Math3101/Math5303
11. Parallel programming

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

Version: May 29, 2017

Introduction

A process is a running instance of an executable program. At any given moment, a computer typically runs a large number of processes that time-share the available CPU cores. Most of these processes are single-threaded, running on one CPU core at any given instant, and use few system resources.

However, in scientific computing it is common for a long-running, single-threaded process to use 100% of a CPU core. The aim of parallel programming is to speed up the execution of such a program by using multiple cores.

Consider a serial program that runs for T_{ser} seconds and a parallel version of the program that produces the same (or nearly the same) results in T_{par} seconds. The ratio $T_{\mathrm{ser}}/T_{\mathrm{par}}$ is called the parallel speedup. At best, if the parallel program uses N processor cores, we might hope for a speedup of N.

Parallel hardware

A typical personal computer has 2 or 4 CPU cores. A server motherboard might have two CPUs, each with 8 cores. A cluster computer consists of a number of processor nodes each with several CPU cores, and connected with a high-speed network.

The cores on a single node can access the RAM on that node quickly, whereas data transfers between cores on different nodes are much slower (but still fast compared to ordinary networks).

Parallel programming for a moderate number of cores on a single node can use a shared memory approach. A more complicated distributed memory approach is needed to work with multiple nodes.

Trivial parallelism

One easy use case for parallel computing is to run multiple instances of the same serial program with different inputs simultaneously.

The Fortran program `maxeig` uses the power method to compute the largest eigenvalue of the $N \times N$ matrix $[a_{ij}]$ with entries

$$a_{ij} = \frac{1 + |i-j|^{\alpha-1} h^{\alpha}}{\Gamma(\alpha)}, \qquad h = 1/N.$$

The program takes two command-line arguments: N and α. For example, after using the makefile to build the program, you can do

```
./maxeig 500 0.5
```

and find that the largest eigenvalue is 1.418839 when $N = 500$ and $\alpha = 0.5$.

The shell scripts `run_serial.sh` and `run_parallel.sh` both compute the largest eigenvalue for $N = 10,000$ and two different choices of α.

```
time sh run_serial.sh
time sh run_parallel.sh
```

The only difference is that `run_parallel.sh` runs `maxeig` in the background by appending & to the shell command, with the result that the second instance of `maxeig` starts without waiting for the first instance to finish.

Open the system monitor and compare the CPU loads for the two scripts.

The module `MaxEig.jl` implements the power method in Julia and uses it for the same matrix as in the Fortran program `maxeig.f08`.

The Julia script `run_serial.jl` uses the map function to call `maxeig` with two sets of arguments. The second script `run_parallel.jl` instead uses the pmap (parallel map) function.

Julia workers

The -p 4 flag tells Julia to start with 4 worker processes (which would be appropriate for a PC with 4 CPU cores). Try

```
julia -p 2
```

and then

```
@everywhere println("hello from process ", myid())
```

The master process has id number 1, and the 4 worker processes have id numbers 2–3.

The -L flag loads a file on all processors. On a dual-core laptop, we do

```
julia run_serial.jl
julia -p 2 -L MaxEig.jl run_parallel.jl
```

Parallel libraries

The easiest way to achieve (non-trivial) parallel execution is via libraries. Julia's dense matrix operations use a parallel BLAS library by default when the matrices are sufficiently large.

Monitor the loads on the CPU cores when Julia executes the following commands.

```
A = rand(5000,5000);
B = rand(5000,5000);
@time C = A*B;
BLAS.set_num_threads(2)
@time C = A*B;
BLAS.set_num_threads(1)
@time C = A*B;
```

Julia's fast Fourier transform functions call the single-threaded FFTW library by default, but you can use multithreaded versions if they are available on your system.

```julia
A = rand(10000,10000);
B = rand(10000,10000);
C = complex(A, B);
@time fftw!(C)
FFTW.set_num_threads(2)
@time fftw!(C)
FFTW.set_num_threads(4)
@time fftw!(C)
```

A Monte-Carlo simulation

Consider the 1D boundary-value problem

$$\begin{aligned} -\big(\kappa(x,\omega)u_x(x,\omega)\big)_x &= 0 \quad \text{for } 0 < x < L, \\ -\kappa(x,\omega)u_x(x,\omega) &= \gamma \quad \text{at } x = 0, \\ u(x,\omega) &= 0 \quad \text{at } x = L, \end{aligned} \tag{1}$$

where $\omega = (\omega_1, \omega_2, \ldots, \omega_M)$ is a vector of independent random variables, each uniformly distributed in the interval $[-1/2, 1/2]$, and

$$\kappa(x,\omega) = \bar{\kappa}(x) + \sum_{m=1}^{M} \omega_m \psi_m(x).$$

The basis functions $\psi_n(x)$ are of the form

$$\psi_m(x) = \frac{C}{m^s} \sin \frac{m\pi}{L} x.$$

The expected value of $\kappa(x)$ is

$$\mathbb{E}(\kappa(x)) = \int_{[-1/2,1/2]^M} \kappa(x,\omega)\,d\omega = \bar{\kappa}(x),$$

