Performance and Profiling

Jim Stone

10 November 2016

Outline

Performance

First principle of optimization

Don't

Drawbacks of optimization:

- Sacrifice clarity
- Sacrifice maintainability

Second principle of optimization

But you must.

Importance of optimization:

- The problems you can solve are determined by the performance of your code.
- Access to shared resources (clusters) often requires demonstrating good performance.

Measuring Performance

Quanta of time on a computer is a **clock period** (or cycle), which is the inverse of frequency at which processor runs.

Processor frequency is variable (e.g. turbo-boost) on most CPUs, which complicates performance measurements.

Generally,

$\text{performance} \propto 1/(\text{time})$

Goal is to maximize performance (minimize time) in each step of computation (calculations, I/O, communication, etc.)
Performance can be measured using

- profiling tools like `gprof` and VTune
- a *benchmark* real-world application program or kernel run on hardware being tested.

For example, the Linpack benchmark is used for the top500 list.

We’re spoiled by Moore’s Law

Prediction by Gordon Moore, 1965.

The transistor density in integrated circuits will be doubled every two years.

Since performance scales with transistor density, Moore’s Law has been interpreted as a prediction about the former as well.

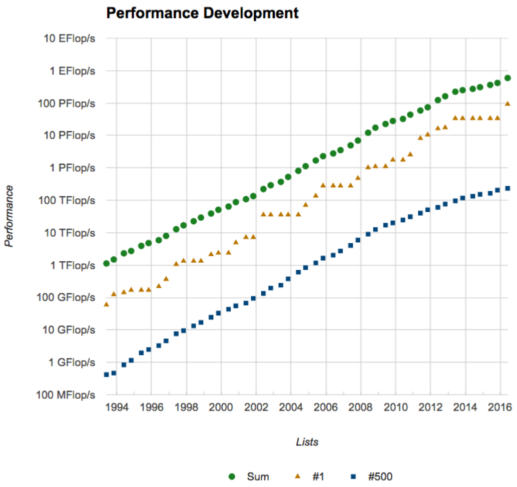
Actually, performance doubles about every 18 months.

Shows no sign of ending (see plot next slide from `top500.org`).

But, we are forced to use mass-produced commodity processors that were not designed for scientific computation

Hardware

Increasing importance of computation driven in part by the staggering growth in performance of the world's fastest computers.



Koomey's Law

Koomey et al. 2011

The energy efficiency of integrated circuits will be doubled every eighteen months.

Currently power consumption is a limiting factor in HPC centers.

A $\sim 10^6$ increase in energy-efficiency of processors is still possible before fundamental physical limits will be encountered.

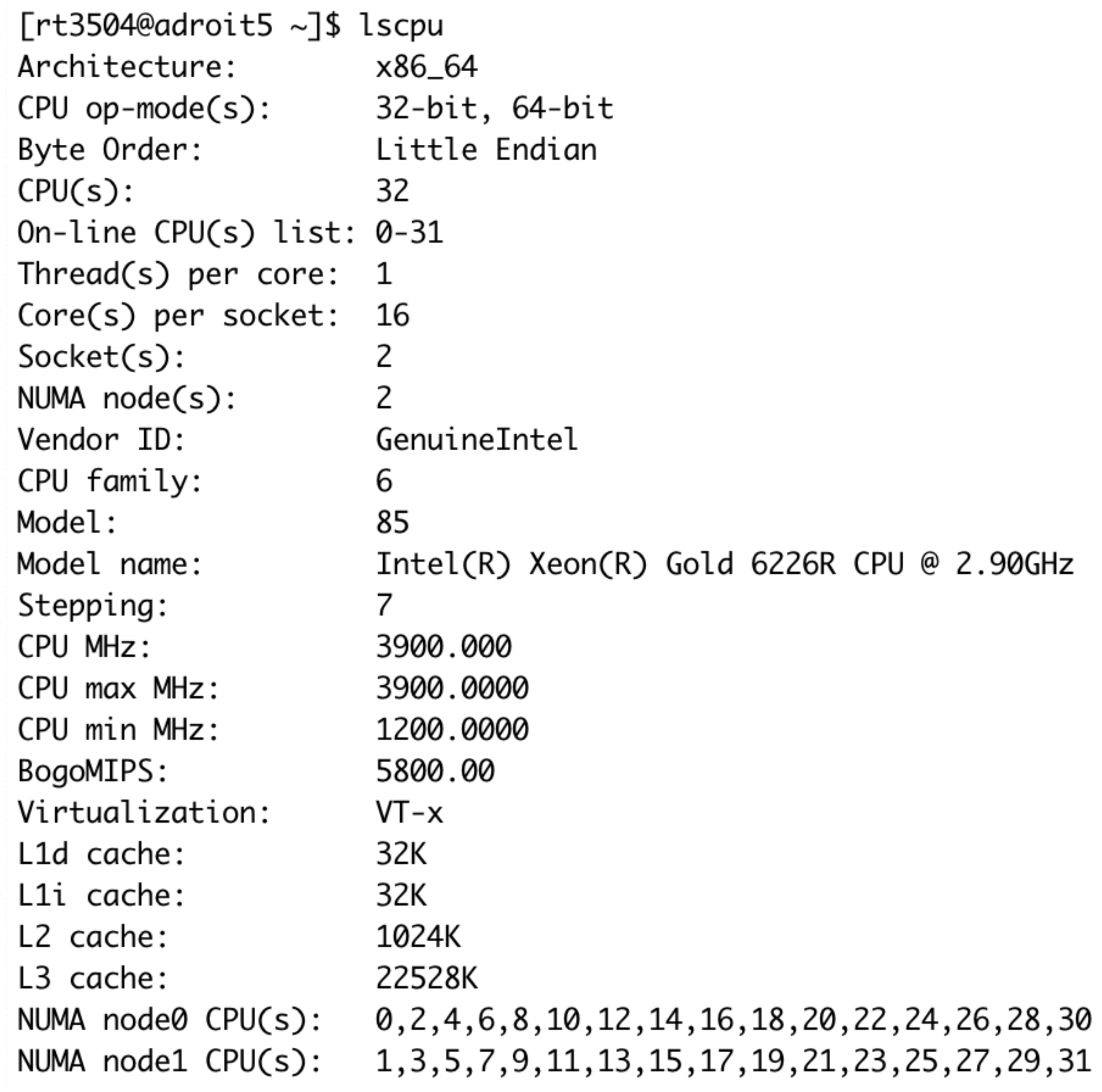
Koomey's Law will allow HPC centers to double performance every 18 months, at fixed power usage, by adding more cores.

