Math3101/Math5303
10. Improving Performance of Julia Programs

A/Prof William McLean,
School of Maths and Stats, UNSW
Session 1, 2017

Version: May 23, 2017

Before trying to improve performance

- Choose the best algorithm(s) for the job.
- Use high quality libraries wherever possible (easy in Julia).
- Apply *test-first programming* to ensure your code works correctly (and stays that way).
- Aim to achieve a clean overall design by decomposing the problem into functions and designing composite types as appropriate.
- Organise functions into modules.
- Refactor code to make it simpler wherever possible.
- Keep a backup copy of a working version (or better still use a version control system like git).
- Isolate the most resource-intensive parts of your code in small functions (computational kernels).

Profiling tools

Use profiling tools to verify that identify bottlenecks.

We have seen how to use the `time()` function to measure the CPU time used in a code fragment:

```
start = time()
<stuff you want to time>
finish = time()
elapsed = finish - start
```

You can easily time a complete function (or any expression) using the `@elapsed` macro.

```
include("Elliptic_ver2.jl")
include("profile_Richardson.jl")
@elapsed resid=iterate(Richardson!)
```

The first time you execute a function, the elapsed time will include the time taken to compile your code. Subsequent function calls will therefore be faster.

Another useful macro is `@allocated`, which tells you the number of bytes allocated during evaluation of an expression.

```
1 @allocated resid=iterate(Richardson!)
```

An unusually large amount of memory usage is often a sign of some inefficiency.

The `@time` macro is often more convenient, because it combines the output of `elapsed` and `allocated`. More verbose output is given by `@timev`.

Avoid unnecessary memory allocation

We will use some functions defined in `memory_example.jl`.

Many array operations generate temporary array variables. For example, if A and B are 1024×1024 arrays, then doing (twice)

```
1 @time A += B;
2 @time add!(A, B);
```

shows that Julia allocated 8MB for a temporary array to hold $A + B$ in the first case, but allocated almost nothing in the second.

Now compare the memory usage of `first_expm` and `second_expm`. These functions both approximate the matrix exponential e^A by computing

$$\sum_{k=0}^{N} \frac{A^k}{k!} = I + A + \frac{A^2}{2!} + \cdots + \frac{A^N}{N!}.$$

To ensure that $\|A\| \leqslant 1/2$ so that $N = 15$ suffices, use

```
1 A = test_matrix(1024);
```

Getting more detailed information

The `@profile` provides a line-by-line sampling profiler.

```
include("Elliptic_ver2.jl")
include("profile_Cholesky.jl")
@time band_solve!(gr)
@time band_solve!(gr)
@profile band_solve!(gr)
Profile.print()
```

The number of samples for a given line of code should reflect the proportion of the runtime spent executing that line. You might find it easier to work with the alternative "flat" output format:

```
Profile.print(format=:flat)
```

The `ProfileView` package provides profiling information in a graphical format.

```
using ProfileView
ProfileView.view()
```

Array processing

The script `array_order.jl` illustrates the fact that data locality improves the efficiency of the CPU cache. Wherever possible, you should process data whose physical addresses are close together in RAM. In practice, this means organising your loops so that the first index of each array varies most rapidly, then the second, and so on. Thus, we should access a matrix column-by-column and not row-by-row.

By default, Julia generates code to check that all array indices remain in bounds. For example, doing

```
1 A = rand(3,3)
2 x = A[3,5]
```

will generate a `BoundsError`. The `@inbounds` macro tells Julia not to do this for the current statement, which speeds up the code but with unpredictable results if an index is ever out of bounds.

Type stability

Julia can generate fast machine code from a generic function like `f(x)=x^2`. Here, `x` could be an `Int64` or `Float64` or a square `Matrix{Float64}` or ....

Functions are a natural unit of optimisation because Julia knows the argument types at run time, and can compile accordingly.

Important to avoid changing the type of a variable. In the following function, `s` starts as an `Int64` but then changes to a `Float64`.

```
1  function harmonic(N)
2      s = 0
3      for n = 1:N
4          s += 1/n
5      end
6      return s
7  end
```

We should instead initialise `s` as `s=0.0`.

Single-instruction, multiple-data (SIMD) operations

Given a loop like

```
1  s = 0.0
2  for i = 1:N
3      s += a[i] * b[i]
4  end
```

modern CPUs are able to process several consecutive loop iterations simultaneously. Intel first introduced this capability in 1997 with its MMX (multimedia extensions a.k.a. matrix-math extensions) instruction set, subsequently expanded to SSE (streaming-SIMD extensions) and AVX (advanced vector extension).

These SIMD instructions can significantly reduce execution times.

SIMD operations are not possible if the order of the loop iterations is significant, such as in the following example:

```
a[1] = 1.0
for j = 1:N
    a[j+1] = 3 + 1 / a[j]
end
```

Use the `@simd` to tell Julia that it is safe to vectorise a loop.

```
s = 0.0
@simd for i = 1:N
    @inbounds s += a[i] * b[i]
end
```

The Julia documentation points out that `@simd` is an experimental feature that might change in future.

Comparison with Fortran and C

The module `Elliptic_ver2.jl` defines three extra versions of the `Richardson!` function.

- `Richardson_ver2!`: a pure Julia, SIMD version.
- `ser_Richardson!`: wrapper around a serial Fortran implementation.
- `par_Richardson!`: wrapper around a parallel (OpenMP) Fortran version.
- `C_Richardson!`: wrapper around a (serial) C version.

Compare the performance of these functions, starting with the original version:

```
1 include("Elliptic_ver2.jl")
2 include("profile_Richardson.jl")
3 @time resid=iterate(Richardson!);
4 @time resid=iterate(Richardson!);
```