Нижегородский государственный университет им. Н.И. Лобачевского

Parallel numerical methods

Laboratory Work
Solving Sparse Linear Systems by Iterative Methods: Problem of Heat Diffusion in a Plate

Supported by Intel

K.A. Barkalov
Software Department

Contents

Purposes of work

- ***The purpose of this laboratory work*** is to see how linear systems with sparse matrices are solved using iterative methods via example of a stationary problem of heat diffusion in a rectangular plate at given temperature conditions at the plate edges.

Objectives of work (1)

- Studying the Successive Over Relaxation method to solve linear systems with general matrices.
- SOR method development to solve linear system with a special matrix (block five-diagonal matrix with the same number on each individual diagonal).
- Development of infrastructure for mass experiments.
- Development of a consecutive SOR method implementation to solve linear system with a block five-diagonal matrix.
- Developed SOR method convergence analysis

Objectives of work (2)

- Development of a so-called evident parallel implementation involving method modification based on Intel® Cilk Plus.
- Evident parallel implementation scalability analysis
- Development of a pipelined parallelization scheme based on Intel® Threading Building Blocks.
- Modified parallel implementation scalability analysis

Test infrastructure

CPU	No. 2 Intel Xeon E5520 (2.27 GHz)
RAM	16 Gb
OS	Microsoft Windows 7
Framework	Microsoft Visual Studio 2008
Compiler, profiler, debugger	Intel Parallel Studio XE 2011
Libraries	Intel® Threading Building Blocks 3.0 for Windows, Update 3 (part of Intel® Parallel Studio XE 2011)

STATIONARY PROBLEM OF HEAT DIFFUSION IN A PLATE

Legend

- l_1, l_2 – lateral lengths of a rectangular plate.
- $U(x, y)$ – plate temperature in the point (x, y) belonging to $G,\ G = \{(x, y)\colon\ x \in [0, l_1], y \in [0, l_2]\}$.
- $f(x, y)$ – total exposure of a rectangular plate to external sources and flows in the point (x, y).

Problem Statement (1)

- ☐ The stationary problem of heat diffusion in a plate is described by a differential Poisson equation.

$$\Delta U = U_{xx} + U_{yy} = -f(x,y) \qquad (1)$$

$$G = \{(x,y)\colon\ x \in [0,l_1], y \in [0,l_2]\}$$

- ☐ For a complete description of a stationary process, set the temperature conditions at the plate edge:

$$\begin{aligned} U(0,y) &= \mu_1(y) \\ U(l_1,y) &= \mu_2(y) \\ U(x,0) &= \mu_3(x) \\ U(x,l_2) &= \mu_4(x) \end{aligned} \qquad (2)$$

Problem Statement (2)

- ❑ To be definite, use the following functions as $f(x,y), \mu_1(y), \mu_2(y), \mu_3(x), \mu_4(x)$

$$f(x,y) = 10\, sin\frac{\pi x}{l_1} sin\frac{\pi y}{l_2}, \tag{3}$$

$$\mu_1(y) = y(l_2 - y)\, cos\frac{\pi(l_2 - y)}{l_2} cos\frac{\pi y}{l_2}, \tag{4}$$

$$\mu_2(y) = \ -y(l_2 - y)\, sin\frac{\pi(l_2 - y)}{l_2}, \tag{5}$$

$$\mu_3(x) = x(l_1 - x)\, cos\frac{\pi(l_1 - x)}{l_1} cos\frac{\pi x}{l_1}, \tag{6}$$

$$\mu_4(x) = -x(l_1 - x)\, sin\frac{\pi(l_1 - x)}{l_1} \tag{7}$$

COMPUTATION SCHEME

Difference scheme (1)

- ❑ Introduce a $x_i = ih, y_j = jk$ grid where $h = \frac{l_1}{n}, k = \frac{l_2}{m}$ within G.
- ❑ Approximate the differential equation using a difference scheme (9).

