Adding Vectors and Computing Dot Product in Parallel

Sam Magura

February 10, 2014
Definitions

Let \( \mathbf{u}, \mathbf{v} \) be vectors in \( \mathbb{R}^n \).

Addition:

\[
(u_1, \ldots, u_n) + (v_1, \ldots, v_n) = (u_1 + v_1, \ldots, u_n + v_n)
\]

Dot product:

\[
\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i
\]
Why does this matter?

Many algorithms need fast vector operations
  - For example: Conjugate Gradients
  - Iterative method for solving linear systems

Conjugate Gradients uses:
  - Dot product of a vector with itself
  - Vector addition and subtraction
  - Scalar multiplication
Outline

Question: How to do addition and dot product in parallel?

- Part 1: Vector addition
- Part 2: Dot product
- Part 3: Measuring the speedup from parallelism
Part 1: Vector addition

\( \mathbf{u} \) and \( \mathbf{v} \) are vectors

- with length \( n \)

The sequential version

```c
void vector_add(double *result, double *u, double *v, int n){
    for(int i = 0; i < n; i++){
        result[i] = u[i] + v[i];
    }
}
```

The parallel version

```c
void vector_add(double *result, double *u, double *v, int n){
    #pragma omp parallel for
    for(int i = 0; i < n; i++){
        result[i] = u[i] + v[i];
    }
}
```
Part 2: Dot Product

The sequential version

```c
double vector_dot_nopara(double *u, double *v, int n){
    double result = 0;

    for(int i = 0; i < n; i++){
        result += u[i] * v[i];
    }

    return result;
}
```
Dot product in parallel

The parallel version

```c
double vector_dot(double *u, double *v, int n) {
    double result = 0;

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        result += u[i] * v[i];
    }

    return result;
}
```

Does this work?
Atomic/critical

No! May give the wrong answer

- Problem if two threads assign to result at same time

Solution: atomic and critical

- Only let one thread execute the code at a time

```c
double vector_dot_atomic(double *u, double *v, int n){
    double result = 0;

    #pragma omp parallel for
    for(int i = 0; i < n; i++){
        //or #pragma omp critical
        result += u[i] * v[i];
    }

    return result;
}
```
Reduction

Tell the compiler the purpose of the loop:

▶ to compute a sum

Allows compiler to optimize

▶ Compute local results then aggregate into global result

double vector_dot_reduction(double *u, double *v, int n){
    double result = 0;

    #pragma omp parallel for reduction(+:result)
    for(int i = 0; i < n; i++){
        result += u[i] * v[i];
    }

    return result;
}
Compiling

```bash
gcc -Wall -std=gnu99 -O3 -fopenmp ...
```

- `-O3` tells GCC to create an optimized executable
- Makes a big difference!
# Comparison

<table>
<thead>
<tr>
<th>Version</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>0.119s</td>
</tr>
<tr>
<td>reduction</td>
<td>0.053s</td>
</tr>
<tr>
<td>atomic</td>
<td>8.390s</td>
</tr>
<tr>
<td>critical</td>
<td>11.153s</td>
</tr>
</tbody>
</table>

- **Baseline**
- **Fast**
- **Super slow!**
- **Uber slow!!**

- On a 4-core computer
- Vector has length $10^8$ (0.8 GB of memory)
- Measured by averaging many trials
Why are atomic and critical so slow?

OpenMP has to do a lot internally to prevent simultaneous assignments
  ▶ Threads have to wait for each other

**Conclusion:**
Use **reduction** when computing sums!
OpenBLAS

Should also compare to an external library

- BLAS (Basic Linear Algebra Subroutines)

I chose OpenBLAS, which uses parallelism

- But dot product is computed sequentially :(
  - Highly optimized
  - Very complicated compared to my code
Comparison to OpenBLAS

Wow! OpenBLAS (1 thread) is almost as fast as reduction (4 threads)
Part 3: Measuring the speedup from parallelism

Want to compare parallel to sequential
  ▶ Parallel could be slower than sequential!

Want to try many different values of $n$
  ▶ Recall $n = \text{length of vectors}$
  ▶ Before, $n$ was fixed at $10^8$

Need a way to time our functions
Profiling tools

Unix `time` command
- Prints both real (elapsed) time and user (CPU) time

GNU `gprof` utility
- Shows time taken by each function

C standard library function `gettimeofday`
- Provides microsecond precision
Profiling

- One execution of function is too fast to time accurately
- Take average over \( k \) executions of function

- How to choose \( k \)?
  - \( k \) too big \( \leftarrow \) takes a long time
  - \( k \) too small \( \leftarrow \) inaccurate results

- Can be hard to find a good \( k \)
  - Good \( k \) depends on size of input
Relative speed

Definition:

$$\text{Relative speed} = \frac{\text{Time(parallel)}}{\text{Time(sequential)}}$$

Hopefully relative speed $> 1$
Why is parallelism ineffective for $n > 10^5$?
Results: dot product

Why the drop between $n = 10^{5.5}$ and $n = 10^6$?
Automate the process

Use Python to automate:

- Testing your program
- Generating plots

My Python script

1. Runs my C program for different $n$
2. Parses the output
3. Creates plot (using matplotlib)
4. Saves plot to file
Conclusions

Use **reduction** when computing sums

Profiling programs can be hard

Parallelism doesn’t help much for large vectors
  - for my code
  - I have no idea why
  - “large” means $n \geq 10^6$
Email me (srmagura@ncsu.edu)

If you have questions on
  ► Final projects
  ► C
  ► Python + matplotlib
  ► LaTeX
  ► Bagels
  ► My hair
  ► Super Smash Bros.