PAWSEY supercomputing centre

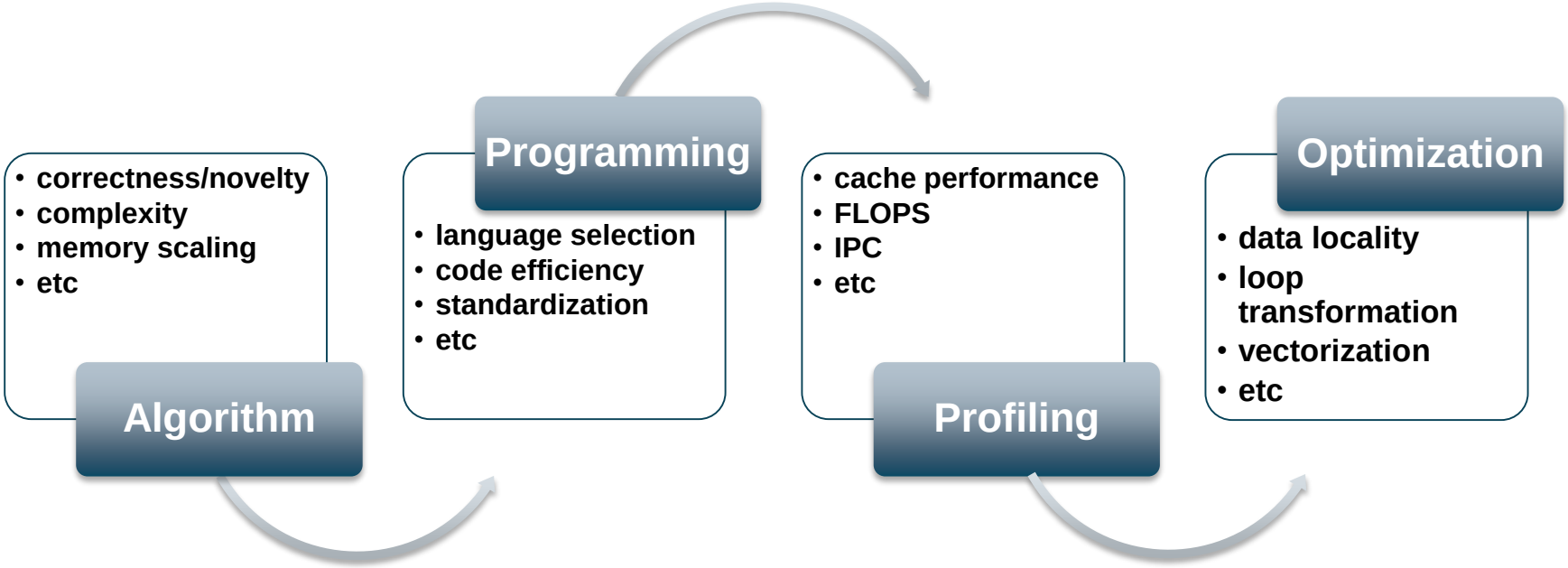
Optimising Serial Code

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



Parallelization

Performance

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.

Development Effort

1 Magnus day = 48 years on a dual-core desktop.

- ⇒ It is worth it to spend months of development time to save years/decades of runtime!
- ⇒ This might go against common opinions.

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

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

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

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!

Timing routines

- C / C++

  clock – CPU time, CLOCKS_PER_SEC

  time – real time, seconds

- Fortran

  cpu_time – CPU time, microseconds

  date_and_time – real time, microseconds

- glibc

  gettimeofday – microseconds since epoch

- MPI

  MPI_Wtime – real time, MPI_Wtick units

CHOOSING AN ALGORITHM

PAWSEY
supercomputing centre

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>

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 $\#t.n.(n\text{-}1)$ calculations = $O(t*n^2)$

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

Scaling

- Consider memory scaling as well as compute scaling. **Memory is finite**.
- E.g. QR tridiagonal eigensolver is $O(n^2)$ in memory, while MR^3 tridiagonal eigensolver is $O(n)$ in memory.

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

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.

