OpenMP vs. MPI

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

Performance Considerations

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

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

2

1. Coverage and Granularity

- Recall Amdahl’s Law

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

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

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

3

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.

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

4

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 …

5

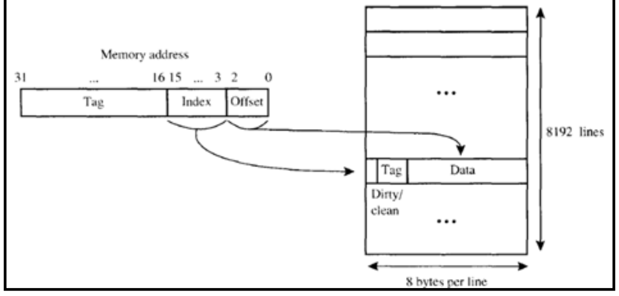
3. Data locality

- 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.

6

3. Data locality
An example cache system

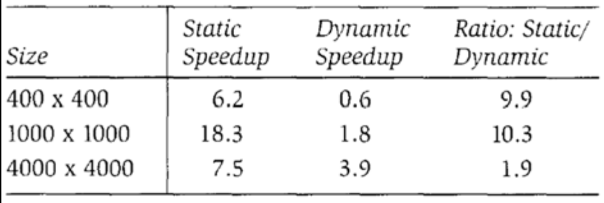
- 32-bit address, each address refers to a byte → 4GB memory
- 64KB cache, 8192 entries of 8 bytes



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.

9

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

10

4. Synchronization

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

11

Review of Parallel Programming

- Distributed memory
  - Standard: MPI
  - Explicit message passing, highly scalable
  - Standard compilers, library functions
- Accelerator
  - OpenCL, CUDA
  - 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

12

Lecture 15

Parallel Sorting Algorithm

13

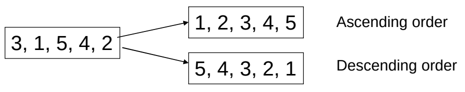
Outline

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

14

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.



15

Issues in Sorting on Parallel Computers (1)

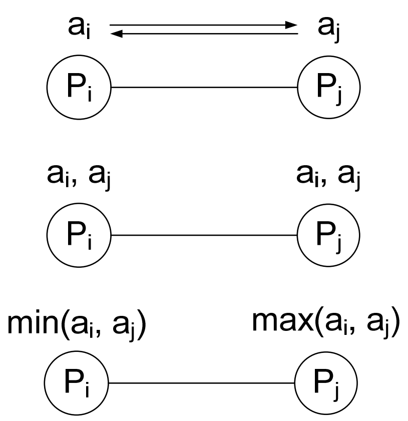
1. Where the Input and Output Sequences are Stored
   - Data can be distributed among processes
     - Sorting is part of other algorithms
   - Data can be stored initially on one process, then distributed to all processes
   - It is common to have the same order of sorted numbers as the process numbering.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

16

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.

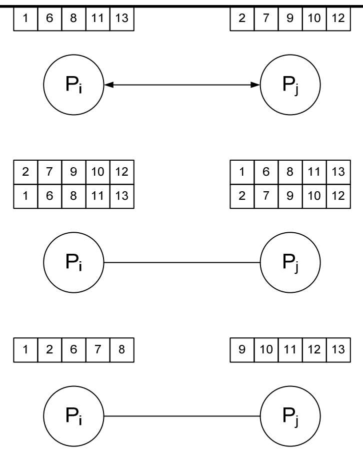
17



18

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



Bubble sort

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

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);
        }
    }
}
```

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,
```

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.

25

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);
        }
    }
}
```

26

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,
```

