Parallel Lab

Fall’ 23

Optimizations & Blockers

Code Motion

avoid redundant computation

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        arr[i*100 + j] = j;
    }
}
```

```
int tmp = 0;
for (int i = 0; i < n; i++) {
    tmp = i*100;

    for (int j = 0; j < n; j++) {
        arr[tmp + j] = j;
    }
}
```

Strength Reduction

replace expensive operations

```
c = 7;
for (i = 0; i < N; i++) {
    y[i] = c * i;
}
```

```
c = 7;
tmp = 0;
for (i = 0; i < N; i++) {
    tmp = tmp + c;
    y[i] = tmp;
}
```

- Also consider: bitwise operations over others (x << 3 vs x * 2 ^ 3)

Common Subexpression Elimination

storing and retrieving vs. repeat calculation

```
a = b * c * d + f;
e = b * c * d * g;
```

```
tmp = b * c * d;

a = tmp + f;
e = tmp * g;
```

Procedure Calls

minimize calls to other functions

```
for (i = 0; i < strlen(s); i++) {
    for (j = i; j < strlen(s); j++) {
        ...
    }
}
```

```
length = strlen(s);

for (i = 0; i < length; i++) {
    for (j = i; j < length; j++) {
        ...
    }
}
```

Memory Aliasing

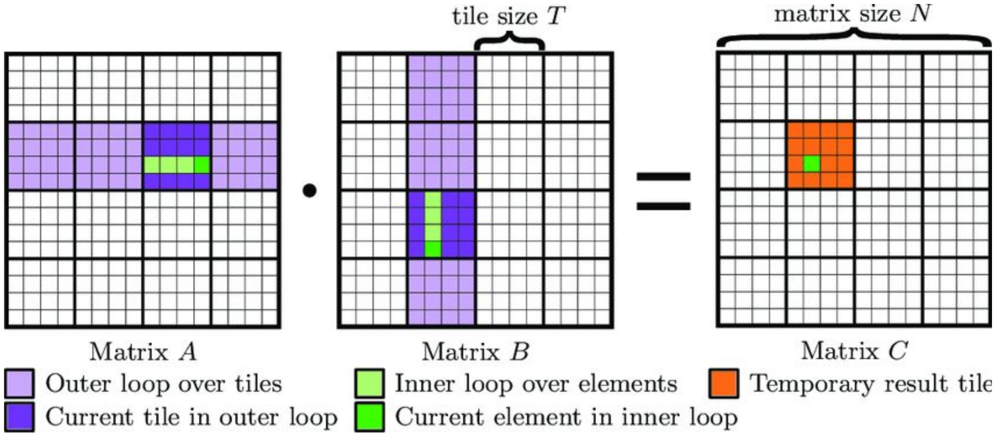
eliminate compiler variable reload (explicitly telling the compiler than the loop can be unrolled)

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        arr1[i] += arr2[i*n + j];
    }
}
```

```
for (i = 0; i < n; i++) {
    int tmp = 0;
    for (j = 0; j < n; j++) {
        tmp += arr2[i*n + j];
    }
    arr1[i] = tmp;
}
```

Tiling

reduce access latency
(very important)



<pre>for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    dst[j*n + i] = src[i*n + j];
   }
}</pre>
	<pre>for (i = 0; i < n; i+=BLOCK) {
	 for (j = 0; j < n; j+=BLOCK) {
	  for (a = i; a < i+BLOCK; a++) {
	   for (b = j; b < j+BLOCK; b++) {
	     dst[b*n + a] = src[a*n + b];
	   }
	}</pre>

Parallelism

Pipelining & Instruction-Level Parallelism

Loop Unrolling

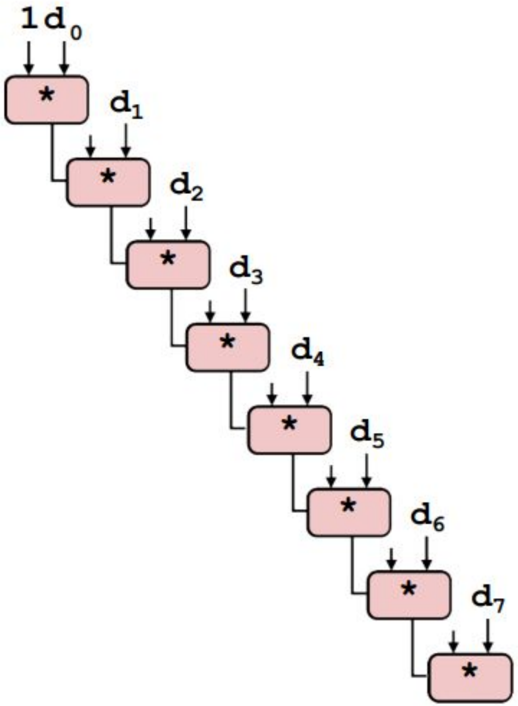
reduce iteration count

```
for (i = 0; i < n; i++) {
    arr[i] = i;
}
```

