Introduction to Parallel Programming

Section 12.
Parallel Methods for Partial Differential Equations

Gergel V.P., Professor, D.Sc., Software Department
Contents…

- Problem Statement
- Methods for Solving the Partial Differential Equations
- Parallel Computations for Shared Memory Systems:
  - Problem of Blocking in Mutual Exclusion
  - Problem of Indeterminacy in Parallel Calculations
  - Race Condition of Threads
  - Deadlock Problem
  - Elimination of Calculation Indeterminacy
  - Parallel Wave Computation Scheme
  - Block-structured (Checkerboard) Decomposition
  - Load Balancing
Contents

- Parallel Computations for Distributed Memory Systems:
  - Data Decomposition Schemes
  - Striped Decomposition
  - Parallelization of Data Communications
  - Collective Communications
  - Block-structured (Checkerboard) Decomposition
  - Computational Pipelining (Multiple Wave Computation Scheme)
  - Overview of Data Communications in Solving Partial Differential Equations

- Summary
Introduction

- *Partial Differential Equations (PDE)* are widely used for developing models in various scientific and technical fields
- Analysis of mathematical models based on differential equations is provided by the numerical methods
- The performed computation is greatly time-consuming

*Numerical solving of partial differential equations is a subject of intensive research*
Problem Statement

Let's consider the *numerical solving of the Dirichlet problem for the Poisson equation* as the case study for PDE Calculations. This problem can be formulated as follows:

\[
\begin{align*}
\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} &= f(x, y), \quad (x, y) \in D, \\
\quad u(x, y) &= g(x, y), \quad (x, y) \in D^0, \\
D &= \{(x, y) \in D : 0 \leq x, y \leq 1\}
\end{align*}
\]
Methods for Solving the Partial Differential Equations…

- **Method of Finite Differences:**
  - The solution domain is represented as a discrete set *(grid)* of points *(nodes)*,
  - The solution sequence uniformly converges to the Dirichlet problem solution, while the solution error is of $h^2$ order

\[
\begin{align*}
D_h = \{(x_i, y_j) : x_i &= ih, y_j = jh, \ 0 \leq i, j \leq N + 1, \\
h &= 1/(N + 1), \\
\frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}}{h^2} &= f_{ij} \\
u_{ij} &= 0.25 (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{ij})
\end{align*}
\]
Methods for Solving the Partial Differential Equations...

The Gauss-Seidel method...

\[ u_{ij}^k = 0.25(u_{i-1,j}^k + u_{i+1,j}^{k-1} + u_{i,j-1}^k + u_{i,j+1}^{k-1} - h^2 f_{ij}) \]

Calculation complexity

\[ T = k m N^2 \]

where
- \( N \) - number of nodes for each dimension,
- \( m \) - number of operations for one node,
- \( k \) - number of iterations
Algorithm 1: The Sequential Gauss-Seidel Algorithm

```c
// Algorithm 12.1
do {
    dmax = 0; // maximum deviation of values u
    for (i=1; i<N+1; i++)
        for (j=1; j<N+1; j++) {
            temp = u[i][j];
            u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
            dm = fabs(temp-u[i][j]);
            if (dmax < dm) dmax = dm;
        }
} while (dmax > eps);
```

Code
A Computational Example

\[
\begin{align*}
  f(x, y) &= 0, \quad (x, y) \in D, \\
  100 - 200x, &\quad y = 0, \\
  100 - 200y, &\quad x = 0, \\
  -100 + 200x, &\quad y = 1, \\
  -100 + 200y, &\quad x = 1,
\end{align*}
\]

\[
\begin{align*}
  N &= 100 \\
  \epsilon &= 0.1 \\
  k &= 210
\end{align*}
\]
Parallel Computations for Shared Memory Systems...

- The possible way for obtaining software for parallel computations – rewriting the existing sequential programs
- Rewriting can be implemented either automatically by a compiler or directly by a programmer
- The second approach prevails as the possibilities of automatic program analysis for generating parallel versions of programs are rather restricted
- The application of new algorithmic languages oriented at parallel programming leads to the necessity for a considerable reprogramming of the existing software
The possible problem solution is the application of some means "outside of programming language". For instance, they may be directives or comments which are processed by a special preprocessor before the program is compiled.

Directives can be used to point out different ways to parallelize a program, while the original program text remains the same.

The preprocessor replaces the parallelism directives by some additional program code (as a rule in the form of addressing the procedures of a parallel library).

If there’s no preprocessor, the compiler would ignore directives and construct the original sequential program code.
Parallel Computations for Shared Memory Systems

The unity of the program code for sequential and parallel calculations reduces the difficulties in parallel programs’ development and maintenance.