- $O(N^2)$ 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

Choosing an Algorithm

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

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

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 large 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	LuaJIT	Java
	gcc 4.8.2	0.3.7	2.7.9	3.1.3	R2014a	3.8.1	10.0	V8 3.14.5.9	go1.2.1	gsl-shell 2.3.1	1.7.0_75
fib	0.57	2.14	95.4	529	4260	9210	167	3.68	2.20	2.02	0.96
parse_int	4.67	1.57	20.5	54.3	1530	7570	17.7	2.29	3.78	6.09	5.43
quicksort	1.10	1.21	46.7	248	55.9	1530	48.5	2.91	1.09	2.00	1.65
mandel	0.87	0.87	18.8	59.0	60.1	394	6.12	1.86	1.17	0.71	0.68
pi_sum	0.83	1.00	21.1	14.4	1.28	260	1.27	2.15	1.23	1.00	1.00
rand_mat_stat	0.99	1.74	22.3	16.9	9.82	30.4	6.20	2.81	8.23	3.71	4.01
rand_mat_mul	4.05	1.09	1.08	1.63	1.12	1.06	1.13	14.58	8.45	1.23	2.35

Comparison from http://julialang.org/.
Times relative to C (gcc 4.8.2). Lower is better.

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

Fortran Compiler Comparison

	GCC	Intel	Cray
fib	1	4.41	3.16
parse_int	1	0.77	0.77
mandel	1	0.50	-
quicksort	1	0.91	0.99
pi_sum	1	0.55	0.55
rand_mat_stat	1	0.80	0.47
rand_mat_mul	1	0.75	0.15

Comparison code perf.f90 from http://julialang.org/.

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.

- 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/PawseySupercomputing/Optimising-Serial-Code.git

It's case sensitive. Download them!

Have a look at the Makefile.

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 *jobid.slurm*.

Exercise 2: matmul (2)

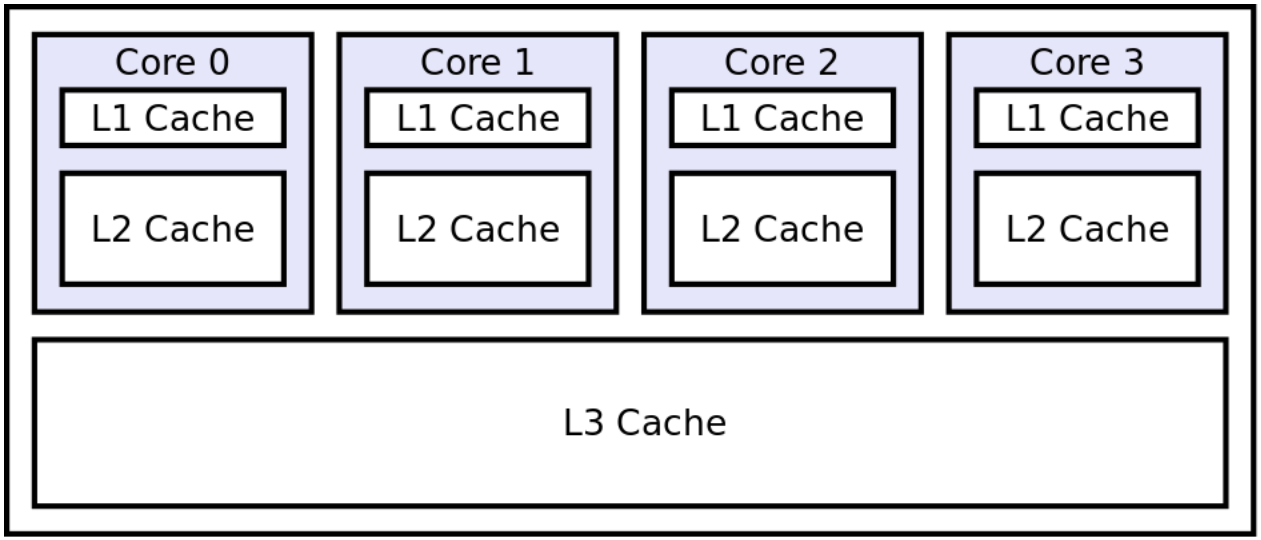
Compare the timings:

