SF2568: Parallel Computations for Large-Scale Problems

Lecture 3: Parallel Techniques and Introduction to Fortran/C

January 22, 2024

Acknowledgements

These slides are an extension of slides by Michael Hanke and Niclas Jansson.

Mandelbrot set

Definition

Let f be a function of two complex variables. For any $z \in \mathbb{C}$ consider the iteration

$$z_{k+1} = f(z_k, d).$$

A **Mandelbrot set** is a set of points d in the complex plane that are quasi-stable, i.e., the sequence $\{z_k\}$ remains bounded for $z_0 = 0$.

Problem

Visualise a Mandelbrot set in the complex plane for the function

$$f(z) = z^2 + d.$$

Algorithm

- Assume that we have N different colours $c_1, \ldots, c_N$
- Choose a bound $b > 0$
- For each d, iterate f:
  - If $|z_n| < b$ after N iterations, assume d to be quasi-stable and assign colour c_N to d
  - If $|z_n| \geq b$ and $|z_{k-1}| < b$, assign colour c_k to d

Note: All quasi-stable points must be within a circle with its center at the origin and a radius of b

Sequential Code

We use MATLAB syntax for brevity

```
function count = cal_pixel(d, b, N)
  count = 1; % Initialize counter
  z     = 0; % Initialize sequence z

  % Iterative through sequence
  while ( abs(z) < b ) && (count < N)
    % Determine next value in sequence
    z     = z*z+d;
    count = count + 1; % Increment counter
  end % End of while loop
end % End of cal_pixel
```

Sequential Code

Assume a viewing area of $w \times h$ pixels. This area must be mapped to the square$[-b, b] \times [-b, b]$. The pixel (x, y) is mapped to d according to

```
dx    = 2*b/(w-1);       % Step size in x
dy    = 2*b/(h-1);       % Step size in y
dreal = x*dx-b;          % Real part
dimag = y*dy-b;          % Imaginary part
d     = dreal+1i*dimag;  % Pixel mapped to d
```

Sequential Code

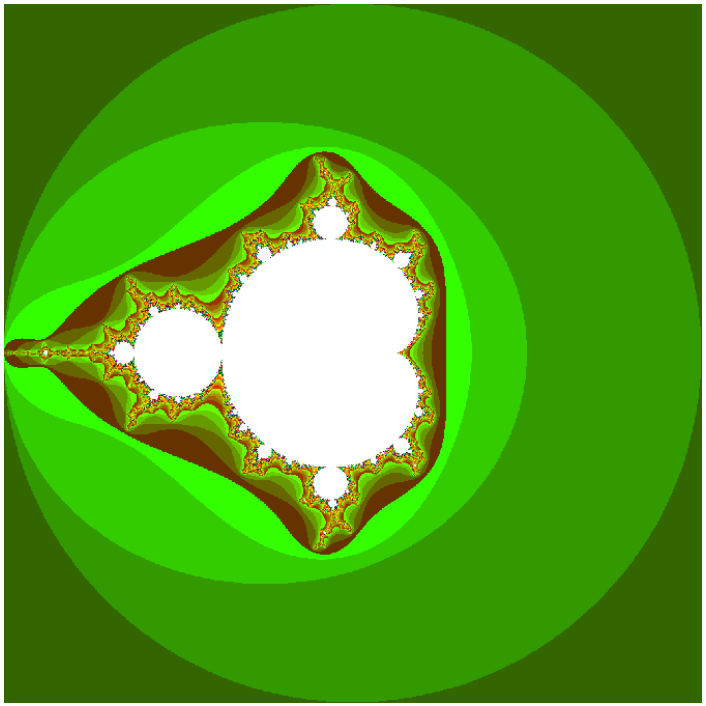
The complete code:

```
% Parameters
N  = ...;        % Max number in sequence
b  = ...;        % Bound
w  = ...;        % Figure width
h  = ...;        % Figure height
dx = 2*b/(w-1); % Step size in x
dy = 2*b/(h-1); % Step size in y
% Initialize color matrix
colvals = zeros(w, h);

for y = 1:h % Loop through figure height
    % Imaginary part
    dimag = y*dy-b;
    for x = 1:w % Loop through figure width
        % Real part
        dreal = x*dx-b;
        % Pixel mapped to d
        d     = dreal+1i*dimag;
        % Evaluate for colour
        colvals(x, y) = cal_pixel(d, b, N);
    end % End of x loop
end % End of y loop

% Generate image
image(colvals')
```

For $b = 2$, $N = 256$,
$w = h = 2048$



Mandelbrot – Parallelization

Observations

- The innermost computations (including cal_pixel) are completely independent of each other
- The computational domain can naturally be split into disjoint subdomains

This algorithm is embarrassingly parallel!