However, running code on millions of cores presents new challenges.

- Algorithms must scale nearly perfectly.
- Mean time between failure of nodes becomes hours. Codes must be able to adapt to failures.

Understanding CPU Model and Architecture

- CPU is the heart of the machine – it reads in, decodes, and executes machine instructions, working on memory and peripherals.
- CPUs and other resources are managed by the Linux kernel.
- In Linux you can use `lscpu` or `cat /proc/cpuinfo` to get the CPU architecture details.



```
[rt3504@adroit5 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    1
Core(s) per socket:    16
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz
Stepping:              7
CPU MHz:               3900.000
CPU max MHz:           3900.0000
CPU min MHz:           1200.0000
BogoMIPS:              5800.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              22528K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
```

CPU, sockets, cores and threads

- A core is a hardware term that describes the number of independent central processing units in a single computing component (die or chip).
- A socket is the interface between the CPU and the motherboard. Adroit has two sockets and 16 cores in each socket so there is 32 physical cores available on this server. The work on the node is called MIMD as Multiple Instructions Multiple Data.
- A thread is a software term for the basic ordered sequence of instructions that can be processed by a single core. A core with two hardware threads can execute instructions on behalf of two different software threads without the overhead of switching between them. On adroit there is only 1 thread per core.
- The number of logical cores, which equals “Thread(s) per core” × “Core(s) per socket” × “Socket(s)” i.e. 1x16x2=32. Adroit has a total of 32 logical cores. You can check this also using: `> nproc --all`
- Based on CPU MHz and CPU max MHz, the clock rate is 3.9 GHz.

Understanding cache memory

CPU cache is important for the efficiency of your applications. There are different levels of cache memory and they are used to keep data close to the CPU.

- L1 Cache
    - The L1 cache has the smallest amount of memory (often between 1K and 64K) and is directly accessible by the CPU in a single clock cycle, which makes it the fastest as well.
    - It stores the most frequently used data that remain in L1 until some other data becomes more frequent. If so, it is moved to the bigger L2.

Understanding cache memory

CPU cache is important for the efficiency of your applications. There are different levels of cache memory and they are used to keep data close to the CPU.

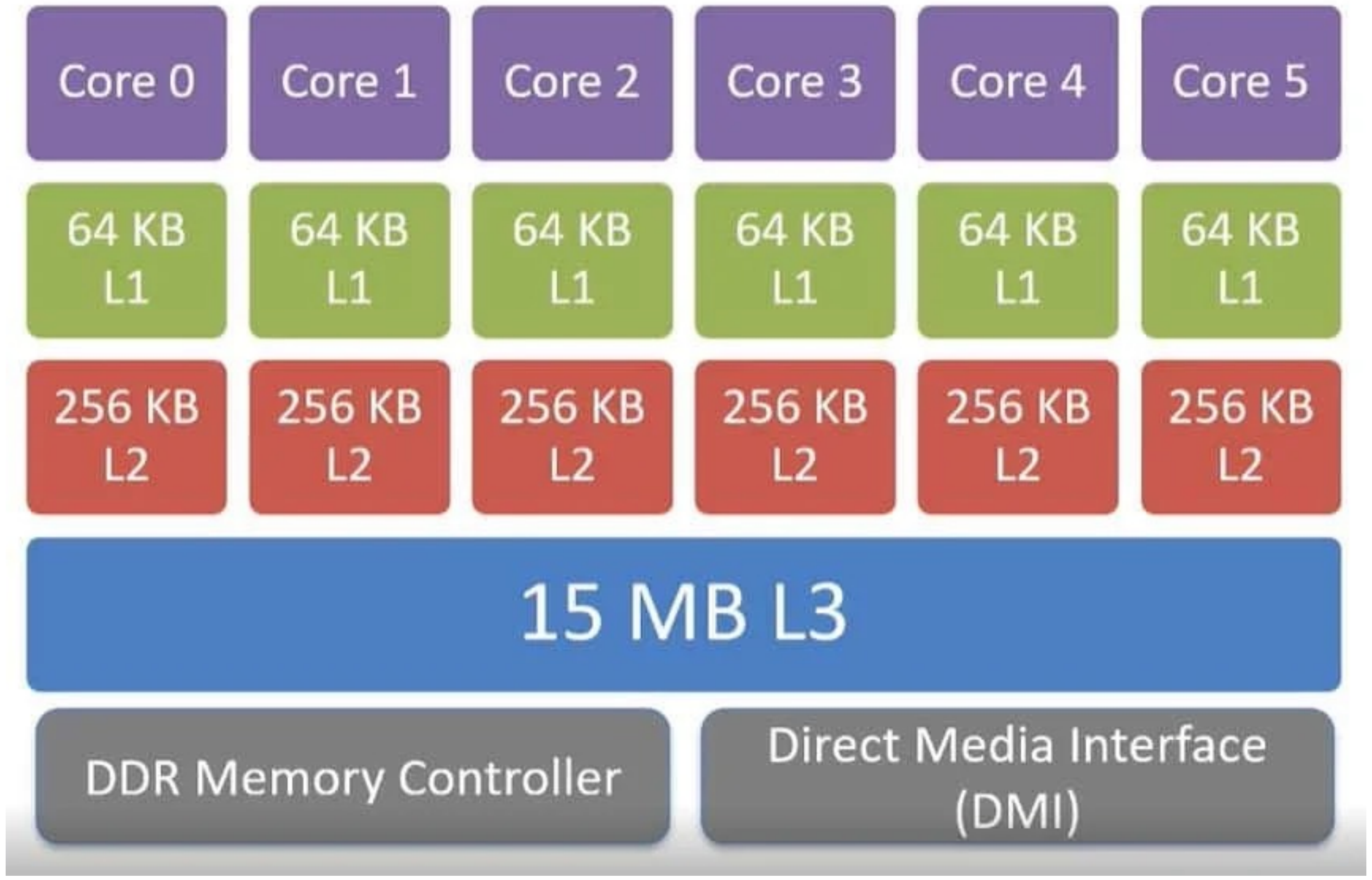
- L2 Cache
    - The L2 cache is the middle level, with a larger amount of memory (up to several megabytes) adjacent to the processor, which can be accessed in a small number of clock cycles.
    - Less frequently used data are moved from L2 to L3.

