Parallel Processing: Why, When, How?

SimLab2010, Belgrade
October, 5, 2010
Rodica Potolea
Parallel Processing Why, When, How?

• Why?
  • Problems “too” costly to be solved with the classical approach
  • The need of results on specific (or reasonable) time

• When?
  • Are there any sequences which are better suited for parallel implementation
  • Are there situations when is better NOT to parallelize

• How?
  • Which are the constraints (if any) when we need to pass from sequential to parallel implementation
Outline

- Computation paradigms
- Constraints in parallel computation
- Dependence analysis
- Techniques for dependences removal
- Systolic architectures & simulations
- Increase the parallel potential
- Examples on GRID
Outline

- Computation paradigms
- Constraints in parallel computation
- Dependence analysis
- Techniques for dependences removal
- Systolic architectures & simulations
- Increase the parallel potential
- Examples on GRID
Computation paradigms

• Sequential model –
  • referred to as Von Neumann architecture
  • basis of comparison
  • he simultaneously proposed other models
  • due to the technological constraints, at that time, only the sequential model has been implemented
  • Von Neumann could be considered as the founder of parallel computers as well (conceptual view)
• Flynn’s taxonomy
  (according to instruction/data flow single/multiple)
  • SI SD
  • SI MD  single control unit dispatches instructions to each processing unit
  • MI SD  the same data is analyzed from different points of view (MI)
  • MI MD  different instructions run on different data
Parallel computation

Why parallel solutions?
- Increase speed
- Process huge amount of data
- Solve problems in real time
- Solve problems in due time
Parallel computation Efficiency

- Parameters to be evaluated (in sequential view)
  - Time
  - Memory
- Cases to be considered
  - Best
  - Worst
  - Average
- Optimal algorithms
  - $\Omega() = O()$
  - An algorithm is optimal if the running time in the worst case is of the same order of magnitude with function $\Omega$ of the problem to be solved
Parallel computation Efficiency

contd.

- **Order of magnitude:** a function which expresses the running time and $\Omega$ respectively
  - if polynomial function, the polynomial should have the same maximum degree, regardless the multiplicative constant
    - Ex: $\Omega() = c_1n^p$ and $O() = c_2n^p$ it is considered $\Omega = O$, written as $\Omega(n^2) = O(n^2)$
  - Algorithms have $O() \geq \Omega$ in the worst case
  - $O() \leq \Omega$ could be true for best case scenario, or even for the average case
    - Ex Sorting: $\Omega(n \log n)$ and $O(1)$ best case scenario for some algs.
Parallel computation Efficiency

contd.

• Do we really need parallel computers?
  • Any “computing greedy” algorithms better suited for parallel implementation?

• Consider 2 classes of algorithms:
  • Alg1: polynomial
  • Alg2: exponential

• Assume the design of a new system which increases the speed of execution \( V \) times

• ? How does the maximum dimension of the problem that can be solved in the new system increases? i.e. express \( n = f(V) \)
Parallel computation Efficiency

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Oper. Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1 (old)</td>
<td>$O(n^k)$</td>
<td>$T$</td>
</tr>
<tr>
<td>M2 (new)</td>
<td>$O(n^k)$</td>
<td>$T/V$</td>
</tr>
</tbody>
</table>

$Vn^k = (n_2)^k = Vn_1^k = (v^{1/k} n_1)^k$

$n_2 = v^{1/k} * n_1 \implies n_2 = c * n_1$
Parallel computation Efficiency

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Oper. Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1(old):</td>
<td>$2^n T$</td>
<td></td>
</tr>
<tr>
<td>M2(new):</td>
<td>$2^n T/V$</td>
<td></td>
</tr>
<tr>
<td>V2^n</td>
<td>$T$</td>
<td></td>
</tr>
</tbody>
</table>

$V = 2^n \Rightarrow 2^n = V$  
$2^{n_1} = 2^{\lg V + n_1}$  
$n_2 = n_1 + \lg V \Rightarrow n_2 = c + n_1$
Parallel computation Efficiency contd.

Speed of the new machine in terms of the speed of the old machine: \( V_2 = V \cdot V_1 \)

\[ \text{Alg1: } O(n^k): \quad n_2 = V^{1/k} \cdot n \]

\[ \text{Alg2: } O(2^n): \quad n_2 = n + \log V \]

Conclusion: BAD NEWS!!
Parallel computation Efficiency contd.

