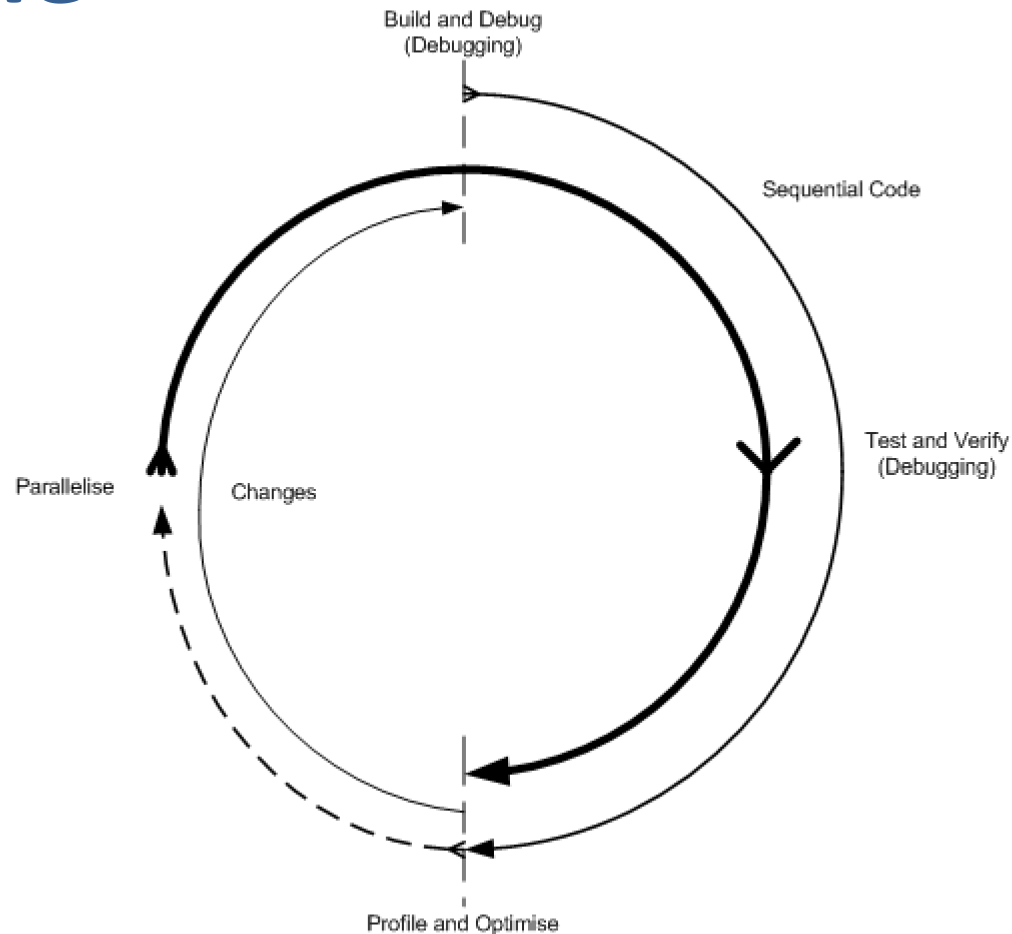
University of East Anglia

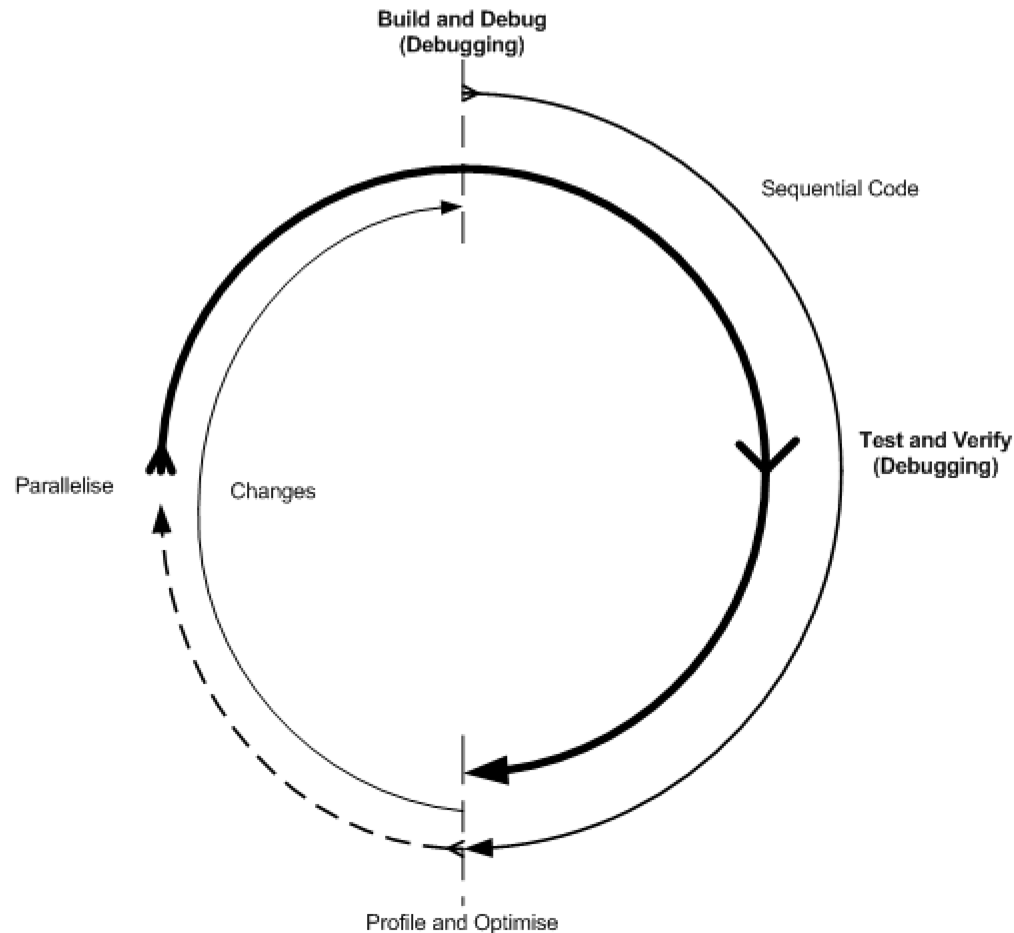
Debugging, Profiling and Optimising Scientific Codes

Wadud Miah
Research Computing Group

Scientific Code Performance Lifecycle



Debugging Scientific Codes



Software Bugs

- A bug in a program is an unwanted or unexpected property of software that causes it to behave incorrectly;
- Debugging is the practice of locating and fixing bugs in programs;
- Everyone has to debug as no one writes perfect software first time round;
- There are a number of ways to debug and using a debugger is one way.

Types of Bugs

- Syntax errors – code does not conform to rules of language;
- Floating point – divide by zero, over- and under-flow;
- Memory – array out of bounds, bad memory allocation;
- Logic – infinite loops, if conditions;
- Parallel codes – deadlock and race conditions.

Available Tools

- Syntax errors – compiler;
- Floating point – compiler and GDB;
- Memory – compiler, GDB, Valgrind, DDT;
- Logic – GDB and DDT;
- Parallel – DDT.

Floating Point Representation (1)

- The IEEE-754 standard is used to represent floating points numbers $(-1)^{s}\ (1.m) \times 2^{e-127}$ where s = sign, m = mantissa (or significand) and e = exponent;
- Floating point numbers are represented by finite precision, so rounding occurs: near, trunc, +inf and –inf;
- Evaluation order is important: $a + (b + c) \neq (a + b) + c$ using finite precision arithmetic;

Floating Point Representation (2)

- Every floating point operation returns a result;
- This can be a normal number where error is bounded;
- Or can be subnormal between zero and the smallest number;
- +0.0, -0.0, +inf, -inf;
- NaN (not a number): 0/0, log(-1), (-1)**0.5.