- 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?
- 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

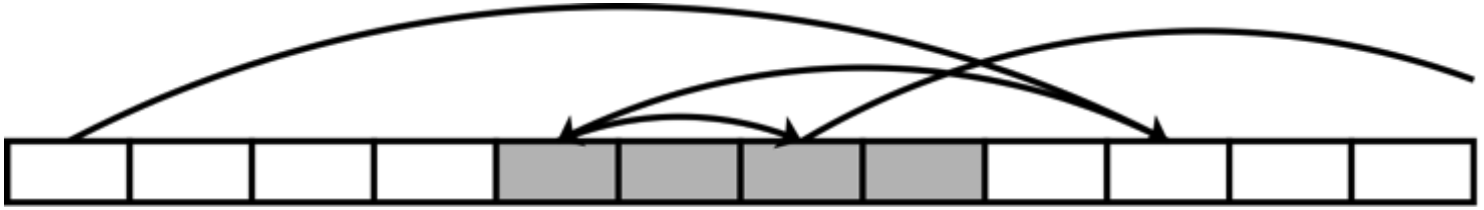
Source: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

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



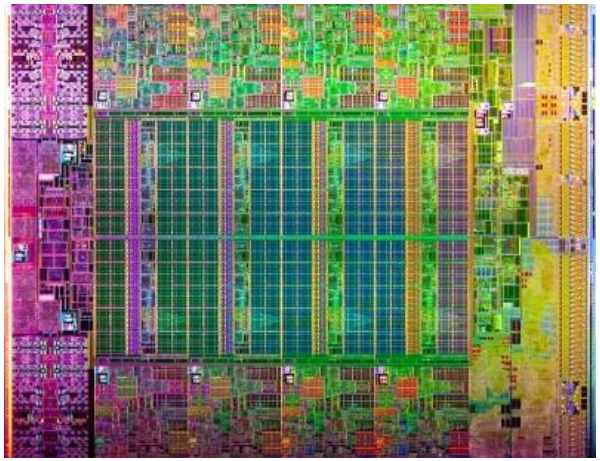
Translation Lookaside Buffer

- 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. 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)

Being cache-friendly

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



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 clock cycle.
- Instructions Per Cycle (e.g. 4 for a Xeon) is an average. 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.

Loop Blocking (1)