$$\frac{\left(U_{i-1,j} - 2U_{i,j} + U_{i+1,j}\right)}{h^2} + \frac{\left(U_{i,j-1} - 2U_{i,j} + U_{i,j+1}\right)}{k^2} = -f_{i,j},$$

$$i = \overline{1, n-1}, j = \overline{1, m-1}$$

$$U_{0,j} = \mu_{1,j}, \qquad j = \overline{0, m} \qquad (8)$$

$$U_{n,j} = \mu_{2,j}, \ j = \overline{0, m}$$

$$U_{i,0} = \mu_{3,i}, \ i = \overline{0, n}$$

$$U_{i,m} = \mu_{4,i}, \ i = \overline{0, n}$$

Difference scheme (2)

- Boundary conditions are solved for unknown variables at boundary nodes.
- Substitute the respective unknown variables for their values.
- Consider the first differential scheme equation ($i = 1$).
- In a similar way, cases when $i = n - 1$ и $i = \overline{2, n - 2}$ can be considered.

Difference scheme (3)

$$\frac{(U_{0,j}-2U_{1,j}+U_{2,j})}{h^2}+\frac{(U_{1,j-1}-2U_{1,j}+U_{1,j+1})}{k^2}=-f_{i,j},\ \ j=\overline{2,m-2}$$

$$\frac{U_{2,j}}{h^2}-2\left(\frac{1}{h^2}+\frac{1}{k^2}\right)U_{1,j}+\frac{U_{1,j-1}}{k^2}+\frac{U_{1,j+1}}{k^2}=-f_{i,j}-\frac{1}{h^2}\mu_{1,j},\ \ j=\overline{2,m-2}$$

$$\frac{(U_{0,1}-2U_{1,1}+U_{2,1})}{h^2}+\frac{(U_{1,0}-2U_{1,1}+U_{1,2})}{k^2}=-f_{i,1},\ \ j=1$$

$$\frac{U_{2,1}}{h^2}-2\left(\frac{1}{h^2}+\frac{1}{k^2}\right)U_{1,1}+\frac{U_{1,2}}{k^2}=-f_{i,1}-\frac{1}{k^2}\mu_{3,1}-\frac{1}{h^2}\mu_{1,1},\ \ j=1$$

$$\frac{(U_{0,m-1}-2U_{1,m-1}+U_{2,m-1})}{h^2}+\frac{(U_{1,m-2}-2U_{1,m-1}+U_{1,m})}{k^2}=-f_{i,j},\ \ j=m-1$$

$$\frac{U_{2,m-1}}{h^2}-2\left(\frac{1}{h^2}+\frac{1}{k^2}\right)U_{1,m-1}+\frac{U_{1,m-2}}{k^2}=-f_{i,j}-\frac{1}{h^2}\mu_{1,m-1}-\frac{1}{k^2}\mu_{4,1}$$

$$,\qquad j=m-1$$

Putting a differential system

- Form the vector of unknowns $\vec{U}$ as follows:

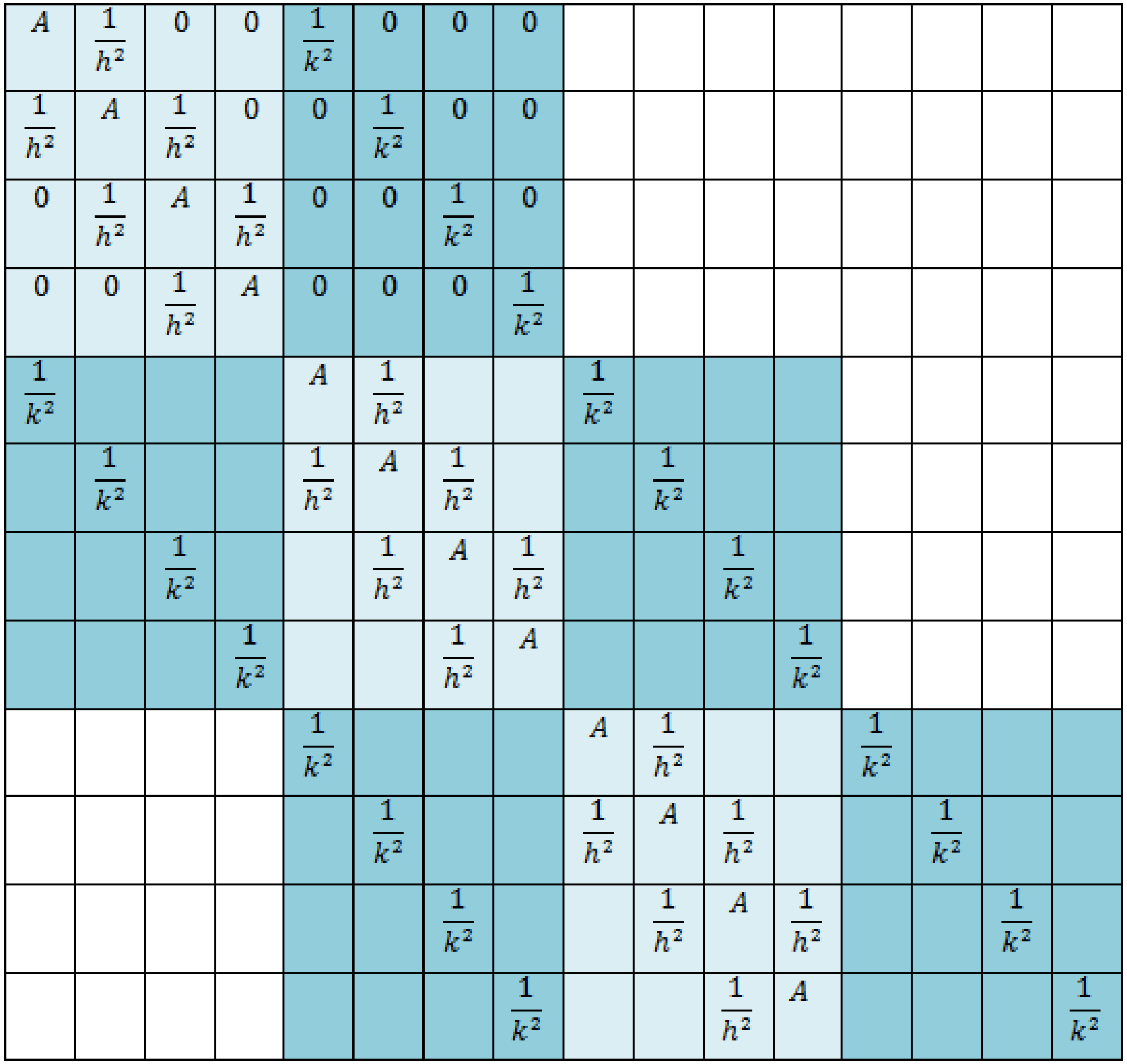
$$\vec{U} = (U_{1,1}, \quad U_{2,1}, \dots, U_{n-1,1},$$
$$U_{1,2}, U_{2,2}, \dots, U_{n-1,2}, \dots,$$
$$U_{1,m-1}, U_{2,m-1}, \dots, U_{n-1,m-1})$$

interior grid nodes are listed as they are located along the coordinate variation axis x.

- Then the differential system can be put it a matrix form like $A\overrightarrow{U = F}$ where $\vec{F}$ is the vector of the function $f(x, y)$ values at the interior nodes minus boundary condition remainders and the matrix A is a block five-diagonal one.

Put the system of differential equations in a matrix form (2)

- ❑ Example of such a matrix for $n = 5, m = 5$ (where $A = -2\left(\frac{1}{h^2} + \frac{1}{k^2}\right)$).



A	$\frac{1}{h^2}$	0	0	$\frac{1}{k^2}$	0	0	0								
$\frac{1}{h^2}$	A	$\frac{1}{h^2}$	0	0	$\frac{1}{k^2}$	0	0								
0	$\frac{1}{h^2}$	A	$\frac{1}{h^2}$	0	0	$\frac{1}{k^2}$	0								
0	0	$\frac{1}{h^2}$	A	0	0	0	$\frac{1}{k^2}$								
$\frac{1}{k^2}$				A	$\frac{1}{h^2}$			$\frac{1}{k^2}$							
	$\frac{1}{k^2}$			$\frac{1}{h^2}$	A	$\frac{1}{h^2}$			$\frac{1}{k^2}$						
		$\frac{1}{k^2}$			$\frac{1}{h^2}$	A	$\frac{1}{h^2}$			$\frac{1}{k^2}$					
			$\frac{1}{k^2}$			$\frac{1}{h^2}$	A				$\frac{1}{k^2}$				
				$\frac{1}{k^2}$				A	$\frac{1}{h^2}$			$\frac{1}{k^2}$			
					$\frac{1}{k^2}$			$\frac{1}{h^2}$	A	$\frac{1}{h^2}$			$\frac{1}{k^2}$		
						$\frac{1}{k^2}$			$\frac{1}{h^2}$	A	$\frac{1}{h^2}$			$\frac{1}{k^2}$	
							$\frac{1}{k^2}$			$\frac{1}{h^2}$	A				$\frac{1}{k^2}$