Parallelisation Strategy

- Assume that we have PQ processors
- Subdivide the viewing area into PQ rectangles and assign the computation to the processors
- Each rectangle has dimensions $w_p \times h_q$. The computation time on processor (p, q) is

$$t_{comp,pq} = \alpha \, N \, w_p \, h_q \, t_a$$

where t_a is the time for one arithmetic operation.
The factor $0 < \alpha \leq 1$ is slightly dependent of N, w, h, p, q (number of iterations)

Implementation

- Let's make life easier and set $Q = 1$. Moreover, assume that w is divisible by P
- Then each processor gets a vertical strip of the viewing area

$$w_p = \frac{w}{P}, \qquad h_p = h$$

- The offsets (coordinates of the upper left corner of the rectangles) are

$$x_{off} = p \cdot \frac{w}{P}, \qquad y_{off} = 0$$

Implementation: SPMD

```
% p and P are known to each process
% w, h, N, b are assumed to be initialized
dx   = 2*b/(w-1); % Step size in x
dy   = 2*b/(h-1); % Step size in y
wp   = w/P;       % Number of steps of width on local processor
hp   = h;         % Number of steps of height on local processor
xoff = p*w/P;     % Local offset in x
yoff = 0;         % Local offset in y
for x = 0:wp-1 % Loop through local width
  % Real part
  dreal = (x+xoff)*dx-b;
  for y = 0:h-1 % Loop through local height
    % Imaginary part
    dimag = (y+yoff)*dy-b;
    % Pixel mapped to d
    d = dreal+i*dimag;
        % Evaluate for colour
        colvals(x, y) = cal_pixel(d, b, N);
    end % End of y loop
end % End of x loop

% Generate image ...
```

Implementation: SPMD

```
if p == 0 % I am the leader
  for q = 1:P-1
    colorvals(q*wp:(q+1)*wp-1,1:h) = receive(q);
  end
display(color);
else % I am a follower
  send(0,color);
end
```

Performance Evaluation

- Assume that all processes start at the same time
- The first synchronization point is the send/receive operations,

$$t_{comp} = \max_{0 \leq p < P} (\alpha N h w / P) t_a = (\alpha N h w / P) t_a$$

- There are $P - 1$ communication steps, each transferring hw/P data items

$$t_{comm} = (P - 1)(t_{startup} + t_{data} h w / P)$$

- Hence,

$$T_P = (\alpha N h w / P) t_a + (P - 1)(t_{startup} + t_{data} h w / P)$$

Speedup

- In this case, $T_S^* \approx T_1$ such that

$$S_P = \frac{\alpha N h w t_a}{(\alpha N h w / P) t_a + (P-1)(t_{startup} + t_{data} h w / P)}$$

$$= P \left(\frac{1}{1 + \frac{P(P-1)}{\alpha N h w} \frac{t_{startup}}{t_a} + \frac{(P-1)}{\alpha N} \frac{t_{data}}{t_a}} \right)$$

Conclusion

The speedup is nearly optimal if $\frac{P(P-1)}{\alpha N h w} \frac{t_{startup}}{t_a} \ll 1$ which holds if

- either $hw \gg P$,
- or N is very large

Brief Introduction to Fortran/C

- You should be able to read and understand a C or Fortran program, so that you can modify it
- You should be able to write a program in C or Fortran, given that you know how to do it in another programming language

Note

Anything is allowed, such as reading the manual-pages and books, browsing the internet and asking questions

Introduction

- Both Fortran and C are procedural languages, and must be compiled
- C is one of the most widely used programming languages (General purpose programming language)
  - Systems programming (operating systems, embedded systems)
  - Compilers and Interpreters (Perl, Python, . . .)
  - Scientific computing (libraries e.g. FFTW, accelerators e.g. OpenCL)
  - And of course numerous end-user applications
- Fortran comes from FORmula TRANslation, originally developed for easy implementation of mathematical formulae, vector, and matrix operations
- Fortran is not case sensitive

