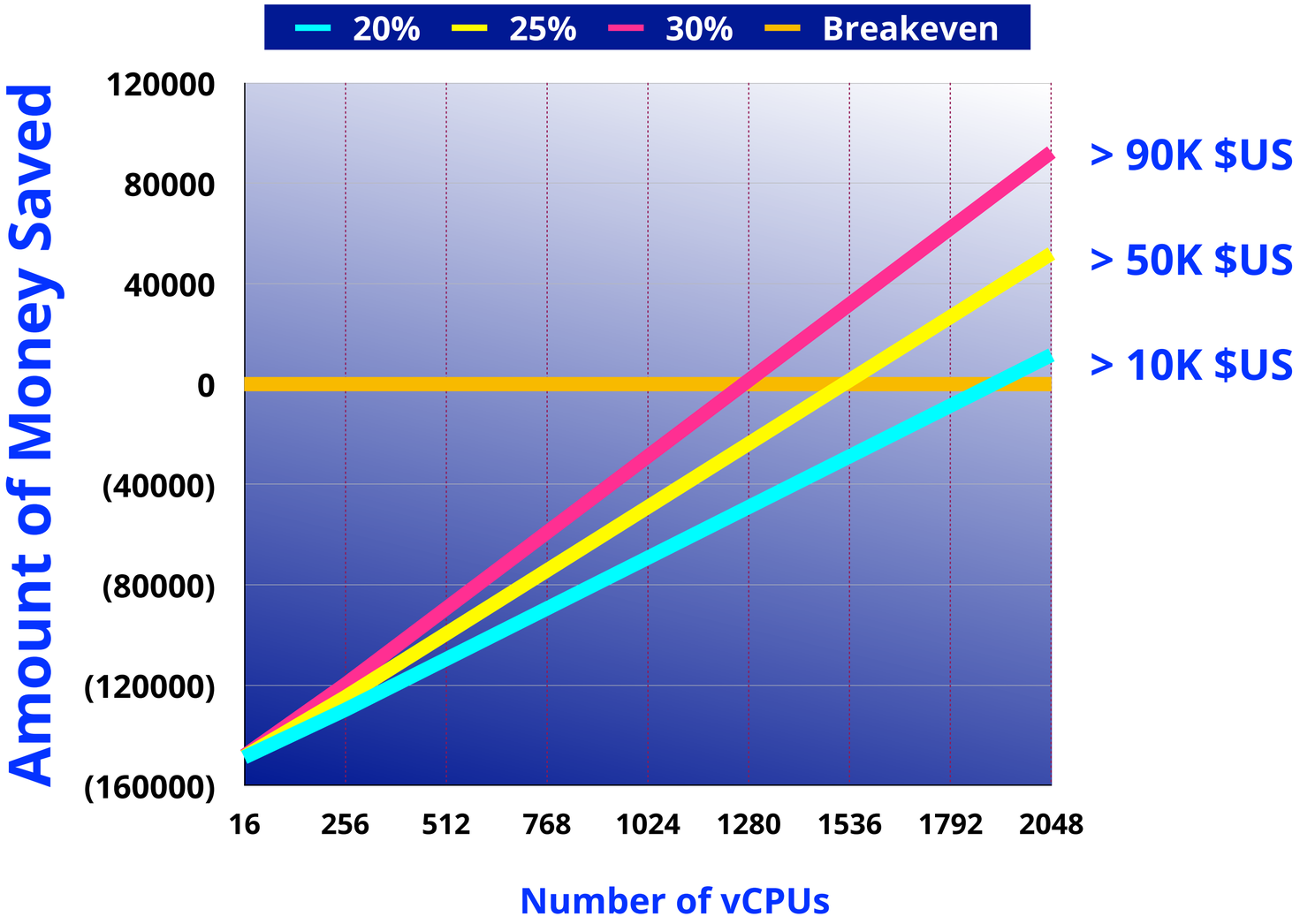
c7g.4xlarge*	USA (North California)
vCPU	16
Memory	32 GB
Cost/hour ($US)	0.7208
Cost/year ($US)	6314
Salary Performance Geek ($US)	150K

*) Compute Optimized, 7-th instance generation, processor family “g” (AWS Graviton) - lowest cost in this class

https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html#instance-type-names

https://aws.amazon.com/ec2/pricing/on-demand/

And the Winner is ...



The Impact of ~~Always Incorrect~~ AI

The Challenge in Performance Tuning

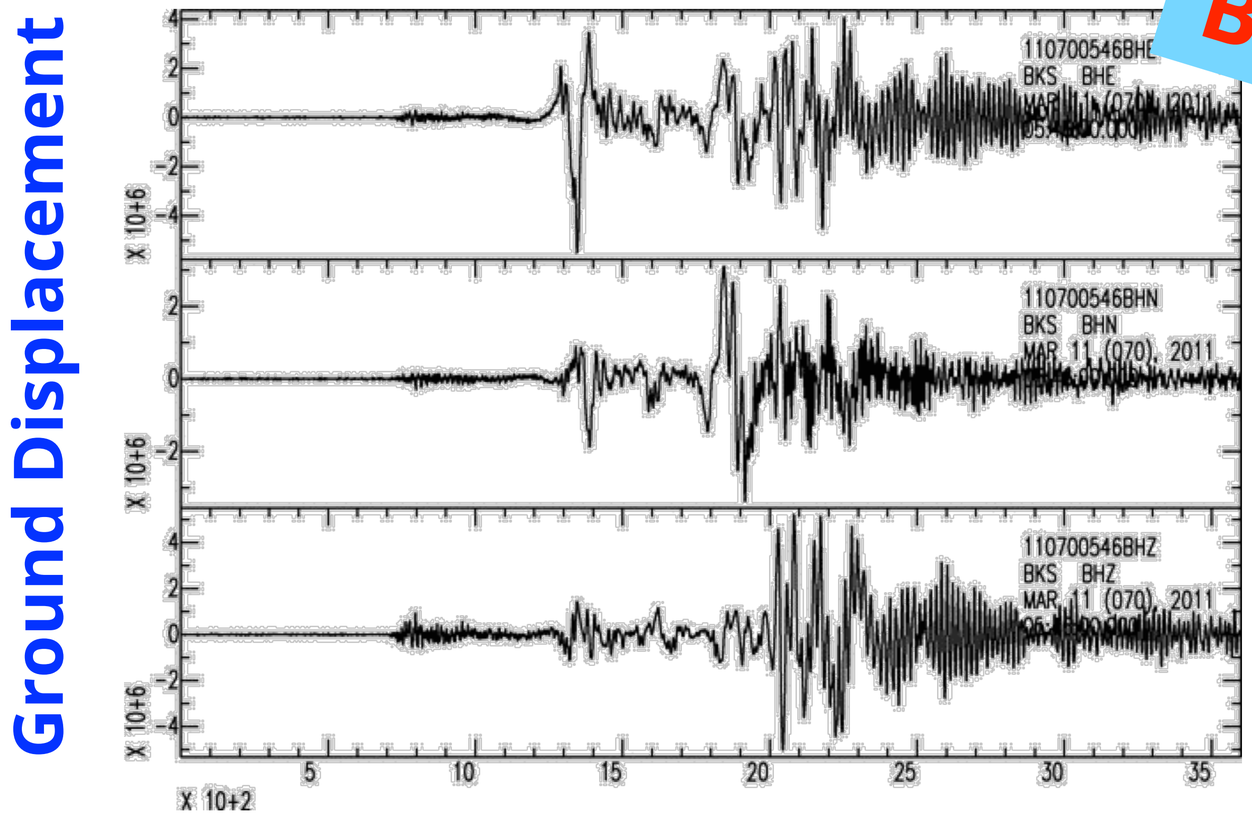
What Is The Reference?

Is What You Observe "Normal"?

Challenge: What is Normal?