Are there algorithms that really need exponential running time?
- backtracking
- avoid it! “Never” use!

Are there problems that are inherently exponential?
- NP-complete, NP-hard problems
- “difficult” problems - we do not face them in real-world situations often?
- Hamiltonian cycle, sum problem, …
Parallel computation Efficiency contd.

- Parameters to be evaluated for parallel processing
  - Time
  - Memory
  - Number of processors involved
  - Cost = Time * Number of processors
    \( c = t \times n \)

- Efficiency
  - \( E = O() / \text{cost} \)
  - \( E \leq 1 \)
Parallel computation Efficiency contd.

- **Speedup**
  - $\Delta = \text{best sequential execution time/time of parallel execution}$
  - $\Delta \leq n$

- **Optimal algorithms**
  - $E=1, \Delta = n$
  - $\text{Cost} = O()$
  - A parallel algorithm is optimal if the cost is of the same order of magnitude with the best known algorithm to solve the problem in the worst case

- **Other**
Efficiency cont

- Parallel running time
  - Computation time
  - Communication time (data transfer, partial results transfer, information communication)

- Computation time
  - As the number of processors involved increases, the computation time decreases

- Communication time
  - Quite the opposite!!!
Efficiency cont

Diagram of running time

\[ T \sim f(N) \]

5 Oct. 10

Rodica Ptoolea - CS Dept., T.U. Cluj-Napoca, Romania
Outline

- Computation paradigms
- Constraints in parallel computation
- Dependence analysis
- Techniques for dependences removal
- Systolic architectures & simulations
- Increase the parallel potential
- Examples on GRID
Constraints in parallel computation

- Data/code to be distributed among the different processing units in the system
- Observe the precedence relations between statements/variables
- Avoid conflicts (data and control dependencies)
- Remove constraints (where possible)
Constraints in parallel computation contd.

- **Data parallel**
  - Input data split to be processed by different CPU
  - Characterizes the SIMD model of computation

- **Control parallel**
  - Different sequences processed by different CPU
  - Characterizes the MIMD and MISD models of computation
  - Conflicts should be avoided
    
    ```
    if constraint removed then parallel sequence of code
    else sequential sequence of code
    ```
Outline

• Computation paradigms
• Constraints in parallel computation
• Dependence analysis
• Systolic architectures & simulations
• Techniques for dependences removal
• Increase the parallel potential
• Examples on GRID
Dependence analysis

- Dependences represent the major limitation in parallelism
- **Data dependencies**
  - Refers to read/write conflicts
    - Data Flow dependence
    - Data Antidependence
    - Output dependence
- **Control dependencies**
  - Exist between statements
Dependence analysis

• Data Flow dependence
  • Indicates a write before read ordering of operations which must be satisfied
  • Can NOT be avoided therefore represents the limitation of the parallelism’s potential

Ex: (assume sequence in a loop on i)

S1: \( a(i) = x(i) - 3 \) //assign \( a(i) \), i.e. write in its memory location
S2: \( b(i) = a(i) / c(i) \) //use \( a(i) \), i.e. read from the cor. mem loc.

• \( a(i) \) should be first calculated in S1, and only then used in S2. Therefore, S1 and S2 cannot be processed in parallel!!!
Parallel Algorithms cont.

Dependencies analysis

• Data Antidependence
  • Indicates a read before write ordering of operations which must be satisfied
  • Can be avoided (techniques to remove this type of dependence)
    Ex: (assume sequence in a loop on i)

S1: \( a(i) = x(i) - 3 \) // write later
S2: \( b(i) = c(i) + a(i+1) \) // read first

• \( a(i+1) \) should be first used (with its former value) in S2, and only then computed in S1, at the next execution of the loop. Therefore, several iterations of the loop cannot be processed in parallel!
Parallel Algorithms cont.