Floating Point Exceptions

- Underflow – result is a subnormal number;
- Overflow – result is ±inf;
- Division by zero;
- Invalid operation – sqrt(-1), 0.0/0.0, inf, -inf;
- Loss of precision which occurs very often, e.g. 2.0/3.0;
- By default, most compilers do not stop when exceptions occur.

Trapping Floating Point Exceptions

- Allows the stopping of execution of a program when exceptions occur;
- PGI: -Ktrap=fp;
- GNU: -ffpe-trap=zero,overflow,invalid
- Intel: -fpe0;
- The above switches to be used when compiling and linking.

Linux Signals

- When programs raise exceptions, a signal is sent to the executing process causing it to terminate in special ways;
- Common signals: SIGFPE (8) – floating point exception; SIGSEGV (11) segmentation violation; SIGKILL (9) kill program; SIGTERM (15) terminate nicely; SIGABRT (6) abort;
- SIGSEGV, SIGABRT, SIGFPE – terminate and creates core dump; SIGKILL, SIGTERM – terminate.

Core Files

- A core file contains the memory image of a program in execution and the value of its variables and subroutines;
- Core files are produced by the Linux operating system to help users debug their programs;
- To enable the creation of core files, type:

```
ulimit -c unlimited
```

- Note: core files can be huge!

Debuggers

- GDB: GNU debugger - most common;
- IDB: Intel debugger;
- PGDB: PGI debugger;
- Compile using the `-g` flag to enable the inclusion of debugging information;
- Switch off all optimisations using `-O0` (i.e. capital oh zero). Need to explicitly specify this flag as some compilers switch on optimisations by default.

Debugging Methods

- Interactive debugging allows user to walk through code execution;
- Breakpoints allow the debugger to stop when reaching a line in the code;
- Watchpoints allow the debugger to stop when a variable changes;
- Post-mortem debugging allows user to go to problematic line in code after code has crashed.

Practical 1 – Using GDB

- Interactive debugging:

```
gfortran -g program1.f90 \
            -o program1
gdb ./program1
```

  Step through code and print variable values;
- Post-mortem debugging:

```
gfortran -g program2.f90 \
            -o program2; ./program2
gdb ./program2 ./core.PID
```

  where *PID* is the process ID. Find the cause of the crash.

Compiler Memory Debugging

- Commands below enable array bounds checking which has an overhead;
- `gfortran -g -fbounds-check \`
  `program.f90`
- `ifort -check bounds program.f90`
- `pgf90 -C program.f90`
- Once bug has been located and fixed, re-compile without array bounds checking as the program runs faster.

Valgrind Memory Debugging

- Provides a suite of tools for debugging and profiling executables;
- Compile without optimisation and with debugging information on (using the `-g` flag);
- memcheck is the most common, which checks for: segmentation faults, accessing memory after freeing, using un-initialised variables, double freeing, memory leaks and memory overlapping.

Practical 2 - Using Valgrind

- Load the valgrind module:

```
module load valgrind/3.6.1
```

- Compile with debugging:

```
gfortran -g program3.f90 \
         -o program3
valgrind ./program3
```

