Optimising Serial Code

Advancing Science with Pawsey

Learning Objectives

- Choose an algorithm for good performance
- Choose a language for good performance
- Understand the importance of standard conformance
- Write code that can be optimised for a modern CPU
- Locate bottlenecks in a serial code and address them

Course Roadmap

Algorithm

- correctness/novelty
- complexity
- memory scaling
- etc

Programming

- language selection
- code efficiency
- standardization
- etc

Profiling

- cache performance
- FLOPS
- IPC
- etc

Optimization

- data locality
- loop transformation
- vectorization
- etc

Parallelization

Performance

PAWSEY supercomputing centre

Motivation

- Science ***will*** be limited by the code and computer.
- Development effort may be secondary to code performance.
- Many gains come with small effort.
- Code optimisation ***can*** force good programming styles.

Is it worth the time?

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK: 50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

From XKCD webcomic: https://xkcd.com/1205/

Development Effort

1 Magnus day = 49 years on a dual-core desktop.

$\Rightarrow$ It is worth it to spend months of development time to save years/decades of runtime!

$\Rightarrow$ This might go against common opinions.

You still need to consider unit testing, ease of collaboration and code extension.

PAWSEY supercomputing centre

Know When to Stop Developing

Focus your effort!

1. Start with a good algorithm.
2. Write code with performance in mind.
3. Profile the code to determine where to spend your optimising effort.
4. Optimise the code, then profile again, and repeat until satisfactory performance achieved.

Know when to stop trying! Set an effort limit or a runtime goal.

MEASURING PERFORMANCE

PAWSEY
supercomputing centre

Busy vs Effective

- Low processor utilisation ***does*** mean sub-optimal performance.
- High processor utilisation ***does not*** mean optimal performance.
- Why?
  - processor could be doing something inefficient
  - processor could be polling

PAWSEY supercomputing centre

Time

- What matters is real (your) time. Use the clock, we call this ***walltime***.
- You *can* use routines like **gettimeofday**, **MPI_Wtime** to measure program runtime.
- Don't use many timing calls to profile the code. This is slow and alters the profile. Instead use a profiler, which we will do later!
- CPU clock speed can vary between runs!

PAWSEY
supercomputing centre

Timing routines

- C / C++

  clock_t **clock**(void): Returns CPU time in “clock ticks” since initial call. (Use CLOCKS_PER_SEC to convert.). Resolution is microseconds.

  time_t **time**(time_t *second): Returns real time in seconds since unix epoch 00:00:00 Jan 1, 1970. (Output argument “second” is also assigned with that value if it is not a NULL pointer.). Resolution is in seconds.

- Fortran

  CPU_TIME(TIME): Returns the processor time in seconds since the start of the program through “TIME”, which is a REAL with INTENT(OUT). Resolution in miliseconds.

  DATE_AND_TIME([date,time,zone,values]): Returns character and binary data on the real-time clock and date. Resolution is miliseconds.

- glibc

  int **gettimeofday**(struct timeval *tp, struct timezone *tzp): Returns the current calendar time as the elapsed time since the epoch in the structure “tp”. Resolution is microseconds.

- MPI

  double **MPI_Wtime**(void): Returns time in seconds since an arbitrary time in the past. Resolution is indicated by: double **MPI_Wtick**(void).

CHOOSING AN ALGORITHM

Algorithms

Many common algorithms have:

- a period of time
- increments within that period
- elements of interest
- ways to visit these elements
- a calculation <do something>

Algorithm A
 	
Grid
	For t = t1 .. tn {over time increments}
	    For i = x_1 .. x_n
	        For j = y_1 .. y_n
	            For k = z_1 .. z_n
	                <Do something>

Algorithm B
 	
N-body
	For t = t1 .. tn {over time increments}
	    For each element
	        For each other element
	            <Do something>

PAWSEY supercomputing centre

Complexity

At this stage, we can make a few observations:

Algorithm A will do $\#t.\#x.\#y.\#z$ calculations = $O(t*n^3)$

Algorithm B will do $\#t.n.(n-1)$ calculations = $O(t*n^2)$

… this is the **order** or algorithmic **complexity**.

Scaling

- Consider memory scaling as well as compute scaling. **Memory is finite**.

	Compute Time	Memory
3D-grid with time	$O(t*n^3)$	$O(n^3)$
N-body	$O(t*n^2)$	$O(n)$

- Other examples: QR tridiagonal eigensolver is $O(n^2)$ in memory, while MR^3 tridiagonal eigensolver is $O(n)$ in memory.

Example Alternative Algorithm

- Instead of calculating every pairwise interaction (brute force) in the N-body problem, represent hierarchies of bodies with multipoles in a spacial decomposition (influence of very distant bodies is negligible).
- $O(N^2)$ interactions becomes O(N log N), even O(N) for adaptive expansion.

N	Brute force $O(N^2)$	Multipole model O(N log N)
1,000	1,000,000	7,000

PAWSEY supercomputing centre

Choosing an Algorithm

There may be multiple algorithms to solve a problem. Consider:

- E.g. O(n^3) vs O($n \log n$) or Brute Force vs Clever approach with adequate results.
- Parallelisability constraints imposed by the algorithm.
- Domain decomposition often leads to good scaling and parallelisability. Load balancing may become an issue.

PAWSEY supercomputing centre

Prefactor

- If two algorithms have the same prefactor, does one have fewer expensive operations? E.g. square roots, exponentials, complex numbers, sin, cos etc.
- Algorithms that scale well might have a more expensive prefactor and be slower for small *n*. Perhaps use two algorithms and set a transition point based on *n*.

Rewrite your Maths

Minimise the number of array accesses and floating point operations, don't just go for elegant maths or copy a journal article verbatim. E.g.

- $\sum_{ij} X_{ij} Y_{ij}$ instead of $tr(X^T Y)$.
- c$\sum_i A_i$ instead of $\sum_i cA_i$.
- For symmetric matrices, you ***might*** gain from working with upper/lower triangles.
- Don't transpose a symmetric matrix.

Simple Tricks

- x^n – for small n make sure *n* is an integer. Then the compiler can change this to $x * x$.
- *sqrt(x)* is generally faster than $x^{0.5}$.
- test $x^2 + y^2 < r^2$ instead of $\sqrt{x^2 + y^2} < r$.
- $\sum_{i \neq j} \frac{M_i M_j}{|r_{ij}|}$ if problem is symmetric then don't double the effort.
- Bring constants outside of loops, such as $\frac{1}{4\,\pi\varepsilon_0}$. The compiler *should* do this anyway.

PAWSEY supercomputing centre

CHOOSING A LANGUAGE

Choosing a language

Choose the right language for the job.

- Performance.
- Potential to use OpenMP, OpenACC, CUDA, and/or MPI.
- Flexible to new technologies.
- Portable.
- Available performance and debugging tools.
- Ease of programming.

Perhaps a hybrid approach, e.g. python/C or python/Fortran

Other Language Considerations

Our Goals:

- Want to convert algorithms into code (without too much effort).
- Want to help the compiler produce the fastest code (without too much effort).

Some Considerations:

  - Array / matrix support
  - Complex number support
  - Aliasing
  - Static typing

Aliasing

Aliasing is when some variables *might* refer to the same memory location. This introduces constraints on compiler optimisations. (e.g. prevents code reordering).