Conversion of sequential programs to parallel ones by means of directives’ application allows to implement the stage-by-stage technology of parallel software development that is greatly valued in programming.
OpenMP Technology

- To specify program fragments that can be executed in parallel, the programmer adds directives (C/C++) or comments (Fortran) into the program.
- These directives (or comments) allow to determine the parallel regions of the program.

As a result of this approach the program can be represented as a sequence of interleaved serial (one-thread) and parallel (multi-thread) parts of the code.

- Such type of computing is usually referred as the fork-join (or pulsatile) parallelism.
Algorithm 1.2: The first variant of the Gauss-Seidel parallel algorithm

```c
// Algorithm 1.2
omp_lock_t dmax_lock;
omp_init_lock (dmax_lock);
do {
    dmax = 0; // maximum deviation of values u
#pragma omp parallel for shared(u,n,dmax) private(i,temp,d)
    for ( i=1; i<N+1; i++ ) {
#pragma omp parallel for shared(u,n,dmax) private(j,temp,d)
        for ( j=1; j<N+1; j++ ) {
            temp = u[i][j];
            u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+ u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
            d = fabs(temp-u[i][j])
            omp_set_lock(dmax_lock);
            if ( dmax < d ) dmax = d;
            omp_unset_lock(dmax_lock);
        } // the end of inner parallel region
    } // the end of outer parallel region
} while ( dmax > eps );
```
The Results of Computational Experiments

<table>
<thead>
<tr>
<th>Grid size</th>
<th>Gauss-Seidel sequential method (algorithm 12.1)</th>
<th>Parallel algorithm 12.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$k$</td>
<td>$T$</td>
</tr>
<tr>
<td>100</td>
<td>210</td>
<td>0,06</td>
</tr>
<tr>
<td>200</td>
<td>273</td>
<td>0,34</td>
</tr>
<tr>
<td>300</td>
<td>305</td>
<td>0,88</td>
</tr>
<tr>
<td>400</td>
<td>318</td>
<td>3,78</td>
</tr>
<tr>
<td>500</td>
<td>343</td>
<td>6,00</td>
</tr>
<tr>
<td>600</td>
<td>336</td>
<td>8,81</td>
</tr>
<tr>
<td>700</td>
<td>344</td>
<td>12,11</td>
</tr>
<tr>
<td>800</td>
<td>343</td>
<td>16,41</td>
</tr>
<tr>
<td>900</td>
<td>358</td>
<td>20,61</td>
</tr>
<tr>
<td>1000</td>
<td>351</td>
<td>25,59</td>
</tr>
<tr>
<td>2000</td>
<td>367</td>
<td>106,75</td>
</tr>
<tr>
<td>3000</td>
<td>370</td>
<td>243,00</td>
</tr>
</tbody>
</table>

$k$ – the number of iterations,
$T$ – the execution time,
$S$ – the speedup
Estimation of the Approach

- The developed parallel algorithm provides the solution to the given problem.
- It can be used up to $N^2$ processors for program execution.
- There are the excessively high synchronization of the parallel regions of the program.
- Low level of processors’ load.

Low efficiency
Problem: *Blocking in Mutual Exclusion*…

- Each parallel thread after processing values must check (and probably change) the value $d_{max}$.
- The permission for using the variable has to be obtained by one thread only. The other threads must be blocked. After the shared variable is released the next thread may get control, etc.
Problem: *Blocking in Mutual Exclusion*

As a result a multithread parallel program turns practically into a sequentially executable code
Algorithm 1.3: The Second Variant of the Gauss-Seidel Parallel Algorithm

```c
// Algorithm 12.3
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
    dmax = 0; // maximum deviation of values u
#pragma omp parallel for
shared(u,n,dmax)private(i,temp,d,dm)
    for ( i=1; i<N+1; i++ ) {
        dm = 0;
        for ( j=1; j<N+1; j++ ) {
            temp = u[i][j];
            u[i][j] = 0.25*(u[i-1][j]+u[i+1][j] + 
                          u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
            d = fabs(temp-u[i][j]);
            if ( dm < d ) dm = d;
        }
        omp_set_lock(dmax_lock);
        if ( dmax < dm ) dmax = dm;
        omp_unset_lock(dmax_lock);
    }
} // the end of parallel region
} while ( dmax > eps );
```

