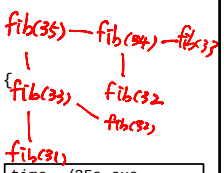
Lecture 13

Review

- Compiler Directives
  - Conditional compilation
  - Parallel construct
  - Work-sharing constructs
    - for, section, single
  - *Work-tasking*
  - *Synchronization*
- *Library Functions*
- *Environment Variables*

35a.cpp




```
unsigned long fib(unsigned long n) {
  if(n<2) return n;
  unsigned long x, y;
  #pragma omp task shared(x)
  x = fib(n-1);
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;
}
int main() {
  #pragma omp parallel
  {
    #pragma omp single nowait
    cout << fib(35) << " ";
  }
  return 0;
}
```

```
time ./35a.exe
9227465

real    0m5.806s
user    0m9.345s
sys     0m2.004s
```


Compiler directives
task *construct,* if *clause*

- When the if clause argument is false
  - The encountering task is suspended.
  - The new task is executed immediately by the encountering thread.
  - The data environment is still local to the new task...
  - ...and it's still a different task with respect to synchronization.
  - The parent task resumes when the task finishes.
- It's a user directed optimization
  - when the cost of deferring the task is too great compared to the cost of executing the task code.
  - to control cache and memory affinity.
  - Avoid creating small tasks

A "Hands-on" Introduction to OpenMP*, T. Mattson and L. Meadows

36a.cpp

```
unsigned long fib(unsigned long n) {
  if(n<2) return n;
  unsigned long x, y;
  #pragma omp task shared(x) if(0)
  x = fib(n-1);
  y = fib(n-2);

  return x+y;
}
int main() {
  #pragma omp parallel
  {
    #pragma omp single nowait
    cout << fib(35) << " ";
  }
  return 0;
}
```

if (false) makes the encountering thread immediately executes the new task (and suspend the current running task …)

```
time ./36a.exe
9227465

real    0m4.606s
user    0m4.792s
sys     0m0.000s
```


Compiler directives
task *construct,* untied *clause*

- By default, tasks are tied to the thread that first executes them
  - not the creator
- Tied tasks can be scheduled as the implementation wishes
  - Constraints:
    - Only the thread that the task is tied to can execute it
    - A task can only be suspended at a suspend point
      - task creation, task finish, taskwait, barrier
    - If the tasks is not suspended in a barrier it can only switch to a direct descendant of all tasks tied to the thread
- Untied tasks
  - Can be suspended at any point
  - Can switch to any task
  - Be careful: variable scoping

OpenMP

Compiler directives
Library functions
Environment variables

OpenMP Library Functions

- #include <omp.h>
- Once you use OpenMP library functions, the code can no longer be compiled on compilers that doesn't support OpenMP.
- Three categories
  - Timing
    - omp_get_wtime(): get a wallclock time reference
    - omp_get_wtick(): get the resolution of wallclock time.
  - Thread information and control
  - Locking

Threading

omp_get_num_procs(): get the number of available processors

omp_get_num_threads(): get the number of threads currently using in parallel execution
omp_get_max_threads(): get the max. number of threads that can be used for parallel execution
omp_get_thread_num(): get the identifier (~ MPI_Comm_rank()) of the current thread.
omp_in_parallel(): returns true if the code is in a parallel region

omp_set_num_threads(): suggest the number of threads to be used in parallel execution.

Locking

- Locks are used for synchronization
- OpenMP Locks are represented by "**lock variables**", which can only be accessed through OpenMP locking functions.
- An OpenMP lock has one of the following three states: uninitialized, unlocked, or locked.

Locking

omp_init_lock(): initializes a simple lock.

omp_destroy_lock(): uninitializes a simple lock.

omp_set_lock(): **waits** until a simple lock is available, and then sets it.
omp_test_lock(): **tests** a simple lock, and **sets** it if it is available.

omp_unset_lock(): unsets a simple lock.