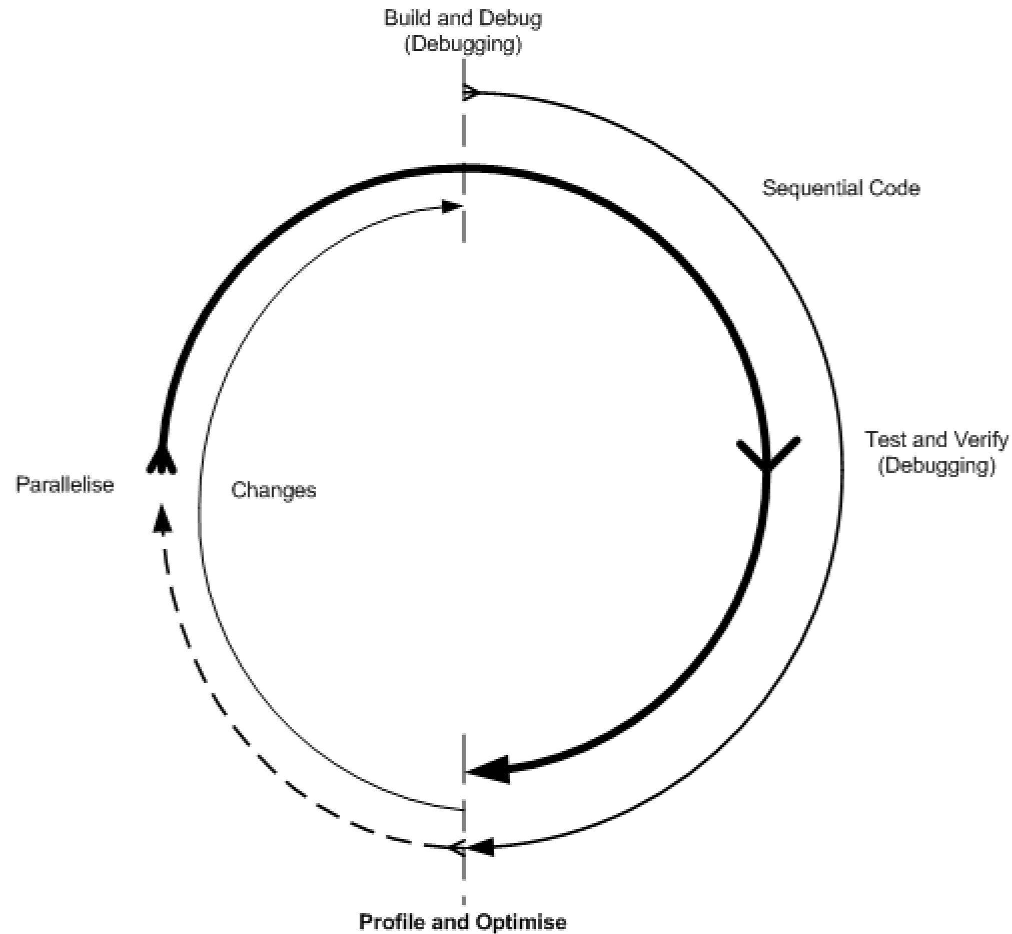
Debugging - Reference

"The Art of Debugging with GDB, DDD and Eclipse", N. Matloff and P. Salzman;

GDB: http://sources.redhat.com/gdb/

Valgrind:
http://valgrind.org/docs/manual/QuickStart.html

Profiling Scientific Codes



Profiling - Motivation (1)

- Now that you know how to debug your code, you now have a working code which can be further improved;
- Scientific codes can exploit faster compute architectures as new CPUs are being released;
- To do this, a user must focus on parts of codes that can benefit from optimisation;
- Profiling monitors the code's behaviour during execution;

Profiling - Motivation (2)

- Helps identify compute intensive parts of codes even if you are aware of the code;
- Focus can be put on such parts of the code for greater speed up and scalability;
- Optimise sequential versions of your code first to ensure optimal performance, and then parallelise;
- Profiling will help you optimise your code for sequential execution.

Profiling Methods

- Statistical profiling gives a high level overview of application performance;
- Instrumentation method gives finer details about performance, e.g. cache hits, floating point operations per second;
- Start with statistical methods, and if further performance is required, use instrumentation methods.

Profiling Tools

	GProf	PAPI	Scalasca
Portability	Yes	Yes	Yes
Robustness	Not so much for parallel codes	Yes	Yes
Metrics	Flat profiles Call graphs Timing	Hardware counters	Gprof + PAPI
Intrusive	No	YES	Yes with PAPI
Data collection	Statistical	Instrumentation using hardware counters	Statistical/ Tracing
Serial/parallel codes	Serial	Serial	Parallel

Compiling for Profiling

- To compile codes for profiling, use:

```
gfortran -pg -o code code.f90
```

or in two steps (compiling and linking):

```
gfortran -pg -c code.f90 -o code.o
gfortran -pg code.o -o code
```

- When program ends successfully, the file `gmon.out` is produced;
- Older files are over-written, so rename/move if required.

