Math 4997-1

Lecture 6: Shared memory parallelism

Patrick Diehl

https://www.cct.lsu.edu/~pdiehl/teaching/2020/4997/

Notes

Notes

Reminder

Notes

Lecture 5

What you should know from last lecture

- Operator overloading
- Header and class files
- CMake

Notes

Shared memory parallelism

Notes

Definition of parallelism

- We need multiple resources which can operate at the same time
- We have to have more than one task that can be performed at the same time
- We have to do multiple tasks on multiple resources the same time

Notes

Amdahl's Law (Strong scaling) [1]

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

where S is the speed up, P the proportion of parallel code, and N the numbers of threads.

Example

A program took 20 hours using a single thread and only the part took one hour can not be run in parallel, we will get $P = 0.95$. So the theoretical speed up is $\frac{1}{(1-0.95)} = 20$.

Parallel computing with many threads is only beneficial for highly parallelizable programs.

Notes

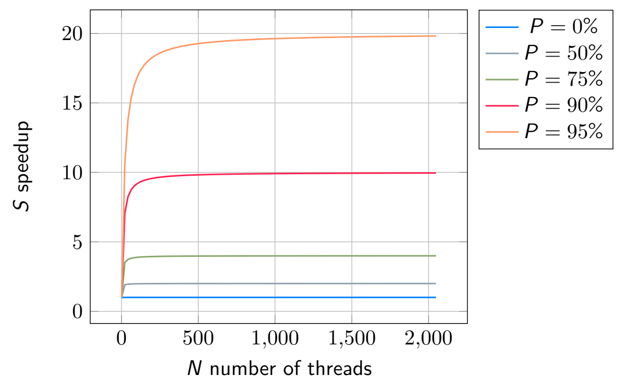


Figure: Plot of Amdahl's law for different parallel portions of the code.

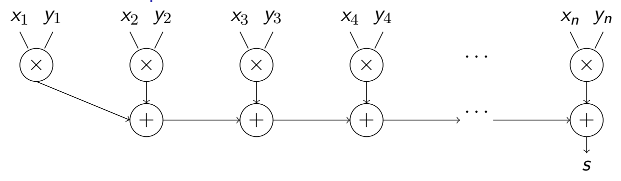
Notes

Example: Dot product

$$S = \mathbf{X} \cdot \mathbf{V} = \sum_{i}^{N} x_i y_i$$
$$\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$$
$$\mathbf{Y} = \{y_1, y_2, \ldots, y_n\}$$

$$S = (x_1 y_1) + (x_2 y_2) + \ldots + (x_n y_n)$$

Flow chart: Sequential

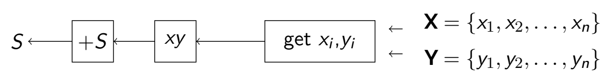


Notes

Parallelism approaches

Pipeline parallelism

- Used in vector processors
- Data passes between successive stages
- Used in execution pipelines in all general microprocessors
- Exploits
  - Fine grain parallelism
  - High clock speeds
  - Latency hiding


S
$+S$
xy
get x_i, y_i
$\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$
$\mathbf{Y} = \{y_1, y_2, \ldots, y_n\}$

More details [6]

Notes

Parallelism approaches

Single instructions and multiple data (SIMD)

- All perform same operation at the same time
- But may perform different operations at different times
- Each operates on separate data
- Used in accelerators on microprocessors
- Scales as long as data scales

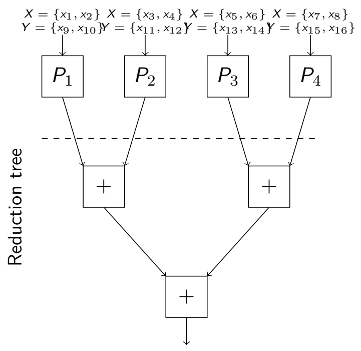
SIMD is part of Flynn's taxonomy, a classification of computer architectures, proposed by Michael J. Flynn in 1966 [4, 2].

Notes

Flow chart: SIMD

Algorithm

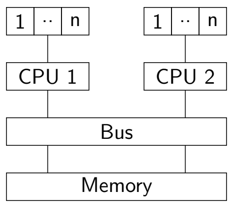
1. $S = 0$
2. Get x_{i+1}, y_{i+1}
3. Compute xy
4. Add to S
5. More data, go to 2
6. Send S to reduce
7. Stop


$X = \{x_1, x_2\}$ $X = \{x_3, x_4\}$ $X = \{x_5, x_6\}$ $X = \{x_7, x_8\}$
$Y = \{x_9, x_{10}\}$ $Y = \{x_{11}, x_{12}\}$ $Y = \{x_{13}, x_{14}\}$ $Y = \{x_{15}, x_{16}\}$
P_1
P_2
P_3
P_4
Reduction tree
$+$

Reduction tree: Exploits fine grain functions and need global communications

Notes

Uniform memory access (UMA)



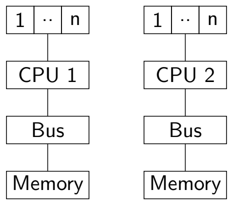
Access times

- Memory access times are the same

More details [3, 5].

Notes

Non-uniform memory access (NUMA)



Access time to the memory depends on the memory location relative to the CPU.

Access times

- Local memory access is fast
- Non-local memory access has some overhead

Notes

Parallel algorithms

Notes

Parallel algorithms in C++ 17[2]

- C++17 added support for parallel algorithms to the standard library, to help programs take advantage of parallel execution for improved performance.
- Parallelized versions of 69 algorithms from `<algorithm>`, `<numeric>` and `<memory>` are available.