27

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)
   }
}
```

28

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: $\underbrace{O\left(\frac{n}{p}\log\left(\frac{n}{p}\right)\right)}_{\text{Local sort}} + \underbrace{O(n)}_{\text{Communication}} + \underbrace{O(n)}_{\text{Comparison}}$

29

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:

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

30

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: $O(n^2)$

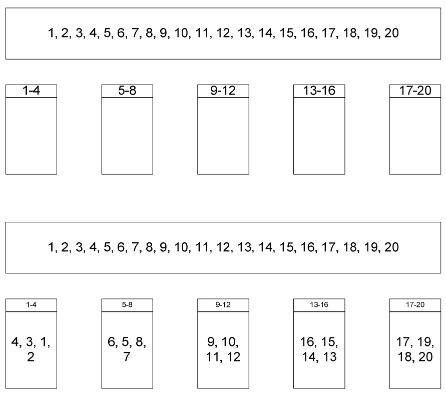
```
void quick_sort(const int N, int *dat, int left, int
  right)
{
  int i;
  int x;          // current pivot value
  int s;          // current pivot position
  if(left<right) {
      // we choose the left-most element to be the pivot
      x = dat[left];
      s = left;
      // go through the current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

Quick sort in action

```
84, 39, 78, 79, 91, 19,        50, 25, 12, 75, 87,  0,
19, 39, 78, 79, 84, 91,         0, 25, 12, 50, 87, 75,
19, 39, 78, 79, 84, 91,         0, 25, 12, 50, 87, 75,
19, 39, 78, 79, 84, 91,         0, 12, 25, 50, 87, 75,
19, 39, 78, 79, 84, 91,         0, 12, 25, 50, 75, 87,
```

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.



Sample Sort

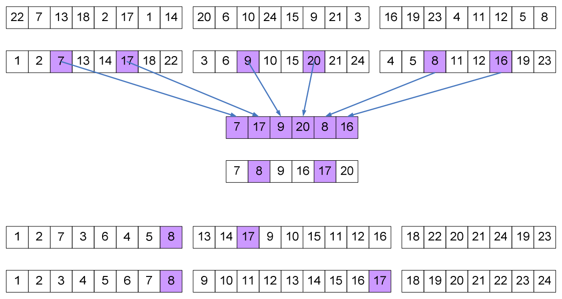
- Improved bucket sort!
- A sample of size *s* is selected from the n-element sequence
- Sort the sampled data and choose m-1 elements (splitters)
- These elements (splitters) divide the same into m equal-sized buckets.
- Proceed with bucket sort using the defined buckets

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)

37

Parallel Sample Sort



38

Parallel Matrix Algorithm

Vectors

Matrix-Vector product

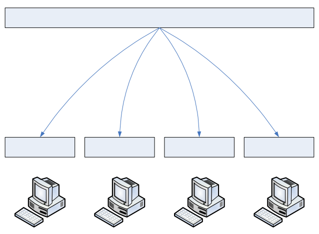
Matrix-matrix multiplication

39

Parallel Linear Algebra Algorithms

Vectors (1)

- Vectors are essentially 1-D arrays.
- The only natural way to decompose vectors is linear decomposition.



40

Parallel Linear Algebra Algorithms

Vectors (2)

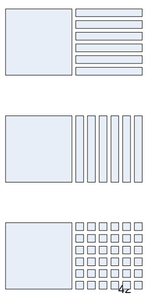
- In the past lectures, we've calculated:
  - Vector length (L2-norm) of a vector.
    - MPI_Scatterv, MPI_Reduce
  - Vector inner-product of two vectors.
    - A bad algorithm using Broadcast.
    - MPI_Scatterv should have been used.
      - Broadcast vector size
      - Scatter vector1
      - Scatter vector2
      - innerDot = cblas_ddot(vector1, vector2)
      - Reduce innerDot with MPI_SUM

41

Parallel Linear Algebra Algorithms

Matrix decomposition

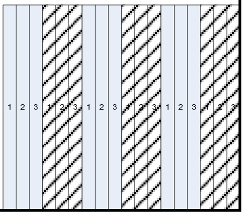
- For matrices, there are more choices for data decomposition
  - 1-D decomposition
    - 1-D block row
    - 1-D block column
  - 2-D decomposition
    - Harder to program
    - May be good for load-balance
    - May be good for Gauss elimination



42

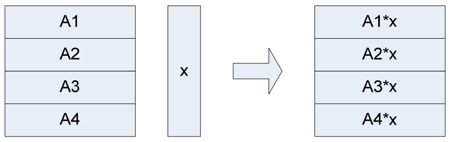
Parallel Linear Algebra Algorithms
Matrix distribution

- Block distribution
  - Each block (column, row, …) is distributed onto a single processor
- Cyclic distribution
  - Each block is furthered scattered onto all available processes. This is good for Gauss elimination and for many other algorithms in terms of load-balancing.