Loop blocking/tiling/strip-mining …

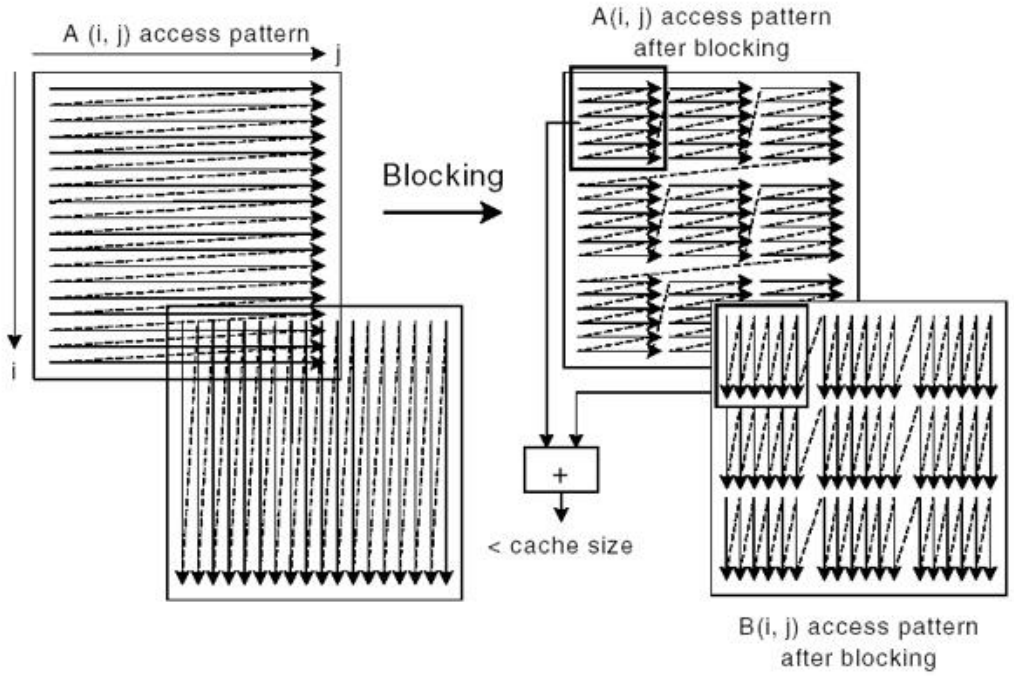
This code potentially contains many cache misses.

```
do j=1,jmax
  do i=1,imax
    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.
- 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.

Contiguous Memory

- Aim to work through contiguous chunks of memory. Avoid unnecessary striding.
- C/C++ and Fortran have different loop orders! (Column vs Row)

Fortran

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

C /C++

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

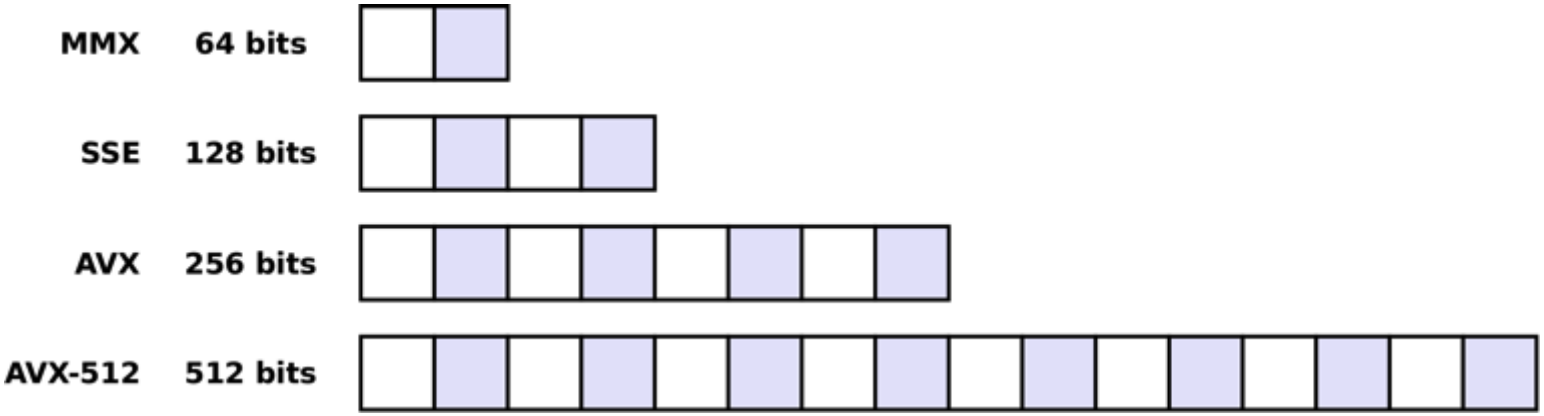
Pointers

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 accelerator memory.
- Hard to optimise cache use, e.g. with linked lists vs contiguous arrays.

MMX / 3DNOW / SSE / AVX

- These are single-instruction-multiple-data 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.



Inlining

- Function calls take time. You can remove this time by placing the code into the calling routine.
- Compilers ***can*** inline code for you. 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.

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 (HDF, HDF4, or HDF5) is the name of a set of file formats and libraries designed to store and organize large amounts of numerical data.

Exercise 3: 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.

PROFILING

PAWSEY
supercomputing centre

Profiling phases

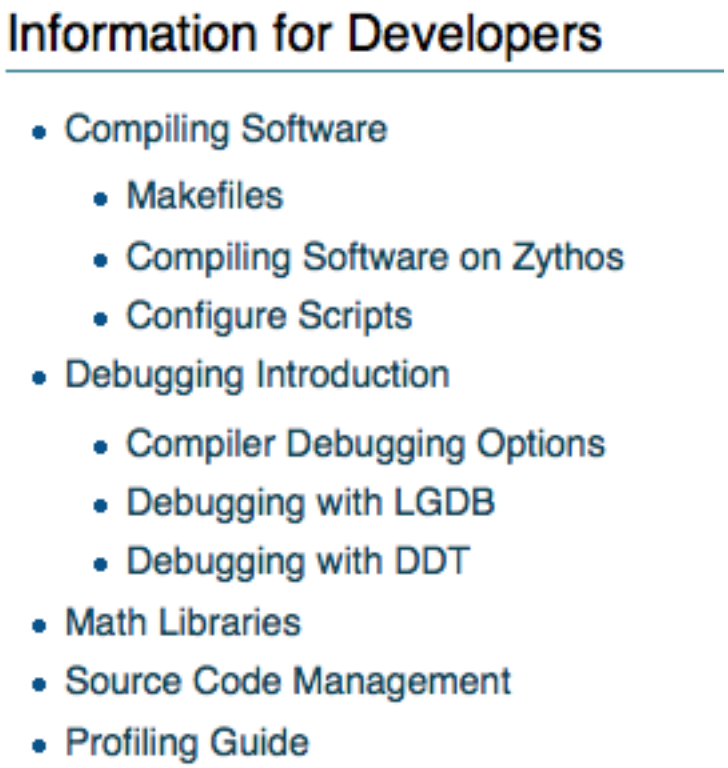
- **Instrumentation:** compile the source code with extra compiler flags that enable the recording of performance-relevant events.
- **Measurement & analysis:** run the instrumented application on a **representative** test case. Usually the instrumented application is much slower than the original one.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

Profilers

- On Cray supercomputers, CrayPAT.
- Tau: http://www.cs.uoregon.edu/research/tau/home.php
- Scalasca: http://www.scalasca.org/
- Intel VTune: https://software.intel.com/en-us/intel-vtune-amplifier-xe

Profilers

- Profiling guide at Pawsey https://support.pawsey.org.au/documentation/display/US/Profiling



PAWSEY supercomputing centre

Full Profiling with CrayPAT

Sampling experiment

Instrumentation

- load `perftools` module
- 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
- Output file: `myapp+pat+*.xf` (or directory of `.xf` files for large runs)
- `pat_report myapp+pat+*.xf > myapp_report.txt` (also generates `.ap2` file that can be viewed with Apprentice2)

Performance examination

- Read output file, or use Apprentice2 with the `.ap2` file.

Full Profiling with CrayPat

Tracing experiment

Instrumentation

- `pat_build -O myapp+pat+*.apa`
  - Essentially pat_build -w -T funcs -g grps -u myapp