Dependencies analysis

- **Output dependence**
  - Indicates a **write before write** ordering of operations which must be satisfied
  - Can be avoided (techniques to remove this type of dependence)
    
    Ex: (assume sequence in a loop on i)

    \[
    \begin{align*}
    S1: & \quad c(i+4) = b(i) + a(i+1) \quad // assign first here \\
    S2: & \quad c(i+1) = x(i) \quad // and here after 3 executions of the loop
    \end{align*}
    \]

- \(c(i+4)\) is assigned in S1; after 3 executions of the loop, the same element in the array is assigned in S2. Therefore, several iterations of the loop cannot be processed in parallel.
Parallel Algorithms cont.

Dependencies analysis

- Each dependence defines a dependence vector
  - They are defined by the distance between the loop where they interfere
    (data flow)
    
    S1: \( a(i) = x(i) - 3 \) \( d = 0 \) on var a
    
    S2: \( b(i) = a(i) / c(i) \)

    (data antidep.)

    S1: \( a(i) = x(i) - 3 \) \( d = 1 \) on var a
    
    S2: \( b(i) = c(i) + a(i+1) \)

    (output dep.)

    S1: \( c(i+4) = b(i) + a(i+1) \) \( d = 3 \) on var c
    
    S2: \( c(i+1) = x(i) \)

- All dependence vectors within a sequence define a dependence matrix
Parallel Algorithms cont.

Dependencies analysis

- Data dependencies
  - Data Flow dependence (wbr) cannot be removed
  - Data Antidependence (rbw) can be removed
  - Output dependence (wbw) can be removed
Outline

- Computation paradigms
- Constraints in parallel computation
- Dependence analysis
- **Techniques for dependences removal**
- Systolic architectures & simulations
- Increase the parallel potential
- Examples on GRID
Techniques for dependencies removal

- Many dependencies are introduced artificially by programmers.
- Output and antidependencies = **false** dependencies.
- Occur not because data is passed but because the same memory location is used in more than one place.
- Removal of (some of) those dependencies increase the parallelization potential.
Techniques for dependencies removal  contd.

• Renaming
  
  S1: \( a = b \times c \)
  
  S2: \( d = a + 1 \)
  
  S3: \( a = a \times d \)

• Dependence analysis:
  
  • S1->S2 (data flow on a)
  • S2->S3 (data flow on d)
  • S1->S3 (data flow on a)
  • S1->S3 (output dep. on a)
  • S2->S3 (antidependence on a)
Techniques for dependencies removal  contd.

- **Renaming**
  
  \[ S1: \ a = b \times c \]
  \[ S2: \ d = a + 1 \]
  \[ S'3: \ newa = a \times d \]

- **Dependence analysis:**
  
  - \( S1 \rightarrow S2 \) (data flow on \( a \)) \( \text{holds} \)
  - \( S2 \rightarrow S'3 \) (data flow on \( d \)) \( \text{holds} \)
  - \( S1 \rightarrow S'3 \) (data flow on \( a \)) \( \text{holds} \)
  - \( S1 \rightarrow S'3 \) (output dep. on \( a \))
  - \( S2 \rightarrow S'3 \) (antidependence on \( a \))
Techniques for dependencies removal contd.

• Expansion
  
  for i = 1 to n
  S1: \( x = b(i) - 2 \)
  S2: \( y = c(i) \times b(i) \)
  S3: \( a(i) = x + y \)

• Dependence analysis:

  • S1->S3 (data flow on \( x \) – same iteration)
  • S2->S3 (data flow on \( y \) – same iteration)
  • S1->S1 (output dep. on \( x \) – consecutive iterations)
  • S2->S2 (output dep. on \( y \) – consecutive iterations)
  • S3->S1 (antidep. on \( x \) – consecutive iterations)
  • S3->S2 (antidep. on \( y \) – consecutive iterations)
Techniques for dependencies removal contd.

• **Expansion** (loop iterations become independent)
  
  \[
  \text{for } i=1 \text{ to } n \\
  S'1: \ x(i) = b(i) - 2 \\
  S'2: \ y(i) = c(i) \times b(i) \\
  S'3: \ a(i) = x(i) + y(i)
  \]

• **Dependence analysis:**
  
  • S1->S3 (data flow on x) holds
  • S2->S3 (data flow on y) holds
  • S1->S1 (output dep. on x)
  • S2->S2 (output dep. on y)
  • S3->S1 (antidep. on x) replaced by the same data flow on x
  • S3->S2 (antidep. on y) replaced by the same data flow on y
Techniques for dependencies removal contd.