Understanding cache memory

CPU cache is important for the efficiency of your applications. There are different levels of cache memory and they are used to keep data close to the CPU.

- L3 Cache
    - The L3 cache, even slower than L1 and L2, may be twice as fast as the main memory (RAM).
    - Each core may have its own L1 and L2 cache; but they all share the L3 cache.
    - Size and speed are the main criteria that change between each cache level: L1 < L2 < L3.
    - Whereas original memory access may be 100 ns, for example, the L1 cache access can be 0.5 ns.

Understanding cache memory



Example of cache memory architecture for a 6 core socket

Understanding cache memory

Cache is organized in lines, and each cache line can be used to store a specific line in memory.

Each CPU has its separate cache and its own cache controller.

If a processor references main memory, it first checks cache for the data. If it's there, then it's referred to as a **cache hit**. If it's not there, then you've got a **cache miss**.

A new cache line fill is triggered by a cache miss. It means that data is loaded from main memory, which is slow.

So, a cache miss involves extra activity which translates into poor performance.

Controlling data movement

With high-level programming languages, you can partially control data movement.

- Cache to Register: no control but usually fast
- RAM to cache: indirect control (cache-aware algorithms, first touch strategy)
- Disk to RAM: complete control

Read / Write operations require explicit statement in the code. Some variants are faster than other (direct RAM memory copy versus write function).

Some algorithms are better at exploiting the cache hierarchical structure (cache-aware or cache-friendly algorithms that minimize cache misses). Try to exploit maximally data already in cache (data re-use for more cache hits).

For parallel execution, follow the default hardware strategy based on first touch assignment of the data.

Why understand the hardware

Surprisingly common complaint

My code runs fine on a small problem, but when I try it on a bigger problem, it slows to a crawl. What is going on?

- Most likely, the larger problem is too big for the memory of the machine
  - Data structures too big to fit in the cache (will run much slower)
  - Data structures too big for the main memory (will run much, much slower)

Why understand the hardware

Surprisingly common complaint

My code runs fast on large problems, but is 3-4x slower on small problems. What is going on?

- Most likely, vectorization is inefficient on small problems
  - Loop limits smaller than vector length of registers
  - Might be able to redesign data to make vectorization more efficient
- But what is this vectorization anyways?

Computer Architecture

If you plan to write efficient code on modern parallel processors, you have to understand how those processors work[1].

Virtually all processors use *register-to-register* architecture.

Data processed by CPU enters and leaves via registers (usually 80 or more bits in size)

```
C = A + B    becomes    Load R1, A
                        Load R2, B
                        Add R3, R1, R2
                        Store R3, C
```

Each operation requires at least one clock period, so in this example we expect the execution time to be at least four clock periods.

[1] See Hennessy & Patterson *Computer Architecture*, 5th ed., Appendix G to see how floating point arithmetic *really* works.

Pipelining: most important hardware optimization

Same sequence of instructions are often repeated many times (for example, work inside for/do loops).

Can optimize by designing processor so that different steps in sequence can execute at the same time, like a pipeline.

Clock cycle:	1	2	3	4	5	6	7	8	9
instruction i	IF	ID	ME	EX	WB				
instruction i+1		IF	ID	ME	EX	WB			
instruction i+2			IF	ID	ME	EX	WB		
instruction i+3				IF	ID	ME	EX	WB	
instruction i+4					IF	ID	ME	EX	WB

IF=instruction fetch; ID=instruction decode; EX=execute; ME=memory reference; WB=write back.

Pipeline in example takes 9 clock cycles to complete 5 instructions.
Once pipeline is full, a result is produced every clock cycle.
Un-pipelined processor would take 5*5 = 25 cycles
Pipelining is an example of *instruction level parallelism* (ILP)

Vector/SIMD Processors

Vectorization is an example of *data parallelism*. Identical instructions are performed on multiple datum simultaneously. Schematic 2-element vector pipeline:

Clock cycle:	1	2	3	4	5	6	7
element i	LD	AD	ML	WB			
element i+1	LD	AD	ML	WB			
element i+2		LD	AD	ML	WB		
element i+3		LD	AD	ML	WB		
element i+4			LD	AD	ML	WB	
element i+5			LD	AD	ML	WB	
element i+6				LD	AD	ML	WB
element i+7				LD	AD	ML	WB