- Fortran

  Assumes no aliasing. Avoid unnecessary use of pointers (stick to ***allocatable***).

- C

  Use the ***restrict*** keyword, since C99 standard.

- C++

  __***restrict***__ etc are non-standard but in some compilers. Not portable.

Avoid unnecessary use of pointers.

Static Typing

- With ***static typing***, the type is known at compile-time.
- Compiler can check that variables passed to functions are compatible with the functions.

- Permits optimisations at compile-time.
- Improves reproducibility, reduces errors.
- API is well documented for others.
- Fortran, C, C++ all support static typing.

Language performance

	Fortran	Julia	Python	R	Matlab	Octave	Mathematica	JavaScript	Go	SciLua	Java
	gcc 7.3.1	1.0.0	3.6.3	3.5.0	R2018a	4.2.2	11.3	V8 6.2.414	go1.9	1.0.0-b12	1.8.0_17
fibonacci	0.98	1.32	94.2	263	17.6	10045	132.1	3.51	1.80	1.18	3.63
parse_integer	6.87	2.19	19.76	50.37	178.2	574.4	22.66	5.03	0.96	0.97	3.17
quicksort	1.18	0.99	37.57	57.93	2.36	2221.3	44.48	4.28	1.24	1.56	2.98
mandelbrot	0.70	0.68	65.66	195.5	9.84	5812	18.29	1.134	0.77	1.00	1.42
pi_sum	1.00	1.01	14.77	11.69	1.01	317.5	1.45	1.08	1.00	0.99	1.08
matrix_statistics	1.54	1.63	17.73	20.97	8.09	46.24	7.49	13.97	6.08	1.70	5.02
matrix_multiply	1.15	0.97	1.18	8.26	1.16	1.21	1.18	31.77	1.43	1.08	8.07

Comparison from https://julialang.org/benchmarks/.
Times relative to C (gcc 7.3.1). Lower is better.
Ran on a single core (serial execution) on an Intel® Core™ i7-3960X 3.30GHz CPU with 64GB of 1600MHz DDR3 RAM, running openSUSE LEAP 15.0 Linux.

These are not optimal results, just how a typical researcher might program in those languages.

PAWSEY supercomputing centre

Fortran Compiler Comparison

	GCC	Intel	Cray
fibonacci	1	4.41	3.16
parse_integer	1	0.77	0.77
mandelbrot	1	0.50	-
quicksort	1	0.91	0.99
pi_sum	1	0.55	0.55
matrix_statistics	1	0.80	0.47
matrix_multiply	1	0.75	0.15

Comparison code perf.f90 from https://julialang.org/benchmarks/.

Runtimes normalised to the GNU runtimes. Lower is better.

Results on Magnus, all using “ftn –O3 perf.f90” under relevant compiler environments.

STANDARDS CONFORMANCE

Standard Conformance

- Your code will run on different systems and different compilers over the years.
- Standards conformance significantly improves the chance of **reproducibility**. (good for science!)
- Reproducibility means “apples vs apples” performance comparisons between systems and compilers, and **between profiling runs**.

- Do not rely on compiler extensions!

How to write portable code (1)

- Use your compiler to check.
  - gcc -std=c99 -pedantic myfile.c
  - gcc -std=c++98 -pedantic myfile.cxx
  - gfortran -std=f95 -pedantic myfile.f90
- Develop with multiple compilers. Cray compilers are very pedantic.
- Download a copy of the standard (or a draft).

How to write portable code (2)

- Try initialising variables to other values, do you get the same answer?.
  - gfortran -finit-real=inf -finit-logical=false -finit-character=t …
- Try runtime checking. This is slower than compile-time checking.
- gfortran -fcheck=all

Portable Makefiles

- Don’t hard code compiler-specific flags.
  - There is no standard for compiler flags, just for the code itself.
  - -Wall is not portable among compilers

- Use Makefile variables, make them easy to find and modify:

```
CFLAGS=-g -Wall
#CFLAGS=-O3
```

http://www.gnu.org/software/make/manual/

Exercise 1

Exercises are publicly available via Github:

git clone https://github.com/PawseySC/Optimising-Serial-Code.git

It's case sensitive. Download them!

Have a look to the various "`Makefile`"s.

Exercise 2: matmul (1)

In the exercise directory:

```
cd matrix && make matrix
```

This produces the executables `matrix.O0`, `matrix.O1`, `matrix.O2`, `matrix.O3` (Have a quick look at the `Makefile`).

Run them through the queue:

```
sbatch run_matrices.slurm
```

Output is in the SLURM output file `slurm-JOBID.out`

Exercise 2: matmul (2)

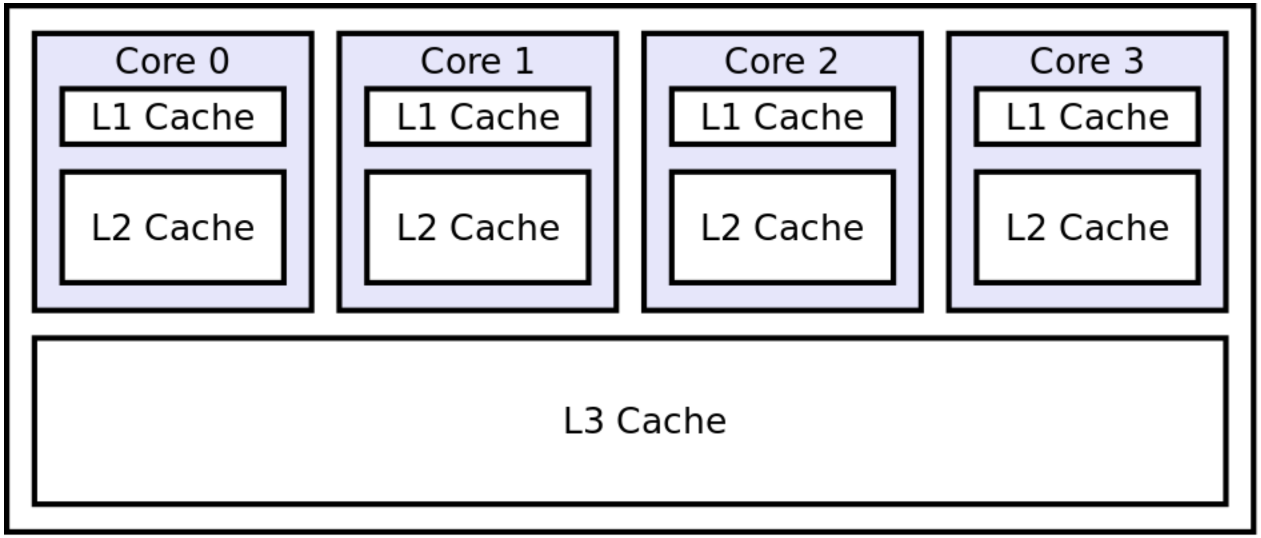
Compare the timings:

- What effect does optimisation level have on calls to the external math routine dgemm, to the intrinsic matmul, to manual looping of matrix multiplication?
- Is there a single best method for any matrix size at high optimisation?
- What is the main difference between matmul3 and matmul4.
- If you have time, try with a different compiler.

MODERN COMPUTERS AND OPTIMISATION

CPUs

- Most CPUs have multi-level caches, to reduce the time taken to access data.
- A CPU can do a lot of work in the time it takes to access main memory.



