

Parallel & Distributed Computing: Lecture 10

Alberto Paoluzzi

November 15, 2018

- 1 Matrix products (continue)
- 2 Decomposition Techniques
- 3 Parallel Algorithm Models

Matrix products (continue)

The evolution of lower bounds for matrix multiplication

Until the late 1960s computing the product C of two $n \times n$ matrices required a cubic number of operations, as the **fastest algorithm known** was the **naive algorithm** which runs in $O(n^3)$ time

In 1969, **Strassen** gave the first **subcubic time algorithm** for matrix multiplication, running in $O(n^{2.808})$ time

This amazing discovery spawned a long line of research which gradually reduced the **matrix multiplication exponent** ω over time.

The first result to break 2.5 was by **Coppersmith and Winograd** who obtained $\omega < 2.496$.

Three years later, **Coppersmith and Winograd** combined Strassen's technique with a novel form of analysis based on large sets avoiding arithmetic progressions and obtained the bound of $\omega < 2.376$, **unchanged for more than twenty years**

Cache oblivious matrix multiplication

Source: Michael Bader and Christoph Zenger, Cache oblivious matrix multiplication using an element ordering based on the Peano curve, Linear Algebra and its Applications, 2006

Algorithm 1 multiplication of two n -by- n matrices

```
for i from 1 to n do
  for j from 1 to n do
    C[i,j] := 0;
    for k from 1 to n to
      C[i,j] := C[i,j] + A[i,k] * B[k,j];
    end do;
  end do;
end do;
```

Cache oblivious matrix multiplication

Source: Michael Bader and Christoph Zenger, Cache oblivious matrix multiplication using an element ordering based on the Peano curve, Linear Algebra and its Applications, 2006

Algorithm 1 multiplication of two n-by-n matrices

```
for i from 1 to n do
  for j from 1 to n do
    C[i,j] := 0;
    for k from 1 to n to
      C[i,j] := C[i,j] + A[i,k] * B[k,j];
    end do;
  end do;
end do;
```

Algorithm 2 multiplication of two n-by-n matrices (revisited)

```
// matrix C is assumed to be initialized
for all triples (i,j,k) in {1..n}x{1..n}x{1..n} do
  C[i,j] := C[i,j] + A[i,k] * B[k,j];
end do;
```

Cache oblivious matrix multiplication

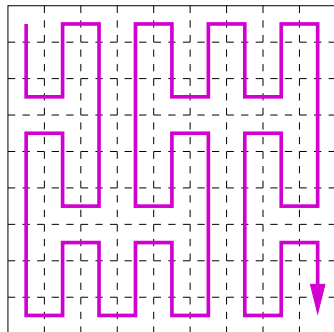
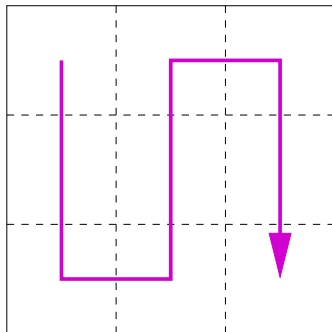
Source: Michael Bader and Christoph Zenger, Cache oblivious matrix multiplication using an element ordering based on the Peano curve, Linear Algebra and its Applications, 2006

Algorithm 2 multiplication of two n-by-n matrices (revisited)

```
// matrix C is assumed to be initialized
for all triples (i,j,k) in {1..n}x{1..n}x{1..n} do
    C[i,j] := C[i,j] + A[i,k] * B[k,j];
end do;
```

- may be executed in **any order**, because of the commutativity
- find **optimal serializations** of the **loop**
- shows **better locality** of the element access, and can benefit from the **presence of cache memory**