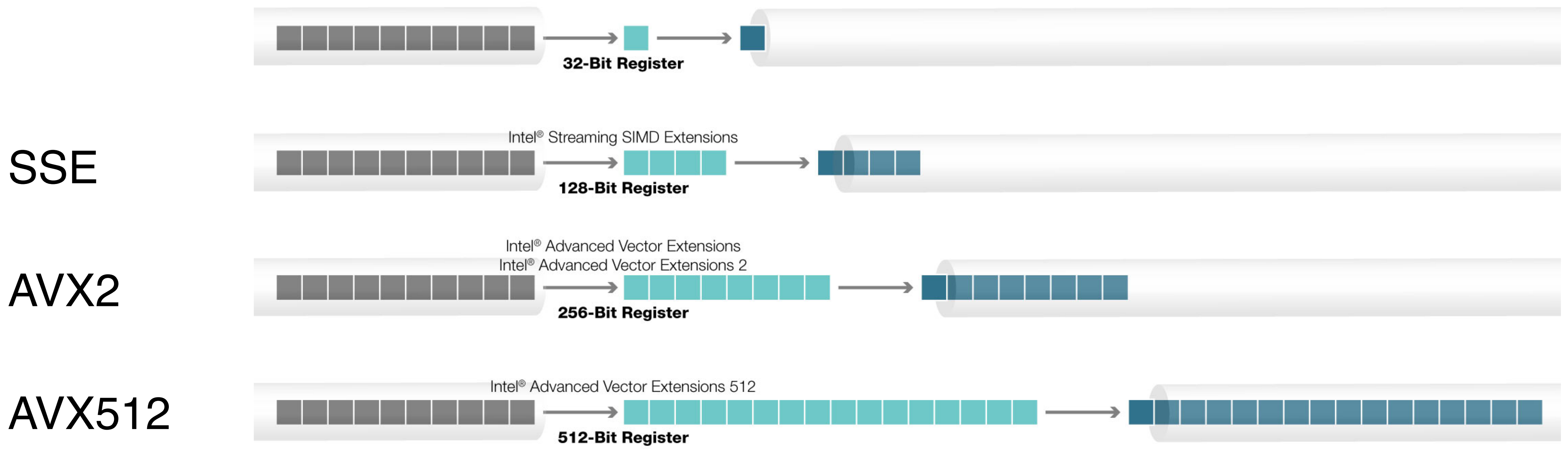
LD=load; AD=add; ML=mulitply; WB=write back

When pipeline is full, we get two results (4 flops thanks to *chaining*) every clock period! On latest processors, vector length is now 512 bits (16 single precision words).

Understanding vector operations

Intel AVX-512 is a set of CPU instructions that impacts compute, storage, and network functions. The number 512 refers to the width, in bits, of the register file, which sets the parameters for how much data a set of instructions can operate upon at a time. This is called SIMD as Single Instruction Multiple Data.

Intel AVX-512 doubles the width of the register compared to its predecessor, and it also doubles the number of registers to further decrease latency. It also contains additional optimizations to further accelerate tasks for modern workloads



32-Bit Register
SSE
Intel® Streaming SIMD Extensions
128-Bit Register
AVX2
Intel® Advanced Vector Extensions
Intel® Advanced Vector Extensions 2
256-Bit Register
AVX512
Intel® Advanced Vector Extensions 512
512-Bit Register

Role of compilers

Today, most code is written in one of a small number of languages. Hardware is now designed to optimize instructions produced by compilers of that language. Compilers can

- Integrate procedures into calling code (inline)
- Eliminate common sub-expressions (do algebra!)
- Eliminate unnecessary temporary variables (reduces loads/stores)
- Change order of instructions (e.g. move code outside loop)
- Pipeline
- Vectorize loops automatically
- Optimize register allocation

In short, compilers can do a lot!
So let the compiler do all the work.

Compilation Options for GNU Compiler

- Just type `> man gfortran` or `> man gcc`
- -O0: no optimization useful for debugging
- -O1: conservative optimization, does not increase code size
- -O2: normal optimization, vectorization and loop optimization
- -O3: aggressive optimization, takes longer and risky
- -fopenmp: activate OpenMP parallelization
- -g: activate extra information for debuggers
- -x f95-cpp-input: activate preprocessor directives

```
> gfortran -O3 -fopenmp -x f95-cpp-input -DNDIM=2 code.f90 -o code
```

Memory Design

Stores both data and instructions, organized into bits, bytes, words, and double words.

Order of bits in byte is standardized, but not order of bytes in word.

- **Little endian:** leading byte last (little end): 76543210 (Intel)
- **Big endian:** leading byte first (big end): 01234567 (IBM)

Most memory today is **Dynamic Random Access Memory** (DRAM) - bits stored in 2D array, accessed by rows and columns. Typical access time 100ns.

DRAM comes on **Dual Inline Memory Modules** (DIMMs).

Since 1998, memory on DIMMs doubles every 2 yrs. This is slower than Moore’s Law, and is leading to a growing memory/processor performance mismatch.

Reading data from RAM is SLOW!

DRAM must be constantly refreshed. During a refresh, the memory cannot be accessed. Typically 5% of reads have to wait for a refresh to finish.

Reading destroys data in DRAM, so it must be re-written after a read.

Both introduce latency.

Memory bus operates at slower clock speed than CPU. Typical bandwidth is 4 GB/s. That is about one byte per clock period.

Good performance requires maximizing flops per memory access (ideally should be bigger than one).

Hierarchical Memory

Exploits *principle of locality* to increase memory performance

- Most programs access adjacent memory locations sequentially, e.g. location M at time t; location $M+1$ at time $t+1$.
- Build memory closest to processor using fastest design (cache)
- Main memory built from slower (less expensive) DRAM
- Additional memory can be built from even cheaper disks

Usually cache is subdivided into several levels (L1, L2,[L3]). Data is transferred between levels in blocks: *cache line*