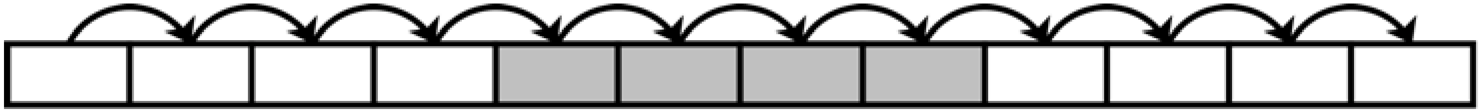
Data Locality

Location	Access time	Access time (cycles)
Register	<1ns	-
L1 cache	1ns	4
L2 cache	4ns	10
L3 cache	15-30ns	40-75
Memory	60ns	150
Solid state disk	50us	130,000
Hard disk	10ms	26,000,000
Tape	10sec	26,000,000,000

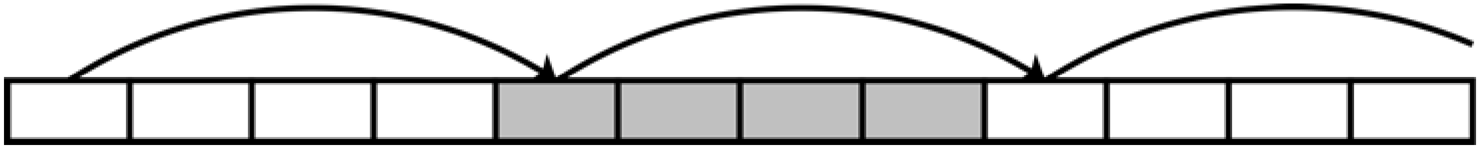
Source: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

Cache: Access Patterns

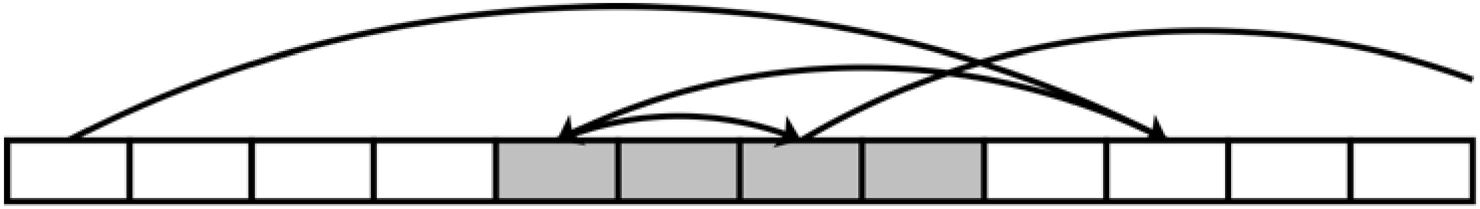
- Sequential access results in a higher rate of cache hits



- Striding access has a low rate of hits, however modern processors can detect striding access patterns and pre-fetch cache lines



- Random access is typically the worst, with a low rate of hits and no ability to predict subsequent access locations



PAWSEY supercomputing centre

Translation Lookaside Buffer

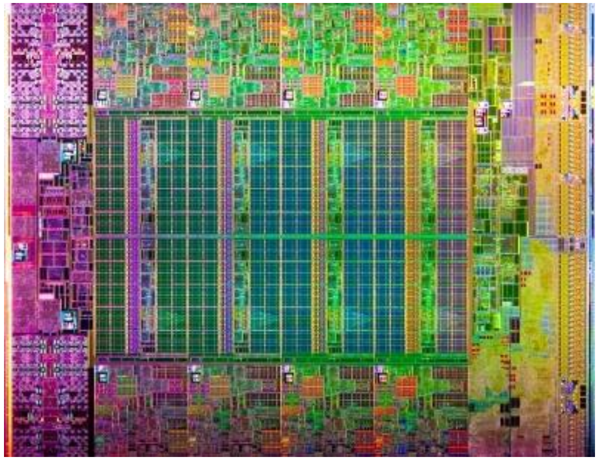
- The Translation Page Table maps the memory seen by the program (Virtual Memory) into physical memory. It sits in main memory and is slow.
- The Translation Lookaside Buffer is a cache of recently used **mappings (not actual content)**. Like the main caches, aim to work within VM pages. These are typically 4kB.

- On a Cray you can use larger pages.
- Before **compiling**: `module load craype-hugepages2M` (size **does not** matter)
- **Runtime**: `module load craype-hugepages`*XXX*`M` (size **does** matter)

PAWSEY supercomputing centre

Being cache-friendly

- Read and write to contiguous chunks of memory. Data is transferred in a *cache line*.
- Avoid *cache misses*.



PAWSEY supercomputing centre

Writing to Memory

- Don’t store data in RAM if you don’t need to. Use local temporary variables instead.
- These could be optimised into registers.
- In particular, don’t use global variables as local temporary variables.
- Similarly, avoid using array sections for temporary storage.

Instruction Pipelining

- Modern CPUs can complete multiple Instructions Per Cycle (IPC), also known as Instructions Per Clock.
- An average for Xeon is 4. You need to keep the pipeline full to achieve this.

Loop Unrolling

Keep data in registers/cache via loop unrolling / blocking with inlining. This can also improve IPC.

e.g. `a(n)=somefunc(n) + a(n-1)`

Unroll with stride 2:

```
do n=1,nmax,2
  a(n)=somefunc(n) + a(n-1)
  a(n+1)=somefunc(n+1) + a(n)
end do
```

Loop Counts

- Inner loops should have high iteration counts, since loops themselves have a non-negligible cost.
- Counter to this, very small inner loops may get unrolled away. In this case do it manually.

Good

```
do j=1,10
  do n=1,10000
    work
  end do
end do
```

Bad

```
do n=1,10000
  do j=1,10
    work
  end do
end do
```

Knowing Loop Counts

- There is more potential for compiler optimisation when the loop count is known before the loop is started.
- Use “do”, “for” loops. Avoid “while” and “do … while” loops.

Contiguous Memory

- Aim to work through contiguous chunks of memory. Avoid unnecessary striding.
- In **Fortran**, the consecutive elements of a column reside next to each other (**column-major order**).
- In **C/C++**, the consecutive elements of a row reside next to each other (**row-major order**)

Fortran

```
do j=1,10
  do i=1,10
    A(i,j)=something
```

C /C++

```
for (i=0;i<10;i++)
  for (j=0;j<10;j++)
    a[i][j]=something
```

PAWSEY supercomputing centre

Loop Blocking (1)