Gathering Statistics for Profiling

- When executing code to create `gmon.out`, execute a "typical" run;
- Ensure input parameters and arguments represent a typical execution of your program;
- To summarise a number of runs with gprof, use:

```
gprof --sum gmon.out.1 gmon.out.2
```

This will create a `gmon.sum` file which is the summary of two runs.

Creating GProf Reports

- Basic usage of gprof is:

  `gprof program [gmon.out] [> output]`

- Useful options are:

  `gprof -l` : line by line profiling

  `gprof -e func` : do not print information `func` and its children

  `gprof -f func` : only print information about `func` and its children

- Above switches can be combined.

Analysing GProf Reports

- Flat profile - how much time each function spent and how many times it was executed;
- Functions sorted by the most expensive;
- Call graph - shows which functions called other functions;
- Distinguish between a) functions that are slow because they do a lot of work b) functions that are slow because they call many other functions c) functions that are slow because they call slow functions.

Flat Profile Report (1)

- Example output:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 100.0      0.01      0.01        1    10.06    10.06  mysub_
  0.00      0.01      0.00        1     0.00    10.06  MAIN__
```

- `% time` - percentage of total running time;
- `cumulative seconds` - running sum of number of seconds accounted for by this function and those listed above;
- `self seconds` - number of seconds accounted for this function only;

Flat Profile Report (2)

- `calls` - number of times this function was invoked;
- `self ms/call` - average number of milliseconds spent in this function per call;
- `total ms/call` - average number of milliseconds spent in this function and its children per call;
- `name` - name of function.

Call Graph Report (1)

- Example output:

```
index % time     self  children    called     name
                 0.00    0.01       1/1           main [3]
[1]    100.0     0.00    0.01       1             MAIN__ [1]
                 0.01    0.00       1/1           mysub_ [2]
-----------------------------------------------
                 0.01    0.00       1/1           MAIN__ [1]
[2]    100.0     0.01    0.00       1             mysub_ [2]
```

- `index` - a unique number given to a function;
- `% time` - time that was spent in this function and its children;
- `self` - total time spent in this function only;

Call Graph Report (2)

- `children` - total time propagated into this function by its children;
- `called` - number of times the function was called;
- `name` - name of function.

GNU Coverage

- GNU coverage (`gcov`) analyses number of times a line is executed in a program;
- To enable coverage, use the command:

```
gcc -fprofile-arcs -ftest-coverage \
prog.c
```

- Then execute and type `gcov prog.c` which creates `prog.c.gcov` which is a text file;
- This produces an annotated file with number of times a line was executed followed by the line.

Performance Application Program Interface (PAPI)

- Fine grained profiling of scientific codes;
- Requires source code modifications - code instrumentation method;
- Reads CPU hardware counters (registers) that measure cache misses, floating point operations;
- Helps the user understand why the code is running slowly at the hardware level;

PAPI Hardware Counters

PAPI counter	Description
PAPI_L1_DCH	Level 1 data cache hits
PAPI_L2_DCH	Level 2 data cache hits
PAPI_L3_TCH	Level 3 total cache hits
PAPI_TOT_INS	Total instructions completed
PAPI_VEC_INS	Vector instructions
PAPI_FP_OPS	Floating point operations

PAPI - Fortran Example

```
include 'f90papi.h'
integer events(2), numevents, ierr;
integer *8, values(2)

numevents = 2
events(1) = PAPI_FP_INS
events(2) = PAPI_TOT_CYC
call PAPIF_start_counters( events, numevents, ierr );
! do some work

call PAPIF_read_counters( values, numevents, ierr );
! do some more work

call PAPIF_stop_counters( values, numevents, ierr );
! ierr should PAPI_OK
```

Scalasca

- Open source GUI based profiling tool;
- Provides statistical and instrumentation methods of profiling;
- Supports C, C++ and Fortran;
- OpenMP and MPI parallel profiling;
- If you're interested in using this tool, then please let me know!

Practical 1 - using gprof

- Compile with profiling information:

```
gcc -pg program4.c \
     -o program4
./program4
gprof ./program4 gmon.out
```

- Compile with instrumentation:

```
gcc -fprofile-arcs -ftest-coverage \
program5.c -o program5
./program5
gcov program5.c
cat program5.c.gcov
```

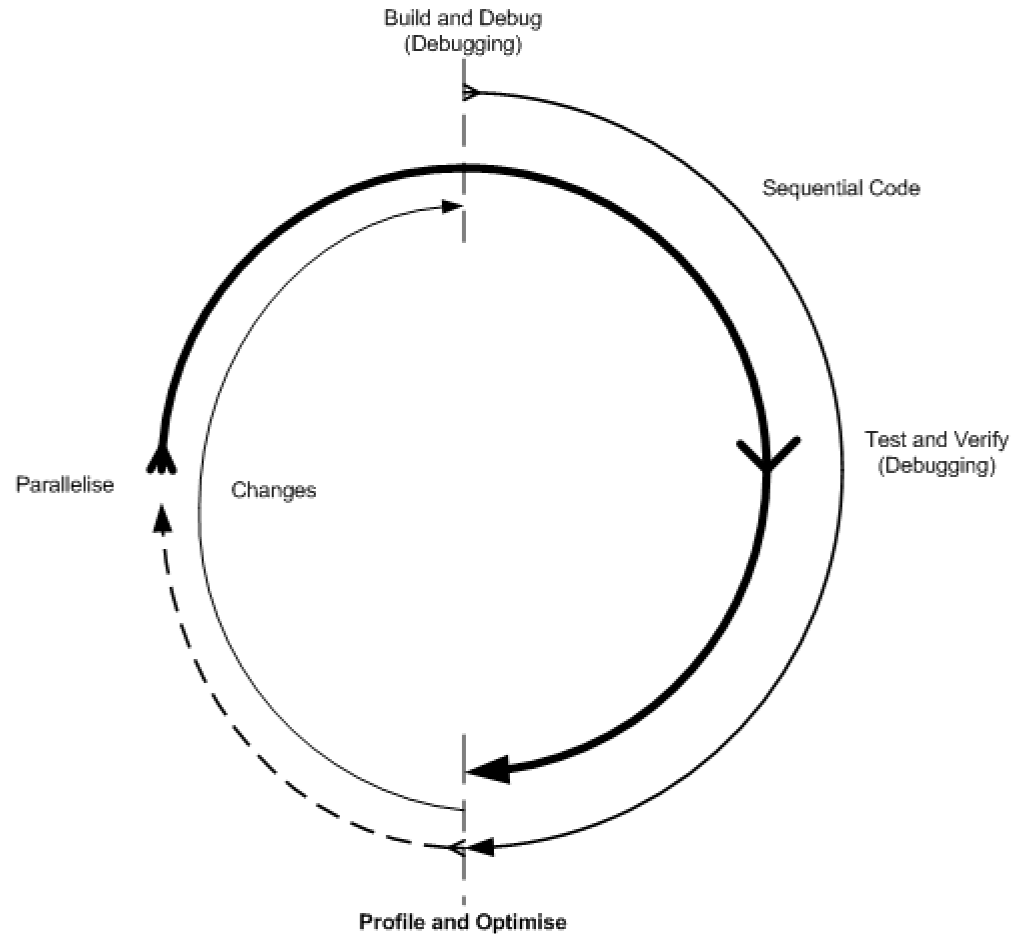
Profiling - Reference

Gprof:
http://sourceware.org/binutils/docs/gprof/

PAPI: http://icl.cs.utk.edu/papi/

Scalasca: http://www.scalasca.org/

Optimising Scientific Codes



Optimisation - Motivation

- Scientific codes not only need to be correct, they also have to perform sufficiently well;
- They subsequently have to be efficient: $R_{max} \rightarrow R_{peak}$;
- Efficiency of around 70% to 80% should be sought (depending on the algorithm);
- Reduce time to results - run more experiments;
- Increase the computational domain or refine it - accurately reflecting natural phenomena.

Strategy for Optimisation

- Choose the right stable algorithm, e.g. $O(n^2)$ or $O(\log n)$ is (obviously) better than $O(n^3)$;
- Use binary file formats, e.g. NetCDF and HDF5, if required;
- Use the best compiler and use the right flags;
- Select your computational units, e.g. CPU or GPU;
- Use available numerical libraries;
- Manually optimise sequential parts of your code - *a prerequisite to parallelism!*

Computational Architectures

- The most common type of CPU is the CISC (complex instruction set computer): Intel Xeon, AMD Opteron;
- Another is the RISC (reduced instruction set computer) CPU: IBM PowerPC;
- EPIC (explicitly parallel instruction computing) CPU: Intel Itanium 2;
- Another is a RISC type: the GPU (graphical processing unit) - great for highly parallel codes.

CPU Architectures (1)

Main Memory

Central Processing Unit

IU1 IR1

IU2 IR2

SSE IR3

FU1 FR1

FU2 FR2

SSE2 FR3

L1 cache

L2 cache

L3 cache

Static RAM

Hard Disk

Backup Tape

Dynamic RAM

time + size

CPU Architectures (2)

- Level 3 cache (static RAM);
- Level 2 cache - faster than level 3 cache;
- Level 1 cache - faster than level 2 cache;
- Registers - zero cycle transfer from processing units, hence fastest;
- Floating point units (single and double precision) and integer units;
- SIMD processing/vector units;
- Instructions are processed at rate of clock cycles (Hz);

Data Types

- Double precision operations require more cycles than single precision operations;
- Single precision operations require more cycles than integer precision;
- Use data types carefully and default to single precision unless more accuracy is required;
- Some applications can work with single precision whereas some require double precision for numerical stability and accuracy.

Memory Access (1)

- Memory is much slower than CPU, so creates a bottleneck in computational architectures;
- C and C++ uses row major ordering;
- Fortran and Matlab uses column major ordering;
- In C and C++, use the code:

