




**Parallel Programming:
Background Information and Tips**



**Oregon State
University**
Mike Bailey
mjb@cs.oregonstate.edu



parallel_background.pptx



mjb - March 22, 2021


1

Three Reasons to Study Parallel Programming

1. Increase performance: do more work in the same amount of time
2. Increase performance: take less time to do the same amount of work
3. Make some programming tasks more convenient to implement

Example:
Decrease the time to compute a simulation

Example:
Create a web browser where the tasks of monitoring the user interface, downloading text, and downloading multiple images are happening simultaneously



mjb - March 22, 2021

2

**Three Types of Parallelism:
1. Instruction Level Parallelism (ILP)**

A program might consist of a continuous stream of assembly instructions, but it is not necessarily executed continuously. Oftentimes it has "pauses", waiting for something to be ready so that it can proceed.

```
A = B + 1;
C = 3;
```

If B is not already in cache, this instruction will **block** while B is fetched from memory


```
Load  B,r0
Add   $1,r0
Store r0,A
Load  $3,r1
Store r1,C
```

Out-of-order execution capability will slide instructions up if they can be executed while waiting for the block to end

```
Prefetch B
Load   $3,r1
Store  r1,C
Load   B,r0
Add    $1,r0
Store  r0,A
```

If a compiler does this, it's called **Static ILP**
If the CPU chip does this, it's called **Dynamic ILP**

This is all good to know, but it's nothing we can control much of.



mjb - March 22, 2021

3

**Three Types of Parallelism:
2. Data Level Parallelism (DLP)**


Executing the same instructions on different parts of the data

```
for( i = 0; i < NUM; i++ )
{
    B[ i ] = sqrt( A[ i ] );
}
```

```
for( i = 0; i < NUM/3; i++ )
{
    B[ i ] = sqrt( A[ i ] );
}
```

```
for( i = NUM/3; i < 2*NUM/3; i++ )
{
    B[ i ] = sqrt( A[ i ] );
}
```

```
for( i = 2*NUM/3; i < NUM; i++ )
{
    B[ i ] = sqrt( A[ i ] );
}
```



mjb - March 22, 2021

4

Three Types of Parallelism: 3. Thread Level Parallelism (TLP)

Executing *different* instructions

Example: processing a variety of incoming transaction requests

Different Tasks/Functions

In general, TLP implies that you have more threads than cores

Thread execution switches when a thread blocks or uses up its time slice

Oregon State University
Computer Graphics

mjb - March 22, 2021

5

Flynn's Taxonomy

$\left\{ \begin{matrix} \text{Single} \\ \text{Multiple} \end{matrix} \right\}$ Instruction, $\left\{ \begin{matrix} \text{Single} \\ \text{Multiple} \end{matrix} \right\}$ Data

Data →

SISD "Normal" single-core CPU	SIMD GPUs, Special vector CPU instructions
MISD ?????	MIMD Multiple processors running independently

Instructions ↓

Oregon State University
Computer Graphics

mjb - March 22, 2021

6

Von Neumann Architecture: Basically the fundamental pieces of a CPU have not changed since the 1960s

The "Heap" (the result of a *malloc* or *new* call), is in here, along with Globals and the Stack

Other elements:

- Clock
- Registers ←
- Program Counter ←
- Stack Pointer ←

} These together are the "state" of the processor

Oregon State University
Computer Graphics

mjb - March 22, 2021

7

What Exactly is a Process?

Processes execute a program in memory. The process keeps a state (program counter, registers, and stack).

Other elements:

- Clock
- Registers
- Program Counter
- Stack Pointer

Oregon State University
Computer Graphics


mjb - March 22, 2021

8

When is it Good to use Multithreading?

- Where specific operations can become blocked, waiting for something else to happen
- Where specific operations can be CPU-intensive
- Where specific operations must respond to asynchronous I/O, including the user interface (UI)
- Where specific operations have higher or lower priority than other operations
- To manage independent behaviors in interactive simulations
- When you want to accelerate a single program on multicore CPU chips

Threads can make it easier to have many things going on in your program at one time, and can absorb the dead-time of other threads.



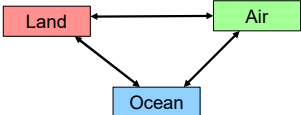
mjb - March 22, 2021

13

Two Ways to Decompose your Problem into Parallelizable Pieces

Functional (or Task) Decomposition

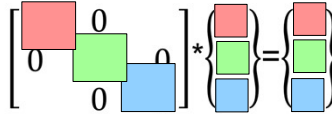
Breaking a task into sub-tasks that represent separate functions. A web browser is a good example. So is a climate modeling program:




"Thread Parallel"

Domain (or Data) Decomposition

Breaking a task into sub-tasks that represent separate sections of the data. An example is a large diagonally-dominant matrix solution:



"Data Parallel"



