1st Assignment - Intro to Parallel Computing

Bryce Mazurowski

October 10, 2023

NCSA Advanced Parallel Computing Cohort Fall 2023
Due date: October 11, 2023

1 Describe the two pure ways of parallelism (data and task). Which one of them has the most capability to speed up computationally intensive workloads?

- Data Parallelism: When a single task is executed on a chunk of data that is split into pieces based on the number of threads available for the task
- Task Parallelism: When multiple independent tasks are executed on a single data source. Each task can be assigned a thread and computed concurrently.
- In general, data parallelism is more capable of speeding up computationally intensive workflows. When working with a large dataset, it is quicker to partition the data and work on a given task concurrently. Task parallelism is also more limited if one task depends on the operations in a different task.

2 What do you perceive as the best advantage of OpenMP? What is its main limitation?

The biggest advantage of OpenMP is its widespread availability. It can be used on any operating system with basically any compiler. The fact that it is simple to work with is a close second.

OpenMP's biggest limitation is that it is only effective on shared memory machines. It cannot handle communication on distributed memory systems.

3 Write/find a simple serial program with a loop (ask ChatGPT if you can't find one).

1. Parallelize the loop using OpenMP directives
2. Measure the time it takes to complete the loop with a different number of OpenMP threads (1 to 16 suggested)
3. Repeat b) 3-6 times, take the averages, and calculate the speedup factor for each number of threads
4. Plot your results
5. Discuss the scalability of your parallelization. Is it far from ideal? Why?

3.1 Code

The code takes a vector of complex numbers and computes the number of iterations for the Mandelbrot set.

$$f(z) = z^2 + c$$

where z and c are complex numbers. z is initialized to zero and the components of c are randomly generated.

3.1.1 Serial

```
#include <iostream>
#include <vector>
#include <complex>
#include <sys/time.h>

#define ORD 1 << 27

using std::cout;
using std::endl;

typedef std::complex<float> cFloat;

int mandelbrot(cFloat& c);

int main() {
  // instantiate vector of
  std::vector<cFloat> v;
  float a, b;
  srand(time(NULL));
  for (int iPos = 0; iPos < ORD; iPos++) {
    // make 2 random doubles
    a = rand() % RAND_MAX;
    a = float(a) / (RAND_MAX);
    b = rand() % RAND_MAX;
    b = float(b) / (RAND_MAX);
    // make our complex number
    cFloat z(a, b);
    // add it to the vector
    v.push_back(z);
  }

  // stopwatch variables
  struct timeval startTime, stopTime, elapsedTime;

  // now call saxpy
  gettimeofday(&startTime, NULL);
  std::vector<cFloat>::iterator vPos = v.begin();
  std::vector<cFloat>::iterator vEnd = v.end();
  for (; vPos != vEnd; vPos++) {
    std::cout << "vPos = " << (*vPos) << std::endl;
    // calculate setVal
    const int iTs = mandelbrot(*vPos);
    std::cout << "iTs = " << iTs << std::endl;
  }

  gettimeofday(&stopTime, NULL);

  timersub(&stopTime, &startTime, &elapsedTime);
  cout << "Number of args: " << v.size() << "Elapsed time (s): "
       << elapsedTime.tv_sec + elapsedTime.tv_usec / 1000000.0 << endl;
```

```
  return 0;
}

int mandelbrot(cFloat &c) {
  cFloat z(0.0, 0.0);
  const unsigned int maxIt = 200;
  for (unsigned int iMan = 0; iMan < maxIt; iMan++) {
    if (std::abs(z) > 2.0) {
      // outside of magnitude bounds
      return iMan;
    }
    // square z
    z = z * z + c;
  }
  return maxIt;
}
```

