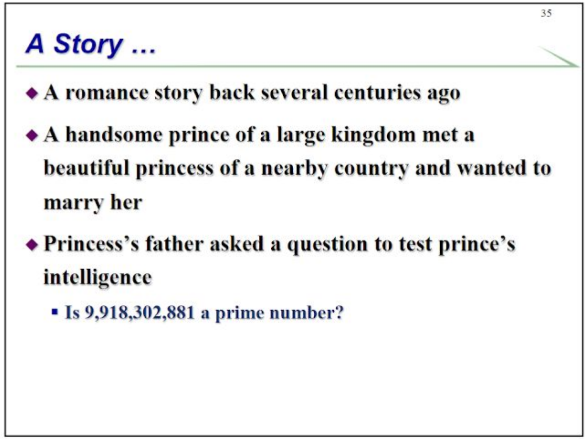
Discussion #2:

Worksheets #1-#4, OpenMP Hints, and Homework #1

Daniel Tan, Jason Lau, Yuze Chi, and Jason Cong

Worksheet #1

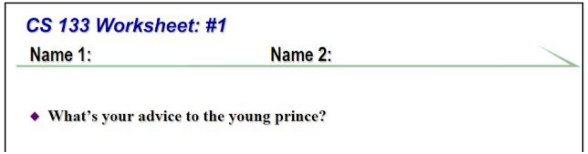


35

A Story …

- A romance story back several centuries ago
- A handsome prince of a large kingdom met a beautiful princess of a nearby country and wanted to marry her
- Princess's father asked a question to test prince's intelligence
  - Is 9,918,302,881 a prime number?

35



CS 133 Worksheet: #1

Name 1: Name 2:

- What's your advice to the young prince?

- **How many potential factors?**
  - √(9,918,302,881) = 99,590
  - Only even prime is 2: ~50,000 to check
  - Also factor out 3, 5, …: ~30,000
  - The prince would need A LOT of meetings!
    - Need hierarchy as given in lecture
- AKS Primality test (https://doi.org/10.4007/annals.2004.160.781): checking prime is **polylog(n)**
  - 2002 paper, 2006 Gödel prize

Worksheet #2

- Are the behaviors of the following two code segments identical? If not, what's the difference?

```
#pragma omp parallel \
 num_threads(2)
{
    foo();
    #pragma omp for
    for (int i = 0; i < 3; i++)
        run(i);
    bar();
}
```

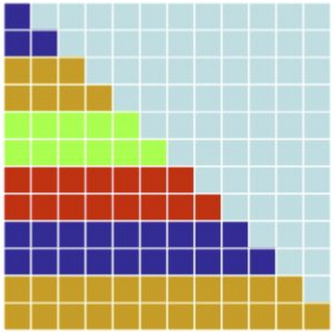
```
#pragma omp parallel \
 num_threads(2)
{
    foo();
    for (int i = 0; i < 3; i++)
        run(i);
    bar();
}
```

- **Only difference: #pragma omp for**
  - How many threads are created?
  - Divides for iterations amongst the threads
  - Pragmas are applied to the next 'structure': loop, if-else, {...}, simple statements
- **What happens if you don't use it?**
  - For loop run across all iterations in threads
  - Usually isn't intended, a common mistake!

Worksheet #3

What happens when we use guided scheduling for the following example?

```
#pragma omp parallel for schedule(guided)
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = ...;
```

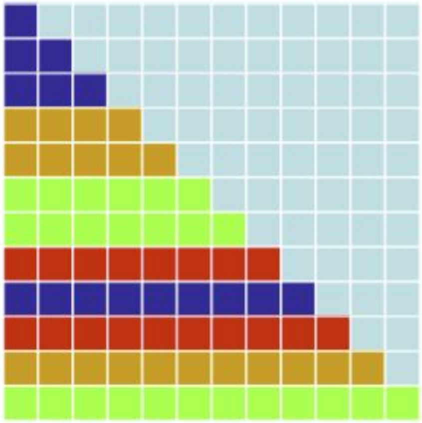


Intel, "Introduction to Parallel Programming," 2009

- **What happen if `schedule(static, 2)`?**
  - Iterations divided evenly
  - Loops with uneven workload distribution

Worksheet #3

```
#pragma omp parallel for schedule(guided)
for (int i = 0; i < 12; i++)
      for (int j = 0; j <= i; j++)
            a[i][j] = ...;
```