Fibonacci Sequence: MATLAB

```
clear;
n = 10;
f = zeros(n, 1); % Useful, not necessary in MATLAB though
f(1:2) = 1;
for i = 3:n
  f(i) = f(i-1) + f(i-2);
end
fprintf('%8d\n', f)
```

Fibonacci Sequence: Fortran

```
program fibonacci
  IMPLICIT NONE
  INTEGER, PARAMETER :: n = 10
  INTEGER, DIMENSION(n) :: f
  INTEGER :: i
  f(1:2) = 1
  do i = 3, n
    f(i) = f(i-1) + f(i-2)
  end do
  write(*,'(10I8)') f
end program fibonacci
```

Fortran Program

- States with `program` and ends with `end program`
- Variables must be declared, and type must be stated!
- Variables are declared at the top of the program
- Always include `IMPLICIT NONE` when initializing the program. Ensures that undeclared variables are illegal

Fibonacci Sequence: C

```c
include <stdio.h>
#define N 10 /* Convention: Use caps: N */
int main(int argc, char **argv) {
  int i, f[N];
  f[0] = 1;
  f[1] = 1;
  for (i = 2; i < N; i++) {
    f[i] = f[i-1] + f[i-2];
  }
  for (i = 0; i < N; i++)
    printf("%8d", f[i]);
  printf("\n");
}
```

Compiling

- Both Fortran and C are **compiled** languages instead of interpreted
  - MATLAB and Python run through an interpreter
  - Compiling code creates an executable file
- Some typical C compilers: `c`, `clang`, **gcc**, `icc`
- Some typical Fortran compilers: `absoft`, `ifort`, **gfortran**

Compiling: Fortran

Fortran

```
gfortran fibonacci.f90 -o fibonacci
./fibonacci
```

- `-o` allows you to name your executable (default if not present is `a.out`)
- Use `-O0` for no optimization, which is useful for debugging
- Use `-O`, `-O2`, `-O3`, `-Ofast` for optimization
- Optimization usually makes the program faster, but not always
- `-fcheck-bounds` Used for debugging (useful for finding segmentation faults)

Compiling: C

C

```
gcc -Wall -o prog prog.c -lm
./prog
```

- `-Wall` enables compiler's warning messages
- `-o` allows you to name your executable (default if not present is `a.out`)
- `-lm` links math.h library
- Use `-O`, `-O2`, `-O3`, `-Ofast` for optimization
- *Always check for correctness of output when using optimization!*

Debugging

Fact

The best and most efficient way of debugging is writing a clear and well structured code.

- Together with your code, develop a debugging strategy
- Several debugging tools exists (e.g. DDD, or DDT for parallel computing)

Debugging

Fact

The best and most efficient way of debugging is writing a clear and well structured code.

- Together with your code, develop a debugging strategy
- Several debugging tools exists (e.g. DDD, or DDT for parallel computing)

Performance Tools

- A profiler gives time spent in various functions (subroutines)
- **gprof** (read the manual...)
  - Compile with `-pg` in C
  - Run `prog`
  - Run `gprof ./prog > prog.prof`
  - Look at statistics in `prog.prof`
- gprof can't be used for parallel programs on Dardel
  - Several other profiles provided e.g. CrayPat, Score-P

The Memory Hierarchy

- Most parallel systems are built from CPU with a memory hierarchy
  - Registers
  - Primary cache
  - Secondary cache
  - Local memory
  - Remote memory - access through the interconnection network
- **As you move down this list, the time to retrieve data increases by about an order of magnitude for each step**
  - Make efficient use of local memory (caches)
  - Minimize remote memory references

Performance Tuning - Cache Locality

- The basic rule for efficient use of local memory (caches):
  Use a memory stride of one
- This means array elements are accessed in the same order they are stored in memory
- Fortran uses **"column-major"** order
  - Want the **leftmost** index in a multi-dimensional array varying most rapidly in a loop
  - Remember indexing of arrays begin at 1 by default
- C uses **"row-major"** order
  - Want the **rightmost** index in a multi-dimensional array varying most rapidly in a loop
  - Remember indexing of arrays begin at 0 by default
- Interchange nested loops if necessary (and possible!) to achieve the preferred order

Accessing Memory: C

Which of the following is faster in C?

```
for (i = 0; i < 10000; i++)
  for (j = 0; j < 10000; j++)
    sum += a[i][j];
```

```
for (j = 0; j < 10000; j++)
  for (i = 0; i < 10000; i++)
    sum += a[i][j];
```

What about in Fortran?