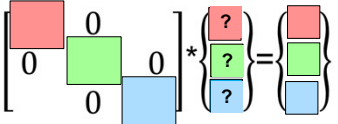
mjb - March 22, 2021

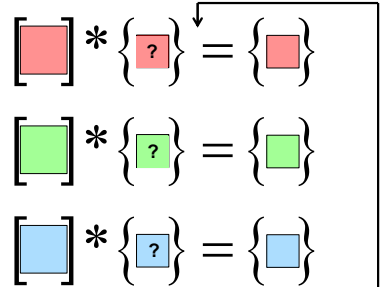
14

Data Decomposition Reduces the Problem Size per Thread


Example: A diagonally-dominant matrix solution

- Break the problem into blocks
- Solve within the block
- Handle borders separately after a Barrier





Barrier
Share results across boundaries



mjb - March 22, 2021

15

Some Definitions

Atomic An operation that takes place to completion with no chance of being interrupted by another thread

Barrier A point in the program where *all* threads must reach before *any* of them are allowed to proceed

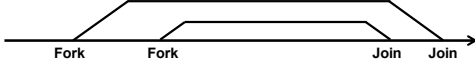
Coarse-grained parallelism Breaking a task up into a small number of large tasks

Deterministic The same set of inputs always gives the same outputs


Dynamic scheduling Dividing the total number of tasks T up so that each of N available threads has *less than* T/N sub-tasks to do, and then doling out the remaining tasks to threads as they become available

Fine-grained parallelism Breaking a task up into lots of small tasks

Fork-join An operation where multiple threads are created from a main thread. All of those forked threads are expected to eventually finish and thus "join back up" with the main thread.



C Fork Fork Join Join



mjb - March 22, 2021

16

Some More Definitions

17

Private variable After a fork operation, a variable which has a private copy within each thread

Reduction Combining the results from multiple threads into a single sum or product, continuing to use multithreading. Typically this is performed so that it takes $O(\log_2 N)$ time instead of $O(N)$ time:

Shared variable After a fork operation, a variable which is shared among threads, i.e., has a single value

Speed-up(N) T_1 / T_N
Speed-up Efficiency $\text{Speed-up}(N) / N$

Static Scheduling Dividing the total number of tasks T up so that each of N available threads has T/N sub-tasks to do



mjb - March 22, 2021

17

Parallel Programming Tips

18



mjb - March 22, 2021

18

Tip #1 -- Don't Keep Internal State

19

```
int
GetLastPositiveNumber( int x )
{
    static int savedX;
    if( x >= 0 )
        savedX = x;
    return savedX;
}
```

← Internal state

If you do keep internal state between calls, there is a chance that a second thread will hop in and change it, then the first thread will use that state thinking it has not been changed.

Ironically, some of the standard C functions that we use all the time (e.g., `strtok`) keep internal state:

```
char * strtok ( char * str, const char * delims );
```

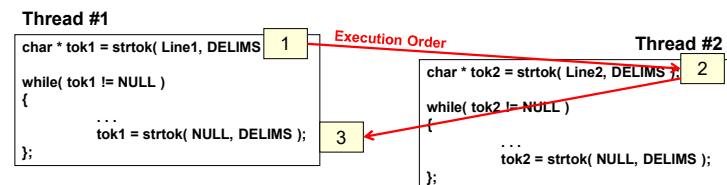


mjb - March 22, 2021

19

Tip #1 -- Don't Keep Internal State

20



1. Thread #1 sets the internal character array pointer to somewhere in Line1[].
2. Thread #2 resets the same internal character array pointer to somewhere in Line2[].
3. Thread #1 uses that internal character array pointer, but it is not pointing into Line1[] where Thread #1 thinks it left it.



mjb - March 22, 2021

20

Tip #1 -- Keep External State Instead


21

Moral: if you will be multithreading, don't use internal static variables to retain state inside of functions.

In this case, using strtok_r is preferred:

```
char * strtok_r( char *str, const char *delims, char **sret );
```

strtok_r returns its internal state to you so that you can store it locally and then can pass it back when you are ready. (The 'r' stands for "re-entrant".)



mjb - March 22, 2021

21

Tip #1 -- Keep External State Instead

22

Thread #1

```
char *retValue1;
char * tok1 = strtok_r( Line1, DELIMS, &retValue1 );


while( tok1 != NULL )
{
    ...
    tok1 = strtok( NULL, DELIMS, &retValue1 );
};
```

Thread #2

```
char *retValue2;
char * tok2 = strtok( Line2, DELIMS, &retValue2 );

while( tok2 != NULL )
{
    ...
    tok2 = strtok( NULL, DELIMS, &retValue2 );
};
```

Execution order no longer matters!



mjb - March 22, 2021

22

Tip #1 -- Note that Keeping Global State is Just as Dangerous

23

Internal state:

```
int
GetLastPositiveNumber( int x )
{
    static int savedX;

    if( x >= 0 )
        savedX = x;


    return savedX;
}
```

Global state:

```
int savedX;

int
GetLastPositiveNumber( int x )
{
    if( x >= 0 )
        savedX = x;

    return savedX;
}
```



mjb - March 22, 2021


23

Tip #2 -- Avoid Deadlock

24

Deadlock is when two threads are each waiting for the other to do something

Worst of all, the way these problems occur is not always deterministic!



mjb - March 22, 2021

24

Tip #3 – Avoid Race Conditions

25

- A Race Condition is where it matters which thread gets to a particular piece of code first.
- This often comes about when one thread is modifying a variable while the other thread is in the midst of using it

A good example is maintaining and using the pointer in a stack data structure:

Thread #1: Pushing:
`p++;`
`*p = incoming;`

Thread #2: Popping:
`outgoing = *p;`
`p--;`

Worst of all, the way these problems occur is not always deterministic!