- **What is `schedule(guided)`?**
  - Dynamic scheduling
  - Start from larger chunks and then shrink

- **Starting block size = number_of_iterations / number_of_threads**
- **Subsequent block size = number_of_iterations_remaining / number_of_threads**

Worksheet #4

CS 133 Worksheet #4
Name 1: Name 2:

Do you have a proposal to improve the efficiency of the following code?
(suggestion is fine, not the exact code)

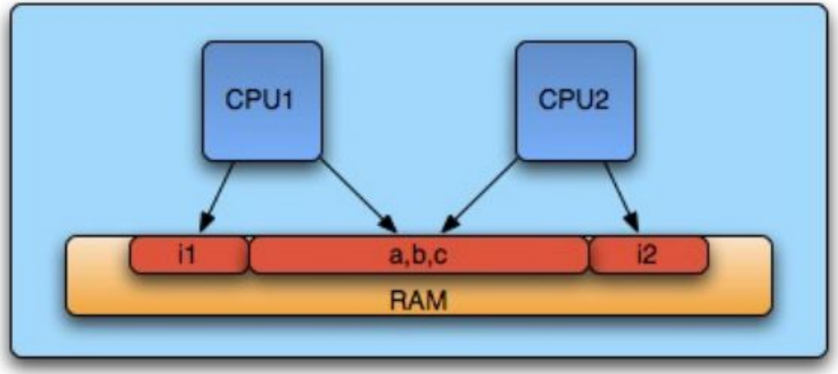
```
#pragma omp parallel for private (index)
for (i = 0; i < elements; i++) {
    index = hash(element[i]);
    #pragma omp critical
    insert_element (element[i], index);
}
```

- **Is it correct?**
- **Is it efficient?**
    - One lock for the whole table.
- **Where the race condition may occur?**
    - Insertion into the same bucket.
    - We don’t want a single lock.
- **Use finer grained lock on each bucket.**
- **Atomic pragma?**

Worksheet #4

- Get rid of `private`?
- Shared-memory
  - Threads have access to the same address space
  - Programmer needs to define what's private

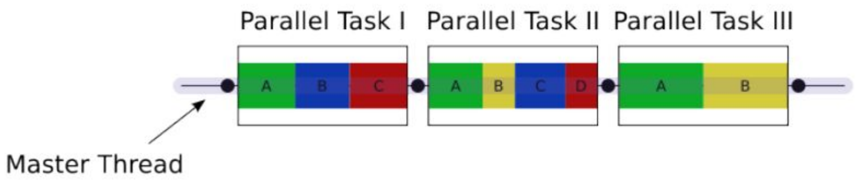
```
for (i = 0; i < elements; i++) {
    index = hash(element[i]);
    #pragma omp critical
    insert_element (element[i], index);
}
```

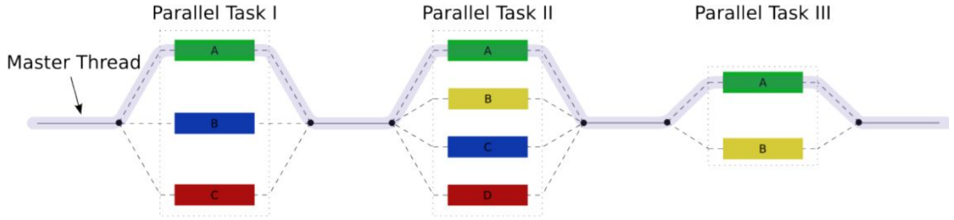


OpenMP Programming Model

- **Fork-join execution model**
  - **Use threads to execute instances of loop/section**

```
#pragma omp parallel for
for (i = 0; i < n; i++) { // Fork
    // do something
} // Join
```





Lab #1: Things to consider

- You should be able to compile and run lab 1&2 using WSL2 (Ubuntu) on Windows
- Which loops to parallelize?
  - Outer loop or Inner loop?
  - Forking and joining multiple times creates unnecessary overhead.
- Static, dynamic or guided scheduling?