Seismographic Data

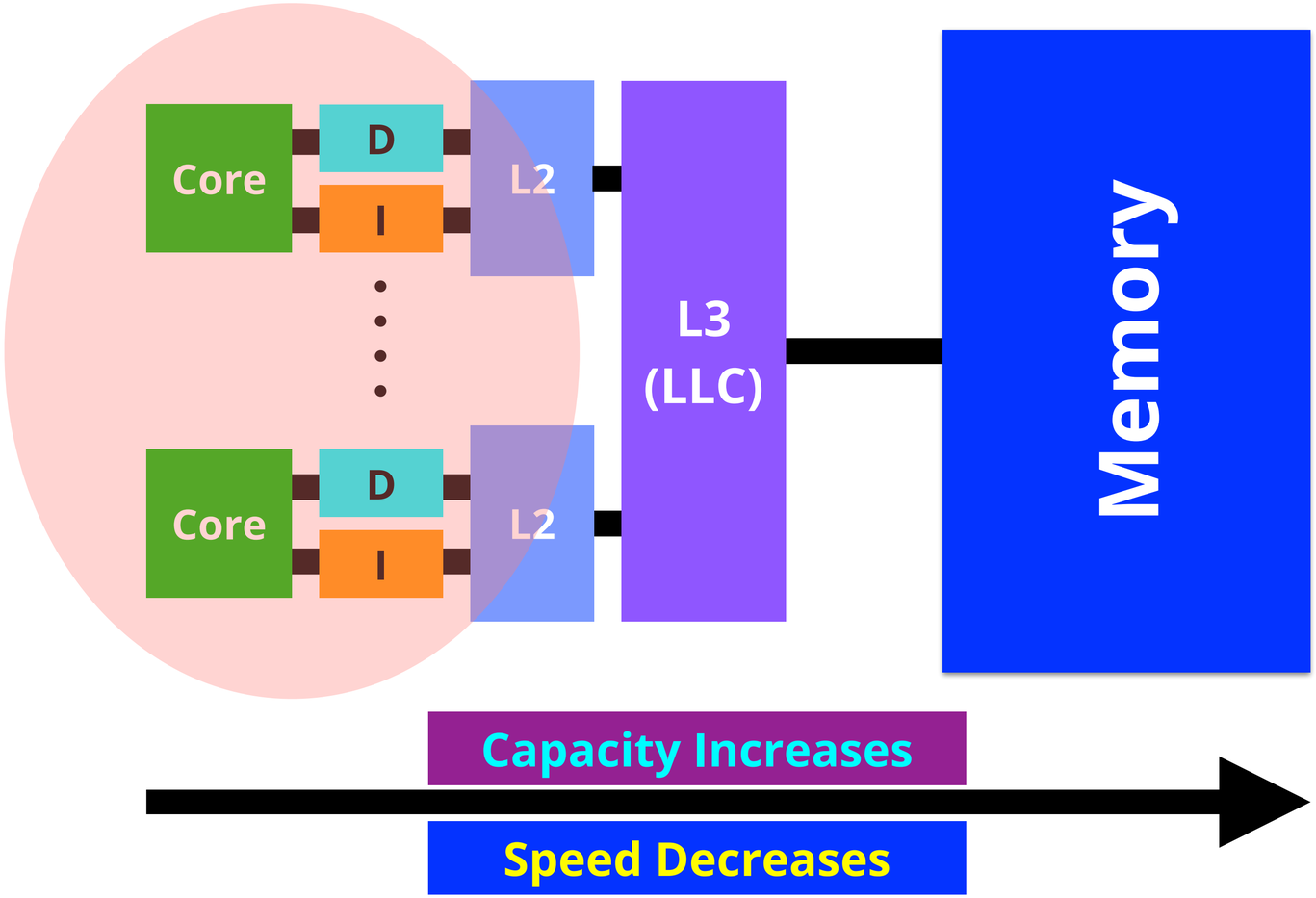
Outliers are BAD news!



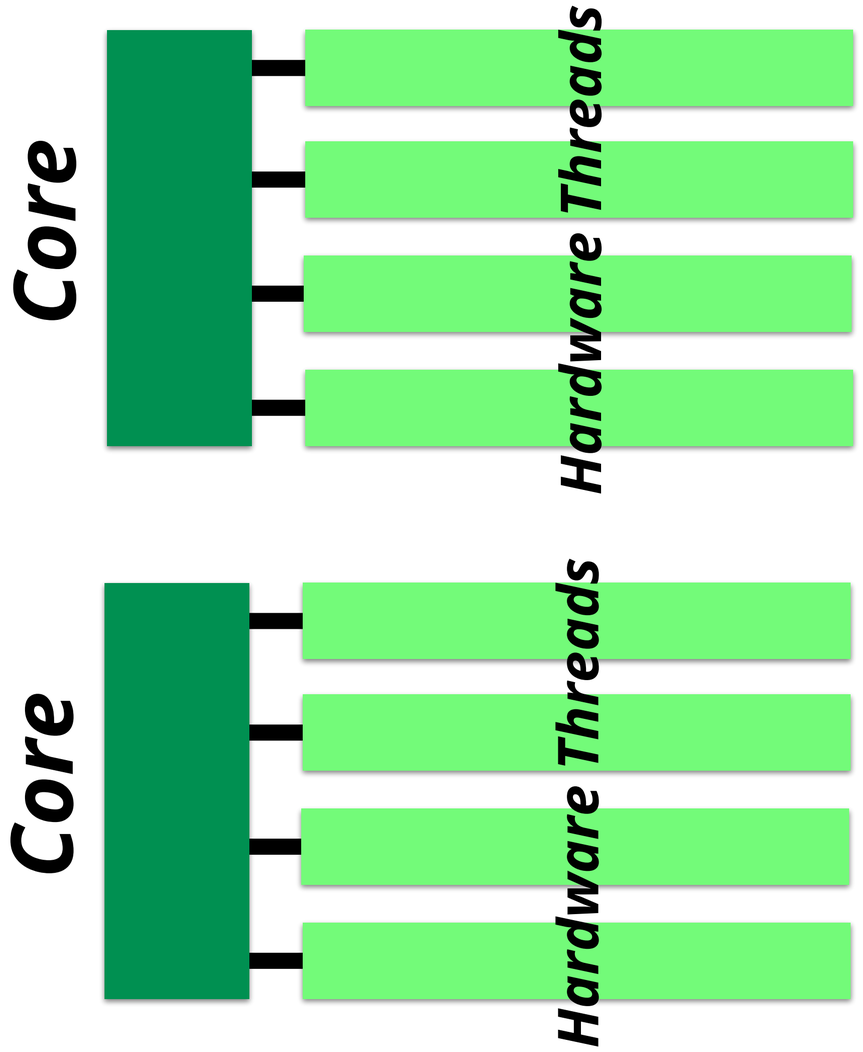
You Probably Want to See Outliers Here Though