- n-1 jumps

```
for (i = 0; i < n-3; i+=4) {
    arr[i] = i;
    arr[i + 1] = i + 1;
    arr[i + 2] = i + 2;
    arr[i + 3] = i + 3;
}
```

- floor((n-1) / 4) jumps
- Unrolled 3x



Reassociation

remove dependencies

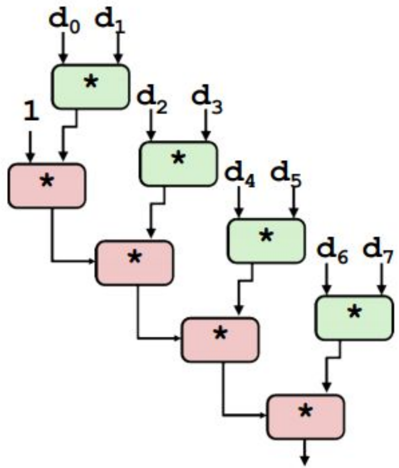
```
for (i = 0; i < n-1; i+=2) {
   sum = (sum + r[i]) + r[i+1];
}
```

- Each operation dependant on completion of the one prior

```
for (i = 0; i < n-1; i+=2) {
   sum = sum + (r[i] + r[i+1]);
}
```

- Right-side sum operations can be done independently

```
x = x OP (d[i] OP d[i+1]);
```



OpenMP

Pragmas/Directives

Pragmas and Directives Overview

Creating a parallel Region

- parallel

Worksharing Constructs

- for
- sections
- single

Synchronization Constructs

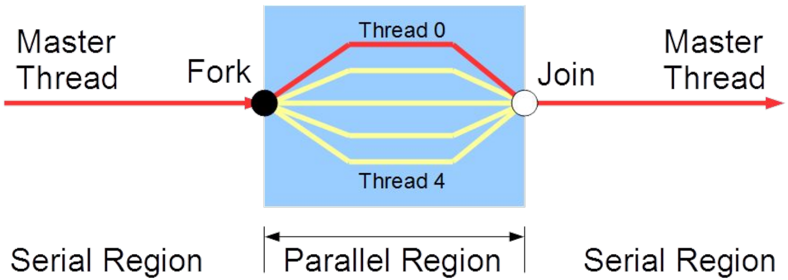
- critical
- atomic
- barrier

Parallel Region Construct

PARALLEL

```
#pragma omp parallel [clauses]
{
    ...//parallel region
)
```

- Indicates a block that should be executed by a team of threads



Worksharing Constructs

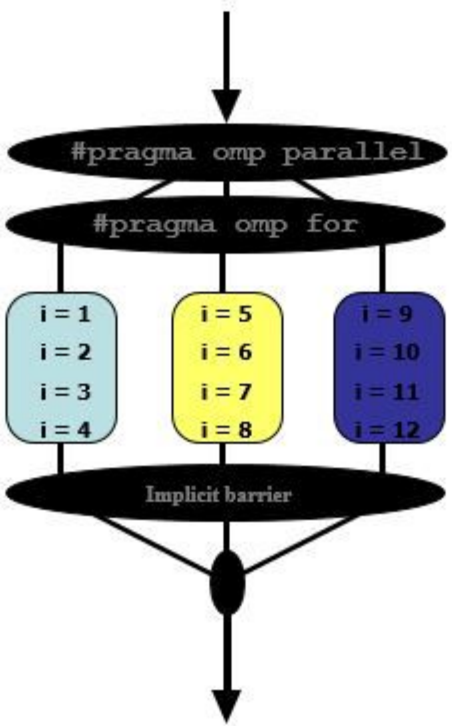
*****must be enclosed within a parallel region*****

*****implicit barriers*****

FOR

```
#pragma omp parallel [clauses]
{
    #pragma omp for [clauses]
    for (int i = 0; i < SOME_N; i++)
    {...}
}
```

- The immediately following for loop is executed in parallel
- Threads are assigned an independent set of loop iterations



SECTIONS

- Encloses a group of sections to be executed in parallel by the threads available in the parallel region
- If the number of threads is greater than the number of sections, some threads will not do work
- If the number of threads is less then the number of sections, some threads will do more than others

```
#pragma omp parallel [clauses]
{
    #pragma omp sections [clauses]
    {
        #pragma omp section
        {...}

        #pragma omp section
        {...}

        //for as many sections
    }
}
```

Note: Combined Worksharing Constructs

PARALLEL FOR

- A *for* pragma inside a parallel region can be written as the following convenient shortcut

```
#pragma omp parallel for [clauses]
for (int i = 0; i < SOME_N; i++) {...}
```