and the variance of $\kappa(x)$ is

$$\begin{aligned}
&\mathbb{E}\big([\kappa(x) - \bar{\kappa}(x)]^2\big) \\
&\quad = \sum_{m=1}^{M} \sum_{m'=1}^{M} \frac{A^2}{m^s (m')^s} \int_{[-1/2,1/2]^M} \omega_m \omega_{m'}\,d\omega \\
&\quad = \sum_{m=1}^{M} \frac{A^2}{m^{2s}} \int_{-1/2}^{1/2} \omega_m^2\,d\omega_m = \frac{A^2}{12}\,\zeta(2s).
\end{aligned}$$

Our aim is to compute the expected value of $u(0)$, that is,

$$\mathbb{E}(u(0)) = \int_{[-1/2,1/2]^M} u(0,\omega)\,d\omega.$$

We assume that $\bar{\kappa}(x)$ and C satisfy

$$\bar{\kappa}(x) \geqslant \frac{C\zeta(s)}{2} + \epsilon,$$

thereby ensuring that

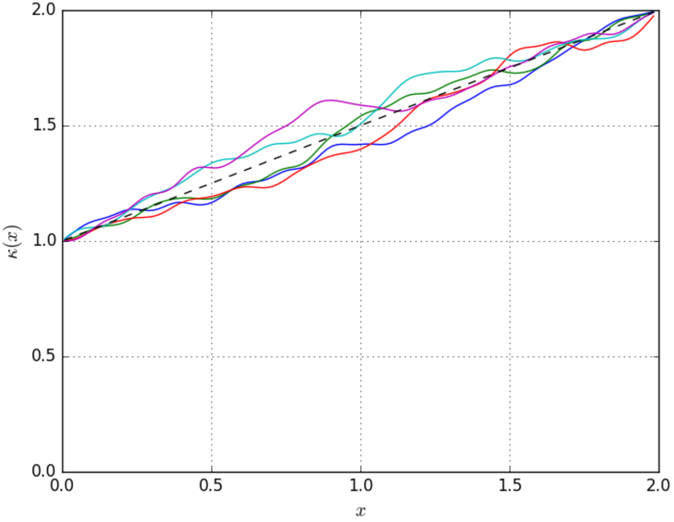
$$\kappa(x,\omega) \geqslant \bar{\kappa}(x) - \frac{C}{2}\sum_{m=1}^{M}\frac{1}{m^s} \geqslant \bar{\kappa}(x) - \frac{C}{2}\zeta(s) \geqslant \epsilon$$

for $0 \leqslant x \leqslant L$ and $\omega \in [-1/2, 1/2]^M$.

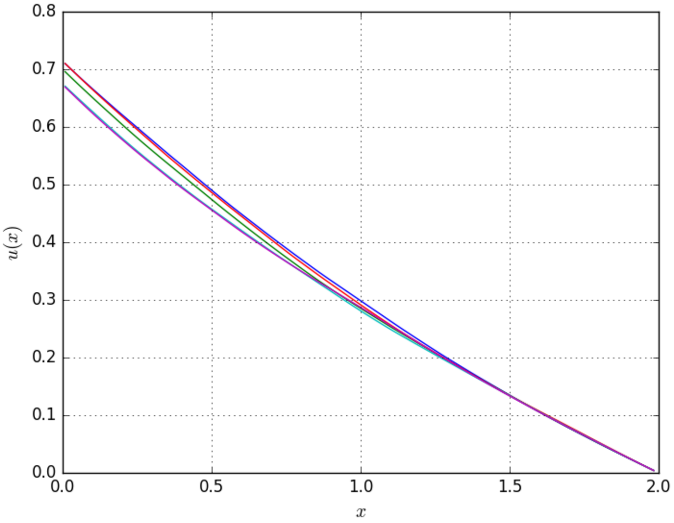
For our computations, we take

$$\bar{\kappa}(x) = 1 + \frac{x}{L}.$$

Five samples of $\kappa(x)$



Corresponding samples of $u(x)$



Reduction

Monte-Carlo simulations are an important and particularly simple use case for parallel computing, because the desired mean can be computed using a standard operation known as a parallel reduction.

The script `serial_MC.jl` estimates $\mathbb{E}\big(u(0)\big)$ using $50,000$ trials. Compare the CPU time required with that for the parallel version `parallel_MC.jl`.

The parallel reduction construct

```
s = @parallel (+) for n = 1:N
        ...
end
```

evaluates the loop body for each n. The sum s is formed by adding together the values produced by the final statement of the loop body for each iteration. The iterations must have no dependencies, since they are farmed out to the worker processes and evaluated concurrently.