• Splitting

  for i=1 to n
  S1: a(i) = b(i) + c(i)
  S2: d(i) = a(i) + 2
  S3: f(i) = d(i) + a(i+1)

• Dependence analysis:

  • S1 -> S2 (data flow on a(i) – same iteration)
  • S2 -> S3 (data flow on d(i) – same iteration)
  • S3 -> S1 (antidep. on a(i+1) – consecutive iterations)

We have a cycle of dependencies: S1 -> S2 -> S3 -> S1
Techniques for dependencies removal contd.

• **Splitting** (cycle removed)

```
for i=1 to n
    S0: newa(i) = a(i+1) //copy the value obtained at the previous step
    S1: a(i) = b(i) + c(i)
    S2: d(i) = a(i) + 2
    S'3: f(i) = d(i) + newa(i)
```

• Dependence analysis:
  • S1→S2 (data flow on a(i) same iteration)
  • S2→S'3 (data flow on d(i) – same iteration)
  • S'3→S1 (antidep. on a(i+1) – consec. iterations)
  • S0→S'3 (data flow on a(i+1) – same iteration)
  • S0→S1 antidep. on a(i+1) – consecutive iterations)
Outline

- Computation paradigms
- Constraints in parallel computation
- Dependence analysis
- Techniques for dependences removal
- **Systolic architectures & simulations**
- Increase the parallel potential
- Examples on GRID
Systolic architectures & simulations

• Systolic architecture
  • Special purpose architecture
  • Pipe of processing units called cells

• Advantages
  • Faster
  • Scalable

• Limitations
  • Expensive
  • Highly specialized for particular applications
  • Difficult to build

=> simulations
Systolic architectures & simulations

- Goals for simulation:
  - Verification and validation
  - Tests and availability of solutions
  - Efficiency
  - Proof of concept
Systolic architectures & simulations

- Models of systolic architectures
  - Many problems in terms of systems of linear equations
  - Linear arrays (from special purpose architectures to multi-purpose architectures)
    - Computing values of a polynomial
    - Convolution product
    - Vector-matrix multiplication
- Mesh arrays
  - Matrix multiplication
  - Speech recognition (simplified mesh array)
Systolic architectures & simulations

- All but one simulated within a software model in PARLOG
  - Easy to implement
  - Ready for run
  - Specification language
- Speech recognition Pascal/C submesh array
  - Vocabulary of thousands of words
  - Real time recognition
Systolic architectures & simulations

• PARLOG example (convolution product)

cel([],[],_,_,[],[]).

cel([X|Tx],[Y|Ty],B,W,[Xo|Txo],[Yo|Tyo]) <-

Yo is Y+W*x,
Xo is B,

cel(Tx,Ty,X,W,Txo,Tyo).
Outline

- Computation paradigms
- Constraints in parallel computation
- Dependence analysis
- Techniques for dependences removal
- Systolic architectures & simulations
- **Increase the parallel potential**
- Examples on GRID
Program transformations

• Loop transformation
  • To allow execution of parallel loops
  • Techniques borrowed from compilers techniques
• **Cycle shrinking**

  - Applies when dependencies cycle of a loop involves dependencies between loops far apart (say distance \( d \))
  - The loop is executed on parallel on that distance \( d \)

  ```
  for i = 1 to n
    a(i+d) = b(i) - 1
    b(i+d) = a(i) + c(i)
  ```

  Transforms into

  ```
  parallel for i1 = 1 to n step d
    a(i+d) = b(i) - 1
    b(i+d) = a(i) + c(i)
  ```
Program transformations contd.

• Loop interchanging
  • Consist of interchanging of indexes
  • It requires the new dep vector to be positive

for j = 1 to n
  for i = 1 to n
    a(i, j) = a(i-1, j) + b(i-2, j)

Inner loop cannot be vectorised (executed in parallel)

for i = 1 to n
  parallel for j = 1 to n
    a(i, j) = a(i-1, j) + b(i-2, j)
Program transformations contd.

- Loop fusion
  - Transform 2 adjacent loops into a single one
  - Reduces the overhead of loops
  - It requires not to violate the ordering imposed by data dependencies

```plaintext
for i = 1 to n
    a(i) = b(i) + c(i+1)
for i = 1 to n
    c(i) = a(i-1) //write before read
for i = 1 to n
    a(i) = b(i) + c(i+1)
    c(i) = a(i-1)
```
Program transformations contd.