Code
The Results of Computational Experiments

<table>
<thead>
<tr>
<th>Grid size</th>
<th>Gauss-Seidel sequential method (algorithm 12.1)</th>
<th>Parallel algorithm 12.2</th>
<th>Parallel algorithm 12.3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$k$</td>
<td>$T$</td>
<td>$k$</td>
</tr>
<tr>
<td>100</td>
<td>210</td>
<td>0,06</td>
<td>210</td>
</tr>
<tr>
<td>200</td>
<td>273</td>
<td>0,34</td>
<td>273</td>
</tr>
<tr>
<td>300</td>
<td>305</td>
<td>0,88</td>
<td>305</td>
</tr>
<tr>
<td>400</td>
<td>318</td>
<td>3,78</td>
<td>318</td>
</tr>
<tr>
<td>500</td>
<td>343</td>
<td>6,00</td>
<td>343</td>
</tr>
<tr>
<td>600</td>
<td>336</td>
<td>8,81</td>
<td>336</td>
</tr>
<tr>
<td>700</td>
<td>344</td>
<td>12,11</td>
<td>344</td>
</tr>
<tr>
<td>800</td>
<td>343</td>
<td>16,41</td>
<td>343</td>
</tr>
<tr>
<td>900</td>
<td>358</td>
<td>20,61</td>
<td>358</td>
</tr>
<tr>
<td>1000</td>
<td>351</td>
<td>25,59</td>
<td>351</td>
</tr>
<tr>
<td>2000</td>
<td>367</td>
<td>106,75</td>
<td>367</td>
</tr>
<tr>
<td>3000</td>
<td>370</td>
<td>243,00</td>
<td>370</td>
</tr>
</tbody>
</table>
Estimation of the Approach

- Considerable decrease in the number of shared variable access
- The maximum possible parallelism decreases to the level of \( N \)
- As a result – a considerable decrease in costs of thread synchronization and a decrease of computation serialization effect

\[ \textgreater \text{The best speedup parameters} \]
Problem: *Indeterminacy in Parallel Calculations*

- The generated sequence of data processing may vary at several program executions with the same initial data.
- The location of threads in the problem domain $D$ may be different - some threads may pass ahead the others and vice versa.
- This thread location structure can vary from execution to execution. The reason of such behavior is a race condition of threads.

*The time dynamics of parallel thread execution should not have an influence on calculations carried out by parallel algorithms and programs.*
Race Condition of Threads

A possible solution: *capture and blocking of the used rows*
Problem: Deadlocks

- For the mutual exclusion of access to the grid nodes a set of semaphores `row_lock[N]` may be introduced. It will allow the threads to block the access to their grid rows.

```
// the thread is processing the row i
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i-1]);
// processing the grid row i
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i-1]);
```

The threads block first rows 1 and 2 and only then pass over to blocking the rest of the rows – deadlock
Deadlock Avoidance

**Approach**: the appropriate order in rows’ blocking

```c
// the thread is processing the row i
omp_set_lock(row_lock[i+1]);
omp_set_lock(row_lock[i]);
omp_set_lock(row_lock[i-1]);
// < processing the grid row i >
omp_unset_lock(row_lock[i+1]);
omp_unset_lock(row_lock[i]);
omp_unset_lock(row_lock[i-1]);
```

*Indeterminacy of calculations is not provided yet*
Elimination of Calculation Indeterminacy

- To eliminate calculation indeterminacy the Gauss-Jacobi method can be used, which use separate places to store the results of the previous and the current iterations
Algorithm 1.4: The Parallel Gauss-Jacobi method...

```
// Algorithm 12.4
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
    dmax = 0; // maximum deviation of values u
#pragma omp parallel for shared(u,n,dmax)
    private(i,temp,d,dm)
    for ( i=1; i<N+1; i++ ) {
        dm = 0;
        for ( j=1; j<N+1; j++ ) {
            temp = u[i][j];
            un[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
            d = fabs(temp-un[i][j]);
            if ( dm < d ) dm = d;
        }
    }
} to be continued
```
**Algorithm 1.4: The Parallel Gauss-Jacobi method**

```c
omp_set_lock(dmax_lock);
    if ( dmax < dm ) dmax = dm;
omp_unset_lock(dmax_lock);
}
} // the end of parallel region
for ( i=1; i<N+1; i++ ) // data update
    for ( j=1; j<N+1; j++ )
        u[i][j] = un[i][j];
} while ( dmax > eps );
```