Recursive construction of the Peano curve



iterations of the Peano curve are generated in a self-similar, recursive process

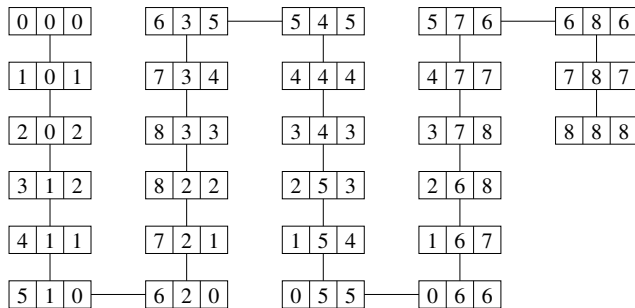
Elements of matrices are stored in Peano-like order

$$\underbrace{\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix}}_{=: A} \underbrace{\begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix}}_{=: B} = \underbrace{\begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}}_{=: C}$$

$$c_k = \sum_{(i,j) \in C_k} a_i b_j$$

$$C_0 = \{(a_0, b_0), (a_5, b_1), (a_6, b_2)\}, \dots \text{ etc } \dots$$

Graph representation of optimal serialization



Graph representation of the operations of a 3×3 (also odd-by-odd) matrix multiplication

after each element operation, either directly re-use a matrix element, or move to its direct neighbour

Recursive construction of the Peano curve

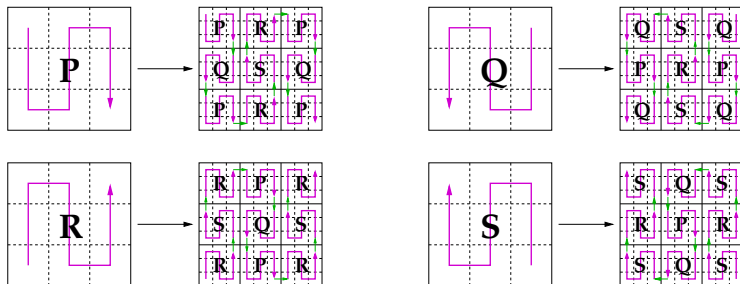
According to **Algorithm 2** to compute the matrix-matrix product, we have to perform two steps:

- ① initialize all $c_k := 0$ for $k = 0, \dots, 8$
- ② for all triples (k, i, j) where $(i, j) \in C_k$, and $k = 0, \dots, 8$ execute:

$$c_k := c_k + a_i b_j$$

- optimally **localized execution order** of the operations, **avoiding jumps** in k , i , and j

Recursive block numbering scheme based on Peano curve



Implementation

$$\underbrace{\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix}}_{=: A} \underbrace{\begin{pmatrix} P_{B0} & R_{B5} & P_{B6} \\ Q_{B1} & S_{B4} & Q_{B7} \\ P_{B2} & R_{B3} & P_{B8} \end{pmatrix}}_{=: B} = \underbrace{\begin{pmatrix} P_{C0} & R_{C5} & P_{C6} \\ Q_{C1} & S_{C4} & Q_{C7} \\ P_{C2} & R_{C3} & P_{C8} \end{pmatrix}}_{=: C}$$

We get the following operations on the matrix blocks:

$$P_{C0} := P_{A0}P_{B0} + R_{A5}Q_{B1} + P_{A6}P_{B2}$$

$$Q_{C1} := Q_{A1}P_{B0} + S_{A4}Q_{B1} + Q_{A7}P_{B2}$$

$$R_{C5} := P_{A0}R_{B5} + R_{A5}S_{B4} + P_{A6}R_{B3}$$

$$S_{C4} := Q_{A1}R_{B5} + S_{A4}S_{B4} + Q_{A7}R_{B3}$$

Algorithmic scheme

$$\begin{aligned}
 P_{C0} &:= 0 \\
 P_{C0} &\stackrel{+}{\leftarrow} P_{A0}P_{B0} \quad (\text{short notation for } P_{C0} := P_{C0} + P_{A0}P_{B0}) \\
 P_{C0} &\stackrel{+}{\leftarrow} R_{A5}Q_{B1} \\
 P_{C0} &\stackrel{+}{\leftarrow} P_{A6}P_{B2}
 \end{aligned} \tag{8}$$

If we just consider the ordering of the matrix blocks, we can see that there are exactly eight different types of block multiplications:

$$\begin{aligned}
 P &\stackrel{+}{\leftarrow} PP & P &\stackrel{+}{\leftarrow} RQ \\
 Q &\stackrel{+}{\leftarrow} QP & Q &\stackrel{+}{\leftarrow} SQ \\
 R &\stackrel{+}{\leftarrow} PR & R &\stackrel{+}{\leftarrow} RS \\
 S &\stackrel{+}{\leftarrow} QR & S &\stackrel{+}{\leftarrow} SS.
 \end{aligned} \tag{9}$$