SOR METHOD

SOR method (1)

- The linear system $A\vec{U} = \vec{F}$ resulting from a stationary heat transfer problem has a symmetric negative definite matrix. The matrix (– *A*) will be positive definite.
- To solve the system (9), use numerical methods applicable to symmetric positive definite matrices:

$$-A\vec{U} = -\vec{F} \tag{9}$$

- The SOR method is a stationary single-step iterative method of linear algebra applicable to symmetric positive definite matrices.

SOR method (2)

- Consider a linear system (10) with a $n*n$ *symmetric positive definite matrix* A:

$$Ax = b \tag{10}$$

- Represent the matrix *A* as a sum of three matrices:

$$A = L + D + R \tag{11}$$

Here, *D* is a $n * n$ diagonal matrix whose principal diagonal coincides with that of the matrix *A*;

L is a $n*n$ lower triangular matrix. Its elements under the main diagonal coincide with the matrix *A* elements, main diagonal being the zero one;

SOR method (3)

R is a $n * n$ upper triangular matrix. Its elements above the main diagonal coincide with the matrix *A* elements, main diagonal being the zero one;

- ❑ Canonical form of the SOR method:

$$\frac{(D + \omega L)\left(x^{(s+1)} - x^{(s)}\right)}{\omega} + Ax^{(s)} = b \qquad (12)$$

Here, $x^{(s)}$, s the approximation obtained at $s + 1$ iteration

ω is the method parameter (number).

SOR method (4)

- The necessary condition of SOR convergence from any initial approximation $x^{(0)}$ to the exact solution x^{*} is fulfillment of $\omega \epsilon (1, 2)$. In case of the symmetric positive definite matrix *A*, this condition is sufficient.
- If $\omega = 1$, the SOR method will be the same as the Seidel method.
- The determine formulas to explicitly compute the next approximation $x^{(s+1)}$ based on the previous one $x^{(s)}$

SOR method (5)

$$(D+\omega L)\left(x^{(s+1)}-x^{(s)}\right)+\omega A x^{(s)}=\omega b$$

$$D x^{(s+1)}+\omega L x^{(s+1)}-D x^{(s)}-\omega L x^{(s)}+\omega A x^{(s)}=\omega b$$

$$D x^{(s+1)}=-\omega L x^{(s+1)}+D x^{(s)}-\omega(A-L) x^{(s)}+\omega b$$

- Given that $A-L=D+R$, we obtain:

$$D x^{(s+1)}=-\omega L x^{(s+1)}+(1-\omega) D x^{(s)}-\omega R x^{(s)}+\omega b \qquad (13)$$

- From (13), record explicit formulas for computation of the new vector $x^{(s+1\,)}$components:

$$a_{ii} {x_i}^{(s+1)}=\omega \sum_{j=1}^{i-1} a_{ij} {x_j}^{(s+1)}+(1-\omega) a_{ii} {x_i}^{(s)}-$$

$$-\omega \sum_{j=i+1}^{n} a_{ij} {x_j}^{(s)}+\omega b_i \qquad (14)$$

SOR method (6)

- As it can be seen from formula (14), to compute the ith component of a new approximation, all smaller index components are taken from the new approximation $x^{(s+1)}$ while all greater index components are taken from the previous one, $x^{(s)}$.
- To implement the method, it is enough to store only one (current) approximation $x^{(s)}$,and to compute the next approximation $x^{(s+1)}$use the formula for all components in series and gradually renew the approximation vector.