**Code**
The results of Computational Experiments

<table>
<thead>
<tr>
<th>Grid size</th>
<th>Sequential Gauss-Jacobi method (algorithm 12.4)</th>
<th>Parallel Gauss-Jacobi method developed on the analogy of the algorithm 12.3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$k$</td>
<td>$T$</td>
</tr>
<tr>
<td>100</td>
<td>5257</td>
<td>1,39</td>
</tr>
<tr>
<td>200</td>
<td>23067</td>
<td>23,84</td>
</tr>
<tr>
<td>300</td>
<td>26961</td>
<td>226,23</td>
</tr>
<tr>
<td>400</td>
<td>34377</td>
<td>562,94</td>
</tr>
<tr>
<td>500</td>
<td>56941</td>
<td>1330,39</td>
</tr>
<tr>
<td>600</td>
<td>114342</td>
<td>3815,36</td>
</tr>
<tr>
<td>700</td>
<td>64433</td>
<td>2927,88</td>
</tr>
<tr>
<td>800</td>
<td>87099</td>
<td>5467,64</td>
</tr>
<tr>
<td>900</td>
<td>286188</td>
<td>22759,36</td>
</tr>
<tr>
<td>1000</td>
<td>152657</td>
<td>14258,38</td>
</tr>
<tr>
<td>2000</td>
<td>337809</td>
<td>134140,64</td>
</tr>
<tr>
<td>3000</td>
<td>655210</td>
<td>247726,69</td>
</tr>
</tbody>
</table>
Estimation of the Approach

- Uniqueness of the calculations
- Use of the additional memory
- Smaller convergence rate

Another possible approach to eliminate the mutual dependences of parallel threads is to apply the red/black row alteration scheme. In this scheme the execution of each iteration is subdivided into two sequential stages:

- At the first stage only the rows with even numbers are processed,
- At the second stage - the rows with odd numbers are used
Red/Black Row Alteration Scheme

Stage 1

Stage 2

- border values
- previous iteration values
- values after stage 1 of the current iteration
- values after stage 2 of the current iteration
Estimation of the Approach…

- No additional memory is required
- The algorithm guarantees uniqueness of calculations, which do not coincide with the results obtained by means of sequential algorithm
- Smaller convergence rate

 Potentiality for the increase in the efficiency of calculations
Estimation of the Approach

The Gauss-Jacobi method

- Use of the additional memory

Red/black row alteration scheme

- Additional memory is not required

- The algorithm guarantees uniqueness of calculations, though the obtained results may not coincide with the results of the sequential calculations

- Calculation schemes demonstrate the convergence rate, which is worse than the original convergence rate of the Gauss-Seidel method
Parallel Wave Computation Scheme…

- Let us now consider the parallel algorithms with the following characteristics - the performed calculations and the obtained results have to be completely identical to the ones of the original sequential method.

- Among such techniques - *the wavefront* or *hyperplane method*.

- The wavefront method can be explained as follows – it is evident that to provide calculations identical as at the original sequential method the following should be taken into account:
  - At the first step the node $u_{11}$ may be processed only,
  - Then – at the second step - the node $u_{21}$ and $u_{12}$ may be recalculated, etc.

As a result at each step the nodes that may be processed form a *bottom-up grid diagonal* with the numbers determined by the step number.
Parallel Wave Computation Scheme…

Growing wave
- border values
- values of the current iteration

Wave peak
- values of the previous iteration nodes, at which values can be recalculated

Decaying wave
Algorithm 1.5: Parallel Algorithm Based on Wave Calculation Scheme...

```c
// Algorithm 12.5
omp_lock_t dmax_lock;
omp_init_lock(dmax_lock);
do {
    dmax = 0; // maximum variation of values u
    // growing wave (nx - wave size)
    for ( nx=1; nx<N+1; nx++ ) {
        dm[nx] = 0;
        #pragma omp parallel for shared(u,nx,dm) private(i,j,temp,d)
        for ( i=1; i<nx+1; i++ ) {
            j = nx + 1 - i;
            temp = u[i][j];
            u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]*h*f[i][j]);
            d = fabs(temp-u[i][j])
            if ( dm[i] < d ) dm[i] = d;
        } // the end of parallel region
    }
}
```
Algorithm 1.5: Parallel Algorithm Based on Wave Calculation Scheme

```c
// decaying wave
for ( nx=N-1; nx>0; nx-- ) {
#pragma omp parallel for shared(u,nx,dm) private(i,j,temp,d)
  for ( i=N-nx+1; i<N+1; i++ ) {
    j = 2*N - nx - I + 1;
    temp = u[i][j];
    u[i][j] = 0.25*(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1]-h*h*f[i][j]);
    d = fabs(temp-u[i][j])
    if ( dm[i] < d ) dm[i] = d;
  } // the end of parallel region
}
#pragma omp parallel for shared(n,dm,dmax) private(i)
  for ( i=1; i<nx+1; i++ ) {
    omp_set_lock(dmax_lock);
    if ( dmax < dm[i] ) dmax = dm[i];
    omp_unset_lock(dmax_lock);
  } // the end of parallel region
while ( dmax > eps );
```