Pseudocoding

Algorithm 3 Recursive implementation of the Peano matrix multiplication

```

/* global variables:
 * A, B, C: the matrices, C will hold the result of AB
 * a, b, c: indices of the matrix element of A, B, and C
 */
peanoMult(int pbsA, int pbsB, int pbsC, int dim)
{
    if (dim == 1) {
        C[c] += A[a] * B[b];
    }
    else
    {
        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a += pbsA; c += pbsC;
        peanoMult( pbsA, -pbsB,  pbsC, dim/3); a += pbsA; c += pbsC;
        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a += pbsA; b += pbsB;

        peanoMult( pbsA,  pbsB, -pbsC, dim/3); a += pbsA; c -= pbsC;
        peanoMult( pbsA, -pbsB, -pbsC, dim/3); a += pbsA; c -= pbsC;
        peanoMult( pbsA,  pbsB, -pbsC, dim/3); a += pbsA; b += pbsB;

        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a += pbsA; c += pbsC;
        peanoMult( pbsA, -pbsB,  pbsC, dim/3); a += pbsA; c += pbsC;
        peanoMult( pbsA,  pbsB,  pbsC, dim/3); b += pbsB; c += pbsC;

        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a -= pbsA; c += pbsC;
        peanoMult( pbsA, -pbsB,  pbsC, dim/3); a -= pbsA; c += pbsC;
        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a -= pbsA; b += pbsB;

        peanoMult( pbsA,  pbsB, -pbsC, dim/3); a -= pbsA; c -= pbsC;
        peanoMult( pbsA, -pbsB, -pbsC, dim/3); a -= pbsA; c -= pbsC;
        peanoMult( pbsA,  pbsB, -pbsC, dim/3); a -= pbsA; b += pbsB;

        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a -= pbsA; c += pbsC;
        peanoMult( pbsA, -pbsB,  pbsC, dim/3); a -= pbsA; c += pbsC;
        peanoMult( pbsA,  pbsB,  pbsC, dim/3); b += pbsB; c += pbsC;

        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a += pbsA; c += pbsC;
        peanoMult( pbsA, -pbsB,  pbsC, dim/3); a += pbsA; c += pbsC;
        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a += pbsA; b += pbsB;

        peanoMult( pbsA,  pbsB, -pbsC, dim/3); a += pbsA; c -= pbsC;
        peanoMult( pbsA, -pbsB, -pbsC, dim/3); a += pbsA; c -= pbsC;
        peanoMult( pbsA,  pbsB, -pbsC, dim/3); a += pbsA; b += pbsB;

        peanoMult( pbsA,  pbsB,  pbsC, dim/3); a += pbsA; c += pbsC;
        peanoMult( pbsA, -pbsB,  pbsC, dim/3); a += pbsA; c += pbsC;
        peanoMult( pbsA,  pbsB,  pbsC, dim/3);
    }
};
  
```

Decomposition Techniques

Introduction

Some commonly used **decomposition techniques** for **achieving concurrency**

- **recursive** decomposition,
- **data**-decomposition,
- **exploratory** decomposition,
- **speculative** decomposition
- **hybrid** decomposition (**mixing the above** methods)