SOR method (7)

- Generalized formula for the software implementation:

$$a_{ii}{x_i}^{(s+1)} = -\omega \sum_{j=1}^{n} a_{ij}{x_j}^{(s)} + a_{ii}{x_i}^{(s)} + \omega b_i \qquad (15)$$

- The SOR method convergence rate depends on the ω parameter selection.
- We know that to solve linear systems of certain classes, the SOR method requires $O(n^2)$ iterations. For a certain ω selection, the method will converge after $O(n)$ iterations.
- There is no general analytical formula to compute the best ω_{opt} value.

SOR method (8)

- For a linear system based on a differential scheme (8), the best parameter ω for the SOR method is known and, if the grid size h and k are the same, it is computed using (16):

$$\omega_{opt} = \frac{2}{1 + 2\,sin\left(\frac{\pi h}{2}\right)} \qquad (\ 16)$$

- Minimum and maximum system matrix eigenvalues will in this case be:

$$\lambda_{min} = \frac{4}{h^2} sin^2\left(\frac{\pi}{2n}\right) + \frac{4}{k^2} sin^2\left(\frac{\pi}{2m}\right) \qquad (17)$$

$$\lambda_{min} = \frac{4}{h^2} cos^2\left(\frac{\pi}{2n}\right) + \frac{4}{k^2} cos^2\left(\frac{\pi}{2m}\right) \qquad (18)$$

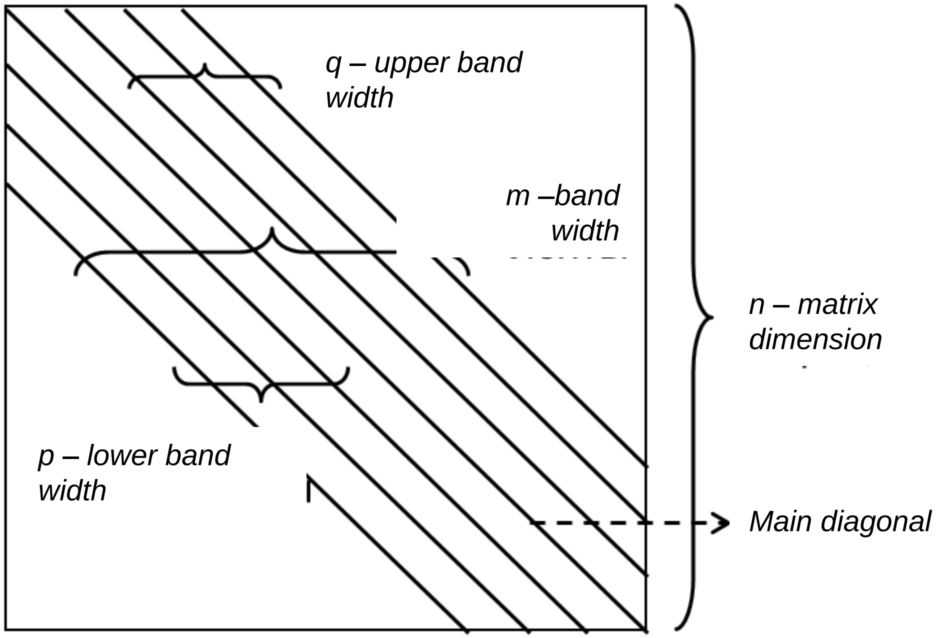
BAND MATRIX STORAGE FORMATS

Band matrix notion (1)

- A A matrix is a *band* one if all its non-zero entries are confined to a band comprising diagonal parallel to the main one.
- If for the matrix A $a_{ij} = 0$ when $i > j + p$ and $a_{ij} = 0$ when $j > i + q$, p is the *lower bandwidth* and q is the *upper bandwidth*.
- $m = p + q + 1$ is called the bandwidth *of the matrix* A.

Band matrix notion (2)

- ❑ Band matrix type:



- ❑ Let us study some band matrix storage types
- ❑ Let us suppose that dimension and bandwidth of the initial matrix A are n and m, respectively.