Code
Parallel Wave Computation Scheme

- The final part of calculations for computing the maximum deviation of values $u$ is the least efficient due to high additional synchronization cost.
- **Chucking (fragmentation)** – the technique of increasing sequential computation blocks to reduce the synchronization cost.
- The possible variant to implement this approach may be the following:

```c
chunk = 200; // sequential part size
#pragma omp parallel for shared(n, dm, dmax)private(i, d)
   for ( i=1; i<nx+1; i+=chunk ) {
      d = 0;
      for ( j=i; j<i+chunk; j++ )
         if ( d < dm[j] ) d = dm[j];
      omp_set_lock(dmax_lock);
      if ( dmax < d ) dmax = d;
      omp_unset_lock(dmax_lock);
   }
```

} the end of parallel region
## The Results of Computational Experiments

<table>
<thead>
<tr>
<th>Grid size</th>
<th>Sequential Gauss-Seidel method (algorithm 12.1)</th>
<th>Parallel algorithm 12.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$k$</td>
<td>$t$</td>
</tr>
<tr>
<td>100</td>
<td>210</td>
<td>0,06</td>
</tr>
<tr>
<td>200</td>
<td>273</td>
<td>0,34</td>
</tr>
<tr>
<td>300</td>
<td>305</td>
<td>0,88</td>
</tr>
<tr>
<td>400</td>
<td>318</td>
<td>3,78</td>
</tr>
<tr>
<td>500</td>
<td>343</td>
<td>6,00</td>
</tr>
<tr>
<td>600</td>
<td>336</td>
<td>8,81</td>
</tr>
<tr>
<td>700</td>
<td>344</td>
<td>12,11</td>
</tr>
<tr>
<td>800</td>
<td>343</td>
<td>16,41</td>
</tr>
<tr>
<td>900</td>
<td>358</td>
<td>20,61</td>
</tr>
<tr>
<td>1000</td>
<td>351</td>
<td>25,59</td>
</tr>
<tr>
<td>2000</td>
<td>367</td>
<td>106,75</td>
</tr>
<tr>
<td>3000</td>
<td>370</td>
<td>243,00</td>
</tr>
</tbody>
</table>
Estimation of the Approach

- Low efficiency of cache use
- In order to increase the computation performance by efficient cache utilization the following conditions need to be provided:
  - The performed calculations use the same data repeatedly (data processing locality),
  - The performed calculations provide access to memory elements with sequentially increasing addresses (sequential access)
- To meet such requirements the procedure of processing some rectangular blocks of the grid should be considered
Block-structured (Checkerboard) Decomposition

- Border values
- Previous iteration values
- Current iteration values
- Nodes which values can be recalculated
- Values, which must be transmitted among the block borders
Algorithm 1.6: Wavefront Method Base on Checkerboard Data Decomposition

//Algorithm 12.6
do {
    // growing wave (wave size is nx+1)
    for ( nx=0; nx<NB; nx++ ) { // NB block number
        #pragma omp parallel for shared(nx) private(i,j)
        for ( i=0; i<nx+1; i++ ) {
            j = nx - i;
            // <processing a block with coordinates (i,j)>
        } // the end of parallel region
    }
    // decaying wave
    for ( nx=NB-2; nx>-1; nx-- ) {
        #pragma omp parallel for shared(nx) private(i,j)
        for ( i=0; i<nx+1; i++ ) {
            j = 2*(NB-1) - nx - i;
            // <processing a block with coordinates (i,j)>
        } // the end of parallel region
    }
    // <calculation of error estimation >
} while ( dmax > eps );

Code
## The Results of Calculation Experiments

<table>
<thead>
<tr>
<th>Grid size</th>
<th>Sequential Gauss-Seidel method (algorithm 12.1)</th>
<th>Parallel algorithm 12.5</th>
<th>Parallel algorithm 12.6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$K$</td>
<td>$T$</td>
<td>$k$</td>
</tr>
<tr>
<td>100</td>
<td>210</td>
<td>0,06</td>
<td>210</td>
</tr>
<tr>
<td>200</td>
<td>273</td>
<td>0,34</td>
<td>273</td>
</tr>
<tr>
<td>300</td>
<td>305</td>
<td>0,88</td>
<td>305</td>
</tr>
<tr>
<td>400</td>
<td>318</td>
<td>3,78</td>
<td>318</td>
</tr>
<tr>
<td>500</td>
<td>343</td>
<td>6,00</td>
<td>343</td>
</tr>
<tr>
<td>600</td>
<td>336</td>
<td>8,81</td>
<td>336</td>
</tr>
<tr>
<td>700</td>
<td>344</td>
<td>12,11</td>
<td>344</td>
</tr>
<tr>
<td>800</td>
<td>343</td>
<td>16,41</td>
<td>343</td>
</tr>
<tr>
<td>900</td>
<td>358</td>
<td>20,61</td>
<td>358</td>
</tr>
<tr>
<td>1000</td>
<td>351</td>
<td>25,59</td>
<td>351</td>
</tr>
<tr>
<td>2000</td>
<td>367</td>
<td>106,75</td>
<td>367</td>
</tr>
<tr>
<td>3000</td>
<td>370</td>
<td>243,00</td>
<td>370</td>
</tr>
</tbody>
</table>
Estimation of the Approach