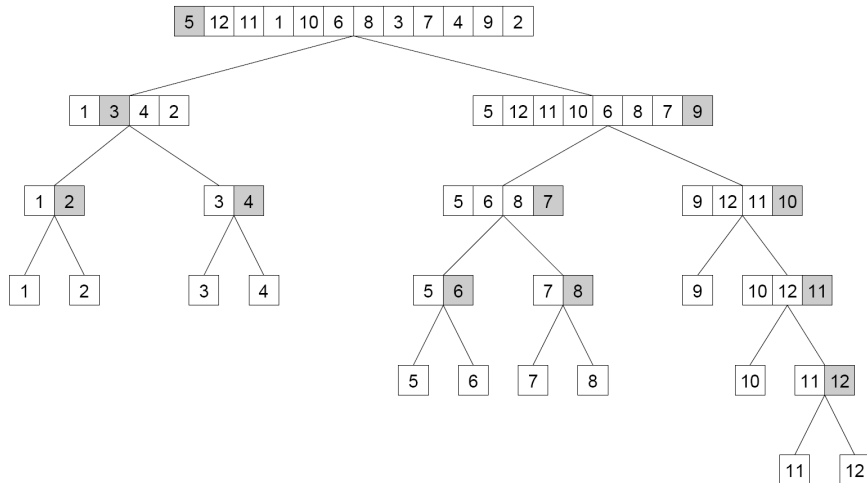
Good starting point for many problems

A **combination of techniques** can be used to obtain **effective decompositions** for a **large set of problems**

Recursive Decomposition (divide-and-conquer)

- **Recursive decomposition** is a method for inducing concurrency in problems that can be solved using the **divide-and-conquer strategy**
- A problem is solved by first **dividing it into a set** of **independent subproblems**.
- **Each subproblem** is solved by **recursively applying** a similar division **into smaller subproblems** followed by a **combination** of their results.

Recursive Decomposition (Quicksort)



also represents the **task graph** for the problem

Recursive Decomposition

A serial program for finding the minimum in an array of numbers A of length n

```
1.  procedure SERIAL_MIN ( $A, n$ )
2.  begin
3.     $min = A[0];$ 
4.    for  $i := 1$  to  $n - 1$  do
5.      if ( $A[i] < min$ )  $min := A[i];$ 
6.    endfor;
7.    return  $min;$ 
8.  end SERIAL_MIN
```

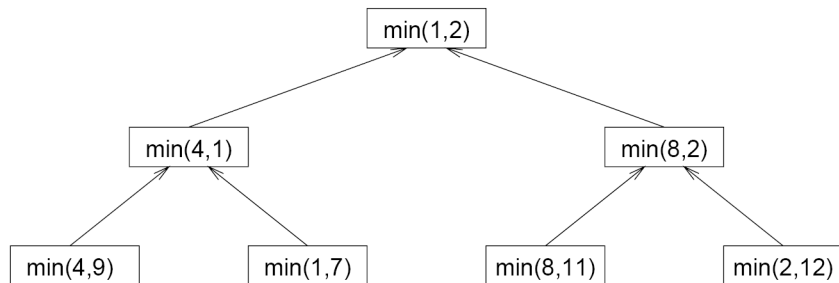
Algorithm 3.1 A serial program for finding the minimum in an array of numbers A of length n .

Recursive Decomposition

```
1.  procedure RECURSIVE_MIN ( $A, n$ )
2.  begin
3.  if ( $n = 1$ ) then
4.     $min := A[0]$ ;
5.  else
6.     $lmin := \text{RECURSIVE\_MIN}(A, n/2)$ ;
7.     $rmin := \text{RECURSIVE\_MIN}(\&(A[n/2]), n - n/2)$ ;
8.    if ( $lmin < rmin$ ) then
9.       $min := lmin$ ;
10.   else
11.      $min := rmin$ ;
12.   endelse;
13. endelse;
14. return  $min$ ;
15. end RECURSIVE_MIN
```

Recursive Decomposition

The **task-dependency graph** for finding the minimum number in the sequence $\{4, 9, 1, 7, 8, 11, 2, 12\}$.



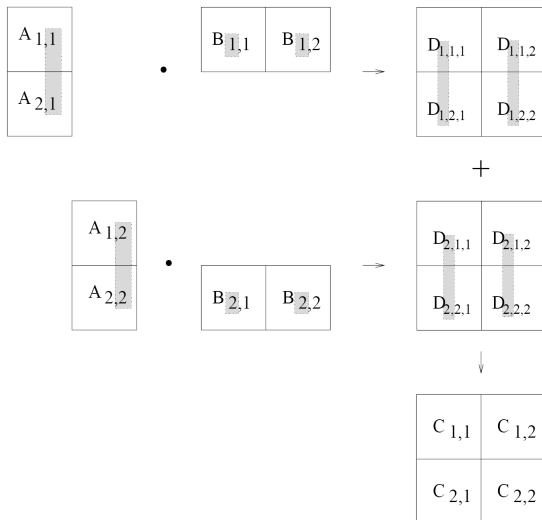
Each node in the tree represents the **task of finding the minimum of a pair** of numbers.