PARALLEL SECTIONS

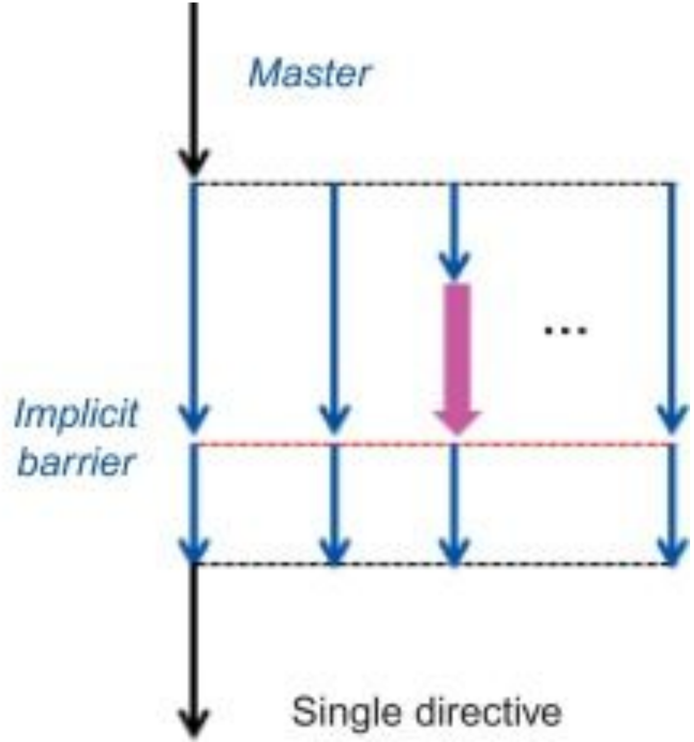
- A *sections* pragma inside a parallel region can be written as the following convenient shortcut

```
#pragma omp parallel sections [clauses]
{
    #pragma omp section
    {...}

    #pragma omp section
    {...}

    //for as many sections
}
```

SINGLE

```
#pragma omp parallel [clauses]
{
    #pragma omp single [clauses]
    {...}
}
```

- Indicates a block within the parallel region that should be executed by a single thread



Synchronization Constructs

```
#pragma omp critical
{
    ...//critical region
}
```

- Indicates a block within the parallel region that is restricted to access by one thread at a time

```
#pragma omp atomic
```

- Indicates an operation to be performed atomically (+, -, /, *, etc.)

```
#pragma omp barrier
```

- Indicates a point within the parallel region where threads must wait until all threads have reached that point

Clauses

- `reduction(operator:list)` **IMPORTANT!**
  - Declares the variables in the comma-separated list to be private to each thread
  - At the end of the reduction, the shared variables are updated by combining the original value with the final value of each of the private copies
  - Operator is one of the following +, *, -, &, ^, |, &&, ||

- `num_threads(int)` **IMPORTANT!**
  - Specifies the size of the team of threads for the immediately following parallel region
- `private(list)`
  - Declares the variables in the comma-separated list to be private to each thread
  - `firstprivate(list)`
    - The private copies are initialized to the values the variables had immediately before entering the construct
  - `lastprivate(list)`
    - The thread that executes the sequentially last iteration/section updates the original variable upon exiting the construct

- `shared(list)`
  - Threads access the same storage area for the variables that appear in list, sharing them

- `default(shared|none)`
  - Specifies scoping attributes for all variables within a parallel region.
  - If not specified, DEFAULT(SHARED) is assumed. C
  - an be overridden by using the private, firstprivate, lastprivate, reduction, and shared clauses.

- `nowait`
  - Overrides implicit barriers

Pragma-Clause Combos

clause	parallel	for	sections	single	parallel for	parallel sections
private						
firstprivate						
lastprivate						
shared						
default						
reduction						
nowait						
num_threads						

green means OK to use together

Runtime Library Routines

- `omp_set_num_threads(int)`
  - Set number of threads for parallel regions
- `omp_get_num_threads()`
  - Get number of threads for parallel regions
- `omp_get_thread_num()`
  - Get the id of the thread currently executing
- `omp_get_num_procs()`
  - Get the number of processors available

The Actual Lab (by Ethan and Aditya)

Phase Objectives

Phase 1: Average pixel, trying to fix buggy OpenMP code
Phase 2: Grayscale conversion, trying to fix buggy OpenMP code
Phase 3: Convolution, parallelizing the code from scratch

Tips

- Start early!
- Parallelism can be scary but just go for it! The more you work with it the more you will get a feel for what needs to be done.
- If you get SEGFAULTS or modify utils.h, do `make clean` and then `make` to do a clean compile.
- This lab isn't meant to trick you. If you get lost or confused, take a look at these slides or try to plan your methodology out ahead of time! A few pragmas can go a long way!
- If your optimized code is as good as the given threshold, you're gucci 😉
  - However the TA will still verify the speeds at the end when the server has much less load
- Correctness >>> speed
- Your TAs and LAs are always here to help you! (Come to office hours 🥺)

Additional Resources

- Official OpenMP Site (tons of cool stuff here!)
- Official Examples
- OpenMP Quick Guide
- Further Learning: ✨CS 133: Parallel and Distributed Computing✨