- Loop collapsing
  - Combines 2 nested loops
  - Increases the vector length

```plaintext
for i = 1 to n
  for j = 1 to m
    a(i,j) = a(i,j-1) + 1

for ij = 1 to n*m
  i = ij div m
  j = ij mod m
  a(i,j) = a(i,j-1) + 1
```
Program transformations contd.

• Loop partitioning
  • Partition into independent problems

  for \( i = 1 \) to \( n \)

  \[
  a(i) = a(i-g) + 1
  \]

  Loop can be split into \( g \) independent problems

  for \( i = 1 \) to \( (n \div g) \times g + k \) step \( g \)

  \[
  a(i) = a(i-g) + 1
  \]

  Where \( k=1,g \)

  Each loop could be executed in parallel
Outline

- Computation paradigms
- Constraints in parallel computation
- Dependence analysis
- Systolic architectures & simulations
- Techniques for dependences removal
- Increase the parallel potential
- **Examples on GRID**
GRID

- It provides access to
  - computational power
  - storage capacity
Examples on GRID

- Matrix multiplication
- Matrix power
- Gaussian elimination
- Sorting
- Graph algorithms
Examples on GRID
Matrix multiplication cont.

Analysis 1D (absolute analysis)

• Optimal sol: $p = f(n)$
Examples on GRID
Matrix multiplication cont.

Analysis 1D (relative analysis)

Computation time (for small dim) is small compared to communication time: hence parallel implementation with 1D efficient on very large matrix dim.
Examples on GRID
Matrix multiplication cont.

Analysis 2D (absolute analysis)

Matrix Multiplication 2D

- Time
- Processors

- 1000x1000
- 700x700
- 300x300
- 100x100
Examples on GRID

Matrix multiplication cont.

Analysis 2D (relative analysis)
Examples on GRID
Matrix multiplication cont.

Comparison 1D - 2D

100000
10000
1000
100
10
1

0 5 10 15 20 25 30 35 40

Time

0 5 10 15 20 25 30 35 40

Processors

Comparison 1D - 2D
Examples on GRID Graph algorithms

- Minimum Spanning Trees

Graph with 7 vertices: MST will have 6 edges with the sum of their weight minimum

Possible solutions: MST weight = 20
Examples on GRID Graph algorithms cont.

- MST Absolute Analysis

- MST Relative Analysis

- Observe the min value shifts to the right as graph’s dimension increases

- Observe the running time improvement with one order of magnitude (compared to sequential approach) for graphs with \( n = 5000 \)
Examples on GRID Graph algorithms cont.

- **Transitive Closure**: source-partitioned

- Each vertex \( v_i \) is assigned to a process, and each process computes shortest path from each vertex in its set to any other vertex in the graph, by using the sequential algorithm (hence, data partitioning)

- Requires no inter process communication
Examples on GRID Graph algorithms cont.

- **Transitive Closure**: source-parallel
- assigns each vertex to a set of processes and uses the parallel formulation of the single-source algorithm (similar to MST)
- Requires inter process communication

![Graph showing time vs. number of processors for transitive closure](image)
Examples on GRID Graph algorithms cont.

- **Graph Isomorphism**: exponential running time!
  - search if 2 graphs matches
  - trivial solution: generate all permutations of a graph \((n!)\) and match with the other graph

![Diagram](https://via.placeholder.com/150)

- Each process is responsible for computing \((n-1)!\) (data parallel approach)
Examples on GRID Graph algorithms cont.

Graph Isomorphism

Time - logarithmic scale

Processors

9 vertices
10 vertices
11 vertices
12 vertices
Conclusions

• The need for parallel processing (why)
  Existence of problems that require large amount of time to be solved (like NP-complete and NP hard problems)

• The constraints in parallel processing (how)
  Dependence analysis: define the boundaries of the problem

• The particularities of the solution give the amount of parallelism (when)
  The parabolic shape: more significant when large communication is involved
  Shift of optimal number of processors
  How parallelization is done matters
  Time up to 2 orders of magnitude decrease
Thank you for your attention!

Questions?