Summer Training

- Integrating PDC into introductory Computer Engineering and Computer Science Courses
- How to modify a course with instrumentation to enable publishing results
- July 31 to August 4, 2023 at Louisiana State University, Baton Rouge
- $5000 stipend
  - $3000 on workshop completion
  - $2000 on submitting an article reporting results
- Must be US citizen or permanent resident to receive stipend

Summer Training

- Target Audience: Instructors teaching beginning Computer Engineering and/or Computer Science Courses in US institutions
- Typical courses include digital logic, CS 1 or 2 (first programming), data structures and discrete math
  - Develop skills and activities to help students for a future in computing
  - Experience hands-on modules for PDC integration
  - Network with other instructors
  - Opportunities to publish your work
  - Prepare your students for the modern workforce

Summer Training

- For more details contact
  - R. Vaidyanathan: vaidy@lsu.edu
  - Sushil Prasad: sushil.prasad@utsa.edu
  - Charles Weems: weems@cs.umass.edu
  - Sheikh Ghafoor: SGhafoor@tntech.edu

Adding Parallelism in CS2

Data structures enable data parallelism

Progressive Motivation

- You don’t appreciate the value of learning something until you have enough experience to understand the problem
- Introducing a topic prior to the stage where it is naturally motivated can have a destructive learning effect — lower engagement, negative associations
- The distance between knowledge available and knowledge needed to solve the problem (cognitive load) also affects motivation
- Introducing a topic with a high cognitive load can have a destructive effect — frustration, loss of confidence, negative associations

PDC Progression

- Drawing attention to common experiences of PDC (CSP, early CS1) — motivating value of studying the subject to serve human needs
  - Real world: Group coordination, teamwork, checkout lines, etc.
  - Computing world: Social media, phone UI, online banking
- Concurrency to make a UI responsive for the user (late CS1)
  - Parallelism for performance isn’t motivated yet
- Parallelism to make a large task faster (CS2)
  - Large data structures, costly algorithms motivate need

PDC Progression

- Advanced concurrency — motivated in web programming, OS
- Performance — illustrates issues in systems class
- Advanced parallelism — motivated by specific subjects
  - Graphics, AI, big data, etc.

Data Parallelism

- Natural way to process many data sets
    - Common concept in linear algebra operations and packages
- For each loops have become common in languages
    - Still iterative, but introduces data parallel thinking
    - Often no access to index because of generality for collection types

```
for(String entry: stringArray) {
  System.out.println("\"" + entry + "\"");
}
```

Unplugged Activity

PDC in Action

Can also do this with distance learning

Search This List of Numbers

27	34	10	3	92	82	55	67	39	41	70	18	25	99	58	43	72	1	44
30	29	28	66	76	93	14	7	81	52	4	85	22	90	8	11	50	0	15

Using everyone in the class to divide up the work

How Did the Process Work?

- Did you elect a leader? How?
- How was the data distributed?
- What information was used to inform the distribution?
- What work was done by each individual? Was it uniform?
- How was the result reported?

Scaling as Motivation for Algorithms

- Consider increase in amount of data
  - How does local work change?
    - Shift to sorting local data to enable binary search
  - How is change of algorithm affected by availability of workers?
- When would there be a major change in strategy?
- Parallelism becomes just another aspect of navigating complexity space

Another Unplugged Activity

PDC in Action

Sort This List of Numbers

27	34	10	3	92	82	55	67	39	41	70	18	25	99	58	43	72	1	44
30	29	28	66	76	93	14	7	81	52	4	85	22	90	8	11	50	0	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Using everyone in the class to divide up the work

How Did the Process Work?

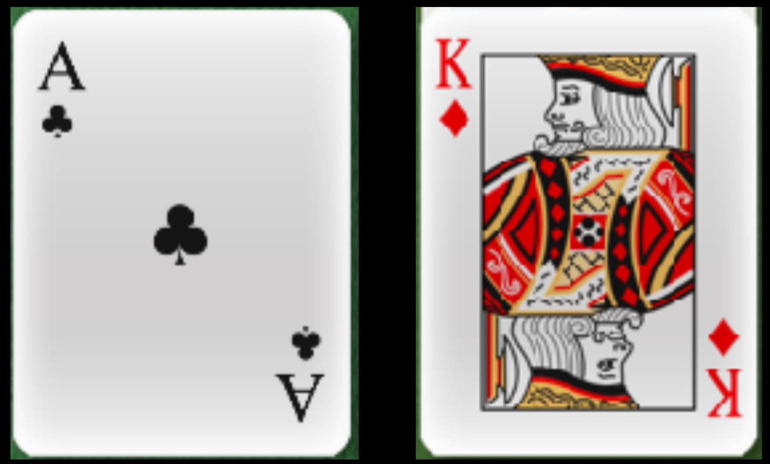
- Did you elect a leader? How?
- How was the data distributed?
- What information was used to inform the distribution?
- What work was done by each individual? Was it uniform?
- How were the work products combined?
- How did you know you were done?