Oregon State University
Computer Graphics
mjb - March 22, 2021

25

BTW, Race Conditions can often be fixed through the use of Mutual Exclusion Locks (Mutexes)

26

Thread #1: Pushing:
`... MutexLock A`
`{`
`1 p++;`
`2 *p = incoming;`
`...`
`}`

Thread #2: Popping:
`... MutexLock A`
`{`
`3 outgoing = *p;`
`4 p--;`
`...`
`}`

Mutex Locks are usually named somehow so that you can have multiple ones with no ambiguity.

We will talk about these in a little while. But, note that, while solving a race condition, we can accidentally create a deadlock condition if the thread that owns the lock is waiting for the other thread to do something

Oregon State University
Computer Graphics
mjb - March 22, 2021

26

Tip #4 -- Sending a Message to the Optimizer: The *volatile* Keyword

27

The *volatile* keyword is used to let the compiler know that another thread might be changing a variable "in the background", so don't make any assumptions about what can be optimized away.

```
int val = 0;
...
while( val != 0 );
```

A good compiler optimizer will *eliminate* this code because it "knows" that, for all time, `val == 0`

```
volatile int val = 0;
...
while( val != 0 );
```

The *volatile* keyword tells the compiler optimizer that it cannot count on `val` being `== 0` here

Oregon State University
Computer Graphics
mjb - March 22, 2021

27

Tip #5 -- Sending a Message to the Optimizer: The *restrict* Keyword

28

Remember our Instruction Level Parallelism example?

```
A = B + 1;
C = 3;
```

To assembly language

```
Load B,r0
Add $1,r0
Store r0,A
Load $3,r1
Store r1,C
```

```
Prefetch B
Load $3,r1
Store r1,C
Load B,r0
Add $1,r0
Store r0,A
```

Optimize by moving two instructions up to execute while B is loading

Oregon State University
Computer Graphics
mjb - March 22, 2021

28

Sending a Message to the Optimizer: The restrict Keyword

```
int *p;
int *q;
...
p = &B;
q = &B;
A = *p + 1;
*q = 3.;
```

Here the example has been changed slightly. This is what worries the out-of-order mechanisms and keeps them from optimizing as much as they could.

29

Assembly language

```
Load [p],r0
Add $1,r0
Store r0,A
Load $3,r1
Store r1,[q]
```

Using the pointers, and using out-of-order processing

```
Prefetch [p]
Load $3,r1
Store r1,[q]
Load [p],r0
Add $1,r0
Store r0,A
```

What's really happening

```
Prefetch B
Load $3,r1
Store r1,B
Load B,r0
Add $1,r0
Store r0,A
```

Uh-oh! B is being loaded at the same time it is being stored into. Who gets there first? Which value is correct?

mb - March 22, 2021

29

Sending a Message to the Optimizer: The restrict Keyword

```
int * restrict p;
int * restrict q;
...
p = &B;
q = &C;
A = *p + 1;
*q = 3.;
```

This is us promising that p and q will *never* point to the same memory location.

30

Assembly language

```
Load [p],r0
Add $1,r0
Store r0,A
Load $3,r1
Store r1,[q]
```

Using the pointers, and using out-of-order processing

```
Prefetch [p]
Load $3,r1
Store r1,[q]
Load [p],r0
Add $1,r0
Store r0,A
```

What's really happening

```
Prefetch B
Load $3,r1
Store r1,C
Load B,r0
Add $1,r0
Store r0,A
```

Now there is no conflict

mb - March 22, 2021

30

Tip #6 – Beware of False Sharing Caching Issues

We will get to this in the Caching notes!

Iteration	Series 1 (Blue)	Series 2 (Red)	Series 4 (Green)
0	1.0	0.8	0.5
1	1.0	0.7	0.5
2	1.0	0.6	0.5
3	1.0	0.6	0.5
4	1.0	0.6	0.5
5	1.0	0.9	0.7
6	1.0	1.0	0.8
7	1.0	2.0	1.0
8	1.0	1.9	1.1
9	1.0	2.0	1.1
10	1.0	2.0	1.2
11	1.0	2.0	1.3
12	1.0	1.9	1.3
13	1.0	2.0	1.3
14	1.0	2.0	1.2
15	1.0	2.0	3.7

mb - March 22, 2021

31