20a.cpp

```
#include <iostream>
using namespace std;
#include "omp.h"
void doSomething() {
  cout << "Executed by many threads at the same time." << endl;}
void alone() {
  cout << "Executed by exactly one thread at any time." << endl;}

int main() {
  int id;
  omp_lock_t Lock;
  omp_init_lock(&Lock);
  #pragma omp parallel
  {
    id = omp_get_thread_num();
    omp_set_lock(&Lock);
    cout << "Thread ID " << id << " entered! " << endl;
    omp_unset_lock(&Lock);
    while(! omp_test_lock(&Lock)) {
        doSomething();
    }
    alone();
    omp_unset_lock(&Lock);
  }
  return 0;
}
```

12

OpenMP

Compiler directives
Library functions
Environment variables

Environment Variables

- **OMP_SCHEDULE**
  - **#pragma omp for schedule(runtime)**
  - Linux, bash: export OMP_SCHEDLE='dynamic, 3072'
  - Windows: set OMP_SCHEDULE=dynamic, 3072
- **OMP_NUM_THREADS**
  - *Suggests* the number of threads for parallel execution

OpenMP vs. MPI

- Easy to incrementally parallelize
- More difficult to write highly scalable programs
- Small API based on compiler directives and limited library routines
- Same program can be used for sequential and parallel execution
- Shared vs private variables can cause confusion
- Portable to all platforms
- Parallelize all or nothing
- Vast collection of library routines
- Possible but difficult to use same program for serial and parallel execution
- variables are local to each processor

Performance Considerations

1. Coverage and Granularity
2. Load balance
3. Locality – avoid false sharing
4. Synchronization

16

Chap. 6 of "Parallel Programming in OpenMP, Chandra et. al.

1. **Coverage** and Granularity

- Recall Amdahl's Law

$$Speedup = \frac{T_1}{T_{NP}} = \frac{T_s + T_p}{T_s + T_p \times \frac{1}{P}} = \frac{\alpha + 1}{\alpha + \frac{1}{P}} \qquad \alpha = \frac{T_s}{T_p}$$

$$Ultimate\ Speedup = 1 + \frac{1}{\alpha}$$

$$Coverage = \frac{T_p}{T_p + T_s} = \frac{1}{1 + \alpha}$$

17

1. Coverage and **Granularity**

- Granularity
  - the extent to which a system is broken down into small parts,
- In parallel computing, granularity means **the amount of computation in relation to communication**
  - **Fine-grained parallelism** means individual tasks are relatively small in terms of code size and execution time. The data are transferred among processors frequently in amounts of one or a few memory words.
  - **Coarse-grained parallelism** is the opposite: data are communicated infrequently, after larger amounts of computation.
- The finer the granularity, the greater the potential for parallelism and hence speed-up, but the greater the overheads of synchronization and communication.
- If the granularity is too fine, the performance can suffer from the increased communication overhead. On the other side, if the granularity is too coarse, the performance can suffer from load imbalance.

18

http://en.wikipedia.org/wiki/Granularity

2. Load balance

- A chain is only as strong as its weakest link
  - How do you parallelize the following code through OpenMP?

```
for(int i=0;i<n-1;i++) {
  for(int j=i+1; j<n;j++) {
    a[i][j] = c * a[i][j];
  }
}
```

•**Static scheduling**: may introduce load imbalance unless chunk size is specified.
•**Dynamic scheduling**: resolve imbalance issue, but increase synchronization overhead
•The load varies regularly with i-index, load balance can be achieved through static scheduling. If the load varies irregularly, then dynamic or guided scheduling needs to be used.
•Scheduling strategy is also affected by **DATA LOCALITY** …

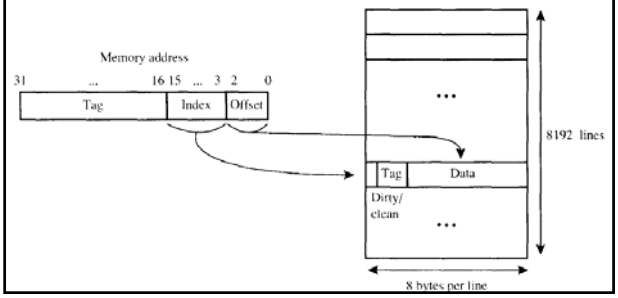
3. Data locality

- On modern cache-based machines, locality is often the **most critical factor** affecting performance.
  - **Write-through cache**: written data is always immediately written back to main memory. Cache is always consistent with main memory.
  - **Write-back cache**: written data does not write to the main memory immediately. Cache is not always consistent with main memory. One location may be updated many times before it gets written back to the main memory.