Unplugged Activity

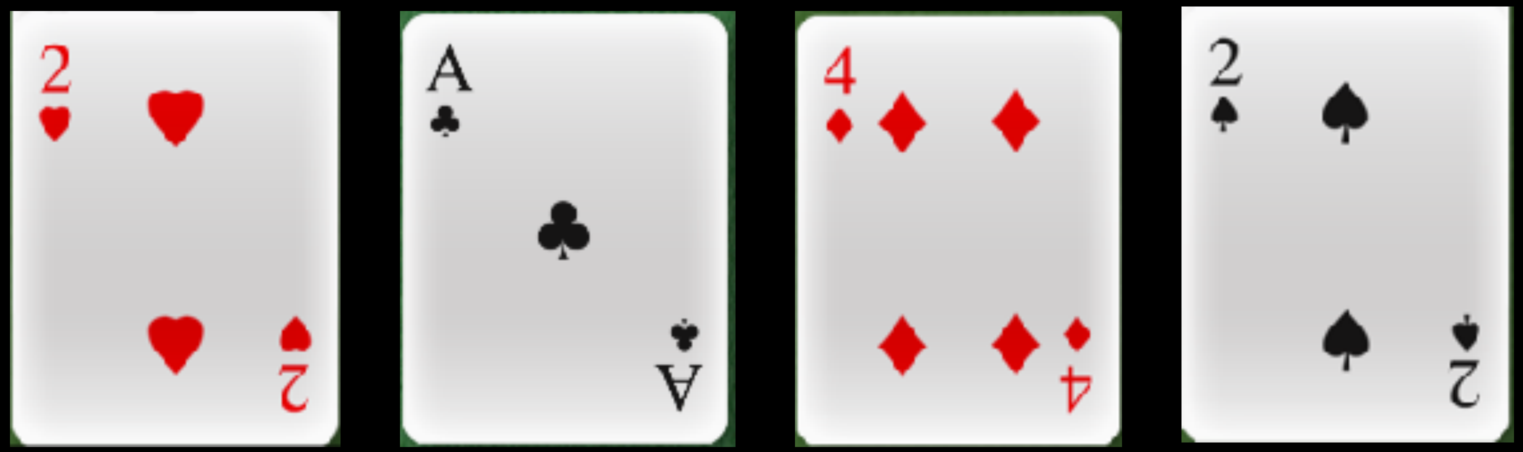
Another Example of PDC in Action

Unplugged Activity, Step 1

- Shuffle deck of cards
- One person sorts the deck
  - Ace is low, King is high

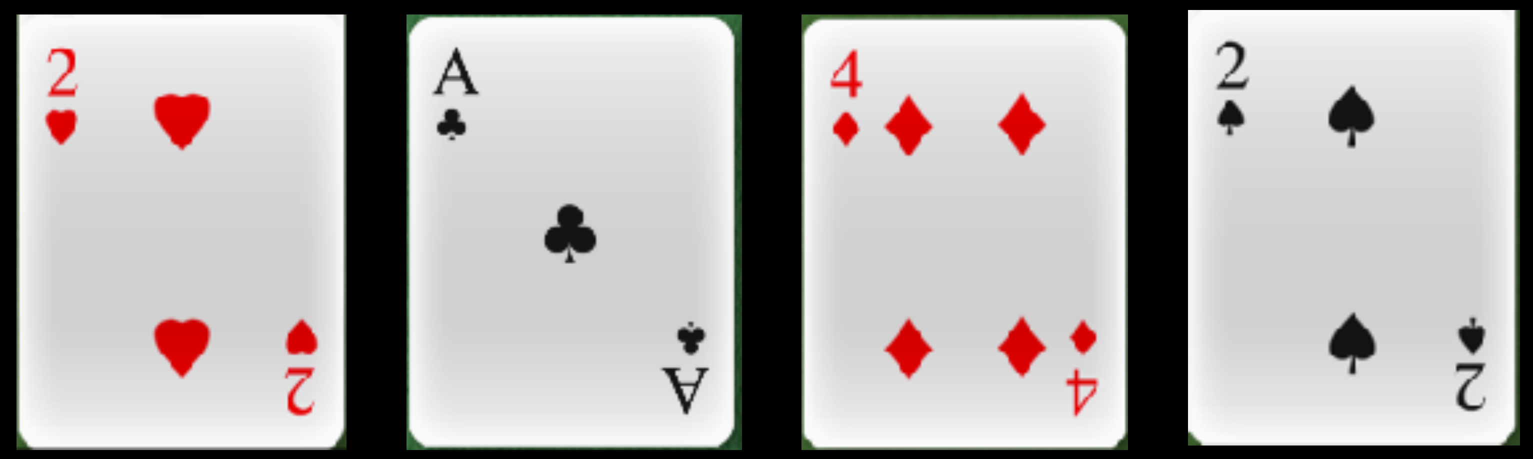


  - Suits ordered: Hearts, Clubs, Diamonds, Spades



- Other person times, watches, identifies strategy being used, describes it
- Reverse rolls

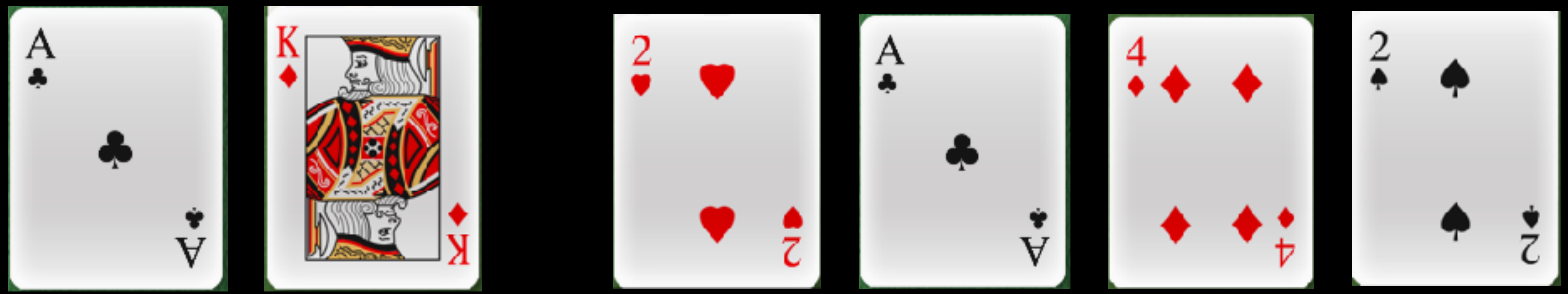
Unplugged Activity, Step 2



- Form groups of three
- Shuffle deck of cards
- Two people sort the deck — Ace is low, King is high
  - Suits ordered: Hearts, Clubs, Diamonds, Spades
- Other team member to time, watch, identify strategy being used, describe
- Reverse rolls, compare strategies and times

Unplugged Activity, Step 3

- Form groups of ten, shuffle deck of cards
- Ten people sort the deck
    - Ace low, King high, suits: Hearts, Clubs, Diamonds, Spades
- Choose a strategy from prior rounds



- Decide on distribution method before starting (may want to account for different speeds)
- Do once for practice, then repeat and record time, compare with prior times

Unplugged Activity, Step 4

- Develop written instructions for the strategy
- Make them work for any number of people
- Be sure that distribution takes advantage of available people too

Likely Outcome

- Students develop some version of parallel merge sort
  - May use a serial distribution algorithm
    - Possibly recursive split
  - Individuals may use different local/serial sorting strategies
  - e.g., Bucket sort
- Merging likely to happen by pairs of students

Prior Knowledge of Merge Sort

- If students already know serial, recursive merge sort (e.g., prior course)
- Some may directly recognize that this is what is happening in parallel
  - If asked to liken it to a known sort, most will make the connection
- That recognition may lead some to a different distribution strategy

Can then ask: What if…

- We repeat this with many more people? (Can also just do this)
  - Fairly clear that dividing work much further won’t improve speed
- Why not?
- Overhead of distribution, coordination, dominates sorting work

Many Variations Possible

- Increase number of decks to see weak scaling relationship
- After initial timing, have a work queue manager distribute chunks of decks in proportion to individual’s speed — effect of load balancing
- Put decks at a distance and set rules for movement to simulate distributed work
- Sort pennies by date, monopoly money by denomination, etc.