with n numbers and p tasks, **complexity** $O(n/p + \log(p))$

Recursive Decomposition

```
1.  procedure RECURSIVE_MIN (A, n)
2.  begin
3.  if (n = 1) then
4.      min := A[0];
5.  else
6.      lmin := RECURSIVE_MIN (A, n/2);
7.      rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
8.      if (lmin < rmin) then
9.          min := lmin;
10.     else
11.         min := rmin;
12.     endelse;
13. endelse;
14. return min;
15. end RECURSIVE_MIN
```

Data Decomposition (matrix product: intermediate decomposition)



Data Decomposition (matrix product: intermediate decomposition)

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,1} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,1} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,1} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,1} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

Task 01: $D_{1,1,1} = A_{1,1}B_{1,1}$

Task 02: $D_{2,1,1} = A_{1,2}B_{2,1}$

Task 03: $D_{1,1,2} = A_{1,1}B_{1,2}$

Task 04: $D_{2,1,2} = A_{1,2}B_{2,2}$

Task 05: $D_{1,2,1} = A_{2,1}B_{1,1}$

Task 06: $D_{2,2,1} = A_{2,2}B_{2,1}$

Task 07: $D_{1,2,2} = A_{2,1}B_{1,2}$

Task 08: $D_{2,2,2} = A_{2,2}B_{2,2}$

Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$

Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Exploratory decomposition

1	2	3	4
5	6	7	8
9	10	7	11
13	14	15	12

(a)

1	2	3	4
5	6	7	8
9	10	11	11
13	14	15	12

(b)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12

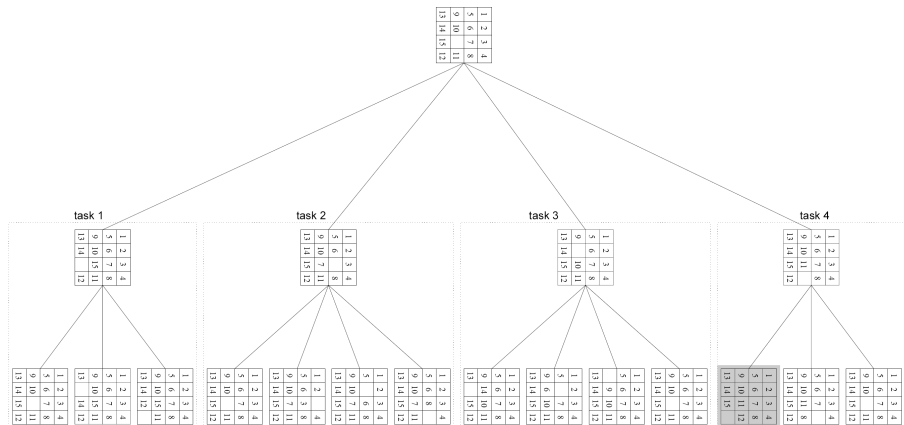
(c)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(d)

Exploratory decomposition is used to decompose problems whose underlying computations correspond to a search of a space for solutions

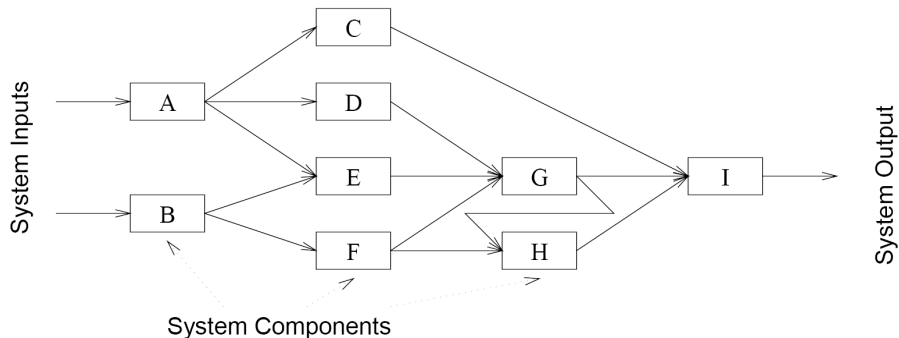
Exploratory decomposition



partition the search space into smaller parts, and search each concurrently, until the desired solutions are found

Speculative decomposition

Speculative decomposition is used when a **program may take** one of many possible computationally **significant branches** depending on the **output** of other computations



An **example** of an application in which **speculative decomposition** is useful is **discrete event simulation**.