Static

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

Dynamic

```
schedule(dynamic):
*   ** **  * * *  *      *  *    **   *  *  * *       *  *   *
  *       *     *    * *     * *   *    *        * *   *    *
 *       *    *     * *   *   *     *  *       *  *  *  *  *   *
   *  *     *    * *    *  *    *    *    ** *  *   *     *   *
```

```
schedule(dynamic, 1):
    *    *     *        *   *    * *  *  *         *  * *  * *
*  *  *   * *     *  * * *    * *      *   ***  *   *         *
 *   *  *  *  *    ** *    *      *  *  * *   *  *   *   *
  *    *     *  **        *  * *    *          *  *    *  * *  *
```

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

```
schedule(dynamic, 8):
                ********                                ********
                        ********        ********
********                        ********        ********
        ********
```

Guided

```
schedule(guided):
                            *********                        *
                ************                     *******  ***
                                     *******                   *
****************                            *****       **    *
```

```
schedule(guided, 2):
                ************                     ****     **
                                     *******         ***    **
                            *********
****************                            *****       **    **
```

```
schedule(guided, 4):
                                     *******
                ************                     ****    ****
                            *********
****************                            *****    ****    ***
```

```
schedule(guided, 8):
                ************                 ********        ***
****************
                                     ********
                            *********                ********
```

Lab #1: Things to consider

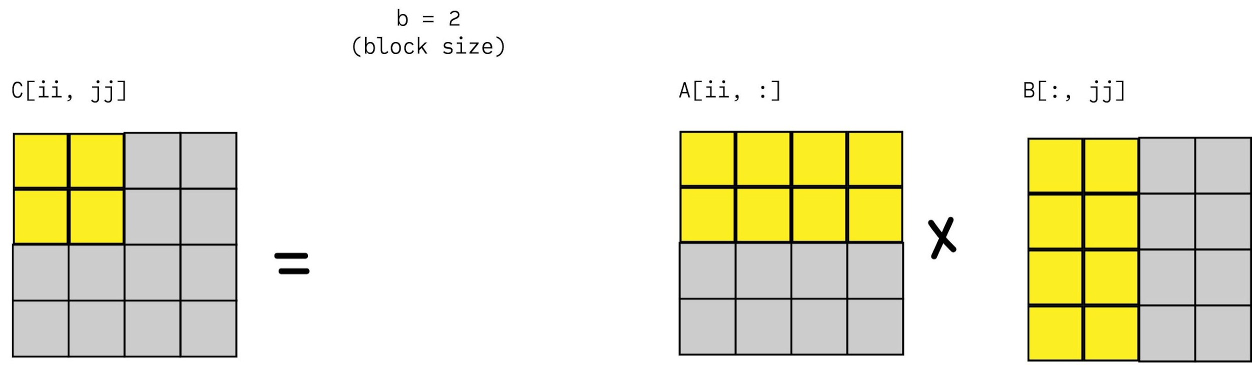
- How many threads?
  - Hyper-threading ≠ more physical cores
- Do I need vectorization?
  - Automatically inferred by compiler: `-march=native`
- Row major access or column major?
  - The sequential version takes over 10 minutes to complete
  - Permute loops, or reorganize the data layout

Cache Memory

Figure: https://medium.com/ai%C2%B3-theory-practice-business/fastai-partii-lesson08-notes-fddcdb6526bb

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Lab #1: Blocked Design?



ii, jj, kk denote block indices while i, j, k denote element indices

```
// C =      A * B
// b = n / N (where b is the block size)
for ii = 1 to N:
  for jj = 1 to N:
    for kk = 1 to N:
      C[ii, jj]  = A[ii, kk] * B[kk, jj]
```

Figure: https://malithjayaweera.com/2020/07/blocked-matrix-multiplication/

Homework #1

2. Your Processors

3. Dennard Scaling

- Dennard scaling states that power consumption per unit area stays the same (or electric field stays constant) with transistor scaling.

- However, this no longer applies beyond a certain size due to increasing leakage. We get more energy benefits
  - by keeping the clock frequency relatively constant, and
  - by using more processor cores.

  That is why computing industry has moved to parallel computing.