Loop blocking/tiling/strip-mining …

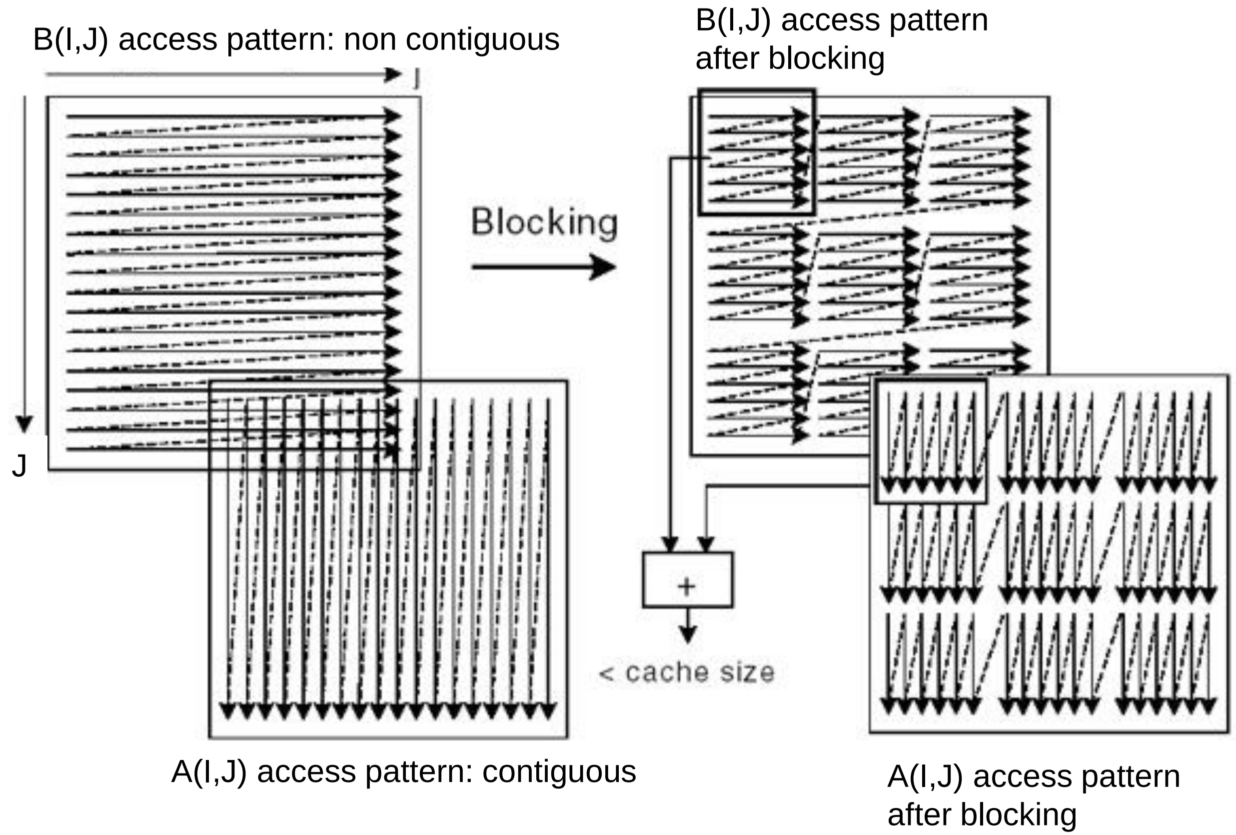
This code potentially contains many cache misses.

```
do J=1,Jmax
  do I=1,Jmax
    A(I,J) = A(I,J) + B(J,I)
  end do
end do
```

A has stride 1, **B** has stride **Jmax**.

- Make the stride of **B** smaller so that small blocks of **A** and **B** sit in cache.
- If **Imax** and **Jmax** are very large, neither A or B can sit in cache in whole.

Loop Blocking (2)



- The Cray Fortran compiler does this for you. Most other compilers currently do not.

Loop Blocking (3)

After applying loop blocking technique

```
do J=1,Jblksize,Jmax
    do I=1,Iblksize,Imax
        do JJ=J,J+Jblksize
            do II=I,I+Iblksize
                A(II,JJ) = A(II,JJ) + B(JJ,II)
            end do
        end do
    end do
end do
```

A has stride 1, **B** has stride **Jblksize**.

- Make the stride of **B** smaller so that small blocks of **A** and **B** sit in cache.
- Only **Iblksize** x **Jblksize** of A or B sit in cache. Not exhausting cache.

Branching

- Remove branches from loops and change the loop bounds. Branches are bad for IPC and pre-emptive cache fetching.
- Avoid GOTO statements (in Fortran, C, C++). They *can* affect cache/register use.

Before:

```
do n=1,nmax
  if (n==1) a(n)=0
  a(n)=somefunc(n)
  if (n==nmax) a(n)=1
end do
```

After:

```
a(1)=0
a(nmax)=1
do n=2,nmax-1
  a(n)=somefunc(n)
end do
```

Loop Fission

- Too much work in loops means that registers and/or instruction cache may get exhausted.
- Perhaps only part of a loop is vectorisable (execute a number of loop-iterations at the same time).
- Break these loops up.

Before:

```
do n=1,nmax
  lots of work
  some I/O
end do
```

After:

```
do n=1,nmax
  lots of work
end do
do n=1,nmax
  some I/O
end do
```

Loop Fusion

Before:

```
do n=1,nmax
  a(n)=somefunc(n)
end do
do n=1,nmax
  b(n)=someotherfunc(n)
end do
```

After:

```
do n=1,nmax
  a(n)=somefunc(n)
  b(n)=someotherfunc(n)
end do
```

- This is usually used in conjunction with other techniques.
- Whether this is beneficial depends on the work inside the loops. Too much work may exhaust registers or cache.

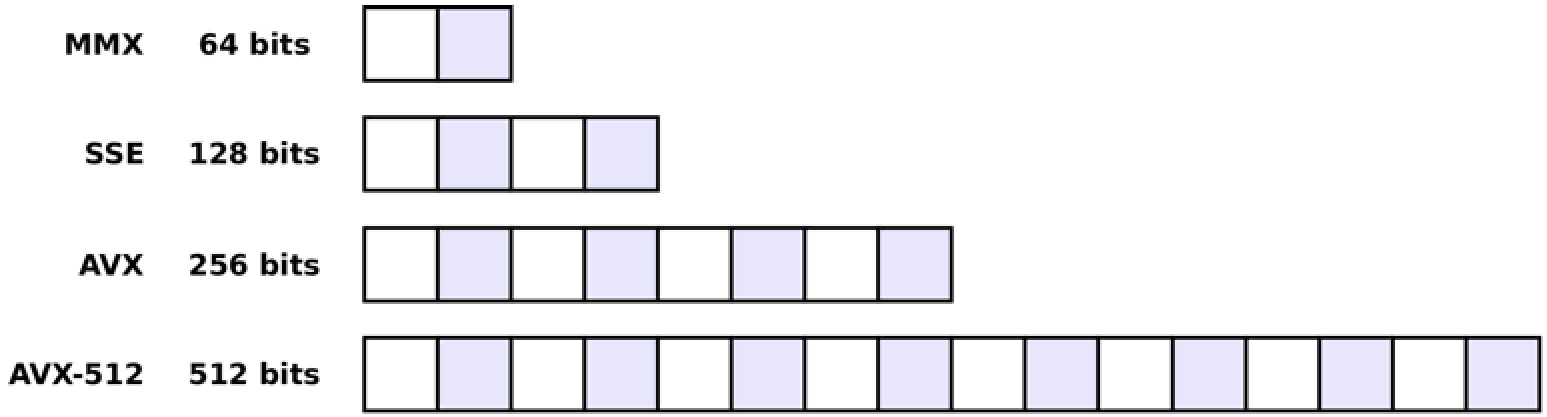
Pointers

Avoid unnecessary use of pointers.

- Pointers ***might*** prevent some compiler optimisations. They should be fine if they have local scope.
- Pointers make it difficult to copy data to GPU memory.
- Hard to optimise cache use, e.g. with linked lists vs contiguous arrays.