Hybrid decomposition

```

1.  procedure COL_LU (A)
2.  begin
3.      for k := 1 to n do
4.          for j := k to n do
5.               $A[j, k] := A[j, k] / A[k, k];$ 
6.          endfor;
7.          for j := k + 1 to n do
8.              for i := k + 1 to n do
9.                   $A[i, j] := A[i, j] - A[i, k] \times A[k, j];$ 
10.             endfor;
11.          endfor;
    /*

```

After this iteration, column $A[k + 1 : n, k]$ is logically the k th column of L and row $A[k, k : n]$ is logically the k th row of U .

```

    */
12.      endfor;
13. end COL_LU

```

Hybrid decomposition

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

$$\begin{array}{l|l|l} 1: A_{1,1} \rightarrow L_{1,1}U_{1,1} & 6: A_{2,2} = A_{2,2} - L_{2,1}U_{1,2} & 11: L_{3,2} = A_{3,2}U_{2,2}^{-1} \\ 2: L_{2,1} = A_{2,1}U_{1,1}^{-1} & 7: A_{3,2} = A_{3,2} - L_{3,1}U_{1,2} & 12: U_{2,3} = L_{2,2}^{-1}A_{2,3} \\ 3: L_{3,1} = A_{3,1}U_{1,1}^{-1} & 8: A_{2,3} = A_{2,3} - L_{2,1}U_{1,3} & 13: A_{3,3} = A_{3,3} - L_{3,2}U_{2,3} \\ 4: U_{1,2} = L_{1,1}^{-1}A_{1,2} & 9: A_{3,3} = A_{3,3} - L_{3,1}U_{1,3} & 14: A_{3,3} \rightarrow L_{3,3}U_{3,3} \\ 5: U_{1,3} = L_{1,1}^{-1}A_{1,3} & 10: A_{2,2} \rightarrow L_{2,2}U_{2,2} & \end{array}$$

Parallel Algorithm Models

Introduction: parallel algorithm models

Source: Introduction to Parallel Computing, Second Edition, By Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Addison Wesley, 2004

An **algorithm model** is typically a way of structuring a parallel algorithm by **selecting** a **decomposition** and **mapping technique** and applying the appropriate strategy to **minimize interactions**

- Data-Parallel
- Task Graph
- Work Pool
- Master-Slave
- Producer-Consumer
- Hybrid Models

Data-Parallel Model

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

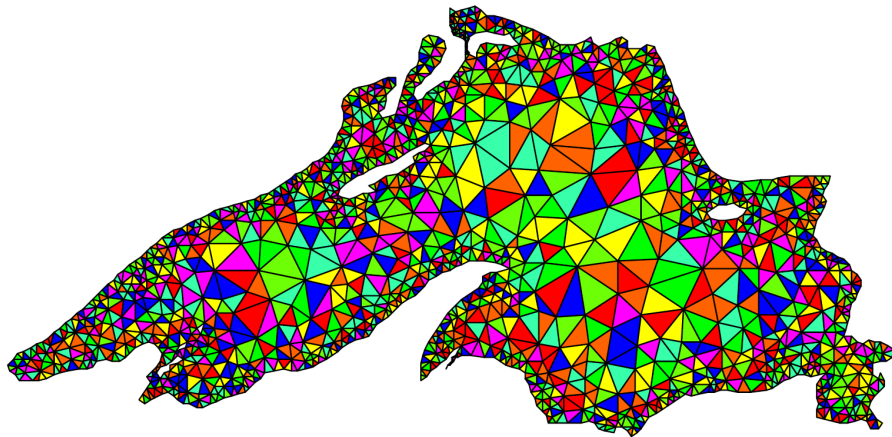
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