Common Relatively Simple Architectures

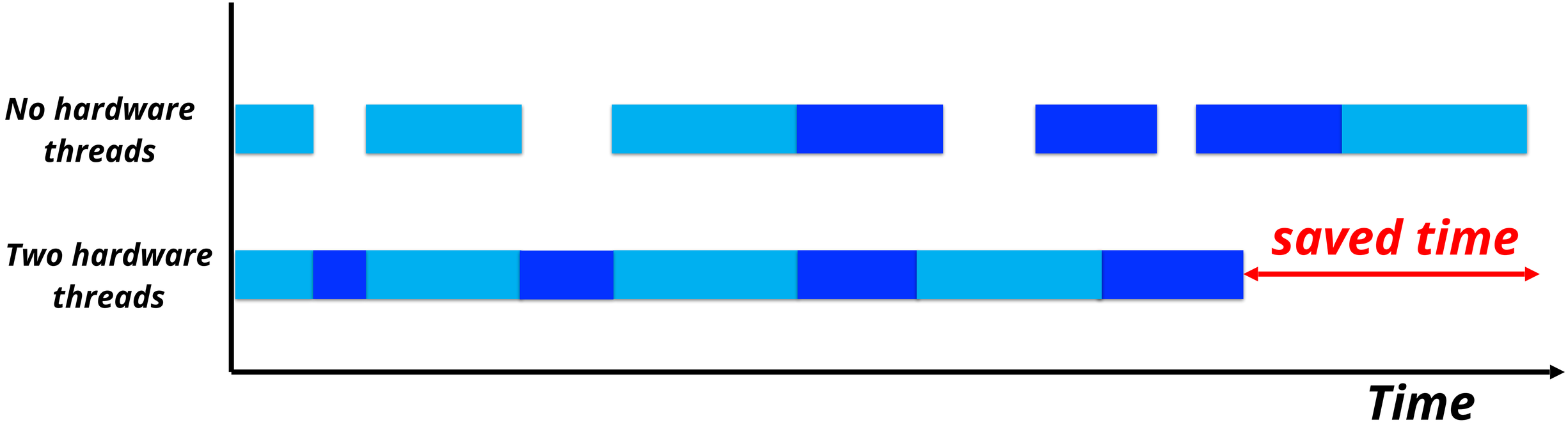
A Typical System



Hardware Threads

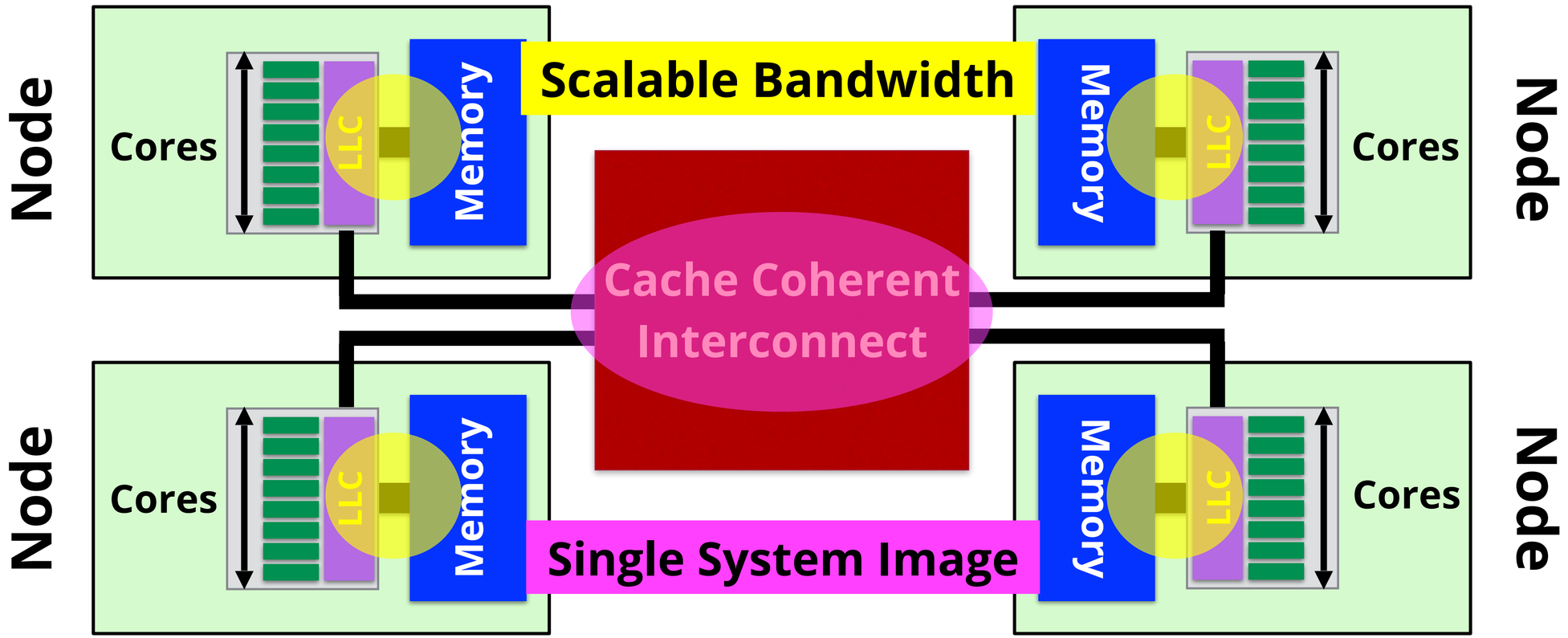


How Hardware Threads Work

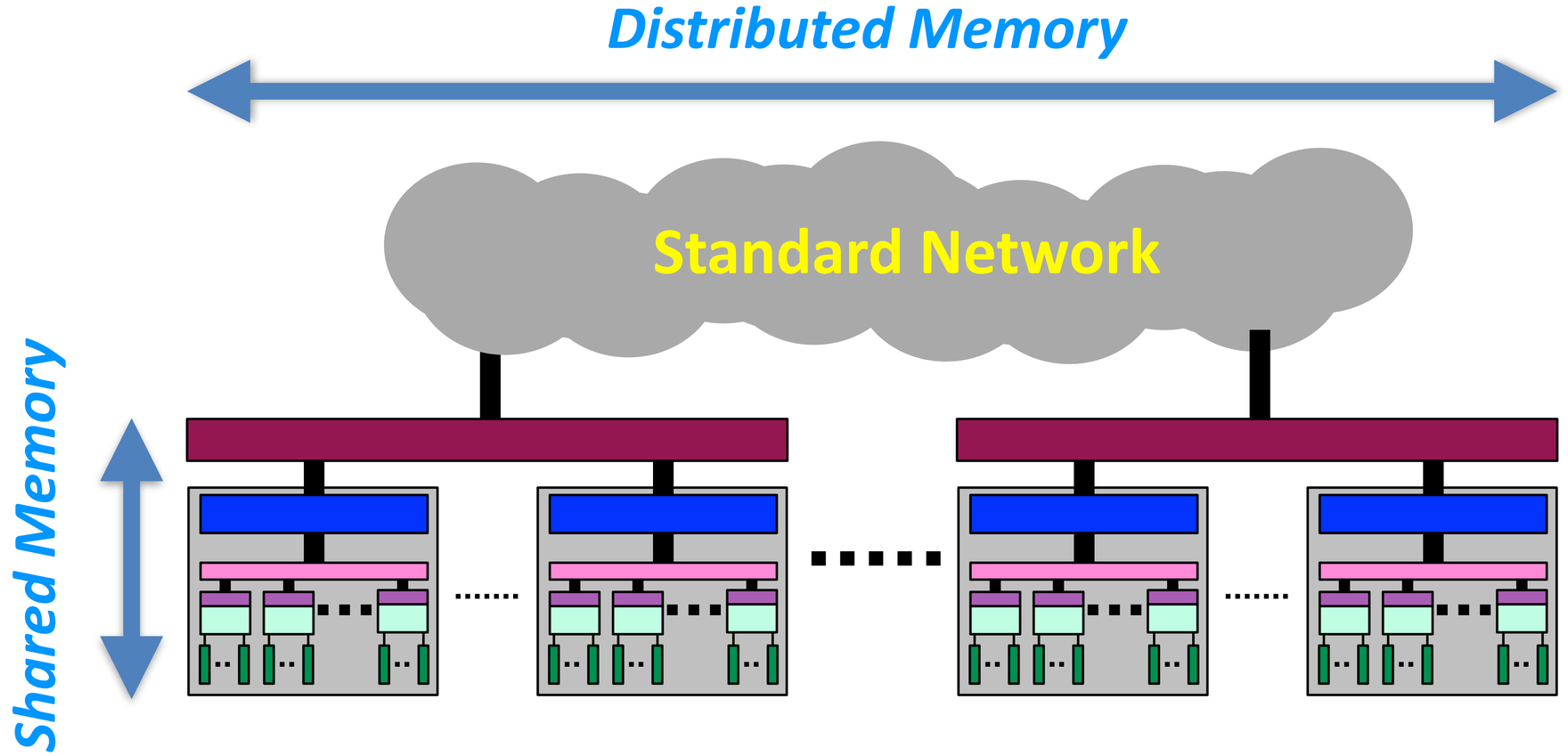


NUMA - The System Most of Us Use Today

A Generic, but very Common and Contemporary NUMA System



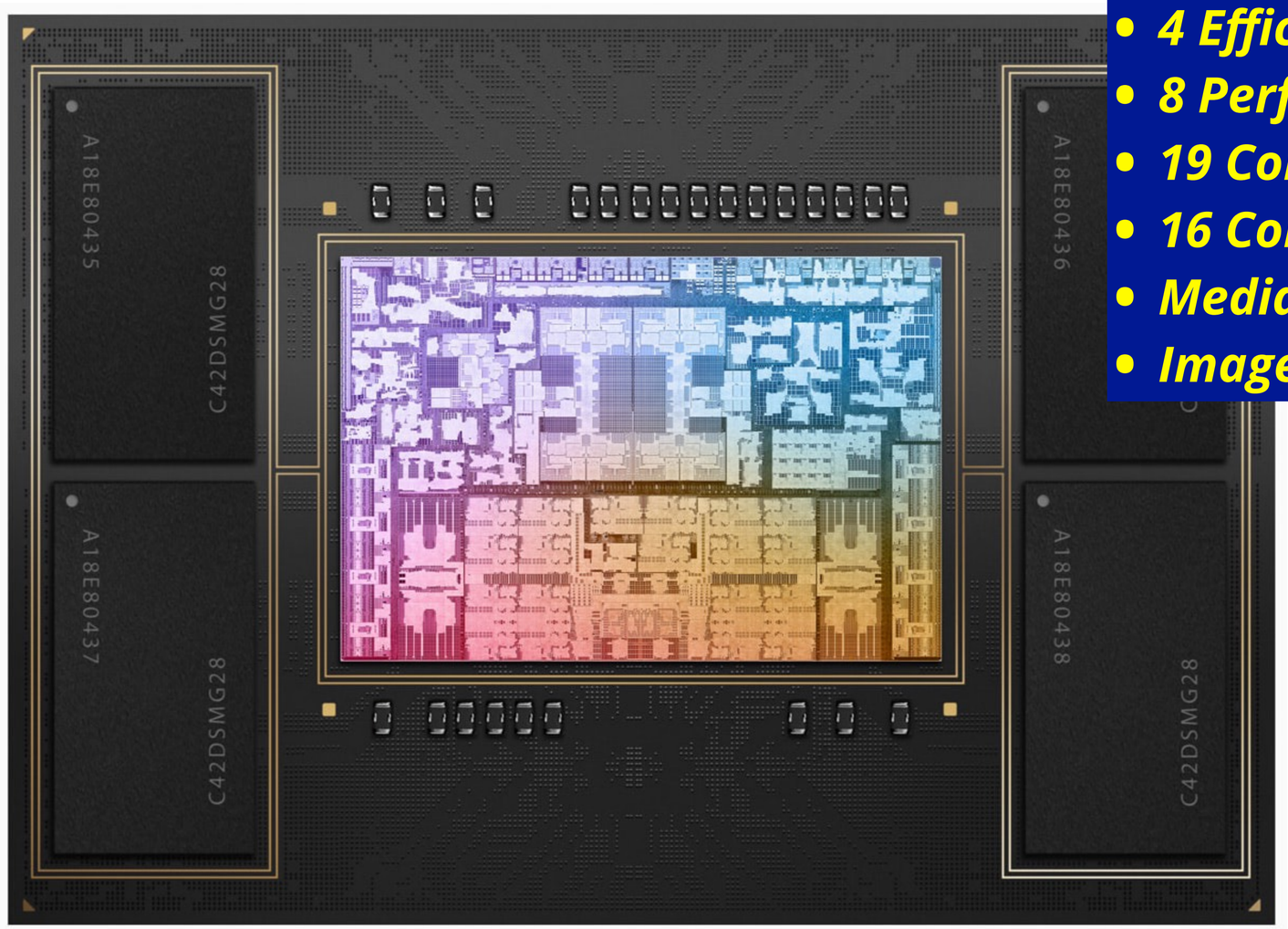
A Hybrid Multi-Level Parallel System



"But these are very complex systems I don't use"

"Let's look at a laptop. Do you think that is any easier?"