```
for ( i = 0; i < Ni; i++ ) {
   for ( j = 0; i < Nj; j++ ) {
      A[i][j] = f( A[i][j], B[i][j] );
   }
}
```

Memory Access (2)

- In Fortran and Matlab, use the code:

```
do j = 0, Nj
   do i = 0, Ni
      A(i,j) = f( A(i,j), B(i,j) );
   end do
end do
```

- Use 1D arrays instead of structs or linked lists;
- Ensure data is contiguous, thus increasing data re-use in cache: increasing spatial locality.

Function Invocation

- Function calls consume around 4 CPU cycles, so ensure they do something useful as stack has to be prepared and arguments copied;
- Inline small functions by prefixing it with the `inline` key word (for C and C++);
- Or use compiler flags to inline functions;
- Inlining replaces the function call with the body of the function;
- Results in a larger binary, but storage is cheaper than CPU cycles.

Conditional Branching (1)

- IF statements reduce instruction streamlining, so avoid them if you can;
- If required, have the first statement as the most likely scenario;
- Examples of this is in for loops - assumes loops condition is true N times, and when it is false, it discards the execution;
- Take IF conditions out of loops;

Conditional Branching (2)

```
for ( i = 0; i < Ni; i++ ) {
  if ( a > 0 ) x[i] = 1.0;
  else x[i] = 0.0;
}
```

Re-write to:

```
if ( a > 0 ) {
  for ( i = 0; i < Ni; i++ ) x[i] = 1.0;
} else {
  for ( i = 0; i < Ni; i++ ) x[i] = 0.0;
}
```

Loop Unrolling (1)

- CPUs have superscalar units so they can execute more than one instruction per cycle;
- Units are called streaming SIMD extensions - SSE, SSE2, SSE3, SSE4 and AVX;
- This is a form of parallelism in sequential codes;
- Compilers can also intelligently unroll loops for you, but you need to use the right flags;

Loop Unrolling (2)

- Loops can be written in this form:

```
for ( i = 0; i < Ni; i += 4 ) {
  U[i] = V[i] + 1.0;
  U[i+1] = V[i+1] + 1.0;
  U[i+2] = V[i+2] + 1.0;
  U[i+3] = V[i+3] + 1.0;
}
```

- If you unroll too much, code will run slower and values will “spill” into main memory.

Floating Point Operations (1)

- Type casting is an expensive task, e.g.