MMX / 3DNOW / SSE / AVX

- These are Single-Instruction-Multiple-Data (SIMD) operations.
- These are not part of standard Fortran / C / C++. They may get deprecated over time.
- Compiler should insert these automatically. Write SIMD-friendly code.



PAWSEY supercomputing centre

Inlining

- Function calls take time. You can remove this time by placing the code into the calling routine.
- Compilers ***can*** inline code for you when using high optimization levels. This is not guaranteed, but you can make it more likely:

- In C / C++, put the code in the same file as the calling routine. Use static functions.
- In C99 / C++, use the **`inline`** keyword.
- In Fortran, put the code in the same file / module as the calling routine. Use the compiler directive "*forceinline*".

PAWSEY
supercomputing centre

COMPILER OUTPUT

Cray Compiler Output

Cray compiler will output annotated version of source file

`ftn -rm mycode.f90`

Outputs *mycode*.lst

Examine annotated file to figure out what's going on

```
%%%    L o o p m a r k   L e g e n d    %%%

Primary Loop Type        Modifiers
------- ---- ----        ---------
A - Pattern matched      a - atomic memory operation
                         b - blocked
C - Collapsed            c - conditional and/or computed
D - Deleted
E - Cloned
F - Flat - No calls      f - fused
G - Accelerated          g - partitioned
I - Inlined              i - interchanged
M - Multithreaded        m - partitioned
                         n - non-blocking remote transfer
                         p - partial
R - Rerolling            r - unrolled
                         s - shortloop
V - Vectorized           w - unwound
```

PAWSEY supercomputing centre

Intel Compiler Output

- Optimisation reports.
- Compiler flag: -qopt-report=3

- Have a look at the man page for other values to opt-report.

Exercise 3: Cray compiler output

- Run:

```
cd compiler_reports
make matrix.cray
```

- Check out the manpage if needed
  - `man crayftn`
- Examine the output in `matrix.lst.O*`
- What has the compiler done with routine calls and loops?
- Can you identify the reason of the timing results from the previous exercise?

Exercise 4: Intel compiler output

- Run:

```
module swap PrgEnv-cray PrgEnv-intel
cd compiler_reports
make matrix.intel
```

- Seek help: `ifort -help reports`
- Examine the output in `matrix.optrpt.O*`
- Might be a bit too much information. Scale back the reporting options in `Makefile`

PROFILING

Profiling phases

- **Instrumentation:** compile the source code with extra compiler flags that enable the recording of performance-relevant events.
- **Measurement & analysis:** run the instrumented application on a **representative** test case. Usually the instrumented application is much slower than the original one.
- **Performance examination:** collect and analyse the measurement results.

PAWSEY supercomputing centre

Profilers

- On Cray supercomputers: Cray Tools

https://pubs.cray.com/content/S-2376/7.0.0/cray-performance-measurement-and-analysis-tools-user-guide/craypat

https://support.pawsey.org.au/documentation/display/US/Profiling+with+Cray+Tools

- Intel VTune: https://software.intel.com/en-us/intel-vtune-amplifier-xe

https://support.pawsey.org.au/documentation/display/US/Profiling+with+Intel+VTune

- Others: gprof, Arm MAP & Performance reports, etc.

https://support.pawsey.org.au/documentation/display/US/User+Training

https://support.pawsey.org.au/documentation/display/US/Training+Material

PAWSEY supercomputing centre

Profilers

- Profiling guide at Pawsey:

https://support.pawsey.org.au/documentation/display/US/Profiling

Profiling

Created by Maciej Cytowski on Sep 04, 2018

Pages in this section:

- Profiling Introduction
- Basic profiling tools
- Profiling with gprof
- Profiling with Arm MAP
- Profiling with Arm Performance Reports
- Profiling with Cray Tools
- Profiling with Intel VTune

Full Profiling with CrayPAT

Sampling experiment

Instrumentation

- module load `perftools`
- Compile code, using Cray compiler wrappers (`ftn, cc, CC`) & **preserving object (`.o`) files**
- `pat_build myapp`
    - Generates executable named `myapp+pat`

Measurement & analysis

- Run `./myapp+pat` as normal, this will generate an
- Output dir: `myapp+pat+XX+YYs/` (or `.xf` file for small runs)
- `pat_report myapp+pat+XX+YYs/ > myapp.sampling.report` (this also generates `.ap2` file that can be viewed with Apprentice2, and a `build-options.apa` file to be used in a tracing experiment)

Performance examination

- Read `myapp.sampling.report` file
- or use Apprentice2 (with X11 forwarding activated: "`ssh –X`"):
  `app2 myapp+pat+XX+YYs/ &`

PAWSEY supercomputing centre

Full Profiling with CrayPat

Tracing experiment

Instrumentation

- First, perform a sampling experiment to generate the file: `myapp+pat+XX+YYs/build-options.apa`
- `pat_build -O myapp+pat+XX+YYs/build-options.apa`
  - Essentially `pat_build -w -T funcs -g grps -u myapp` (can be edited to change `–T funcs` and `–g grps`)
  - Generates executable named `myapp+apa`

Measurement & analysis

- Run `./myyapp+apa` as normal, this will generate an
- Output dir: `myapp+apa+XX+YYt/` (or `.xf` file for small runs)
- `pat_report myapp+apa+XX+YYt/ > myapp.tracing.report` (also generates `.ap2` file that can be viewed with Apprentice2)

Performance examination

- Read `myapp.tracing.report` file
- or use Apprentice2 (with X11 forwarding activated: "`ssh –X`"): `app2 myapp+apa+XX+YYt/ &`

PAWSEY supercomputing centre

Exercise 5: profiling game of life (1)

Profile (sampling) the game_of_life code.

- `module load perftools`
- `cd game_of_life`
- `make game_of_life.cray`
- `pat_build game_of_life.O3.cray`
- `sbatch run_game_profile.slurm`

- `pat_report game_of_life.O3.cray+pat+XX-YYs/ > game_of_life.O3.cray.sampling.report`

Exercise 5: profiling game of life (2)

Examine
`game_of_life.O3.cray.sampling.report`

- Where is all the time spent? What occurs on these lines of the code?
- How good is our cache utilisation?
- Check optimisations in `game_of_life.O3.lst`
- Try again with other levels of optimisation (edit the `Makefile`)

PAWSEY supercomputing centre

Exercise 6: profiling matrix multiplication (1) - **sampling**

Sampling experiment

- `module load perftools`
- `cd profiling`
- `ftn -c –O2 matrix.f90`
- `ftn -o matrix.O2 matrix.o`
- `pat_build matrix.O2`
- `sbatch run_matrix_profile.slurm`
- Generate the report:

`pat_report matrix.O2+pat+XX+YYs/ > matrix.O2.sampling.report`

- Examine `matrix.O2.sampling.report`
- You can also use aprentice2: `app2 matrix.O2+pat+XX+YYs/ &`