An Example - The Apple M2 Pro Processor



- 4 Efficiency Cores
- 8 Performance Cores
- 19 Core GPU
- 16 Core Neural Engine
- Media Engine
- Image Signal Processor

"And You Honestly Think AI Can Master These Kinds Of Beasts?"

"Yes! AI Can Be Trained To Do This!"

"Sure. Let me know when it is ready."

Deep in the Dungeon

We Have These Extremely Large Parallel Systems

Won't that give us unlimited performance?

About Single Thread Performance

Why Is This Still So Important?

We're All In Nicolás's Multicore World Universe After All ;-)

Amdahl's Law*

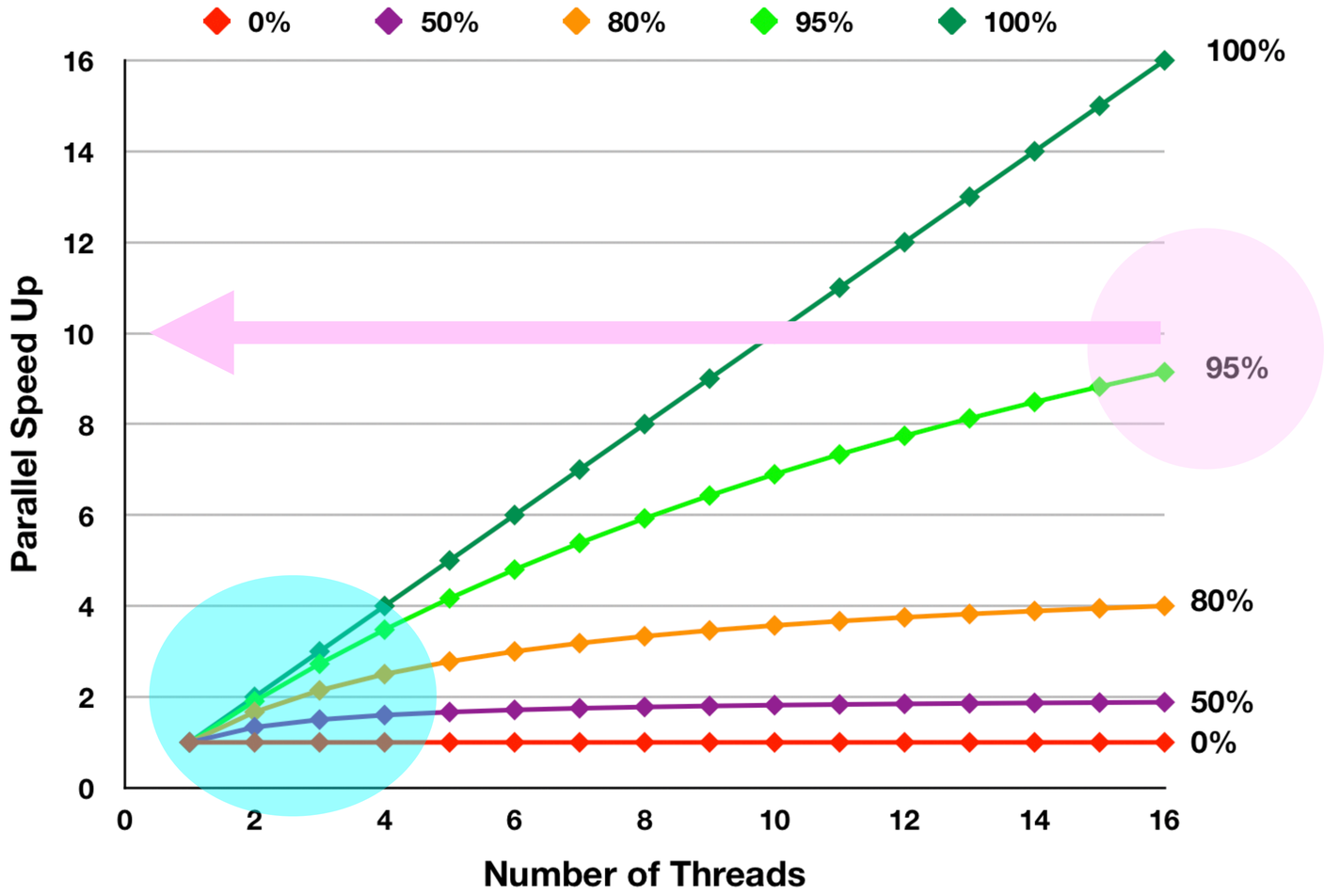
Suppose your application needs 100 seconds to run

If 80% of this run time can execute in parallel, the time using 4 threads is 80/4+20 = 40 seconds

This means that your program is 2.5x faster, not 4x

*) No, this is not another EU legislation on the maximum noise of a vacuum cleaner. It is actually relevant.

Amdahl's Law Using 16 Threads



A Simple Tuning Example

An Example from Graph Analysis

The OpenMP reference version of the Graph 500 benchmark

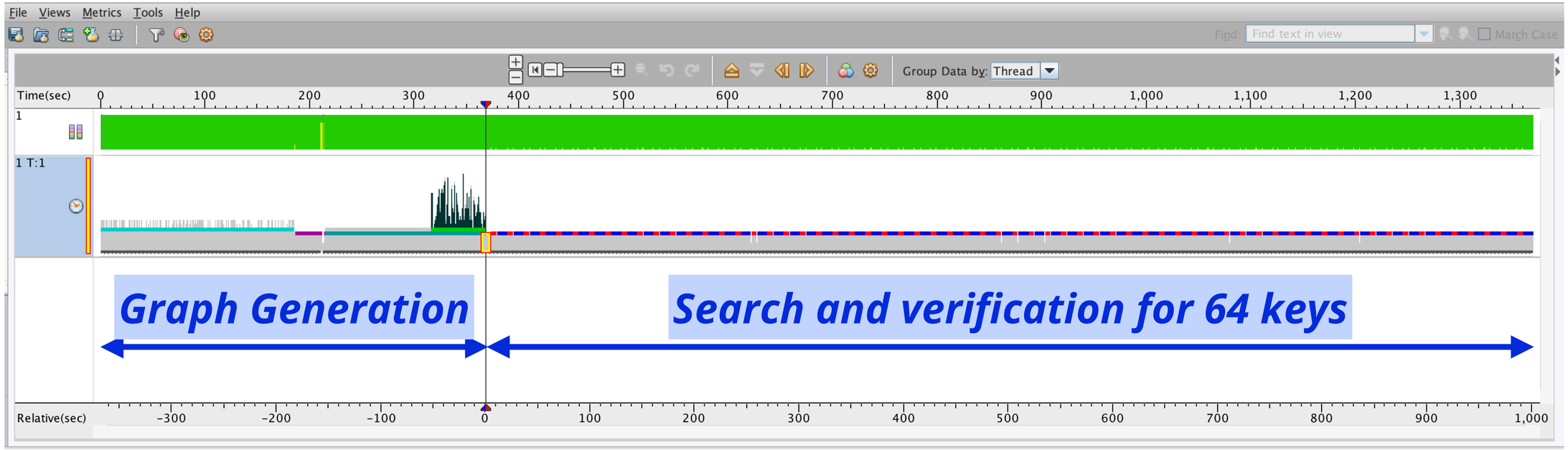
Structure of the code:

- ***Generate an undirected graph of the specified size***
- ***Randomly select a key and conduct a BFS search***
- ***Verify the result is a tree***