3.1.2 Parallel

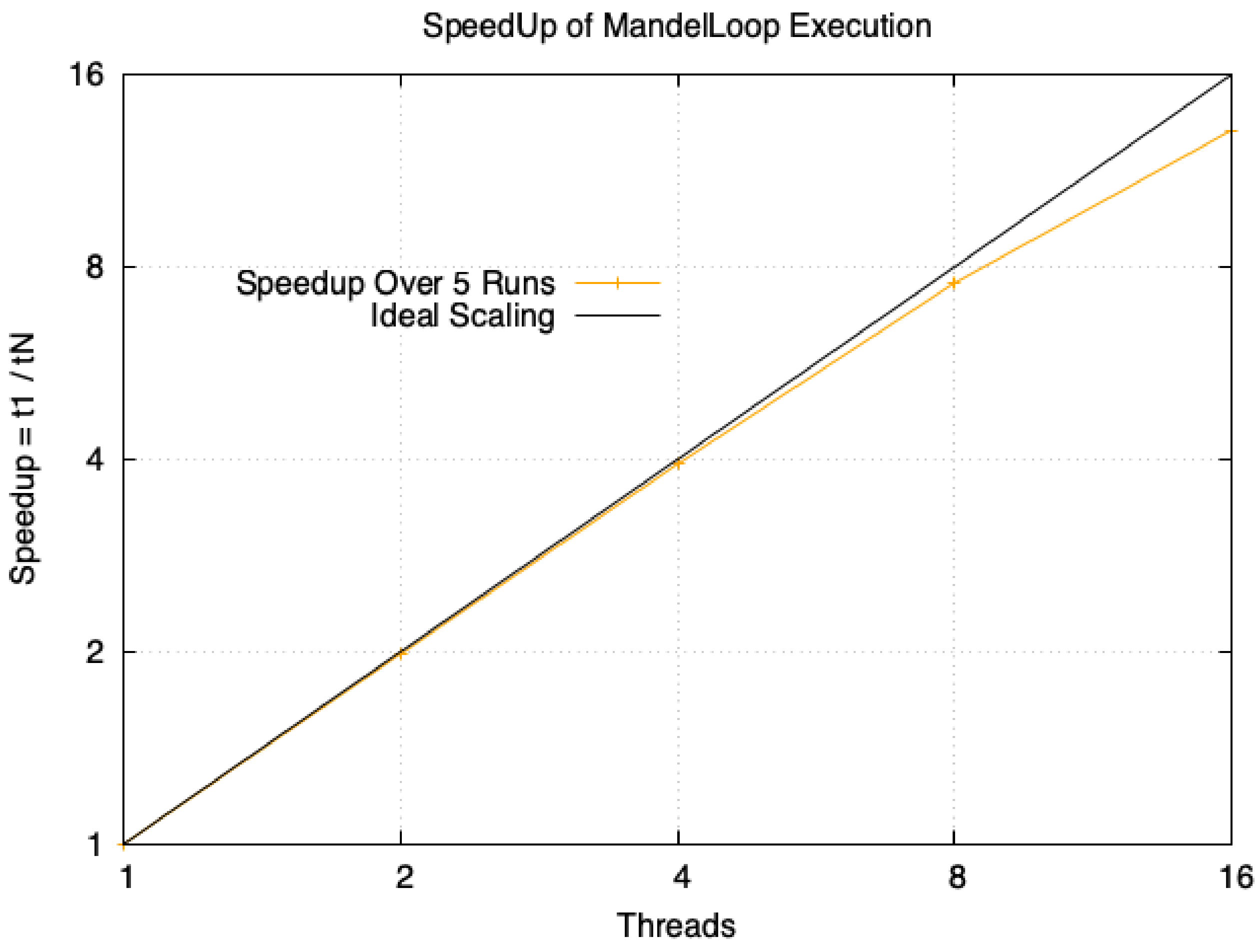
```
#include <iostream>
#include <vector>
#include <complex>
#include <sys/time.h>
#include "omp.h"

using std::cout;
using std::endl;

typedef std::complex<float> cFloat;

int mandelbrot(cFloat& c);

int main(int argc, char* argv[]) {
  int ORD = 1 << 10;
  if (argc != 1) {
    // ran with 1 << 20
    ORD = 1 << std::stoi(argv[1]);
  }
  // instantiate vector of complex numbers
  std::vector<cFloat> v;
  float a, b;
  srand(time(NULL));
  for (int iPos = 0; iPos < ORD; iPos++) {
    // make 2 random doubles
    a = rand() % RAND_MAX;
    a = float(a) / (RAND_MAX);
    b = rand() % RAND_MAX;
    b = float(b) / (RAND_MAX);
    // make our complex number
    cFloat z(a, b);
    // add it to the vector
    v.push_back(z);
  }
```

```
  // stopwatch variables
  struct timeval startTime, stopTime, elapsedTime;

  gettimeofday(&startTime, NULL);
  // note: must initialize iterator in loop
  // or code segFaults
  #pragma omp parallel
  {
    #pragma omp for
    for (std::vector<cFloat>::iterator vPos = v.begin(); vPos < v.end(); ++vPos) {
      // calculate setVal
      const int iTs = mandelbrot(*vPos);
    }
  }

  gettimeofday(&stopTime, NULL);

  timersub(&stopTime, &startTime, &elapsedTime);
  cout << "Number of args: " << v.size() << " Elapsed time (s): "
       << elapsedTime.tv_sec + elapsedTime.tv_usec / 1000000.0 << endl;

  return 0;
}

int mandelbrot(cFloat &c) {
  cFloat z(0.0, 0.0);
  const unsigned int maxIt = 200;
  for (unsigned int iMan = 0; iMan < maxIt; iMan++) {
    if (std::abs(z) > 2.0) {
      // outside of magnitude bounds
      return iMan;
    }
    // square z
    z = z * z + c;
  }
  return maxIt;
}
```

3.2 Results

Run on a vector of 1048576 complex numbers.

threads	run1	run2	run3	run4	run5	avg	speedup
1	4.27445	4.30016	4.28323	4.26816	4.26952	4.279104	1.
2	2.14344	2.17865	2.15289	2.1612	2.14505	2.156246	1.9845157
4	1.08021	1.08807	1.08875	1.09224	1.09027	1.087908	3.9333326
8	0.57001	0.566089	0.564286	0.564507	0.56827	0.5666324	7.5518167
16	0.31242	0.337258	0.361114	0.300969	0.32295	0.3269422	13.088258



The speedup of the program is pretty good. Even with 16 threads it is close to ideal. Although 8 is maybe a better ideal number of threads. This may scale even better with a larger vector, but when I tried the cluster booted me because it took too long with 1 thread.

I think it works well because the task is pretty intensive, sometimes taking 200 iterations for a single input. In serial we have to wait for the that to finish. In parallel we get to work while that happens.

One reason it may not be ideal at higher thread counts is that I am not initializing the vector of complex variables in parallel. This takes time especially for a vector with over a million components. That time is effectively constant for all of the thread counts, and as the time for the Mandelbrot iteration calc goes down it becomes a more significant portion of the execute time.