PAWSEY supercomputing centre

Exercise 6: profiling matrix (2) **sampling**

```
Table 2:  Profile by Group, Function, and Line

  Samp% |  Samp | Imb. |  Imb. | Group
        |       | Samp | Samp% |  Function
        |       |      |       |   Source
        |       |      |       |    Line

 100.0% | 566.0 |   -- |    -- | Total
|---------------------------------------------------------------------------
|  67.3% | 381.0 |   -- |    -- | BLAS
||--------------------------------------------------------------------------
||  38.9% | 220.0 |   -- |    -- | gotoblas_dgemv_n_haswell
||  25.6% | 145.0 |   -- |    -- | __sci_dgemm_
3|        |       |      |       |  bframe/crayblas/src/sci_gemm.c
4|        |       |      |       |   line.58
||==========================================================================
|  24.7% | 140.0 |   -- |    -- | USER
||--------------------------------------------------------------------------
||  23.0% | 130.0 |   -- |    -- | test_matmul3$timeit_
3|        |       |      |       |  serialOptimisation/Optimising-Serial-Code/profiling/matrix.f90
4|  22.3% | 126.0 |   -- |    -- |   line.190
||   1.2% |   7.0 |   -- |    -- | test_matmul1$timeit_
3|        |       |      |       |  serialOptimisation/Optimising-Serial-Code/profiling/matrix.f90
||==========================================================================
|   6.4% |  36.0 |   -- |    -- | ETC
||--------------------------------------------------------------------------
||   4.6% |  26.0 |   -- |    -- | _RANF
|===========================================================================
```

PAWSEY supercomputing centre

Exercise 6: profiling matrix (3) **sampling**

```
Table 3:  Program HW Performance Counter Data

Total


========================================================================
  Total
------------------------------------------------------------------------
  Thread Time                                          0.765879 secs
  CPU_CLK_THREAD_UNHALTED:THREAD_P                1,710,445,839
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                   50,726,596
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                 1,210,638
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                   88,411
  L1D:REPLACEMENT                                   263,414,777
  L2_RQSTS:ALL_DEMAND_DATA_RD                       206,543,602
  L2_RQSTS:DEMAND_DATA_RD_HIT                       160,731,679
  MEM_UOPS_RETIRED:ALL_LOADS                      1,948,873,169
  CPU_CLK                             3.37GHz
  TLB utilization                 1,500.23 refs/miss         2.93 avg uses
  D1 cache hit,miss ratios           86.5% hits             13.5% misses
  D1 cache utilization (misses)       7.40 refs/miss         0.92 avg hits
  D2 cache hit,miss ratio            82.6% hits             17.4% misses
  D1+D2 cache hit,miss ratio         97.6% hits              2.4% misses
  D1+D2 cache utilization            42.54 refs/miss         5.32 avg hits
  D2 to D1 bandwidth              16.074GiB/sec  13,218,790,528 bytes
========================================================================
```

Exercise 6: profiling matrix (4) **sampling**

```
Table 6:  Wall Clock Time, Memory High Water Mark

  Process |   Process | Total
     Time |     HiMem |
          | (MBytes)  |

 0.801315 |      44.1 | Total
```

Exercise 6: profiling matrix multiplication (5) - **tracing**

Tracing experiment

- Edit the build_options file to define tracing options:

```
vim matrix.O2+pat+XX-YYs/build_options.apa
```

Have the following:

```
-g mpi,blas,io,heap
-T test_matmul1$timeit_
-T test_matmul2$timeit_
-T test_matmul3$timeit_
-T test_matmul4$timeit_
-T test_matmul5$timeit_
```

- Build the executable with the defined tracing options:

```
pat_build -O matrix.O2+pat+XX-YYs/build_options.apa
```

Exercise 6: profiling matrix (6) **tracing**

- Edit the job script: `vim run_matrix_profile.slurm`
  Have: `expType=apa`
- `sbatch run_matrix_profile.slurm`

- Generate the report:

`pat_report matrix.O2+apa+XX-YYt/ > matrix.O2.tracing.report`

- Examine: `matrix.O2.tracing.report`
- You can also use aprentice2: `app2 matrix.O2+apa+XX+YYt/ &`

- Use the previous exercises of the matrix multiplication (timing and optimisation listing) to understand the results.
- Repeat the exercise with lower levels of optimisation.

Seek help

- `man pat_build`
- `man pat_report`
- `pat_help all . > all_pat_help`

Exercise 6: profiling matrix (7) **tracing**

```
Table 1:  Profile by Function Group and Function

  Time% |      Time | Imb. |  Imb. |    Calls | Group
        |           | Time | Time% |          |  Function

 100.0% | 0.691078 |   -- |    -- | 1,966.0 | Total
|------------------------------------------------------------------
|  75.5% | 0.521487 |   -- |    -- |     6.0 | USER
||-----------------------------------------------------------------
||  33.3% | 0.230219 |   -- |    -- |     1.0 | test_matmul4$timeit_
||  19.4% | 0.134307 |   -- |    -- |     1.0 | test_matmul3$timeit_
||   8.2% | 0.056589 |   -- |    -- |     1.0 | test_matmul1$timeit_
||   7.3% | 0.050596 |   -- |    -- |     1.0 | test_matmul5$timeit_
||   7.2% | 0.049709 |   -- |    -- |     1.0 | test_matmul2$timeit_
||=================================================================
|  21.6% | 0.149372 |   -- |    -- | 1,302.0 | BLAS
||-----------------------------------------------------------------
||  21.4% | 0.147623 |   -- |    -- |     1.0 | dgemm_
||=================================================================
|   2.9% | 0.020159 |   -- |    -- |   646.0 | ETC
||-----------------------------------------------------------------
||   1.7% | 0.011966 |   -- |    -- |     5.0 | _END
||   1.1% | 0.007688 |   -- |    -- |   384.0 | query_cpu_mask
|==================================================================
```

Exercise 6: profiling matrix (8) **tracing**

```
==================  Observations and suggestions  ===================


D1 cache utilization:

    1.7% of total execution time was spent in 1 functions with D1 cache
    hit ratios below the desirable minimum of 75.0%. Cache utilization
    might be improved by modifying the alignment or stride of references
    to data arrays in these functions.

        D1    Time%   Function
     cache
       hit
     ratio

       0.0%    1.7%   _END


D1 + D2 cache utilization:

    All instrumented functions with significant execution time had
    combined D1 and D2 cache hit ratios above the desirable minimum of
    80.0%.


TLB utilization:

    1.1% of total execution time was spent in 1 functions with fewer
    than the desirable minimum of 200 data references per TLB miss. TLB
    utilization might be improved by modifying the alignment or stride
    of references to data arrays in these functions.

     LS per   Time%   Function
     TLB DM

      100.35    1.1%   query_cpu_mask
```

Exercise 6: profiling matrix (9) **tracing**

```
========================================================================
  USER / test_matmul4$timeit_
------------------------------------------------------------------------
  Time%                                                   33.3%
  Time                                                 0.230219 secs
  Imb. Time                                                  -- secs
  Imb. Time%                                                 --
  Calls                         4.344 /sec                  1.0 calls
  CPU_CLK_THREAD_UNHALTED:THREAD_P                  785,898,184
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                   22,709,160
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                   942,397
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                   11,259
  L1D:REPLACEMENT                                   155,368,214
  L2_RQSTS:ALL_DEMAND_DATA_RD                       130,456,350
  L2_RQSTS:DEMAND_DATA_RD_HIT                        68,567,950
  MEM_UOPS_RETIRED:ALL_LOADS                        794,085,314
  CPU_CLK                        3.46GHz
  TLB utilization              832.67 refs/miss            1.63 avg uses
  D1 cache hit,miss ratios      80.4% hits                19.6% misses
  D1 cache utilization (misses)  5.11 refs/miss            0.64 avg hits
  D2 cache hit,miss ratio       60.2% hits                39.8% misses
  D1+D2 cache hit,miss ratio    92.2% hits                 7.8% misses
  D1+D2 cache utilization       12.83 refs/miss            1.60 avg hits
  D2 to D1 bandwidth         33.776GiB/sec   8,349,206,400 bytes
  Average Time per Call                                0.230219 secs
  CrayPat Overhead : Time        0.0%
========================================================================
```