Performance

- With the GUI, we were interested in responsiveness
- Here we are using parallelism to reduce execution time
  - How do we measure execution time? Get time before, after, subtract

Java

```
long start = System.nanoTime();
// measurement
long elapsed = System.nanoTime() - start;
```

C++

```
#include <chrono>

chrono::time_point<chrono::system_clock> start
chrono::time_point<chrono::system_clock> end

start = chrono::system.clock::now();
// measurement
end = chrono::system.clock::now();
chrono::duration<float> elapsed = end - start;
```

Merge Sort With Timing

#include <chrono>

Note: Templated to allow different types

```
int main(int argc, const char * argv[]) {
    chrono::time_point<chrono::system_clock> start;
    chrono::time_point<chrono::system_clock> end;

    // Initialize the array with random integers
    for (int index = 0; index < MAX_ITEMS; index++) {
        numbers[index] = rand() % 1000000000;
    }

    start = chrono::system_clock::now();                // Record start time

    MergeSort<int>(numbers, 0, MAX_ITEMS-1, temp);    // Run the sort

    end = chrono::system_clock::now();                  // Record end time

    chrono::duration<float> elapsed = end-start;      // Calculate and report time
    cout << "Execution time in seconds = " << elapsed.count() << "\n";

    return 0;
}
```

Chrono

- Provides general set of objects for representing time
- Objects can be a point in time (time_point) or a duration (duration)
- Can be linked to system clock for fine-grained accuracy
- Overloads - operator to subtract time points, giving duration
- Get a time_point for current system time before some process
- Get time_point after, subtract, and report execution time

Recursive MergeSort

```
template<class ItemType>
void MergeSort(ItemType values[], int first, int last, ItemType tempArray[])
{
    if (first < last) //Can still divide incoming piece (otherwise just return)
    {
        int middle = (first + last) / 2;
        MergeSort<ItemType>(values, first, middle, tempArray);    // Recursively sort lower (left) half
        MergeSort<ItemType>(values, middle + 1, last, tempArray); // Recursively sort upper (right) half
        Merge<ItemType>(values, first, middle, middle + 1, last, tempArray); // Merge sorted halves
    }
}
```

Takes in the array to be sorted and indexes indicating the segment to work on
Returns the sorted segment in a second array
Divides the segment into two parts, calling itself on each part
Merges the two sorted parts (tempArray is working space for the merge)

Merge (part 1)

```
template<class ItemType>
void Merge(ItemType values[], int leftFirst, int leftLast,
           int rightFirst, int rightLast, ItemType tempArray[])
{
    int index = leftFirst;
    int saveFirst = leftFirst;
    while ((leftFirst <= leftLast) && (rightFirst <= rightLast))
    {
        if (values[leftFirst] < values[rightFirst]) // If left value is less, take it
        {
            tempArray[index] = values[leftFirst];
            leftFirst++;
        }
        else
        {
            tempArray[index] = values[rightFirst];  // Otherwise take right value
            rightFirst++;
        }
        index++;
    }
```

Merge (part 2)

```
    // Left or right can have extra values (but not both)
    while (leftFirst <= leftLast)
        // Copy remaining items from left half.
    {
        tempArray[index] = values[leftFirst];
        leftFirst++;
        index++;
    }
    while (rightFirst <= rightLast)
        // Copy remaining items from right half.
    {
        tempArray[index] = values[rightFirst];
        rightFirst++;
        index++;
    }
    // Copy temporary array back into values
    for (index = saveFirst; index <= rightLast; index++)
        values[index] = tempArray[index];
}
```

Odds and Ends

Top of main.cpp

```
#include <iostream>
#include <stdlib.h>
#include <thread>
#include <chrono>

using namespace std;

const int MAX_ITEMS = 20000000;
int numbers[MAX_ITEMS];
int temp[MAX_ITEMS];
```

Run it

- What does it report?

```
Execution time in seconds = 3.40077
```

Making it Parallel

- First idea would be to spawn a thread for each side of recursion
- Thread constructor takes name of method and parameters for call
- join for a thread waits until it exits

```
if (first < last)
{
  int middle = (first + last) / 2;
  thread left (ParallelMergeSort<ItemType>, values, first, middle, tempArray, chunkSize);
  thread right (ParallelMergeSort<ItemType>, values, middle + 1, last, tempArray, chunkSize);
  left.join();
  right.join();
  Merge<ItemType>(values, first, middle, middle + 1, last, tempArray);
}
```

Oops…

- This will crash — OS won’t allow a task to allocate this many threads
- Even if it didn’t crash, it would be slower than the serial version
- Why?