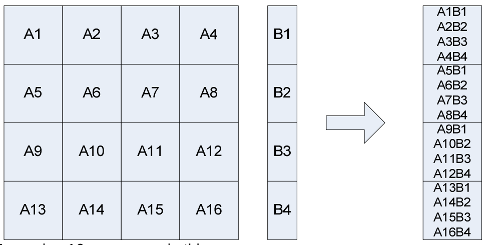


Parallel Linear Algebra Algorithms
Matrix-Vector Product, 1-D algorithm

- Performing y = A*x, where x, y are vectors, and A is a square matrix
- We use 1-D block row decomposition & distribution
- BLAS-2 (cblas_dgemv) is used for performing matrix-vector product on local processes.
- x is broadcasted, A is scattered, and y is gathered.



Parallel Linear Algebra Algorithms
Matrix-Vector Product, 2-D algorithm



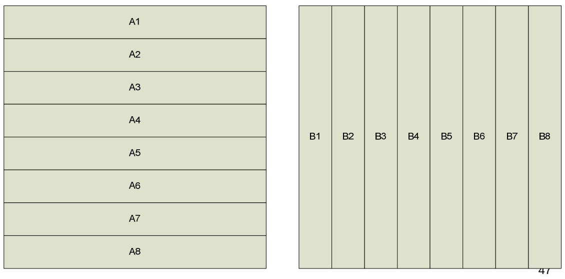
1. Assuming 16 processors in this case
2. Distribute matrices to available processors
3. Scatter vector onto 4 processors (1 – 4, or 1, 6, 11, 16)
4. In each column block, perform one-to-all broadcast of the vector
5. Perform matrix-vector product in each process
6. In each row block, perform all-to-one reduce → Answer

Summary, Matrix-vector product

- 2-D algorithms are usually more scalable
  - 1-D algorithm cannot be applied when the number of column/rows is less than the number of processes.
  - 2-D algorithm can be applied up to the number of entries in the matrix. (So each process has exactly one entry. However, the efficiency will not be good.)

Parallel Linear Algebra Algorithms
Matrix-Matrix Product
Simple algorithm (1)

With 1-D data decomposition for two matrices into 8 divisions (using 8 processes)



Parallel Linear Algebra Algorithms
Matrix-Matrix Product
Simple algorithm (2)

1. Each process does a local multiplication
2. Send the B sub-matrix to the previous process (with wrap around)
3. Loop 1-2 until all submatrices are computed

Parallel Linear Algebra Algorithms
Matrix-Matrix Product
Simple algorithm (3) - Summary

Matrix Layout

A1B1	A1B2	A1B3	A1B4	A1B5	A1B6	A1B7	A1B8
A2B1	A2B2	A2B3	A2B4	A2B5	A2B6	A2B7	A2B8
A3B1	A3B2	A3B3	A3B4	A3B5	A3B6	A3B7	A3B8
A4B1	A4B2	A4B3	A4B4	A4B5	A4B6	A4B7	A4B8
A5B1	A5B2	A5B3	A5B4	A5B5	A5B6	A5B7	A5B8
A6B1	A6B2	A6B3	A6B4	A6B5	A6B6	A6B7	A6B8
A7B1	A7B2	A7B3	A7B4	A7B5	A7B6	A7B7	A7B8
A8B1	A8B2	A8B3	A8B4	A8B5	A8B6	A8B7	A8B8

Calculation Order Color Code

1
2
3
4
5
6
7
8

49

Parallel Linear Algebra Algorithms
Cannon's Algorithm (1) - Decomposition

A1	A2	A3	A4
A5	A6	A7	A8
A9	A10	A11	A12
A13	A14	A15	A16

B1	B2	B3	B4
B5	B6	B7	B8
B9	B10	B11	B12
B13	B14	B15	B16

•Assuming 16 processors are available
•We want to perform calculate C=A*B
•Matrices can be decomposed as shown above

50

Parallel Linear Algebra Algorithms
Cannon's Algorithm (2) - Distribution

A1	A2	A3	A4
A6	A7	A8	A5
A11	A12	A9	A10
A16	A13	A14	A15

B1	B6	B11	B16
B5	B10	B15	B4
B9	B14	B3	B8
B13	B2	B7	B12

•Both A and B are distributed in a skewed way
•A is skewed leftwards
•B is skewed upwards
•Compute local multiplication