4. Top-10 Supercomputers https://www.top500.org/lists/top500/2021/11/

Homework #1

5. Given an integer array `a[ ]` of N elements. Please write an OpenMP function to sort it by the Quicksort algorithm using the task directive. The function header is:
`void quicksort(int *a, int p, int r)`.

```
int main() {
	#pragma omp parallel
	#pragma omp single nowait
	quicksort(a, 0, N-1);
	return 0;
}
```

```
void quicksort(int *a, int p, int r) {
	if ( p >= r ) return;
	if ((r - p) < THRESHOLD) {
	    sequentialSort(a, p, r); return;
	}
	int pivot = partitionArray(a, p, r);
	#pragma omp task
	quicksort(a, p, pivot - 1);
	#pragma omp task
	quicksort(a, pivot + 1, r);
	#pragma omp taskwait
}
```

Homework #1

5. Given an integer array `a[ ]` of N elements. Please write an OpenMP function to sort it by the Quicksort algorithm using the task directive. The function header is:
`void quicksort(int *a, int p, int r)`.

```
int main() {
      quicksort(a, 0, N-1);
      return 0;
}
```

```
void quicksort(int *a, int p, int r) {
      if ( p >= r ) return;
      if ((r - p) < THRESHOLD) {
          sequentialSort(a, p, r); return;
      }
      int pivot = partitionArray(a, p, r);
      #pragma omp parallel sections
      { #pragma omp section
            quicksort(a, p, pivot - 1);
        #pragma omp section
            quicksort(a, pivot + 1, r); }
}
```

Homework #1

6. There is a list of n independent tasks with known (but considerably different) runtimes to be performed by m processors. We order the tasks in a list and assign each task in the order of the list to the first available idle processor until all tasks are completed (so called the list scheduling). Once a processor finishes a task, it requests a new task. Alice sorts the list in decreasing order of the task runtimes and then performs list scheduling. Bob sorts the list in increasing order of the task runtimes and then performs list scheduling. Who do you expect to finish first? Please explain why. *Extra credits for formal proof.*

- **<u>133 isn't a proof-centered course. So, no worries if you don't get this one!</u>**
- **What's our strength as computer science students? Build it and test it out!**
- **Script to simulate the scheduling and "confirm" the claim.**
- **OK. If the proof still piques your interest, let's go!**

Proof for the Scheduling Problem

```
E.g. n=7 m=3    the tasks: 6, 7, 8, 9, 11, 12, 14
```

```
Bob (list scheduling on
ascending list):
processor 0: 6  9  14
processor 1: 7  11
processor 2: 8  12
```

```
Eve (static,1 scheduling
on descending list):
processor 0: 14  9  6
processor 1: 12  8
processor 2: 11  7
```

```
Alice (list scheduling
on descending list):
processor 0: 14  7
processor 1: 12  8  6
processor 2: 11  9
```

Proof outline: Suppose the task durations are ordered ascendingly: $d_0 <= d_1 <= \ldots <= d_{n-1}$

1. **Bob's task assignments (%m) are determined regardless of the durations of the tasks.**
   Tasks 0 to m-1: trivial
   Tasks m to 2m-1:
   …
   Task m*(n//m) to n:

   Induction: For task q*m+r (0<=r<m), the finishing time of processors are:
   $d_0+d_m+\ldots+d_{q*m}$, … , $d_{r-1}+\ldots+d_{q*m+r-1}$, $d_r+\ldots+d_{(q-1)*m+r}$, … , $d_{m-1}+\ldots+d_{(q-1)*m+(m-1)}$
   Processor r is the first to finish!
2. **Eve performs %m assignments but on descending tasks. Eve is as fast as Bob.**
   By definition of (static, 1) scheduling, tasks with same id%m will be on the same processor, like Bob's
3. **Alice is quicker than Eve: next slide**

Proof for the Scheduling Problem Cont.

```
Durations: [14, 12, 11], [9, 8, 7], [6]
```

Task durations: $d_0 >= d_1 >= d_2 >= d_3 >= d_4 >= d_5 >= d_6$

Finishing time for task group j: t_j^E for Eve and t_j^A for Alice

3. Alice is same or faster than Eve, proof:

- Induction: $t_j^A <= t_j^E$ for $j = 0, 1, \ldots, (n//m)+1$
- For group 0: $t_0^E = t_0^A = d_0$
- For group 1:
  - Eve: $t_1^E = t_0^E + d_m$
  - Alice: $t_1^A <= t_0^A + d_m <= t_0^E + d_m$

Generalizes 0->1 to j->j+1

- Take min, increase finishing time, add back in. (A data structure for the finishing times?)
- Keypoint: we push **m** values $<=t_0^A$ into a priority queue and perform the above **m** times?
- All the popped values (start time) are $<=t_0^A$ and all the increments (duration) are $<=d_m$

```
Eve:
processor 0: 14  9  6
processor 1: 12  8
processor 2: 11  7

Alice:
processor 0: 14  7
processor 1: 12  8  6
processor 2: 11  9
```

Q&A