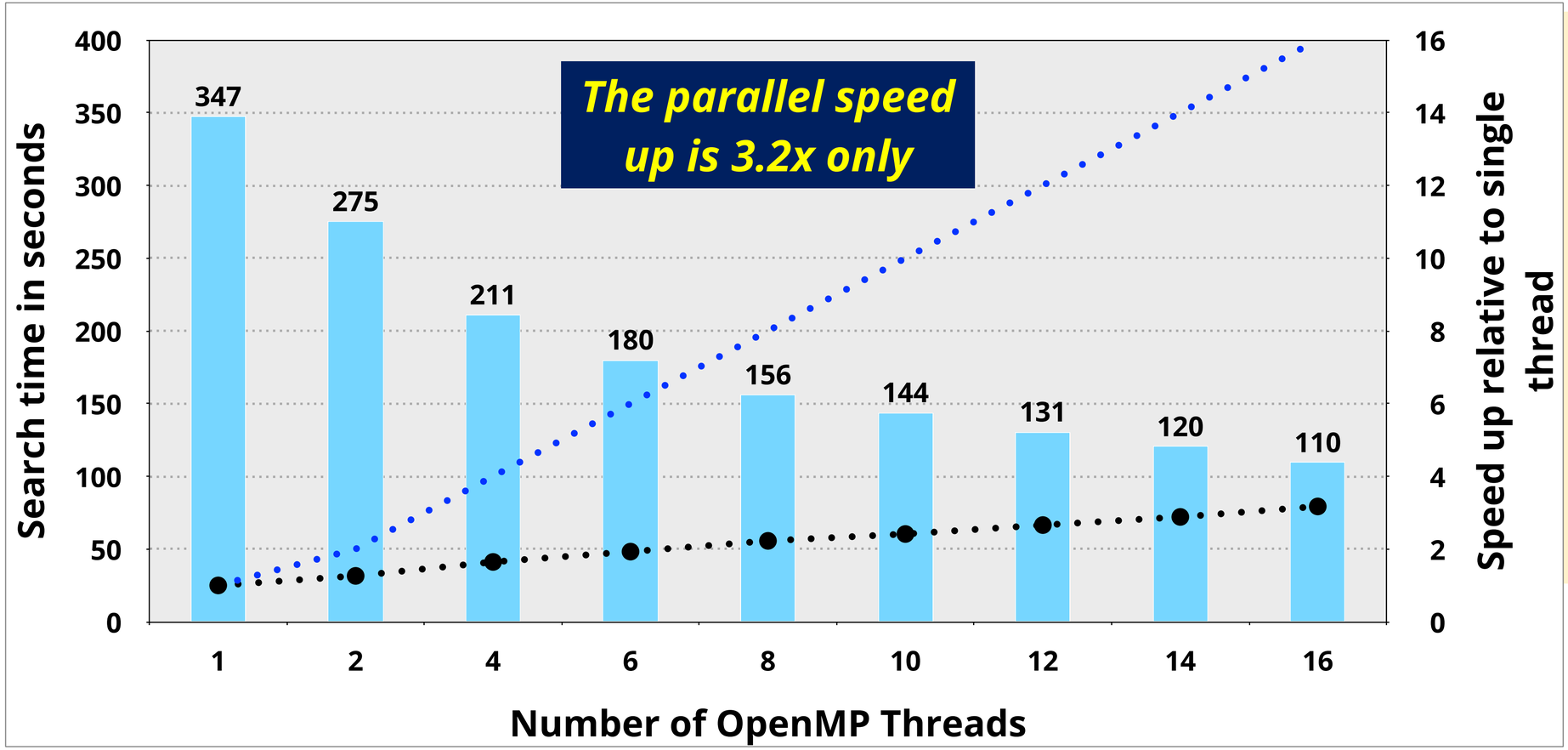
Repeat 64 times

For the benchmark score, only the search time matters

The Dynamic Behaviour



The Scalability is Disappointing



The data is for a 9 GB sized problem (SCALE 24)

Search time reduces as threads are added

Benefit from all 16 (hyper) threads

The 3.2x parallel speed up is disappointing

The parallel scalability is similar for larger graphs

System: A VM with 8 Intel Xeon Platinum 8167M CPU @ 2.00GHz ("Skylake") cores, 16 hardware threads

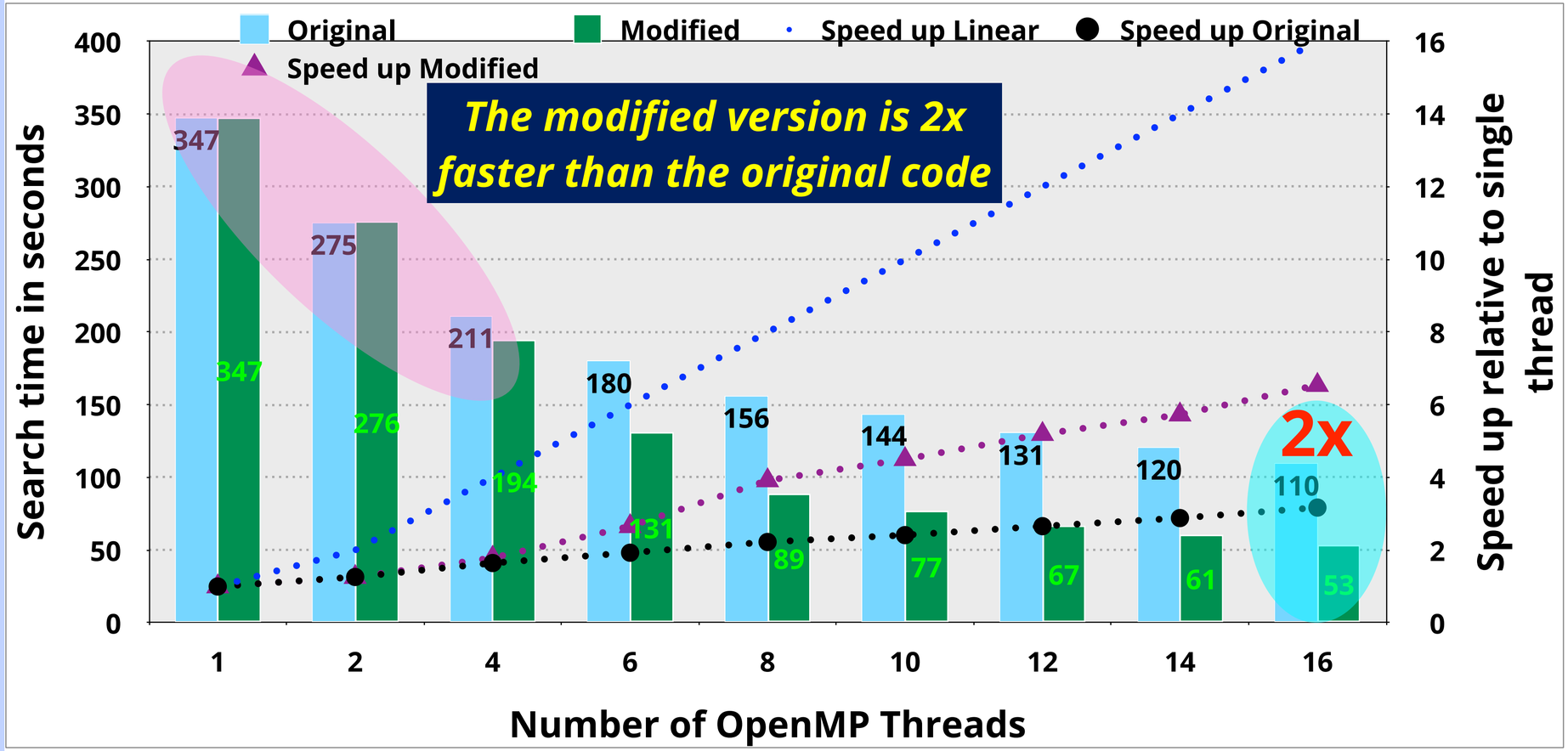
Action Plan

Used the gprofng profiling tool to find the time consuming parts

Found several opportunities to improve the OpenMP part

Although simple changes, the improvement is substantial:

Performance Of The Original and Modified Code

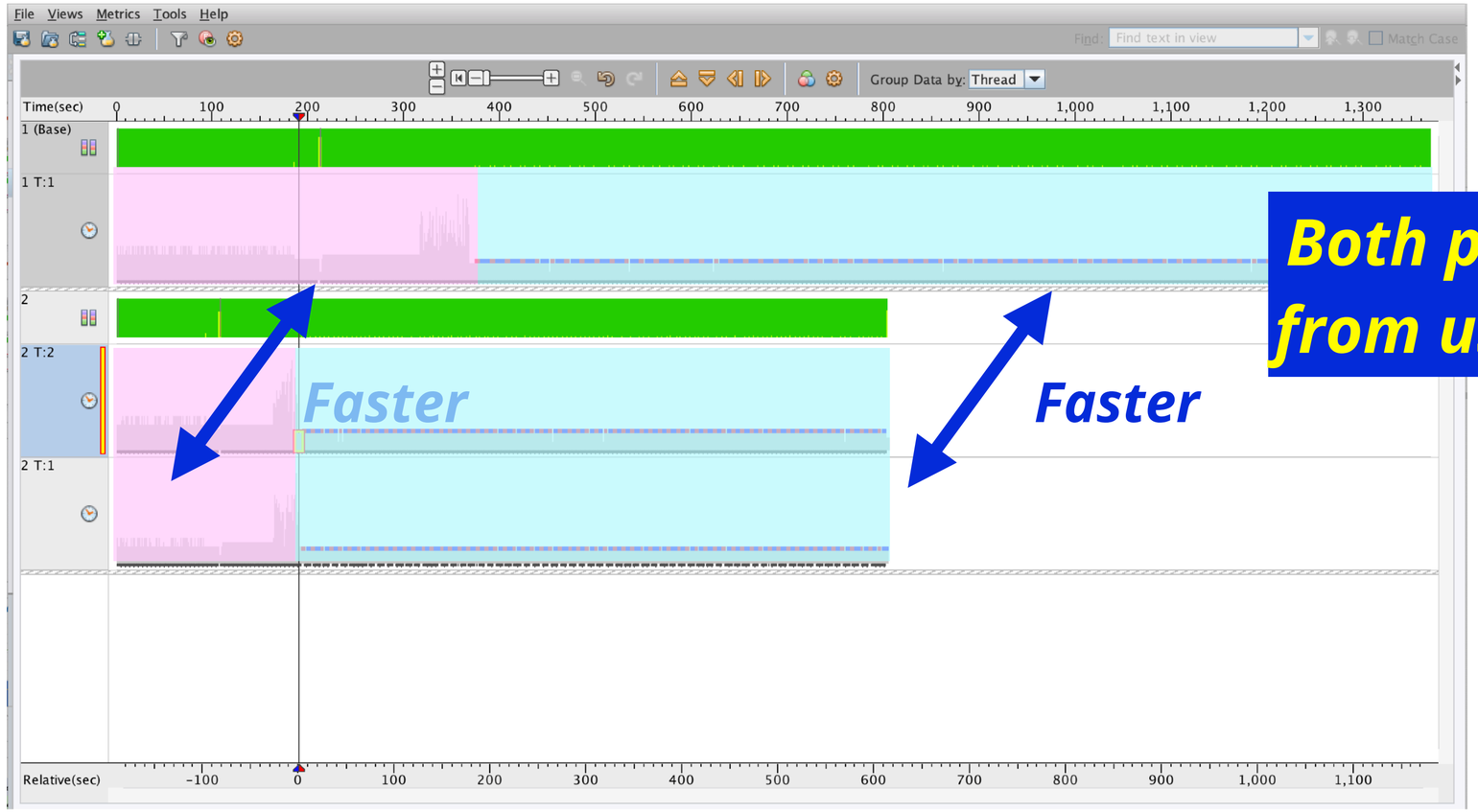


- ✓ *A noticeable reduction in the search time at 4 threads and beyond*
- ✓ *The parallel speed up increases to 6.5x*
- ✓ *The search time is reduced by 2x*

The KEY Question

Are We Done Tuning This Code?