  - Generates executable named `myapp+apa`

Measurement & analysis

- Run `myapp+apa` as normal
- Output file: `myapp+apa+*.xf` (or directory of `.xf` files for large runs)
- `pat_report myapp+apa+*.xf > myapp_report2.txt` (also generates `.ap2` file that can be viewed with Apprentice2)

Performance examination

- Read output file, or use Apprentice2 with the `.ap2` file.

Exercise 4: profiling game of life (1)

Profile the game_of_life code.

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

- `pat_report game_of_life+pat+`*`XXXX-YYY`*`s.xf > game_of_life.report`

Examine game_of_life`.report`

Exercise 4: profiling game of life (2)

Examine game_of_life`.report`

- Where is all the time spent? What occurs on these lines of the code?
- How good is our cache utilisation?

Exercise 5: profiling matrix multiplication (1)

Sampling experiment

- `module load perftools`
- `cd profiling`
- `ftn -c matrix.f90 && ftn -o matrix matrix.o`
- `pat_build matrix`
- `sbatch run_matrix_profile.slurm`
- `pat_report matrix+pat+`*`XXXX-YYY`*`s.xf > matrix_s.report`

Examine `matrix_s.report`

Exercise 5: profiling matrix multiplication (2)

Tracing experiment

- `vim matrix+pat+XXXX-YYYs/build_options.apa`
  - -g mpi,blas,io,heap
- `pat_build -O matrix+pat+XXXX-YYYs/build_options.apa`
- `vim run_matrix_profile.slurm`
  - `srun --export=all -n 1 matrix+apa`
- `sbatch run_matrix_profile.slurm`
- `pat_report matrix+apa+XXXX-YYYs.xf > matrix_t.report`

Examine `matrix_t.report`

Exercise 5: profiling matrix multiplication (3)

Seek help

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

Exercise 5: profiling matrix multiplication – sampling (4)

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

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

 100.0% | 24.0  |    -- |     -- |Total
|-----------------------------------------------------------------
|  58.3% | 14.0  |    -- |     -- |USER
||----------------------------------------------------------------
|  58.3% | 14.0  |    -- |     -- | time_it$timeit_
3        |       |       |        |  opt_serial/opt_F/profile/matrix.f90
||||-------------------------------------------------------------
4|||   4.2% |  1.0  |    -- |     -- |line.24
4|||  54.2% | 13.0  |    -- |     -- |line.51
||||=============================================================
||================================================================
|  41.7% | 10.0  |    -- |     -- |ETC
||----------------------------------------------------------------
||  29.2% |  7.0  |    -- |     -- |dgemm_kernel_a1b1
||   8.3% |  2.0  |    -- |     -- |dgemm_kernel_a1b0
||   4.2% |  1.0  |    -- |     -- |dgemm_itcopy
|================================================================
```