3. Data locality

An example cache system

- 32-bit address, each address refers to a byte → 4GB memory
- 64KB cache, 8192 entries of 8 bytes



3. Data locality

Data locality vs. scheduling

```
for(int i=0;i<n;i++) {
  for(int j=0;j<n;j++) {
    a[i][j] = 2.0 * a[i][j]
  }
}
```

•1 vs. 8 processors
•Dynamic scheduling w/ large enough chunk to minimize overhead
•400x400 → fit into a single processor cache
•1000x1000 → fit into aggregated cache
•4000x4000 → does not fit into cache

Size	*Static Speedup*	*Dynamic Speedup*	*Ratio: Static/ Dynamic*
400 x 400	6.2	0.6	9.9
1000 x 1000	18.3	1.8	10.3
4000 x 4000	7.5	3.9	1.9

3. Data locality

False sharing

```
int local_s[MAX_NUM_THREADS][2]={0};
#pragma omp parallel
{
  int id = omp_get_thread_num();

  #pragma omp for schedule(static)
  for(int i=0;i<n;i++) {
    int index = data[i]%2;
    local_s[id][index]++;
  }

  #pragma omp atomic
    even += local_s[id][0];
  #pragma omp atomic
    odd += local_s[id][1];
}
```

# threads	Time (sec.)
1	0.026304
2	0.06929
3	0.111161
4	0.148231

•local_s is falsely shared
•Each update to local_s invalidates the cache line, resulting in the data of the invalidated cache line being ping-ponged between processors.

3. Data locality

False sharing

```
#pragma omp parallel
{
  int result[2]={0};
  int id = omp_get_thread_num();

  #pragma omp for schedule(static)
  for(int i=0;i<n;i++) {
    int index = data[i]%2;
    result[index]++;
  }

  #pragma omp atomic
    even += result[0];
  #pragma omp atomic
    odd += result[1];
}
```

# threads	Time (sec.)
1	0.0266201
2	0.014734
3	0.01405
4	0.0140409

•Result[] is private, and the OpenMP runtime system ensures it will not fall in the same cache line as others.
•This modified version is free from false sharing

4. Synchronization

- Two kinds of synchronization
  - Barrier
  - Mutual exclusive (MUTEX) ➔ critical section → very expensive
- We want to minimize synchronization as much as possible.

25

Review of Parallel Programming

- Distributed memory
  - **Standard: MPI**
  - Explicit message passing, highly scalable
  - Standard compilers, library functions
- Accelerator
  - **CUDA, OpenCL**
  - Explicit message passing, massive parallel
  - Special compilers, intrinsic functions + library functions
- Shared memory
  - **Standard: OpenMP**
  - Implicit message passing, limited scalability
  - Supported compilers, intrinsic functions + little library functions

26

Parallel Sorting Algorithms

27

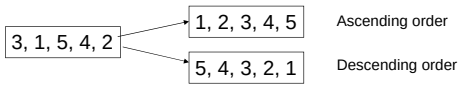
Outline

- Definition
- Issues in Sorting on Parallel Computers
- Bubble sort
- Quick sort
- Bucket and sample sort

28

Definition

- Wikipedia: In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order.



29

Issues in Sorting on Parallel Computers (1)

1. Where the Input and Output Sequences are Stored
   - Data can be distributed among processes
     - Sorting is part of other algorithms
   - Data can be stored initially on one process, then distributed to all processes
   - It is common to have the same order of sorted numbers as the process numbering. e.g. rank=0 process has the smallest numbers in the series, rank=size-1 process has the highest numbers in the series.

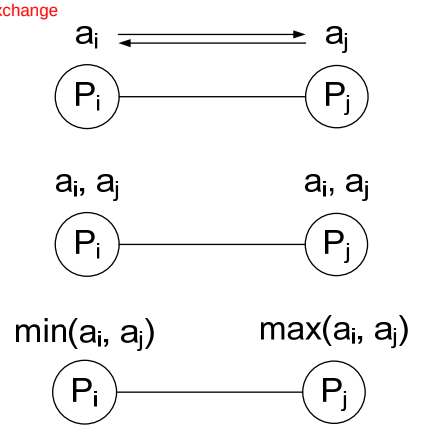
30

Issues in Sorting on Parallel Computers (2)

2. How Comparisons are Performed
   1. *One Element Per Process*
      - compare-exchange: a pair of processes (P_i, P_j) need to compare their elements, a_i and a_j.
      - After the comparison, P_i will hold the smaller and P_j the larger of {a_i, a_j}
      - Process pair exchanges the data, and retain the appropriate one.
      - very poor performance due to latency.