- Block processing is performed on different processors and the blocks are mutually disjoint - as a result, there are no additional costs to support cache coherency of different processors.
- The situations when processors stay idle are possible.

*It is possible to increase the efficiency of calculations.*
Processor Load Balancing

- The block size determines the granularity of parallel computations.
- Choosing the level of granularity it is possible to provide the required efficiency of parallel methods.
- To provide the uniform processor loads (load balancing) all the computational works can be arranged as a job queue.
- In the course of computations the processor, which is already unloaded, may ask for a job from the queue.

_nearby_quote_ A job queue is the general management scheme of load balancing for a shared memory system._
Algorithm 1.7: Load Balancing Based on Job Queue Management Scheme

// <data initialization>
// <loading the initial block pointer into the job queue>
// pick up the block from the job queue (if the job queue is not empty)
while ( (pBlock=GetBlock()) != NULL ) {
    // <block processing>
    // marking the neighboring block readiness for processing
    omp_set_lock(pBlock->pNext.Lock); // right-hand neighbor
    pBlock->pNext.Count++;
    if ( pBlock->pNext.Count == 2 ) PutBlock(pBlock->pNext);
    omp_unset_lock(pBlock->pNext.Lock);
    omp_set_lock(pBlock->pDown.Lock); // lower neighbor
    pBlock->pDown.Count++;
    if ( pBlock->pDown.Count == 2 ) PutBlock(pBlock->pDown);
    omp_unset_lock(pBlock->pDown.Lock);
} // the end of computations, as the queue is empty
Parallel Computations for Distributed Memory Systems

- Many parallel computation problems such as the race condition, deadlocks, serialization are common for the systems with shared and distributed memory.
- The communication of parallel program parts on different processors can only be provided through message passing.
Data Decomposition Schemes

- In the considered the Dirichlet problem there are two different data decomposition schemes:
  - The one-dimensional or striped decomposition of the domain grid,
  - The two-dimensional or block-structured (checkerboard) decomposition of the domain grid
- In case of striped decomposition the domain grid is divided into horizontal or vertical strips
- The number of strips is defined by the number of processors. The size of strips is usually equal
- The strips are distributed among the processors for processing
Striped Decomposition

**Remarks:**

- The border rows of the previous and the next strips should be copied on the processor, which performs processing a strip,
- Border row copying should be performed prior to the beginning of the execution of each method iteration.
Algorithm 1.8: *The Gauss-Seidel Method, the Striped Data Decomposition*

```c
// Algorithm 12.8
// The Gauss-Seidel method, the striped decomposition
// operations performed on each processor
do {
    // <border row exchange between the neighbors>
    // <strip processing>
    // <calculating the computational error dmax>
    while ( dmax > eps ); // eps - the required accuracy
```

Code
Data Distribution between Processors…

- At the first stage each processor transmits its lowest border row to the following processor and receives the analogous row from the previous processor.
- At the second stage processors transmit their upper border rows to the previous neighbors and receive the analogous rows from the following neighbor.
Data Distribution between Processors…

- Carrying out such data transmission operations may be implemented as follows:

```c
// transmission of the lowest border row to the following processor and receiving the transmitted border row from the previous processor
if ( ProcNum != NP-1 ) Send(u[M][*], N+2, NextProc);
if ( ProcNum != 0 ) Receive(u[0][*], N+2, PrevProc);
```

- Such implemented scheme produces the *strictly sequential execution* of data transmission operations.

- Applying *nonblocking* communications may not provide an efficient parallel scheme of processor interactions.
Parallelization of Data Communications

- At the first step all odd processors transmit data, and the even processors receive the data.
- At the second step the processors change their roles: the even processors perform the operation \textit{Send}, the odd processors perform the operation \textit{Receive}.

```c
// transmission of the lowest border row to the following processor
// and receiving the transmitted row from the previous processor
if ( ProcNum % 2 == 1 ) { // odd processor
    if ( ProcNum != NP-1 ) Send(u[M][*],N+2,NextProc);
    if ( ProcNum != 0 ) Receive(u[0][*],N+2,PrevProc);
}
else { // even processor
    if ( ProcNum != 0 ) Receive(u[0][*],N+2,PrevProc);
    if ( ProcNum != NP-1 ) Send(u[M][*],N+2,NextProc);
}
```
Collective Communications

- Operation of accumulating and broadcasting the data may be implemented by the use of the cascade scheme.