PAWSEY supercomputing centre

Exercise 5: profiling matrix multiplication – sampling (5)

```
Table 3:  Program HW Performance Counter Data

Total


================================================================
  Total
----------------------------------------------------------------
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                     24054054
  CPU_CLK_THREAD_UNHALTED:THREAD_P                    758346630
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                    326004
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                    81100
  L1D:REPLACEMENT                                      89587508
  L2_RQSTS:DEMAND_DATA_RD_HIT                          33501445
  L2_RQSTS:ALL_DEMAND_DATA_RD                          43949624
  MEM_UOPS_RETIRED:ALL_LOADS                          726049373
  User time (approx)                 0.251 secs      652162003 cycles
  CPU_CLK                            3.153GHz
  TLB utilization                 1783.45 refs/miss     3.483 avg uses
  D1 cache hit,miss ratios          87.7% hits          12.3% misses
  D1 cache utilization (misses)      8.10 refs/miss     1.013 avg hits
  D2 cache hit,miss ratio           88.3% hits          11.7% misses
  D1+D2 cache hit,miss ratio        98.6% hits           1.4% misses
  D1+D2 cache utilization           69.49 refs/miss     8.686 avg hits
  D2 to D1 bandwidth             10698.431MiB/sec  2812775936 bytes
```

Exercise 5: profiling matrix multiplication – sampling (6)

```
Table 4:  Wall Clock Time, Memory High Water Mark

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

 0.290675 |    48.527 |Total
```

Exercise 5: profiling matrix multiplication – tracing (7)

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


MFLOPS not available on Intel Haswell:

    The document that specifies performance monitoring events for Intel
    processors does not include events that could be used to compute a
    count of floating point operations for Haswell processors: Intel 64
    and IA-32 Architectures Software Developer's Manual, Order Number
    253665-050US, February 2014.


D1 cache utilization:

    All instrumented functions with significant execution time had D1
    cache hit ratios above the desirable minimum of 75.0%.


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:

    All instrumented functions with significant execution time had more
    than the desirable minimum of 200 data references per TLB miss.