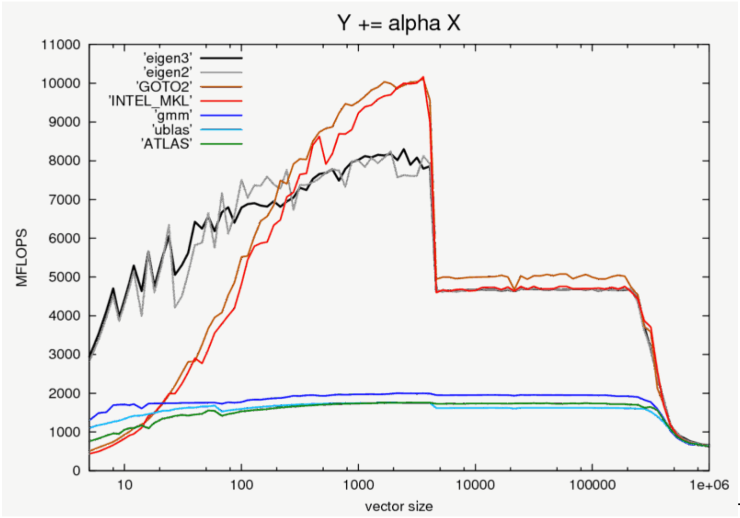
When item at address A is loaded into memory, an entire cache line containing A (and $A+1$, $A+2$, etc) is moved, not just A.

Then, if processor needs $A+1$ on next cycle, it is already in cache.

Cache hit (miss) occurs if data is (is not) in cache when needed. *Hit rate* is fraction of memory requests which are hits.

Goal of Programmer: write code to maximize hit rate

Example: cache efficiency



from http://eigen.tuxfamily.org/index.php?title=Benchmark

Access Time

Effective access time for hierarchical memory

$$t_{\mathrm{eff}} = Ht_{\mathrm{cache}} + (1 - H)t_{\mathrm{main}}$$

where t_{eff} = effective access time

t_{cache} = access time of cache

t_{main} = access time of main memory

H = hit rate

Suppose t_{cache} = 10ns, t_{main}=100ns, H=98%

$$t_{\mathrm{eff}} = (0.98)(10) + (1-0.98)(100) = 11.8\text{ns}$$

Almost as fast as cache!

Write code to maximize cache hits

Always access data contiguously: (1) by ordering loops so inner loop runs over neighboring data elements (2) use stride of one.

```
for (i=0; i<=100; i++) {
  for (j=0; j<=100; j++) {          /* BAD */
    a[j][i] = b[j][i]*c[j][i];
  }
}

for (i=0; i<=100; i++) {
  for (j=0; j<=100; j++) {           /* GOOD */
    a[i][j] = b[i][j]*c[i][j];
  }
}
```

Note: exactly the OPPOSITE ordering is necessary in Fortran.

If there are multiple ways to reference address of a variable (e.g. through pointers), it is said to be *aliased*. In this case, the variable cannot be allocated to a register. *This can greatly reduce performance.* Moral for programmer:

- Use pointer references carefully and sparingly!
- Avoid indirect addressing: a[i[j]]]

Array Storage in Memory

In C and C++, array elements are stored contiguously in memory using a row-major indexing scheme. Fortran uses a column-major indexing scheme.

```
m[0][0] m[0][1] m[0][2]…
```

```c
#include <stdio.h>
#include <time.h>
int m[9999][9999];

void main()

{
    int i, j;
    clock_t start, stop;
    double d = 0.0;

    start = clock();
    for (i = 0; i < 9999; i++)
        for (j = 0; j < 9999; j++)
            m[i][j] = m[i][j] + (m[i][j] * m[i][j]);

    stop = clock();
    d = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("The run-time of row major order is %lf\n", d);
    start = clock();
    for (j = 0; j < 9999; j++)
        for (i = 0; i < 9999; i++)
            m[i][j] = m[i][j] + (m[i][j] * m[i][j]);

    stop = clock();
    d = (double)(stop - start) / CLOCKS_PER_SEC;
    printf("The run-time of column major order is %lf", d);
}
```

```
❯ gcc -Og -o row row.c

~
❯ ./row
The run-time of row major order is 0.257015
The run-time of column major order is 0.690201
```

```
m(1,1) m(2,1) m(3,1)…
```

```fortran
program column

  integer::i,j
  real::tstart,tstop
  integer,dimension(1:9999,1:9999)::m

  call cpu_time(tstart)
  do i=1,9999
     do j=1,9999
        m(i,j)=m(i,j)+m(i,j)*m(i,j)
     end do
  end do
  call cpu_time(tstop)
  write(*,*)'The run-time of row major order is',tstop - tstart

  call cpu_time(tstart)
  do j=1,9999
     do i=1,9999
        m(i,j)=m(i,j)+m(i,j)*m(i,j)
     end do
  end do
  call cpu_time(tstop)
  write(*,*)'The run-time of column major order is',tstop - tstart

end program column
```

```
❯ gfortran -Og column.f90 -o col

~
❯ ./col
 The run-time of row major order is  0.916261017
 The run-time of column major order is  0.148454964
```

Other ways to improve performance

- Save state to reduce overall computation
- Cache frequently-used data structures
- interleave communication and computations for parallel code
- choose the best data design!
- choose the best algorithm!

Other Compiler Optimizations

- Inlining: replace a function call by its content in the calling routine.
- Loop unrolling:

```
do i=1,n
   a(i)=b(i)+d*c(i)
end do
```

$\longrightarrow$

```
do i=1,n-3,4
  a(i)=b(i)+d*c(i)
  a(i+1)=b(i+1)+d*c(i+1)
  a(i+2)=b(i+2)+d*c(i+2)
  a(i+3)=b(i+3)+d*c(i+3)
end do
do j = i,n
  a(j)=b(j)+d*c(j)
end do
```

- Divisions:

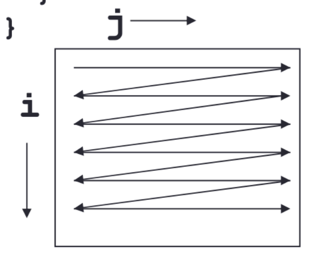
```
do i=1,n
  do j=1,m
    a(i,j)=d(j)/2
  end do
end do
```

$\longrightarrow$

```
tempdiv = 1/2
do i=1,n
  do j=1,m
    a(i,j)=d(j)*tempdiv
  end do
end do
```

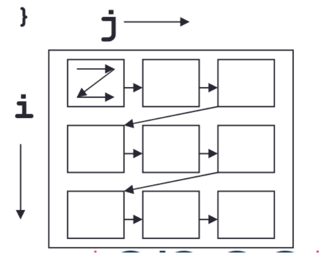
- Loop tiling:

```
for (i=0;i<n;i++){
  for (j=0;j<n;j++){
     a[i][j]+=b[i][j];
  }
}
```



$\longrightarrow$

```
for (ii=0;ii<n;ii+=B){
  for (jj=0;jj<n;jj+=B){
    for (i=ii;i<ii+B;i++){
      for (j=jj;j<jj+B;j++){
         a[i][j]+=b[i][j];
      }
    }
  }
}
```



- Local versus global variables: better cache management.