Recently new feature!

Only recently released compilers (gcc 9 and MSVC 19.14)[1] implement these new features and some of them are still experimental.

Some special compiler flags are needed to use these features:

```
g++ -std=c++1z -ltbb lecture6-loops.cpp
```

[1] https://en.cppreference.com/w/cpp/compiler_support
[2] https://en.cppreference.com/w/cpp/experimental/parallelism

Notes

Example: Accumulate

```
std::vector<int> nums(1000000,1);
```

Sequential[3]

```
auto result = std::accumulate(nums.begin(),
                              nums.end(),
                              0.0);
```

Parallel[4]

```
auto result = std::reduce(
              std::execution::par,
              nums.begin(), nums.end());
```

Important: `std::execution::par` from `#include<execution>`[5]

[3] https://en.cppreference.com/w/cpp/algorithm/accumulate
[4] https://en.cppreference.com/w/cpp/experimental/reduce
[5] https://en.cppreference.com/w/cpp/experimental/execution_policy_tag

Notes

Execution time

Time measurements

```
g++ -std=c++1z -ltbb lecture6-loops.cpp
./a.out
std::accumulate result 9e+08 took 10370.689498 ms
std::reduce result 9.000000e+08 took 612.173647 ms
```

Notes

Execution policies

Notes

Execution policies

- `std::execution::seq`
  The algorithm is executed sequential, like `std::accumulate` in the previous example and using only once thread.
- `std::execution::par`
  The algorithm is executed in parallel and used multiple threads.
- `std::execution::par_unseq`
  The algorithm is executed in parallel and vectorization is used.

Note we will not cover vectorization in this course.

Fore more details: CppCon 2016: Bryce Adelstein Lelbach “The C++17 Parallel Algorithms Library and Beyond”[6]

[6] https://www.youtube.com/watch?v=Vck6kzWjY88

Notes

Be aware of: Data races and Deadlocks

Notes

Be aware of

With great power comes great responsibility!

You are responsible

When using parallel execution policy, it is the programmer's responsibility to avoid

- data races
- race conditions
- deadlocks

Notes

Data race

```
//Compute the sum of the array a in parallel
int a[] = {0,1};
int sum = 0;
std::for_each(std::execution::par,
              std::begin(a),
              std::end(a), [&](int i) {
  sum += a[i]; // Error: Data race
});
```

Data race:

A data race exists when multithreaded (or otherwise parallel) code that would access a shared resource could do so in such a way as to cause unexpected results.

Notes

Solution I: data races

`std::atomic`[7]

```
//Compute the sum of the array a in parallel
int a[] = {0,1};
std::atomic<int> sum{0};
std::for_each(std::execution::par,
              std::begin(a),
              std::end(a), [&](int i) {
  sum += a[i];
});
```

The atomic library[8] provides components for fine-grained atomic operations allowing for lockless concurrent programming. Each atomic operation is indivisible with regards to any other atomic operation that involves the same object. Atomic objects are free of data races.

[7] https://en.cppreference.com/w/cpp/atomic/atomic
[8] https://en.cppreference.com/w/cpp/atomic

Notes

Solution 2: data races

std::mutex[9]

```
//Compute the sum of the array a in parallel
int a[] = {0,1};
int sum = 0;
std::mutex m;
std::for_each(std::execution::par,
              std::begin(a),
              std::end(a), [&](int i) {
  m.lock();
  sum += a[i];
  m.unlock();
});
```

The mutex class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

[9] https://en.cppreference.com/w/cpp/thread/mutex

Notes

Race condition

```
if (x == 5)  // Checking x
{
   // Different thread could change x

   y = x * 2; // Using x
}
// It is not sure if y is 10 or any other value.
```

Race condition

A check of a shared variable within a parallel execution and another thread could change this variable before it is used.

Notes

Solution: Race condition

```
std::mutex m;

m.lock();
if (x == 5)  // Checking x
{
   // Different thread could change x

   y = x * 2; // Using x
}
m.unlock();
// Now it is sure that y will be 10
```

Race condition

A check of a shared variable within a parallel execution and another thread could change this variable before it is used.

Notes

Deadlocks

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

Example (Taken from[10])

Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time.

Example: lecture7-deadlocks.cpp

[10] https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html

Notes

Notes

Summary

Summary

Notes

After this lecture, you should know

- Shared memory parallelism
- Parallel algorithms
- Execution policies
- Race condition, data race, and deadlocks

Further reading:

C++ Lecture 3 - Modern Paralization Techniques[11]: OpenMP for shared memory parallelism and the Message Passing Interface for distributed memory parallelism. Note that HPX which will we cover after the midterm is introduced there.

[11] https://www.youtube.com/watch?v=1DUW5Qw3eck

Notes

References

References I

Notes

[1] Gene M Amdahl.
Validity of the single processor approach to achieving large scale computing capabilities.
In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[2] Ralph Duncan.
A survey of parallel computer architectures.
Computer, 23(2):5–16, 1990.

[3] Hesham El-Rewini and Mostafa Abd-El-Barr.
Advanced computer architecture and parallel processing, volume 42.
John Wiley & Sons, 2005.

[4] Michael J Flynn.
Some computer organizations and their effectiveness.
IEEE transactions on computers, 100(9):948–960, 1972.

References II

[5] Georg Hager and Gerhard Wellein.
Introduction to high performance computing for scientists and engineers.
CRC Press, 2010.

[6] Michael Quinn.
Parallel Programming in C with MPI and OpenMP.
McGraw-Hill Science/Engineering/Math, 2003.

Notes

Notes

Notes

Notes