========================  End Observations  ========================
```

Exercise 5: profiling matrix multiplication – tracing (8)

```
==============================================================================
  BLAS / dgemm_
------------------------------------------------------------------------------
  Time%                                                  21.6%
  Time                                               0.051761 secs
  Imb. Time                                                 -- secs
  Imb. Time%                                                --
  Calls                                  19.317 /sec           1.0 calls
  CPU_CLK_THREAD_UNHALTED:REF_XCLK                      4813452
  CPU_CLK_THREAD_UNHALTED:THREAD_P                    140373589
  DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK                    92870
  DTLB_STORE_MISSES:MISS_CAUSES_A_WALK                   22283
  L1D:REPLACEMENT                                     21372948
  L2_RQSTS:DEMAND_DATA_RD_HIT                          3076814
  L2_RQSTS:ALL_DEMAND_DATA_RD                          4265255
  MEM_UOPS_RETIRED:ALL_LOADS                         121577968
  User time (approx)                     0.052 secs  134649328 cycles  100.0% Time
  CPU_CLK                                2.916GHz
  TLB utilization                     1055.80 refs/miss    2.062 avg uses
  D1 cache hit,miss ratios              82.4% hits         17.6% misses
  D1 cache utilization (misses)          5.69 refs/miss    0.711 avg hits
  D2 cache hit,miss ratio               94.4% hits          5.6% misses
  D1+D2 cache hit,miss ratio            99.0% hits          1.0% misses
  D1+D2 cache utilization              102.30 refs/miss   12.788 avg hits
  D2 to D1 bandwidth                5028.764MiB/sec   272976320 bytes
  Average Time per Call                                0.051761 secs
  CrayPat Overhead : Time                 0.0%
```

PAWSEY supercomputing centre

Exercise 5: profiling matrix multiplication – tracing (9)

```
Table 4:  Heap Leaks during Main Program

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

  100.0% |   0.001 |       2 |Total
|-----------------------------------------
|  100.0% |   0.001 |       2 |time_it$timeit_
|         |         |         | matrix_
|=========================================
```

Exercise 5: profiling matrix multiplication – tracing (10)

```
Table 3:  Heap Stats during Main Program

Total


=========================================
  Total
-----------------------------------------
  Tracked Heap HiWater MBytes     23.139
  MBytes Not Tracked               0.000
  Total Allocs                        14
  Allocs Not Tracked                   0
  Total Frees                         14
  Inferred Frees                       0
  Tracked Objects Not Freed            2
  Tracked MBytes Not Freed         0.001
```

Exercise 5: profiling matrix multiplication – tracing (11)

```
Table 5:  File Output Stats by Filename (maximum 15 shown)

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

 0.000024 | 0.000371 |  15.643670 |    6.0 |  64.83 |Total
|-------------------------------------------------------------------
| 0.000024 | 0.000371 |  15.643670 |    6.0 |  64.83 |stdout
|===================================================================
```

Exercise 5: profiling matrix multiplication – tracing (12)

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

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

 0.288572 |   50.582 |Total
```

CODING HABITS

PAWSEY
supercomputing centre

Global variables

- Avoid global variables unless necessary.
- They may make it difficult to convert to threaded code.

- 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.

Brackets in Fortran

- Try to avoid brackets in Fortran; they force an evaluation and prevent code rearrangement. 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 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.

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.

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

```
Primary Loop Type        Modifiers
    ------- ---- ----        ---------
    A - Pattern matched      a - vector atomic memory operation
                             b - blocked
    C - Collapsed            c - conditional and/or computed
    D - Deleted              f - fused
    E - Cloned
    I - Inlined              i - interchanged
    M - Multithreaded        m - partitioned
    P - Parallel             p - partial
    R - Redundant            r - unrolled
                             s - shortloop
    V - Vectorized           t - array syntax temp used
                             w - unwound
```

Intel Compiler Output

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

- Vectorisation reports.
- Compiler flag: -vec-report=6

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

Exercise 6: Cray compiler output

- Run:

```
cd compiler_reports
ftn -rm -O2 matrix.f90
```

- Check out the manpage if needed
  - `man crayftn`
- Examine the output in matrix.lst
- What has the compiler done with routine calls and loops?

Exercise 7: Intel compiler output

- Run:

```
module swap PrgEnv-cray PrgEnv-intel
cd compiler_reports
ftn -qopt-report=3 -mkl=sequential matrix.f90
```

- Seek help: `ifort -help reports`
- Might be a bit too much information. Scale back the reporting options.

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”

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 addons

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