31

compare-exchange

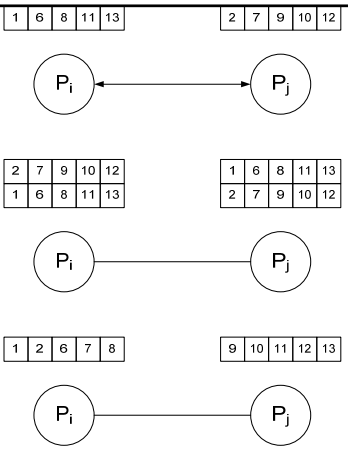


32

Issues in Sorting on Parallel Computers (3)

2. How Comparisons are Performed (cont'd)
   2. *More than One Element Per Process*
      - Assuming p processes (p_0, p_1, p_2, …, p_{p-1})
      - n is the number of elements to be sorted
      - Each process is assigned a block of n/p elements
      - Let A_0, A_1, ... A_{p-1} be the blocks assigned to processes P_0, P_1, ... P_{p-1}
      - We say that A_i<=A_j if every element of A_i is less or equal to every element in A_j.
      - Compare-split algorithm

33



compare-split

34

Bubble sort

35

Bubble Sort

- Complexity: $O(n^2)$
- Slowest sorting algorithm
- Easiest to understand and program
- Compares and exchanges adjacent elements in the sequence to be sorted → hard to parallelize

36

Bubble Sort
Sequential algorithm

```
void bubble_sort(const int N, int *dat) {
    int i, j;

    for(i=N-1; i>=0; i--) {
        for(j=0; j<i; j++) {
            compare_exchange(dat+j, dat+j+1);
        }
    }
}
```

37

Bubble Sort
Sequential algorithm in action

N=6

84, 39, 78, 79, 91, 19,
39, 84, 78, 79, 91, 19,
39, 78, 84, 79, 91, 19,
39, 78, 79, 84, 91, 19,
39, 78, 79, 84, 91, 19,
39, 78, 79, 84, 19, **91**,
39, 78, 79, 84, 19, 91,
39, 78, 79, 84, 19, 91,
39, 78, 79, 84, 19, 91,
39, 78, 79, 19, **84, 91**,
39, 78, 79, 19, 84, 91,
39, 78, 79, 19, 84, 91,
39, 78, 19, **79, 84, 91**,
39, 78, 19, 79, 84, 91,
39, 19, **78, 79, 84, 91**,
19, **39, 78, 79, 84, 91**,

38

Bubble Sort
Odd-even transposition

- Sort n elements in n phases (n is even)
- During the odd/even phase, elements with odd/even indices are compared with their right neighbors, and exchanged when out-of-sequence.
- Each phase requires n/2 compare-exchange operations.
- Easily parallelized.

39

Bubble Sort
Sequential odd-even transposition

```
void bubble_sort_odd_even(const int N, int *dat) {
    int i, j;
    int phase;

    for(i=0;i<N;i++) {
        phase = i % 2;
        for(j=phase;j<N-1;j+=2) {
            compare_exchange(dat+j, dat+j+1);
        }
    }
}
```

40

Bubble Sort
Odd-even transposition algorithm in action

84, 39, 78, 79, 91, 19,
39, 84, 78, 79, 91, 19,
39, 84, 78, 79, 91, 19,
39, 84, 78, 79, 19, 91,
39, 78, 84, 79, 19, 91,
39, 78, 84, 19, 79, 91,
39, 78, 84, 19, 79, 91,
39, 78, 19, 84, 79, 91,
39, 78, 19, 84, 79, 91,
39, 19, 78, 84, 79, 91,
39, 19, 78, 79, 84, 91,
19, 39, 78, 79, 84, 91,
19, 39, 78, 79, 84, 91,
19, 39, 78, 79, 84, 91,
19, 39, 78, 79, 84, 91,
19, 39, 78, 79, 84, 91,

41

Bubble Sort
Parallel odd-Even transposition, pseudo code

```
for(i=0;i<N;i++) {
   if(i%2 == myRank%2) {
     compare_split_min(myRank+1)
   }
   else {
     compare_split_max(myRank-1)
   }
}
```

42

Bubble Sort

Complexity of parallel odd-even transposition (1)

- p: number of processes
- n: number of elements to be sorted
- Each process: s = n/p elements
- p/2 even phases & p/2 odd phases → p phases
- Local sort: quick or merge sort → O(s*log(s))
- Each process in each phase:
  - Communication → O(s) sends and receives n/p
  - Comparison → O(s) to merge two blocks
- Parallel run time: $\underbrace{O\left(\frac{n}{p}\log\left(\frac{n}{p}\right)\right)}_{\text{Local sort}} + \underbrace{O(n)}_{\text{Communication}} + \underbrace{O(n)}_{\text{Comparison}}$