- Obtaining of the maximum value of local errors calculated by the processors may be provided by means of the following technique:
  - At the first step finding of the maximum values for pairwise grouped processor - such calculations may be performed at different processor pairs in parallel,
  - At the second step analogous pairwise calculations may be applied for finding the maximum values among the obtained results, etc.

- According with the cascade scheme it is necessary to perform $\log_2 p$ of parallel iterations to calculate the total maximum value ($p$ is the number of processors).
Algorithm 1.8: The Gauss-Seidel Method, Implementation with Collective Communication Operations

// Algorithm 12.8 - Implementation with Collective Operations
// The Gauss-Seidel method, the striped decomposition
// operations performed on each processor

do {
    // border strip row exchange with the neighbors
    Sendrecv(u[M][*],N+2,NextProc,u[0][*],N+2,PrevProc);
    Sendrecv(u[1][*],N+2,PrevProc,u[M+1][*],N+2,NextProc);
    // <strip processing with the error estimation dm >
    // <calculating the computational error dmax>
    Reduce(dm,dmax,MAX,0);
    Broadcast(dmax,0);
} while ( dmax > eps ); // eps - the required accuracy
## The Results of Calculations Experiments

<table>
<thead>
<tr>
<th>Grid size</th>
<th>Gauss-Seidel sequential method</th>
<th>Parallel algorithm 1.8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$k$</td>
<td>$T$</td>
</tr>
<tr>
<td>100</td>
<td>210</td>
<td>0,06</td>
</tr>
<tr>
<td>200</td>
<td>273</td>
<td>0,35</td>
</tr>
<tr>
<td>300</td>
<td>305</td>
<td>0,92</td>
</tr>
<tr>
<td>400</td>
<td>318</td>
<td>1,69</td>
</tr>
<tr>
<td>500</td>
<td>343</td>
<td>2,88</td>
</tr>
<tr>
<td>600</td>
<td>336</td>
<td>4,04</td>
</tr>
<tr>
<td>700</td>
<td>344</td>
<td>5,68</td>
</tr>
<tr>
<td>800</td>
<td>343</td>
<td>7,37</td>
</tr>
<tr>
<td>900</td>
<td>358</td>
<td>9,94</td>
</tr>
<tr>
<td>1000</td>
<td>351</td>
<td>11,87</td>
</tr>
<tr>
<td>2000</td>
<td>367</td>
<td>50,19</td>
</tr>
<tr>
<td>3000</td>
<td>364</td>
<td>113,17</td>
</tr>
</tbody>
</table>
Striped Wavefront Computations

- To form a wavefront calculations each strip can be represented logically as a set of blocks.
- As a result of such logical structure the wavefront computation scheme may be executed. At the first step the block marked by the number 1 may be processed. Then – at the second step – the blocks marked by the number 2 may be recalculated, etc.
Block-structured (Checkerboard) Decomposition…

- In case of the block-structured (checkerboard) data decomposition the number of the border rows on each processor is increased, which leads correspondingly to a greater number of data communications in the border row transmission (but the number of transmitted elements is reduced).

- The use of the checkerboard scheme of data decomposition is appropriate if the number of grid nodes is essentially large.
Block-structured (Checkerboard) Decomposition...

// Algorithm 12.9
// The Gauss-Seidel method, the striped decomposition
// operations executed on each processor
do {
    // obtaining border nodes
    if ( ProcNum / NB != 0 ) { // nonzero row of processors
        // obtaining data from upper processor
        Receive(u[0][*], M+2, TopProc); // upper row
        Receive(dmax, 1, TopProc);    // computational error
    }
    if ( ProcNum % NB != 0 ) { // nonzero column of processors
        // obtaining data from left processor
        Receive(u[*][0], M+2, LeftProc); // left column
        Receive(dm, 1, LeftProc);       // computational error
        If ( dm > dmax ) dmax = dm;
    }
}
Block-structured (Checkerboard) Decomposition

```c
// <processing a block with computational error dmax>
// transmission of border nodes
if ( ProcNum / NB != NB-1 ) { // processor row is not last
    // data transmission to the lower processor
    Send(u[M+1][*],M+2,DownProc); // bottom row
    Send(dmax,1,DownProc); // computational error
}
if ( ProcNum % NB != NB-1 ) { // processor column is not last
    // data transmission to the right processor
    Send(u[*][M+1],M+2,RightProc); // right column
    Send(dmax,1,RightProc); // computational error
}
// synchronization and distribution of the value dmax
Barrier();
Broadcast(dmax,NP-1);
} while ( dmax > eps ); // eps – the required accuracy
```
Computational Pipelining (Multiple Wavefront Computation Scheme)…

- The wavefront computation efficiency decreases considerably because the processors perform calculations only at the moment when their blocks belong to the wave computation front.