A Comparison Between 1 And 2 Threads

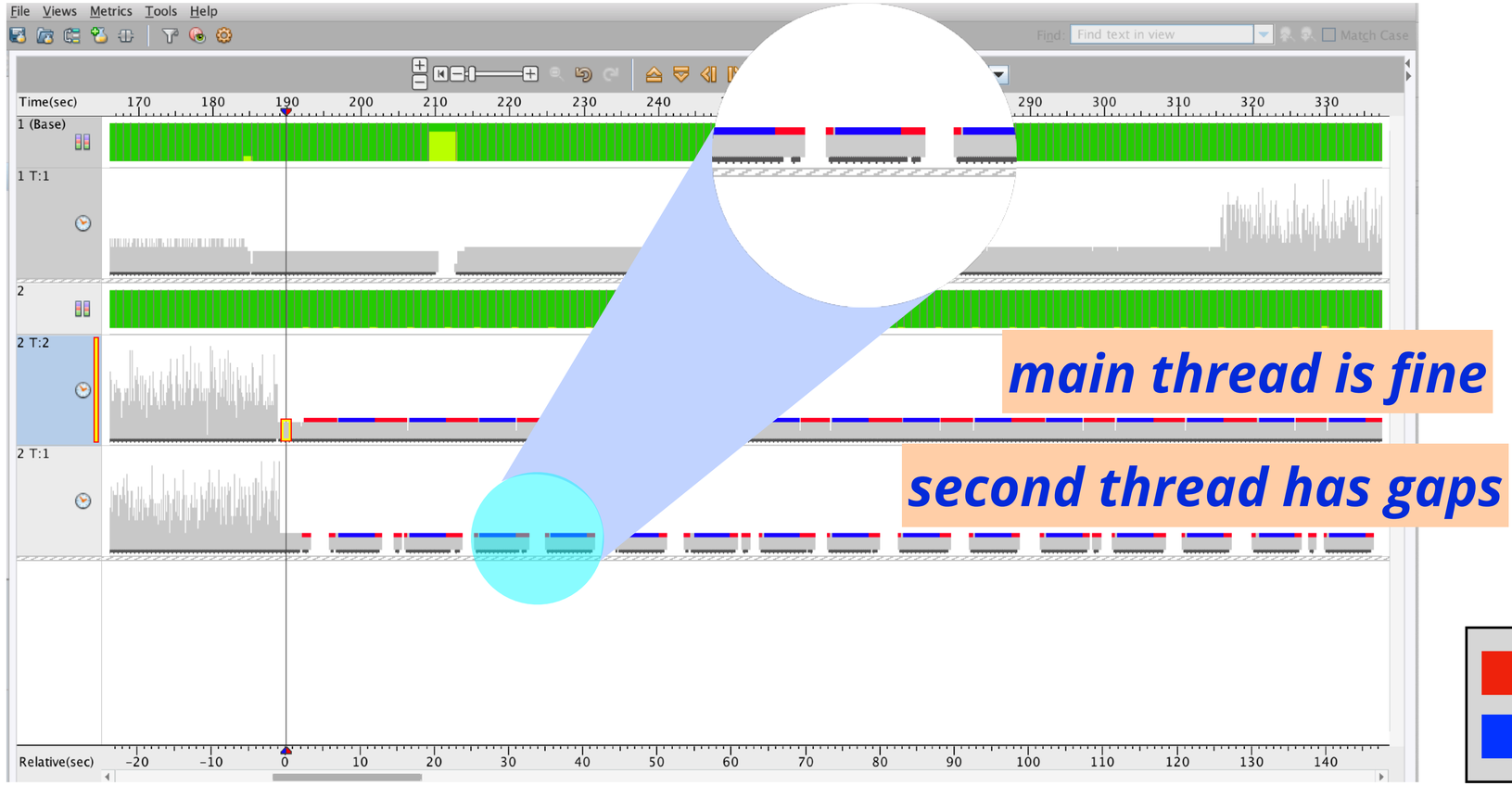


Both phases benefit from using 2 threads

Search
Verification

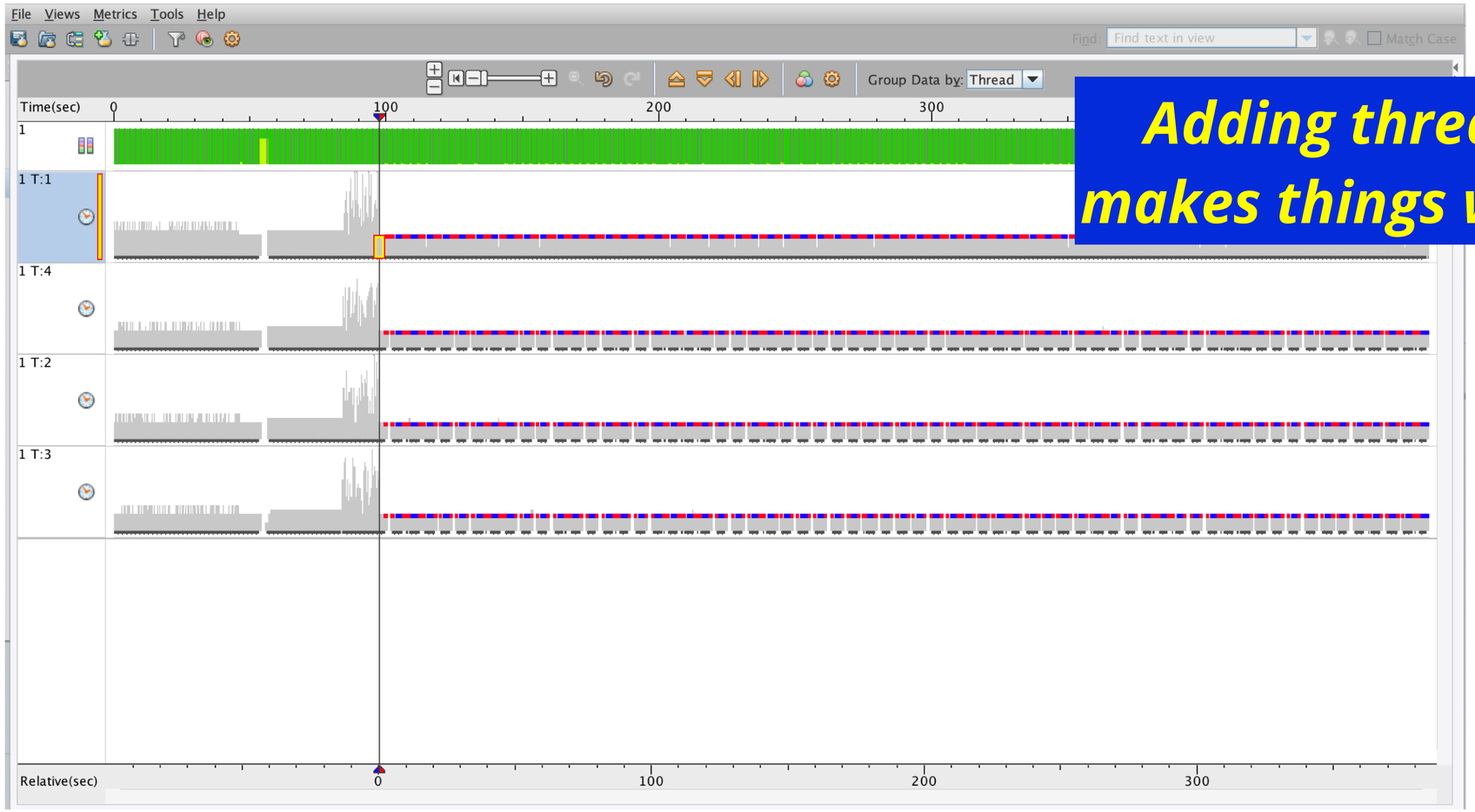
Note: The GNU gprofng profiling tool was used to create this view (https://sourceware.org/binutils)