Overhead

- Like having one person for each card in the decks (104 people)
  - Time to distribute them will be more than sorting
- Each thread takes thousands of instructions to start running
  - If it does less work than that, its creation is costing more
- Solution: chunking
  - Stop recursing and switch to serial sort (serial algorithms still matter)

Parallel-Serial Merge Sort

```
template<class ItemType>
void ParallelMergeSort(ItemType values[], int first, int last, ItemType tempArray[], int chunkSize)
{
    if (first < last)
    {
        int middle = (first + last) / 2;
        if (last-first > chunkSize)         // If enough work left, launch more threads
        {
            thread left (ParallelMergeSort<ItemType>, values, first, middle, tempArray, chunkSize);
            thread right (ParallelMergeSort<ItemType>, values, middle + 1, last, tempArray, chunkSize);
            left.join();
            right.join();
        }
        else                                // Otherwise finish sorting locally
        {
            SerialMergeSort<ItemType>(values, first, middle, tempArray);
            SerialMergeSort<ItemType>(values, middle + 1, last, tempArray);
        }
        Merge<ItemType>(values, first, middle, middle + 1, last, tempArray);
    }
}
```

Local Sort

```
template<class ItemType>
void SerialMergeSort(ItemType values[], int first, int last, ItemType tempArray[])
{
    if (first < last)
    {
        int middle = (first + last) / 2;
        SerialMergeSort<ItemType>(values, first, middle, tempArray);
        SerialMergeSort<ItemType>(values, middle + 1, last, tempArray);
        Merge<ItemType>(values, first, middle, middle + 1, last, tempArray);
    }
}
```

- **We could use any sort here (but we already had this)**
- **Serial and Parallel both use Merge (which is unchanged)**
- **To enable experimentation, main will input the chunk size**

main

```
int main(int argc, const char * argv[]) {
    chrono::time_point<chrono::system_clock> start;
    chrono::time_point<chrono::system_clock> end;

    // Initialize the array with random integers
    for (int index = 0; index < MAX_ITEMS; index++) {
        numbers[index] = rand() % 1000000000;
    }
    cout << "Enter chunk size (<= " << MAX_ITEMS << "): ";
    cin >> chunk;

    start = chrono::system_clock::now();                               // Record start time
    ParallelMergeSort<int>(numbers, 0, MAX_ITEMS-1, temp, chunk);  // Run the sort
    end = chrono::system_clock::now();                                 // Record end time

    chrono::duration<float> elapsed = end-start; // Calculate and report time
    cout << "Done sorting\n";
    cout << "Execution time in seconds = " << elapsed.count() << "\n";

    return 0;
}
```

Run it

- What does it report?

```
Enter chunk size (<= 20000000): 100000
Done sorting
Execution time in seconds = 0.630879
```

```
Enter chunk size (<= 20000000): 5000
Done sorting
Execution time in seconds = 1.16024
```

```
Enter chunk size (<= 20000000): 10000000
Done sorting
Execution time in seconds = 1.83724
```

Speedup?

- Serial time/Parallel time
- Why does my 10-core, 20-thread processor only go 5.4 times faster?
  - Still some overhead
  - Merges are still serial (Amdahl’s law)
  - Cache locality (the array may not fit in cache)
  - OS may have better things to do (also causes variability)

Speedup!

- 5.4X isn't bad
- Consider that a weather forecast for 8 hours from now isn't very useful if it takes 16 hours to compute
- Amdahl's law S_{max} = (1/(1-P + P/S)
    - Example: Million way parallel, 99% parallelizable
    - Speedup = 99.99X (The 1% dominates the time)

A Last Fun Example

HelloThread

```
#include <iostream>
#include <thread>
using namespace std;
void hello (int n){
    cout << "Hello, World #" << n << "\n";
}

int main(int argc, const char * argv[]) {
    thread first(hello, 1);
    thread second(hello, 2);
    cout << "Goodbye, World!\n";
    first.join();
    second.join();
    return 0;
}
```

```
Goodbye, World!
Hello, World #Hello, World #12
```

```
Goodbye, World!
Hello, World #Hello, World #21
```

```
Goodbye, World!
Hello, World #1
Hello, World #2
```

```
Goodbye, World!
Hello, World #Hello, World #1
2
```

```
Hello, World #Hello, World #Goodbye, World!
21
```

Unlike Java, C++ doesn't queue output atomically

Questions?