Data structures

- Why care about data structures?
  - Using a more efficient data structure is one of the best ways of speeding up a code
- Common data structures
  - Arrays
  - Linked lists
  - Hash tables
  - Binary trees
  - Heaps
- Won’t discuss more here, but always remember the importance of choosing the best data structure at the start
  - very hard to change design of data structures later

Use an efficient algorithm

- Most of the speed-up in scientific computing comes from algorithms, NOT from faster computers.
- Examples of algorithm developments
  - Linear algebra
    - iterative methods versus direct solvers
  - Elliptic solvers
    - Multigrid to accelerate Jacobi or Gauss-Seidel iteration
    - ADI, Conjugate gradient methods
    - FFT
  - *n*-body problems
    - tree codes
    - Fast multipole methods
  - etc., etc.

The importance of scaling

- Simple example: Column 8 of *Programming Pearls*
  - Pattern matching code, 4 different algorithms (n^3, n^2, $n \log n$, n)

Problem size n	$1.3n^3$	$10n^2$	$47n \log_2 n$	48n
10^3	1.3 secs	10 msecs	0.4 msecs	0.05 msecs
10^4	22 mins	1 sec	6 msecs	0.5 msecs
10^5	15 days	1.7 min	78 msecs	5 msecs
10^6	41 yrs	2.8 hrs	0.94 secs	48 msecs
10^7	41 millennia	1.7 wks	11 secs	0.48 secs

- Estimate your requirements
  - Use back-of-the-envelope calculations to estimate the time/memory required for your task. Do you need a better algorithm/data structure?

Profiling

- A *profile* measures where a program spends its time
  - Frequency counts of routines
  - Total time spent in each routine (and its children)

Note: the term *profile* was invented by Donald Knuth, in his paper "Empirical study of FORTRAN programs," 1971.

When to worry about performance?

Donald E. Knuth (author of TeX, and *The Art of Computer Programming*):
"There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."
—Donald Knuth, 1974.

And yet. . .

"Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, [s]he will be wise to look carefully at the critical code; but only *after* that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail."

Concentrate on the hot spots

- Say your program spends 90% of its time in one routine, and takes 100 seconds to run
  - Say you cut the run time of that one routine by a factor of 3. Overall code now takes 40 seconds to run (60% less time)
  - Instead, say you cut the run time of the whole rest of the program by a factor of 3. Code now takes 97 seconds to run (3% less time).

The bottom line

- Consider efficient *algorithms* and *data structures* from the beginning: these can make a big difference
    - An $O(n \log n)$ algorithm instead of an $O(n^2)$ algorithm; Jacobi iteration vs. multigrid, FFTs for a Poisson solver.
    - Binary trees, heaps, hash tables
- In general, don't worry too much about efficiency as you write.
- If efficiency is important, *profile* it once it's working, and find the "hot spots" (the critical 3%).
- Design good *interfaces* so that different algorithms may be swapped in.
- Don't sacrifice clarity for the sake of efficiency, especially early on.

Tools

The traditional unix tools are `prof` and `gprof`. Both work by instructing the compiler *and* linker to add extra information. A corollary of this need to compile and link specially is that libraries (*e.g. libc*) won't show up in the profile unless you use `gprof`-enabled versions. Another consequence is that they don't understand dynamically loaded libraries (*e.g.* `import afw`).

PC Sampling

Every so often the system interrupts your program and notes what it's doing; the register pointing to the next instruction to execute is called the *P*rogram *C*ounter, so this approach is often called "PC Sampling".

`gprof` allows access to these system counters.

Modern processors also contain dozens of hardware counters, special registers that record what the processor is doing at all times.

More advanced tools allow access to hardware counters, and therefore more information