Zoom In On The Second Thread



Note: The GNU gprofng profiling tool was used to create this view (https://sourceware.org/binutils)

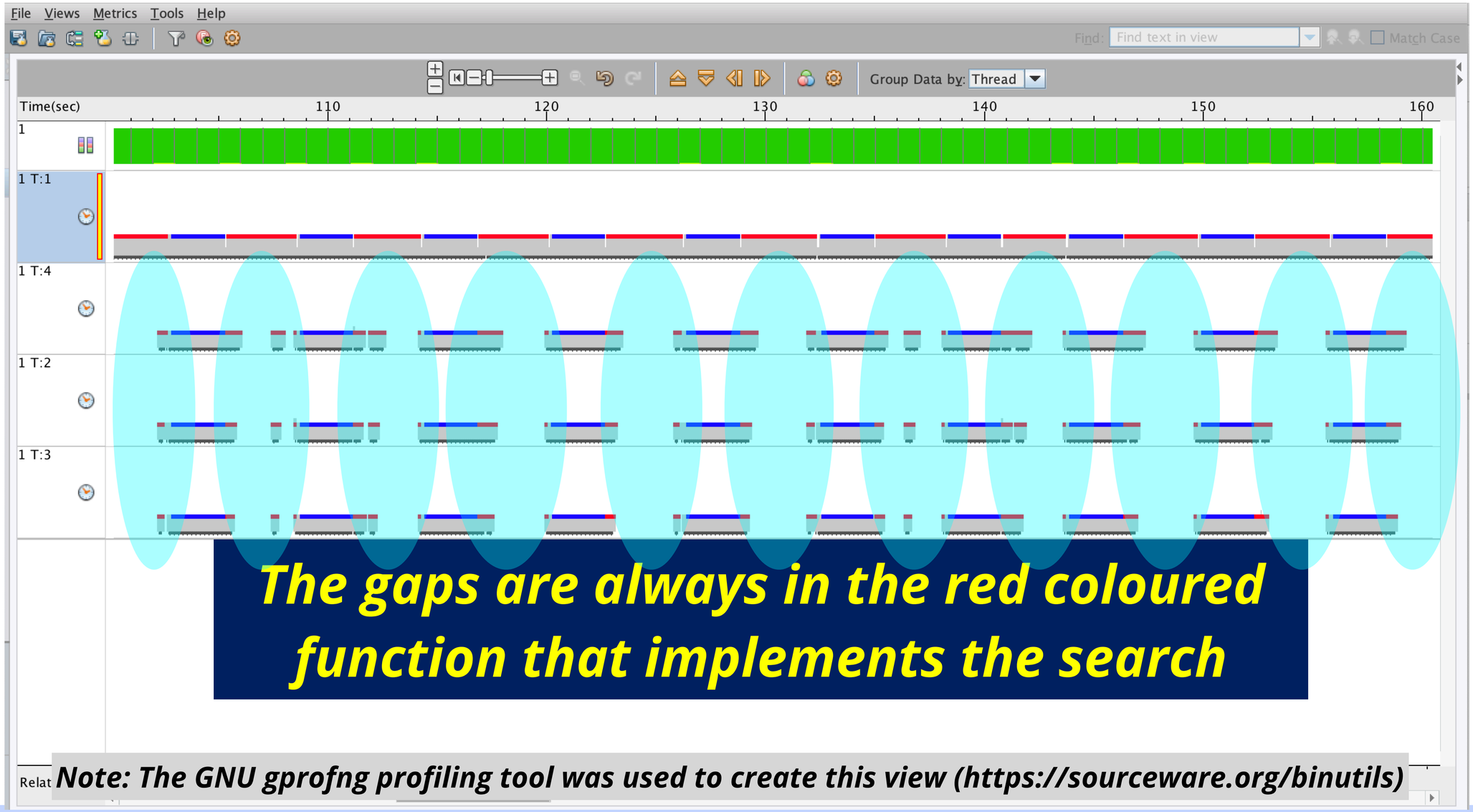
How About 4 Threads?



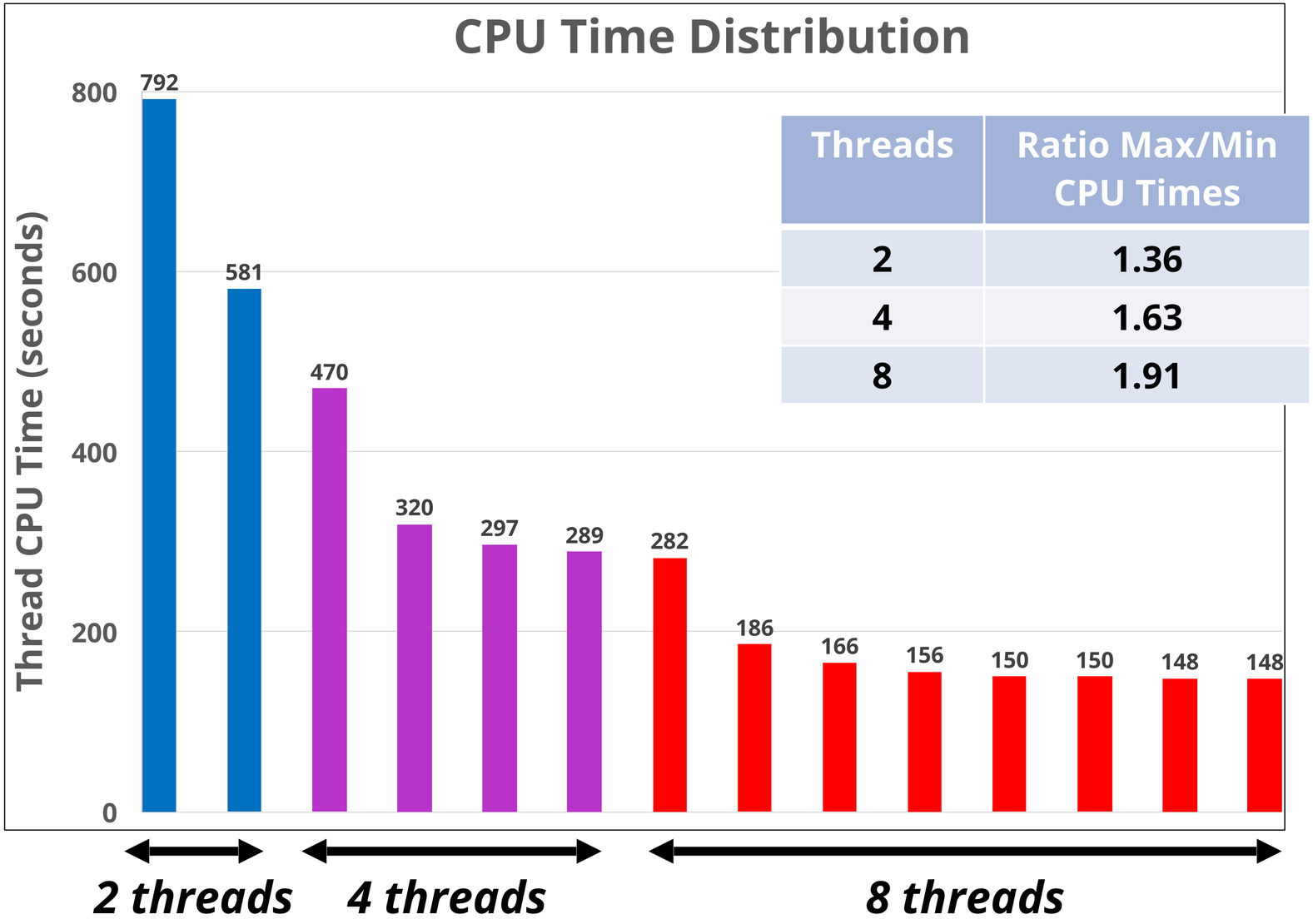
Adding threads makes things worse

Note: The GNU gprofng profiling tool was used to create this view (https://sourceware.org/binutils)

Zoom In Some More



CPU Time Variations



Threads	Ratio Max/Min CPU Times
2	1.36
4	1.63
8	1.91

The load imbalance increases as the thread count goes up

Note: Data obtained with the GNU gprofng profiling tool (https://sourceware.org/binutils)

The Issue

```
#pragma omp for
for (int64_t k = k1; k < oldk2; ++k) {
  const int64_t v = vlist[k];
  const int64_t veo = XENDOFF(v);
  for (int64_t vo = XOFF(v); vo < veo; ++vo) {
    const int64_t j = xadj[vo];
    if (bfs_tree[j] < 0) {
      if (int64_cas(&bfs_tree[j],-1,v)) {
        if (kbuf > (VLOCAL_BUF_LEN-2)) {
          nbuf[kbuf++] = j;
          for (int64_t vk=0; vk<kbuf; ++vk)
            vlist[vlen++] = nbuf[vk];
          kbuf = 0;
        } else {
          int64_t vk;
          assert(kbuf < VLOCAL_BUF_LEN);
          for (int64_t vk=0; vk<kbuf; ++vk)
            vlist[vlen++] = nbuf[vk];
          nbuf[0] = j;
          kbuf = 1;
        } // End of if-then-else
      } // End of if
    } // End of if
  } // End of for-loop on vo
} // End of parallel for-loop on k
```