Exercise 6: profiling matrix (10) **tracing**

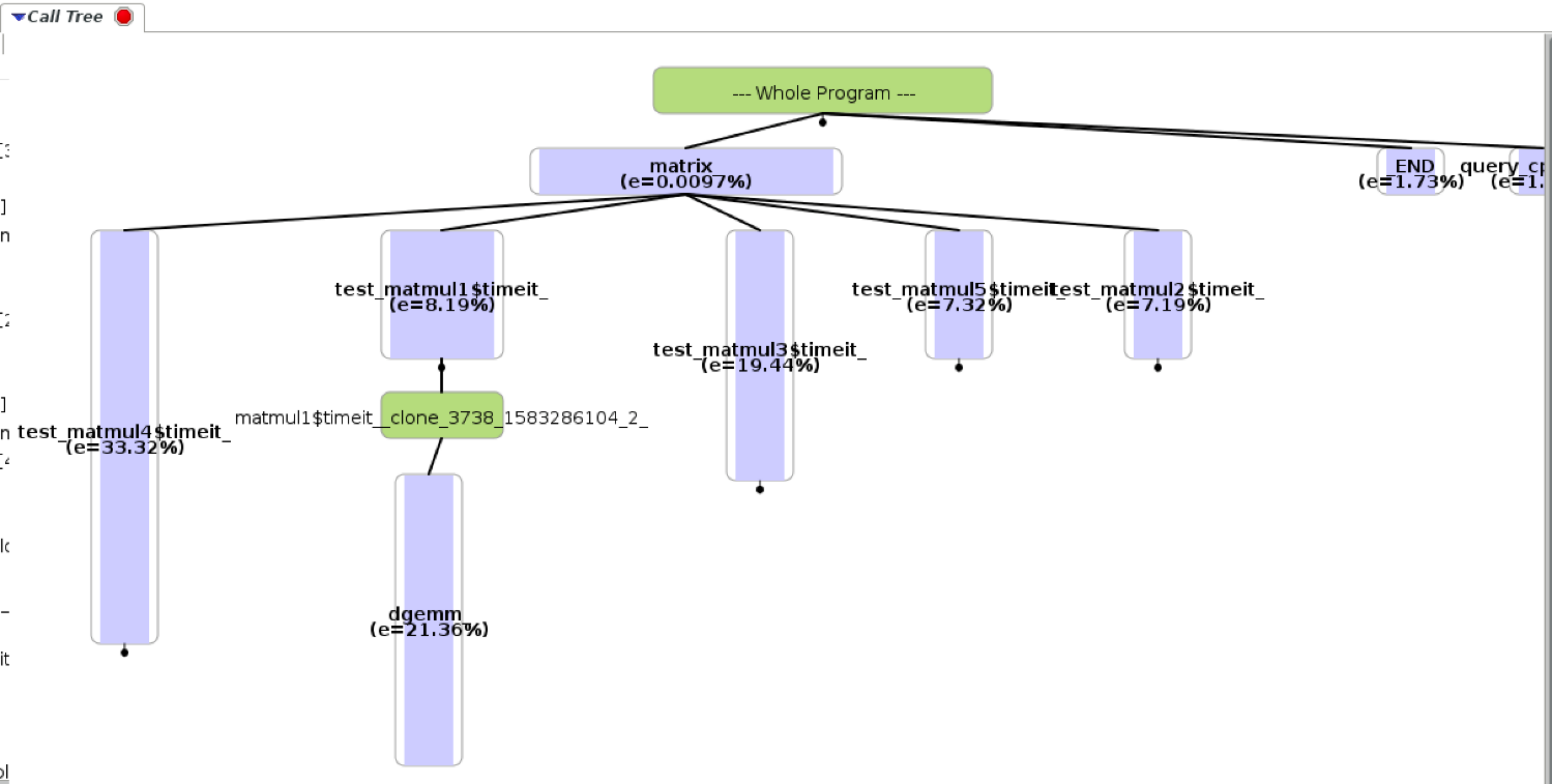
```
==============================================================================
  USER / test_matmul2$timeit_
------------------------------------------------------------------------------
  Time%                                                       7.2%
  Time                                                    0.049709 secs
  Imb. Time                                                     -- secs
  Imb. Time%                                                    --
  Calls                           20.117 /sec                  1.0 calls
  CPU_CLK_THREAD_UNHALTED:THREAD_P                     173,201,511
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                       5,211,090
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                       16,156
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                      15,058
  L1D:REPLACEMENT                                       23,825,001
  L2_RQSTS:ALL_DEMAND_DATA_RD                            9,427,603
  L2_RQSTS:DEMAND_DATA_RD_HIT                           13,346,806
  MEM_UOPS_RETIRED:ALL_LOADS                           224,332,726
  CPU_CLK                           3.32GHz
  TLB utilization                 7,186.93 refs/miss         14.04 avg uses
  D1 cache hit,miss ratios           89.4% hits              10.6% misses
  D1 cache utilization (misses)       9.42 refs/miss          1.18 avg hits
  D2 cache hit,miss ratio           100.0% hits               0.0% misses
  D1+D2 cache hit,miss ratio        100.0% hits               0.0% misses
  D1+D2 cache utilization               -- refs/miss            -- avg hits
  D2 to D1 bandwidth              11.304GiB/sec    603,366,592 bytes
  Average Time per Call                                   0.049709 secs
  CrayPat Overhead : Time             0.0%
```

Exercise 6: profiling matrix (11) **tracing**

```
==============================================================================
  BLAS / dgemm_
------------------------------------------------------------------------------
  Time%                                                    21.4%
  Time                                                  0.147623 secs
  Imb. Time                                                   -- secs
  Imb. Time%                                                  --
  Calls                                 6.774 /sec           1.0 calls
  CPU_CLK_THREAD_UNHALTED:THREAD_P                   139,884,939
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                     4,202,951
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                     26,457
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                     4,623
  L1D:REPLACEMENT                                     32,267,796
  L2_RQSTS:ALL_DEMAND_DATA_RD                         21,348,184
  L2_RQSTS:DEMAND_DATA_RD_HIT                          7,469,941
  MEM_UOPS_RETIRED:ALL_LOADS                         193,382,571
  CPU_CLK                                 3.33GHz
  TLB utilization                    6,222.09 refs/miss        12.15 avg uses
  D1 cache hit,miss ratios              83.3% hits             16.7% misses
  D1 cache utilization (misses)          5.99 refs/miss         0.75 avg hits
  D2 cache hit,miss ratio               57.0% hits             43.0% misses
  D1+D2 cache hit,miss ratio            92.8% hits              7.2% misses
  D1+D2 cache utilization               13.93 refs/miss         1.74 avg hits
  D2 to D1 bandwidth                  8.620GiB/sec   1,366,283,776 bytes
  Average Time per Call                                 0.147623 secs
  CrayPat Overhead : Time                0.0%
```