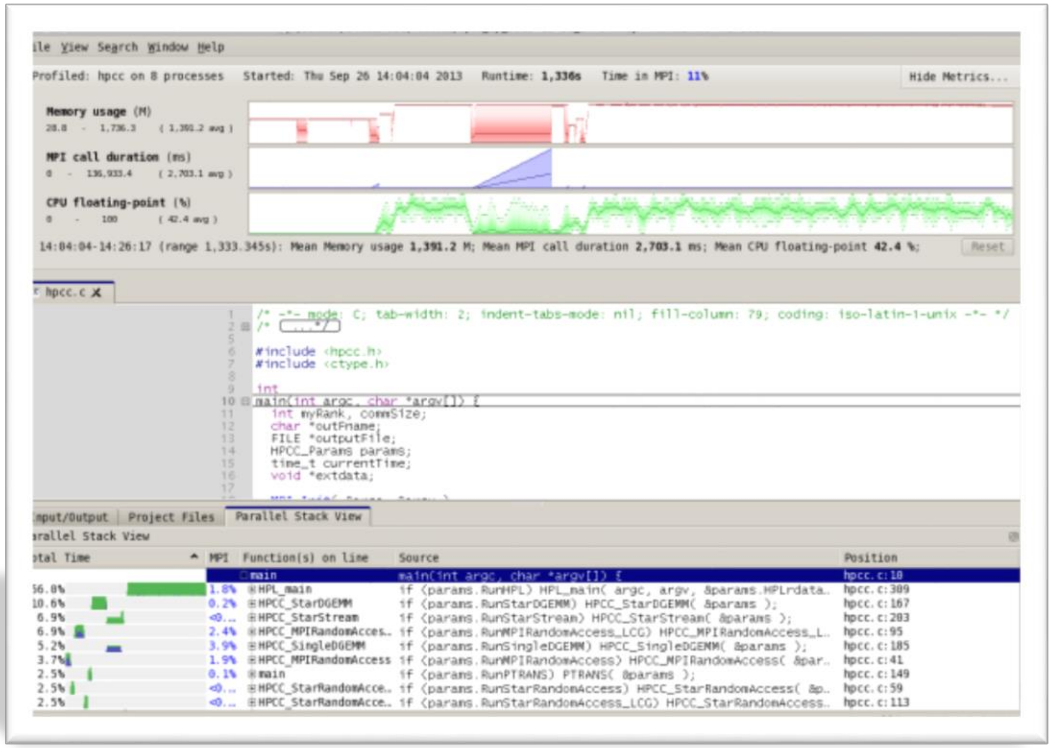
imageAccess *v.* gprof

We can compile and link with -pg and run gprof i

```
  %    cumulative   self          self     total
 time   seconds   seconds calls  us/call  ms/(100*call)  name
 34.47      0.22     0.22  1000  220.63   22.063  add1(Image&, int)
 26.64      0.39     0.17  1000  170.49   17.049  add2(Image&, int)
 17.24      0.50     0.11  1000  110.32   11.032  add3(Image&, int)
 12.54      0.58     0.08  1000   80.23    8.023  set(Image&, int)
  9.40      0.64     0.06  1000   60.17    6.017  Image::Image(int, int)
```

Allinea MAP

- Allinea MAP
  - Commercial profiler
  - C, C++, Fortran
  - Lightweight GUI



Profiled: hpcc on 8 processes   Started: Thu Sep 26 14:04:04 2013   Runtime: 1,336s   Time in MPI: 11%

14:04:04-14:26:17 (range 1,333.345s): Mean Memory usage 1,391.2 M; Mean MPI call duration 2,703.1 ms; Mean CPU floating-point 42.4 %;

```
/* -*- mode: C; tab-width: 2; indent-tabs-mode: nil; fill-column: 79; coding: iso-latin-1-unix -*- */
/* ...*/

#include <hpcc.h>
#include <ctype.h>

int
main(int argc, char *argv[]) {
  int myRank, commSize;
  char *outFname;
  FILE *outputFile;
  HPCC_Params params;
  time_t currentTime;
  void *extdata;
```

rallel Stack View

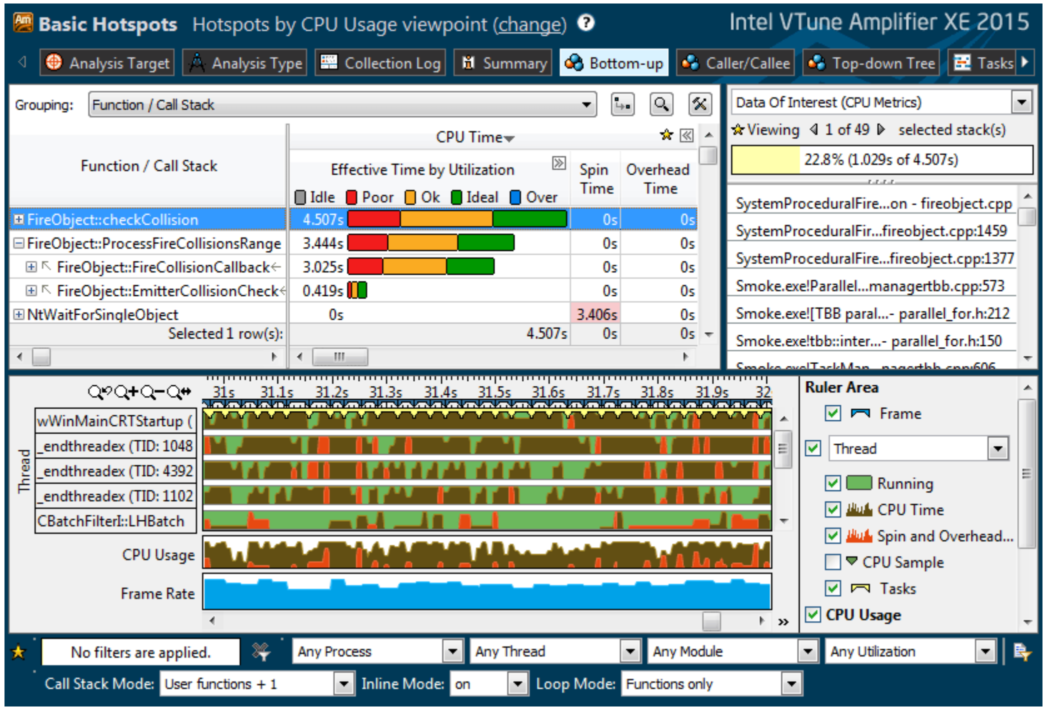
otal Time	MPI	Function(s) on line	Source	Position
		main	main(int argc, char *argv[]) {	hpcc.c:10
56.0%	1.8%	HPL_main	if (params.RunHPL) HPL_main(argc, argv, &params.HPLrdata..	hpcc.c:309
10.6%	0.2%	HPCC_StarDGEMM	if (params.RunStarDGEMM) HPCC_StarDGEMM(&params);	hpcc.c:167
6.9%	<0...	HPCC_StarStream	if (params.RunStarStream) HPCC_StarStream(&params);	hpcc.c:203
6.9%	2.4%	HPCC_MPIRandomAcces..	if (params.RunMPIRandomAccess_LCG) HPCC_MPIRandomAccess_L..	hpcc.c:95
5.2%	3.9%	HPCC_SingleDGEMM	if (params.RunSingleDGEMM) HPCC_SingleDGEMM(&params);	hpcc.c:185
3.7%	1.9%	HPCC_MPIRandomAccess	if (params.RunMPIRandomAccess) HPCC_MPIRandomAccess(&par..	hpcc.c:41
2.5%	0.1%	main	if (params.RunPTRANS) PTRANS(&params);	hpcc.c:149
2.5%	<0...	HPCC_StarRandomAcce..	if (params.RunStarRandomAccess) HPCC_StarRandomAccess(&p..	hpcc.c:59
2.5%	<0...	HPCC_StarRandomAcce..	if (params.RunStarRandomAccess_LCG) HPCC_StarRandomAccess..	hpcc.c:113