51

Parallel Linear Algebra Algorithms
Cannon's Algorithm (3) - Computation

A2	A3	A4	A1
A7	A8	A5	A6
A12	A9	A10	A11
A13	A14	A15	A16

B5	B10	B15	B4
B9	B14	B3	B8
B13	B2	B7	B12
B1	B6	B11	B16

•Send its A submatrix leftwards (with wrap around)
•Send its B submatrix upwards (with wrap around)
•Multiply new submatrices and accumulate it to the answer

52

Parallel Linear Algebra Algorithms
Cannon's Algorithm (4) - Computation

A3	A4	A1	A2
A8	A5	A6	A7
A9	A10	A11	A12
A14	A15	A16	A13

B9	B14	B3	B8
B13	B2	B7	B12
B1	B6	B11	B16
B5	B10	B15	B4

•Send its A submatrix leftwards (with wrap around)
•Send its B submatrix upwards (with wrap around)
•Multiply new submatrices and accumulate it to the answer

53

Parallel Linear Algebra Algorithms
Cannon's Algorithm (5) - Computation

A4	A1	A2	A3
A5	A6	A7	A8
A10	A11	A12	A9
A15	A16	A13	A14

B13	B2	B7	B12
B1	B6	B11	B16
B5	B10	B15	B4
B9	B14	B3	B8

•Send its A submatrix leftwards (with wrap around)
•Send its B submatrix upwards (with wrap around)
•Multiply new submatrices and accumulate it to the answer

54

Parallel Linear Algebra Algorithms
Cannon's Algorithm (6) – Final result

A1B1 A2B5 A3B9 A4B13	A2B6 A3B10 A4B14 A1B2	A3B11 A4B15 A1B3 A2B7	A4B16 A1B4 A2B8 A3B12
A6B5 A7B9 A8B13 A5B1	A7B10 A8B14 A5B2 A6B6	A8B15 A5B3 A6B7 A7B11	A5B4 A6B8 A7B12 A8B16
A11B9 A12B13 A9B1 A10B5	A12B14 A9B2 A10B6 A11B10	A9B3 A10B7 A11B11 A12B15	A10B8 A11B12 A12B16 A9B4
A16B13 A13B1 A14B5 A15B9	A13B2 A14B6 A15B10 A16B14	A14B7 A15B11 A16B15 A13B3	A15B12 A16B16 A13B4 A14B8

55

Parallel Linear Algebra Algorithms
Matrix-Matrix Multiplication

- Comparing the simple (1-D partitioning) and Cannon algorithm (2-D partitioning)
  - With NP processes, 1-D partitioning requires NP communication steps, and 2-D partition requires sqrt(NP) communication steps.
  - 1-D algorithm sends 1/NP of B matrix to the next/previous process in each communication step
  - 2-D algorithm sends 1/NP of A matrix leftwards, and 1/NP of B matrix to upwards in each communication step

56

Comparing 1-D and 2-D partitioning

Per Process!

	Communication Steps		Amount of Communca		Total communcations/	
NP	1-D	2-D	1-D	2-D	1-D	2-D
4	4	2	0.25	0.50	1.00	1.00
9	9	3	0.11	0.22	1.00	0.67
16	16	4	0.06	0.13	1.00	0.50
25	25	5	0.04	0.08	1.00	0.40
36	36	6	0.03	0.06	1.00	0.33
49	49	7	0.02	0.04	1.00	0.29
64	64	8	0.02	0.03	1.00	0.25
81	81	9	0.01	0.02	1.00	0.22
100	100	10	0.01	0.02	1.00	0.20
	"=NP"	"=sqrt(NP)'	"=1/NP"	"=2/NP"	"=Amount * Steps"	

57

Notice

- We're having a discussion on assignments on 6/14? (no class is scheduled from school's point of view!)
- First part (programming) of the final exam will be distributed on 6/14, and to be returned on 6/15 (noon or 5:00PM, haven't decided yet.)
- Final exam on 6/21
  - Written part (open book)
- Presentation on 6/28

58