Irregular workload per k-iteration

Fixed length loop

Irregular length loop

Irregular control flow

Observations and the Solution

The #pragma omp for loop uses default scheduling

The default is implementation dependent, but is static here

In this case, that leads to load balancing issues

The solution: #pragma omp for schedule(dynamic)

Or an even better solution: #pragma omp for schedule(runtime)

Our setting: $ export OMP_SCHEDULE="dynamic,25"

By The Way

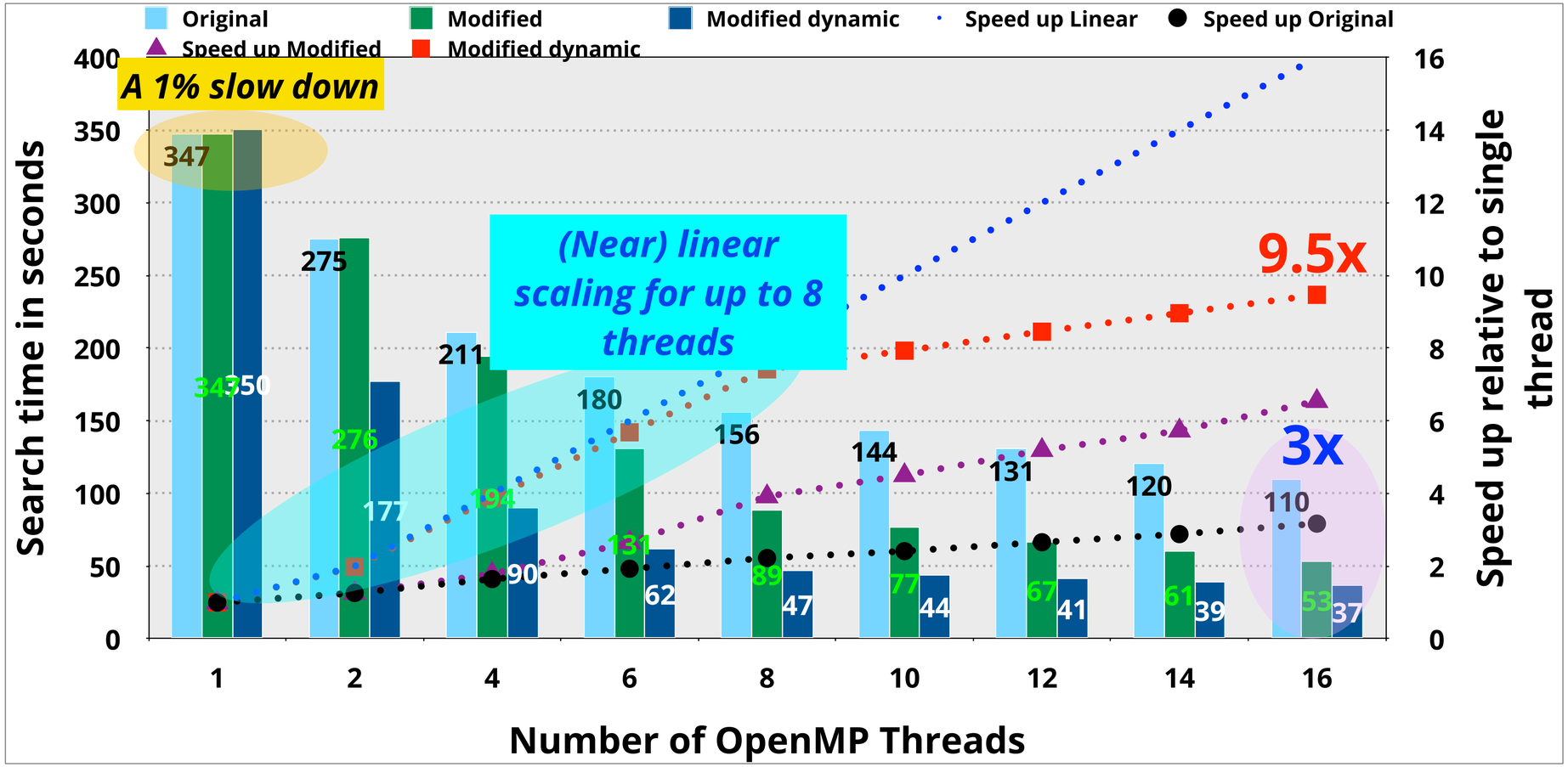
How Do You Know The Chunk Size Should Be 25?

~~Crystal Ball~~

Trial And Error

Devil's Advocate: Can AI realize this and do such things?

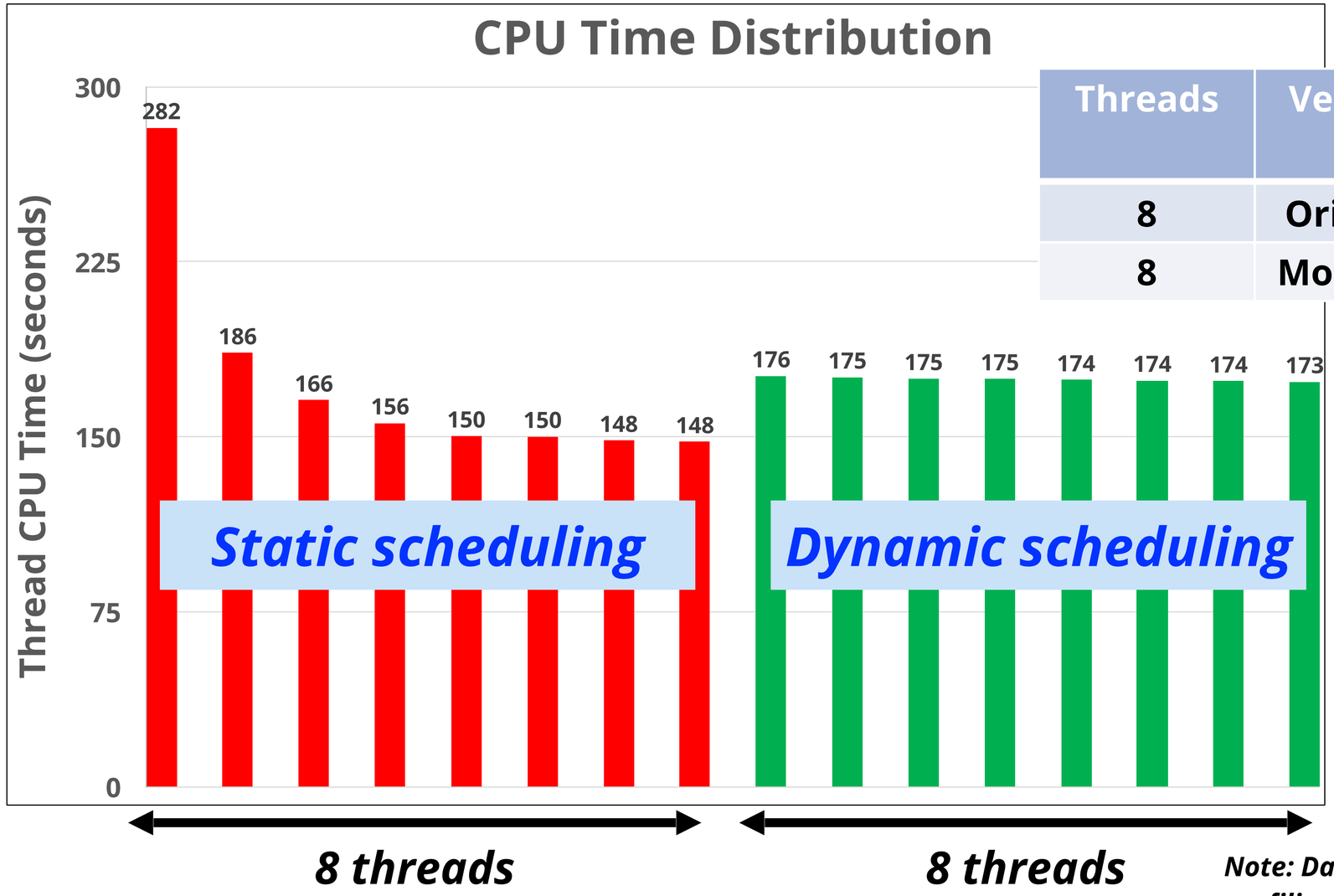
The Performance With Dynamic Scheduling Added



- ✓ A 1% slow down on a single thread
- ✓ Near linear scaling for up to 8 threads
- ✓ The parallel speed up increased to 9.5x
- ✓ The search time is reduced by 3x

The modified version is 3x faster than the original code

The Load Imbalance is Indeed Really Gone



Threads	Version	Ratio Max/Min CPU Times
8	Original	1.91
8	Modified	1.01

Note: Data obtained with the GNU gprofng profiling tool (https://sourceware.org/binutils)

Repeating Myself

Summary - For the Patient

Benefits of improving the application performance:

- ***Save money***
- ***Reduce product development time and/or quality of the product***
- ***Much better understanding of your resource requirements***
- ***Educate your developers to write efficient code from day 1***

Get your application performance expert today!

Can't Resist

"By the way, any progress with your AI efforts?"

"Still working on it."

"Sure. Let me know when it is ready."


Thank You And ... Stay Tuned!
ruud.vanderpas@oracle.com