- To improve the load balancing among the processors a multiple wavefront computation scheme can be applied.

- The multiple wavefront method can be explained as follows: the processors may start processing the blocks of the following wave after executing the current calculation iteration.
### Computational Pipelining (Multiple Wavefront Computation Scheme)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

**Growing wave**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

**Wave peak**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

**Decaying wave**

- Gray squares: blocks of the current iteration
- White squares: blocks of the previous iteration
- Dark gray squares: blocks which nodes can be recalculated
Overview of Data Communications in Solving Partial Differential Equations

<table>
<thead>
<tr>
<th>Communication operation</th>
<th>Processors</th>
<th>Corresponding MPI function</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-to-all distribution of the grid node number</td>
<td>0 1 2</td>
<td>MPI_Bcast</td>
</tr>
<tr>
<td>Scattering strips (blocks) of the grid nodes</td>
<td></td>
<td>MPI_Scatter</td>
</tr>
<tr>
<td>Exchanging borders of neighbor strips or blocks</td>
<td></td>
<td>MPI_Sendrecv</td>
</tr>
<tr>
<td>All-to-all calculating and distributing the computational error</td>
<td></td>
<td>MPI_Allreduce</td>
</tr>
<tr>
<td>Gathering strips (blocks) of the grid nodes</td>
<td></td>
<td>MPI_Gather</td>
</tr>
</tbody>
</table>
Summary

- The ways of parallel algorithm development for the systems with shared and distributed memory are discussed on the example of solving the partial differential equations.
- In case of parallel computations on the systems with shared memory the main attention is given to the OpenMP technology; various aspects concerning with parallel programming are considered.
- In case of parallel computations on the systems with distributed memory the problems of the data decomposition and the information communications between the processors are discussed; striped and checkerboard decomposition schemes are presented.
Discussions

- What are the ways to increase the efficiency of wavefront methods?
- How can the job queue balance the computational load?
- What problems have to be solved in the process of parallel computation on distributed memory systems?
- What basic operations of data communications are used in the parallel methods of the Dirichlet problem?
Exercises

- Develop the parallel algorithm implementation of the wavefront computation scheme including the block-structured data decomposition scheme
- Develop theoretical estimation of the algorithm execution time
- Carry out the computational experiments. Compare the results of computational experiments and the obtained theoretical estimations
References

Next Section

- Parallel Methods for Global Optimization
Author’s Team

Gergel V.P., Professor, Doctor of Science in Engineering, Course Author

Grishagin V.A., Associate Professor, Candidate of Science in Mathematics

Abrosimova O.N., Assistant Professor (chapter 10)
Kurylev A.L., Assistant Professor (learning labs 4,5)
Labutin D.Y., Assistant Professor (ParaLab system)

Sysoev A.V., Assistant Professor (chapter 1)
Gergel A.V., Post-Graduate Student (chapter 12, learning lab 6)
Labutina A.A., Post-Graduate Student (chapters 7,8,9, learning labs 1,2,3, ParaLab system)
Senin A.V., Post-Graduate Student (chapter 11, learning labs on Microsoft Compute Cluster)
Liverko S.V., Student (ParaLab system)
About the project

The purpose of the project is to develop the set of educational materials for the teaching course “Multiprocessor computational systems and parallel programming”. This course is designed for the consideration of the parallel computation problems, which are stipulated in the recommendations of IEEE-CS and ACM Computing Curricula 2001. The educational materials can be used for teaching/training specialists in the fields of informatics, computer engineering and information technologies. The curriculum consists of the training course “Introduction to the methods of parallel programming” and the computer laboratory training “The methods and technologies of parallel program development”. Such educational materials makes possible to seamlessly combine both the fundamental education in computer science and the practical training in the methods of developing the software for solving complicated time-consuming computational problems using the high performance computational systems.

The project was carried out in Nizhny Novgorod State University, the Software Department of the Computing Mathematics and Cybernetics Faculty (http://www.software.unn.ac.ru). The project was implemented with the support of Microsoft Corporation.