Exercise 6: profiling matrix (12) **tracing**

From apprentice2: `app2 matrix.O2+apa+XX+YYt/ &`



Exercise 6: profiling matrix (13) **tracing**

```
Table 3:  Heap Stats during Main Program

Total



======================================================================
  Total
----------------------------------------------------------------------
  Tracked Heap HiWater MBytes     23.418
  MBytes Not Tracked               0.000
  Total Allocs                       134
  Allocs Not Tracked                   0
  Total Frees                        128
  Inferred Frees                       0
  Tracked Objects Not Freed            7
  Tracked MBytes Not Freed         0.247
======================================================================
```

PAWSEY supercomputing centre

Exercise 6: profiling matrix multiplication (14) – **tracing**

```
Table 4:  Heap Leaks during Main Program

 Tracked | Tracked | Tracked | Caller
  MBytes |  MBytes | Objects |
     Not |     Not |     Not |
   Freed%|   Freed |   Freed |

  100.0% |   0.247 |       7 | Total
|-----------------------------------------------------
|   28.8% |   0.071 |       1 | query_cpu_topology
|   28.1% |   0.069 |       1 | _GLOBAL__sub_I_eh_alloc.cc
|   25.6% |   0.063 |       2 | test_matmul1$timeit_
|         |         |         |  matrix_
|   12.7% |   0.031 |       1 | resize_place_table
|    4.8% |   0.012 |       1 | _cray$mt_init
|=====================================================
```

Exercise 6: profiling matrix multiplication (15) – **tracing**

```
Table 5:  File Input Stats by Filename

     Read |      Read |  Read Rate |    Reads | Bytes/ | File Name=!x/^/(proc|sys)/
     Time |    MBytes | MBytes/sec |          |   Call |

 0.012854 | 0.051073 |   3.973411 | 1,393.0 |  38.45 | Total
```

```
Table 6:  File Output Stats by Filename (maximum 15 shown)

    Write |     Write | Write Rate | Writes | Bytes/ | File Name[max15]
     Time |    MBytes | MBytes/sec |        |   Call |

 0.000409 | 0.000796 |   1.947631 |   45.0 |  18.56 | Total
|-----------------------------------------------------------------------
| 0.000409 | 0.000796 |   1.947631 |   45.0 |  18.56 | stdout
|=======================================================================
```

Exercise 6: profiling matrix multiplication (16) – **tracing**

```
Table 9:  Wall Clock Time, Memory High Water Mark

  Process |   Process | Total
     Time |     HiMem |
          | (MBytes)  |

 0.869646 |      46.1 | Total
```

CODING HABITS

Global variables

- Avoid global variables unless necessary.
  They may make it difficult to convert to multithreaded code in further development.

- Pass variables through routine calls. (There is a slight performance overhead).
  In Fortran arguments can be given intent(in), intent(out) attributes. This assists the compiler.
  Scoping in OpenMP becomes much easier.
  May assist in auto-threading by compilers.

Parentheses in Fortran

- Try to avoid parentheses in Fortran; they force an evaluation and prevent code arithmetic rearrangements. Use temporary variables instead.

- Compiler not permitted to rearrange this:

```
a = 2 * (c + d) – 2 * e
```

- Compiler allowed to rearrange this:

```
tmp=c+d
a=2*tmp – 2 * e
```

Fortran Array Notation

Fortran array notation is convenient and easy to read, but current compilers ***are likely to not optimise*** them well.

Some compilers are unlikely to fuse these operations:

```
A(:,:)=1.0
C(:,:)=A(:,:)+B(:,:)
```

In the meantime:

```
do j,1,n
  do i=1,m
    A(i,j)=1.0
    C(i,j)=A(i,j)+B(i,j)
  end do
end do
```

The Cray compiler does fuse array notation!

Low Level Object Oriented

Low level Object Oriented programming has the potential for poor performance. E.g. the below strided (not contiguous) memory accesses.

```
type atom_type
  integer :: atomic_number
  double precision :: mass
  double precision, dimension(3) :: position
end type atom_type

type(atom_type), dimension(n_atoms) :: atom_list

total_mass=0
do i=1,n_atoms
  total_mass=total_mass+atom_list(i)%mass
end do
```

Low Level Object Oriented (2)

Less organised but faster code:

```
integer, dimension(n_atoms) :: atomic_numbers
double precision, dimension(n_atoms) :: masses
double precision, dimension(3,n_atoms) :: positions

total_mass=sum(masses)
```

Set a level for the trade-off between maintainability/extensibility and performance.

Special Case Code

Assume we have a code that handles arrays of varying length, and;

- the code creates temporary arrays;
- in practice an array of length 1 is the most common situation.

Optimisation: write a separate routine for arrays of length 1, and use temporary variables rather than temporary arrays.

PAWSEY supercomputing centre

I/O

- Often the processor is doing little while waiting for I/O.
- Ways to reduce I/O overhead:
  - Use buffering (or don’t turn it off or flush).
  - Output in binary, not formatted text.
  - Use I/O libraries.
- Hierarchical Data Format (HDF5) is the name of a set of file formats and libraries designed to store and organize large amounts of numerical data.

Exercise 7: I/O

Observe the effects of I/O techniques on performance.

- `module load cray-hdf5`
- `cd iobench && make iobench_hdf5`
- `sbatch run_iobench_hdf5.slurm`

Look at the SLURM output file.

Use version control

- Some of your attempts at optimisation will need to be undone. E.g. due to:
  - Incorrect results.
  - Slower performance.

- Use version control software. E.g. git, subversion. You should be using this anyway.
- Use informative comments in check-in.

MATH LIBRARIES

Popular libraries

- BLAS: basic linear algebra such as matrix-vector or matrix-matrix operations.

- LAPACK:
  - Simple matrix/vector operations
  - Linear equations solvers
  - Linear least squares
  - Eigensolvers
  - Singular value decomposition
  - Real + Complex

- FFTW: fast fourier transforms, real/complex

Optimised vendor versions available. e.g. Intel MKL, Cray Libsci, SGI SCSL, IBM ESSL. Some are multi-threaded.

Other libraries

- PLAPACK - better scaling eigensolver (MRRR algorithm)
- PARPACK – sparse eigensolver
- MUMPS – parallel sparse direct solver
- Hypre – parallel linear solver
- Scotch – graph partitioning
- SuperLU – parallel sparse linear solver

- available at cray-tpsl

Intel Math Libraries

Intel MKL includes BLAS, LAPACK and FFT libraries. It consists of multiple libraries – use the Intel advisor to work out the compiler link options:

http://software.intel.com/sites/products/mkl/MKL_Link_Line_Advisor.html

Example output:

```
-L$(MKLROOT)/lib/intel64 -lmkl_intel_lp64 -
lmkl_sequential -lmkl_core –lpthread -lm
```

$(MKLROOT) is set by “module load intel”

PAWSEY supercomputing centre

PetSc / Slepc

- It's an attempt at a black box solver suite. (makes it easy to swap between various solvers in various libraries). It links to common libraries.

- C/C++ and Fortran interfaces
- Linear equation solvers
- Eigensolvers
- Dense and Sparse
- various finite element solver add-ons

Finish

- What's next?:
  - Come to the parallel course.
  - Read optimisation guides from vendors. Intel and Cray in particular.

- Slides are available at

https://support.pawsey.org.au/documentation/display/US/Training+Material

PAWSEY supercomputing centre