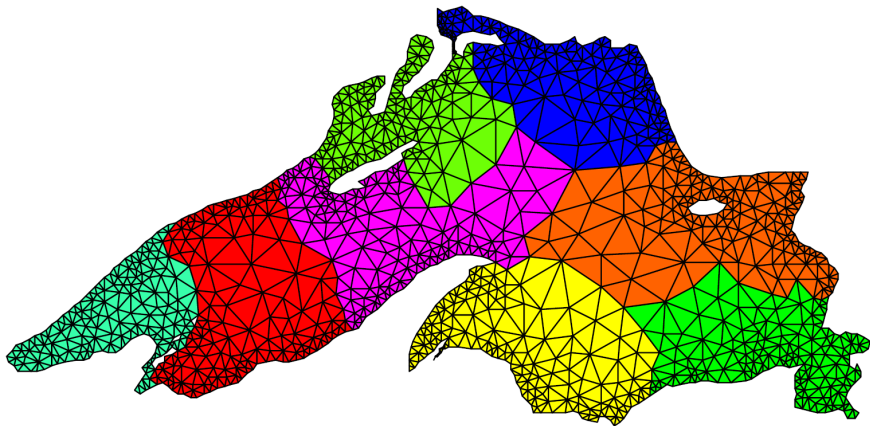
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Data-Parallel (geographical mesh)



Data-Parallel Model (geographical mesh)



Task Graph Model

(a)

1	2	3	4	5	6	7	8
33	21	13	54	82	33	40	72

(b) root = 4

	1	2	3	4	5	6	7	8
<i>leftchild</i>				1				
<i>rightchild</i>				5				

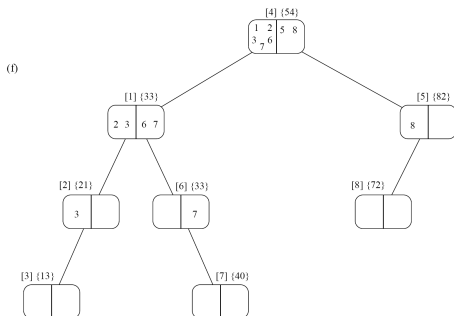
(c)

(d)

	1	2	3	4	5	6	7	8
<i>leftchild</i>	2			1	8			
<i>rightchild</i>	6			5				

	1	2	3	4	5	6	7	8
<i>leftchild</i>	2	3		1	8			
<i>rightchild</i>	6			5	7			

(e)



Work Pool Model

The **work pool** or the **task pool model** is characterized by a **dynamic mapping of tasks onto processes** for load balancing in which **any task** may potentially be performed by **any process**

There is **no desired premapping** of **tasks onto processes**.

In the **message-passing** paradigm, the work pool model is **typically used** when the **amount of data** associated with tasks **is relatively small** compared to the **computation** associated with the tasks.

Master-Slave Model

In the master-slave or the manager-worker model, **master processes generate work** and **allocate it to worker processes**.

The tasks may be **allocated a priori** if the manager **can estimate** the size of tasks or if a **random mapping** can do an adequate job of **load balancing**.

Usually, there is **no desired premapping** of **work to processes**, and **any worker** can do **any job** assigned to it.

This model is generally suitable to **shared-address-space** or **message-passing paradigms** since the **interaction** is naturally **two-way**:

- **managers know** that they need to give out work and
- **workers know** that they need to get work from the manager.

While using the **master-slave model**, care should be taken to ensure that the **master does not become a bottleneck**, which may happen if the tasks are too small (or the workers are relatively fast).

Pipeline or Producer-Consumer Model

In the **pipeline model**, a **stream of data** is passed on **through a succession of processes**, each of which perform some task on it.

This **simultaneous execution** of different **programs** on a **data stream** is called **stream parallelism**.

A **pipeline** is a **chain of producers and consumers**. Each process in the pipeline can be viewed as a **consumer** of a sequence of data items for the process preceding it in the pipeline **and** as a **producer of data** for the process following it in the pipeline.

The pipeline **does not need** to be a **linear chain**; it can be a **directed graph**.

The **pipeline model** usually involves a **static mapping** of tasks onto processes.

Hybrid Models

more than one model may be applicable to the problem, resulting in a hybrid algorithm model

A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.

Hybrid Models

more than one model may be applicable to the problem, resulting in a hybrid algorithm model

A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.