```
dt = dt + 1
i = i + 1.0

dt = dt + 1.0
i = i + 1
```

- CPUs allow fused-multiply add operations in one instruction. Instead of:

```
a = c*d + b
```

write:

```
a = b + c*d
```

Floating Point Operations (2)

- Instead of:

```
r = s**2.0d0
```

use:

```
r = s**2 ! expanded to r = s*s
```

- Instead of:

```
r= -( 1.0d + b )
```

write:

```
r = -1.0d - b;
```

Numerical Libraries (1)

- Vendors have written highly efficient and robust libraries for maths functions;
- Highly optimised for different CPU architectures → use the correct one for your CPU;
- Before writing your own subroutine, see if one already exists - good chance there is so do use an available one;

Numerical Libraries (2)

- BLAS - scalar, vector and matrix operations;
- ATLAS - linear algebra solvers;
- LAPACK - linear algebra factorisations;
- NAG - linear algebra, PDE solvers, random numbers, fast Fourier transform;
- GNU Scientific Library - differential and integral equations, linear algebra and other maths subroutines;
- Intel MKL - linear algebra, fast Fourier, vector maths. Optimised for Intel architectures.

Compilers

- A number of compilers are available that add various levels of optimisation: -O1 (level 1), -O2, -O3 and sometimes -O4;
- List of compilers:
  - GNU compilers;
  - Portland group: very good local and global optimisations;
  - Intel compiler: very good for Intel CPUs.
- Use a proprietary compiler as they are much better than GNU for optimisation.

Level 1 Optimisations (-O1)

- Local block optimisation of code;
- Constant folding: a = 3 + 2 = 5 is evaluated at compilation;
- Common sub-expression elimination `a = x*y + x*y` → `a = t + t` (where `t = x*y`);
- Strength reductions: multiplication and division are replaced by bit shift operations;
- Redundant load and store elimination: redundant assignments are removed, e.g. a = 1; a = b + c; first assignment is removed.

Level 2 Optimisations (-O2)

- Global optimisation of code;
- Branch elimination: removing redundant IF statements;
- Constant propagation: replacing variables with constants;
- Copy propagation: removing references to unwanted variables;
- Dead store elimination: removing unreachable code;
- Invariant code motion: removing code that does not depend on loop.

Level 3 Optimisations (-O3)

- Performs more aggressive levels of optimisation;
- Fused multiply-add instructions are added;
- Short loops are expanded into individual statements;
- Loop unrolling - sequential code is vectorised;
- Function inlining - function calls are replaced by body of function.

Conclusion

- To optimise your code, learn a bit about your architecture and compiler;
- There are lots of compiler documentation available;
- Time your code to ensure benefits:

```
$ time ./a.out
real 0m3.002s; user 0m0.000s; sys 0m0.001s
```

- With a bit of optimisation, you can gain huge performance benefits to ensure scalable codes.

Optimisation - Reference

“Software Optimization for High Performance Computing”, K. Wadleigh and I. Crawford

“Performance Optimization of Numerically Intensive Codes”, S. Goedecker and A. Hoisie

“Introduction to High Performance Computing for Scientists and Engineers”, G. Hager and G. Wellein

Practical 1 - Optimising Codes

- Open an interactive session: `interactive`
- Compile and time program6.f90

```
gfortran program6.f90 -o program6
time ./program6
```

- Re-order the loops so the `k` loop is first;
- Unroll the `k` indices `n` times;
- Cache block the loop.