43

Bubble Sort

Complexity of parallel odd-even transposition (2)

- Parallel run time: $\underbrace{O\left(\frac{n}{p}\log\left(\frac{n}{p}\right)\right)}_{\text{Local sort}} + \underbrace{O(n)}_{\text{Communication}} + \underbrace{O(n)}_{\text{Comparison}}$
- Sequential run time: $O(n\log n)$
- Speed up:

$$\frac{O(n\log n)}{O\left(\frac{n}{p}\log\left(\frac{n}{p}\right)\right)+O(n)}$$

- Efficiency:

$$\frac{1}{1-O\left(\frac{\log p}{\log n}\right)+O\left(\frac{p}{\log n}\right)}$$

44

Quicksort

- A divide and conquer algorithm
  - Select a pivot
  - All entries smaller than pivot goes to the first sequence , and the rest goes to the 2nd sequence
  - Each sequence choose a new pivot and reapply quicksort
- Recursive algorithm
- $O(n\log_2 n)$ complexity in average, worst case scenario: $O(n^2)$

45

```
void quick_sort(const int N, int *dat, int left, int
   right)
{
   int i;
   int x;          // current pivot value
   int s;          // current pivot position
   if(left<right) {
       // we choose the left-most element to be the pivot
       x = dat[left];
       s = left;
       // go through the current sequence
       for(i=left+1;i<=right;i++) {
              if(dat[i] <= x) {
                     s++;
                     swap(dat+s, dat+i);
              }
       }
       swap(dat+left, dat+s);
       quick_sort(N, dat, left, s-1);
       quick_sort(N, dat, s+1, right);
   }
}
```

46

Quick sort in action

```
84, 39, 78, 79, 91, 19,
19, 39, 78, 79, 84, 91,
19, 39, 78, 79, 84, 91,
19, 39, 78, 79, 84, 91,
19, 39, 78, 79, 84, 91,
```

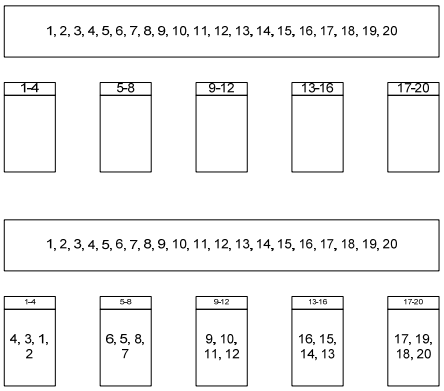
```
50, 25, 12, 75, 87,  0,
 0, 25, 12, 50, 87, 75,
 0, 25, 12, 50, 87, 75,
 0, 12, 25, 50, 87, 75,
 0, 12, 25, 50, 75, 87,
```

47

Bucket Sort

- Bucket sort
  - Assume data is uniformly distributed in [a,b]
  - Divide interval [a,b] into m equally sized buckets
  - Place each element into appropriate bucket
  - Number of elements in each bucket is ~ n/m
  - Sort elements in each bucket
- Parallelize bucket sort → each process gets a bucket
- Uniformly distributed data is overly optimistic.
  - Some buckets may have significantly more elements than others.

48



49

Sample Sort

- Improved bucket sort!
- A sample of size *s* is selected from the n-element sequence
- Sort the sampled data and choose m-1 elements (splitters)
- These elements (splitters) divide the same into m equal-sized buckets.
- Proceed with bucket sort using the defined buckets

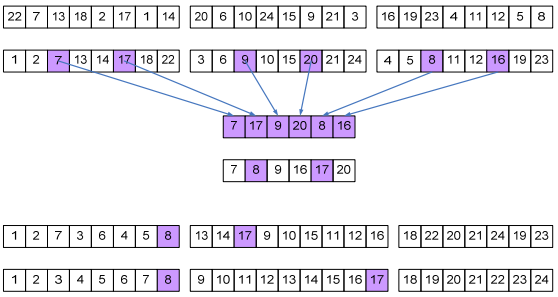
50

Parallel Sample Sort

- Each process does a local sort, chooses p-1 evenly distributed elements, and then sends selected elements onto one process (e.g. p0)
- P0 sorts p*(p-1) elements, then choose p-1 evenly space elements (splitters).
- P0 then broadcasts these p-1 splitters to all processes
- Based on splitters, each process determines which element goes to which process
- Communicate, then each process does local sort (e.g. with quicksort)

51

Parallel Sample Sort



52