- Source code profiling
- Compute, I/O, Memory, MPI bottlenecks
- Getting started:

  https://researchcomputing.princeton.edu/support/knowledge-base/map

Intel VTune

- Intel VTune Amplifer XE
  - Commercial Profiler
  - Extraordinarily powerful (and complicated)
  - Nice GUI



Basic Hotspots Hotspots by CPU Usage viewpoint (change) — Intel VTune Amplifier XE 2015

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization	Spin Time	Overhead Time
FireObject::checkCollision	4.507s	0s	0s
FireObject::ProcessFireCollisionsRange	3.444s	0s	0s
FireObject::FireCollisionCallback	3.025s	0s	0s
FireObject::EmitterCollisionCheck	0.419s	0s	0s
NtWaitForSingleObject	0s	3.406s	0s
Selected 1 row(s):	4.507s	0s	0s

Data Of Interest (CPU Metrics)
Viewing 1 of 49 selected stack(s)
22.8% (1.029s of 4.507s)

SystemProceduralFire...on - fireobject.cpp
SystemProceduralFir...fireobject.cpp:1459
SystemProceduralFire...fireobject.cpp:1377
Smoke.exe!Parallel...managertbb.cpp:573
Smoke.exe![TBB paral...- parallel_for.h:212
Smoke.exe!tbb::inter...- parallel_for.h:150

No filters are applied. Any Process Any Thread Any Module Any Utilization
Call Stack Mode: User functions + 1 Inline Mode: on Loop Mode: Functions only

- Source code profiling
- Shared memory only
  - Serial
  - OpenMP
  - MPI on single node

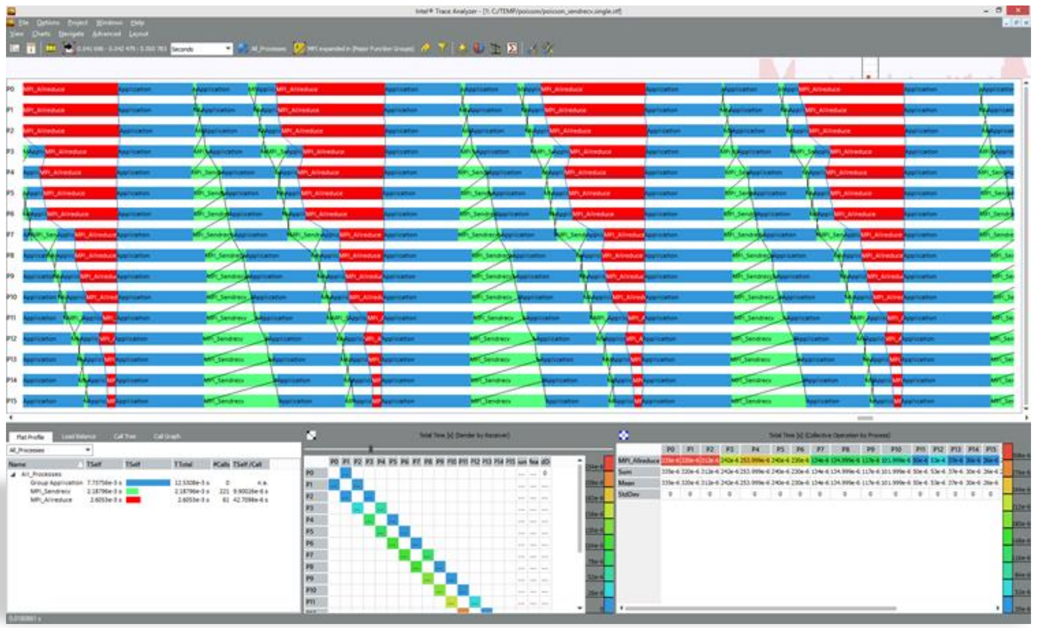
- Getting started:

  https://researchcomputing.princeton.edu/faq/profiling-with-intel-vtun

Intel Trace Analyzer and Collector

- Intel Trace Analyzer and Collector (ITAC)
    - Creates timeline for every process
- Good for MPI scaling & bottlenecks
- Can have large overhead & big files
- Getting started:

https://researchcomputing.princeton.edu/faq/using-intel-trace-analyze



OFFICE of INFORMATION TECHNOLOGY

PICSciE

Lessons Learned

- Don’t optimize prematurely.
- Whenever possible, iterate over arrays in their natural order.
- Use a sampling profiler to get an accurate but imprecise picture of overall performances (including cache and disk I/O) to identify hotspots.

Learn more about programming

- Take APC 524 to learn more about software engineering.
- Take a PICSiE mini-course

  https://researchcomputing.princeton.edu/learn/workshops-live-training

  - Intro to Linux Command Line
  - Intro to Programming using Python
  - Getting Started With the Research Computing Clusters
  - Reproducible HPC Research with Containers: Docker & Singularity
  - Introduction to Version Control using Git
  - Leveraging the NVIDIA A100 GPU AI and HPC

PICSciE is the Princeton Institute for Computational Science and Engineering. Research Computing is collaboration between PICSciE and the Office of Information Technology.