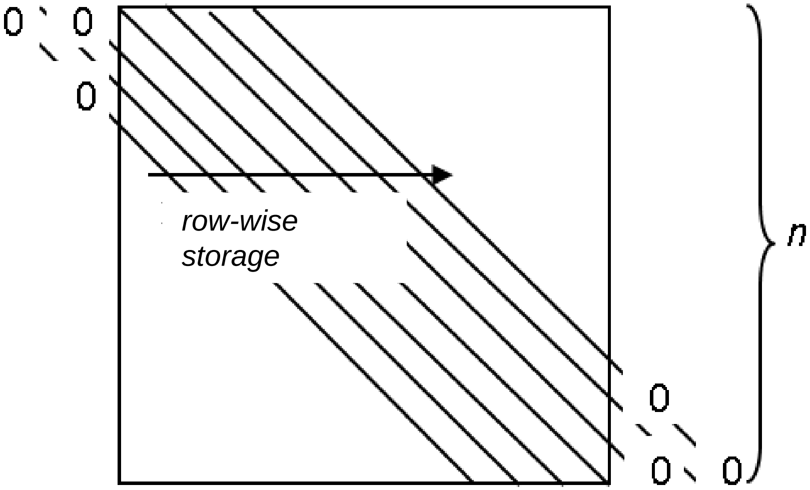
Band format

- *Band format* is used when one can distinguish a dense band of a specific width consisting of nonzeroes.
- If the initial matrix is symmetric, one may store only its lower (or upper) triangle.
- Band format modifications:
  - Band row format
  - Band column format
  - mixed format

Band row format (1)

- *Band row format* to store the initial matrix A uses a $n \times m$ array where nonzeroes of the matrix A are stored row-wise.

- Secondary diagonals are redefined to a n size by adding zeroes at their beginning for the lower triangle and at their end for the upper one.



Band row format (2)

- Example of a matrix stored in the band row format:

Matrix A

$(n = 6, q = 2, p = 1, m = 4)$

1	0	2			
3	4	5	0		
	6	7	8	9	
		10	11	0	12
			0	13	14
				15	16

Storage structure:

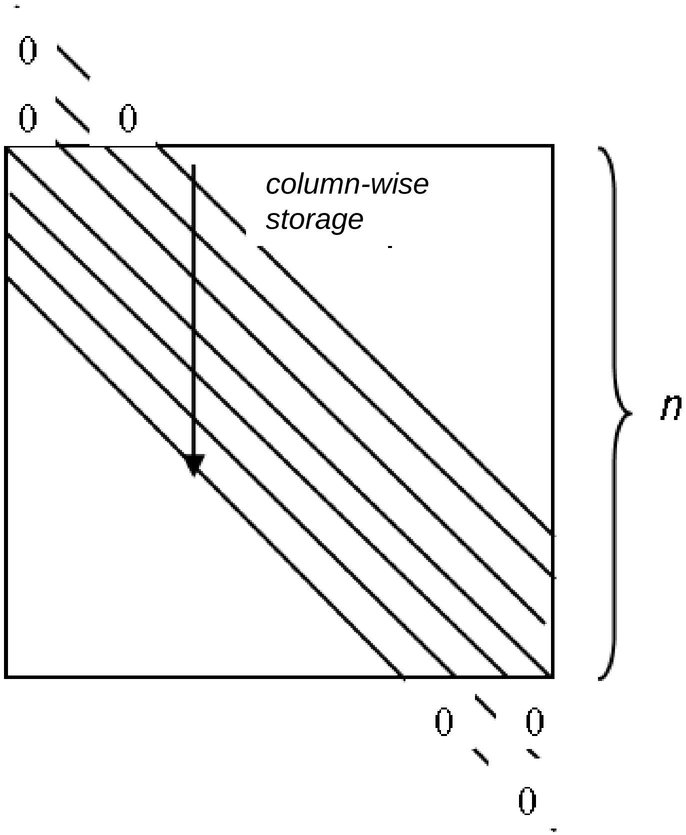
`Matrix`

0	1	0	2
3	4	5	0
6	7	8	9
10	11	0	12
0	13	14	0
15	16	0	0

Band column format (1)

- To store the initial matrix A *band column format* uses a array $n \times m$ with each line containing column nonzeroes of the matrix A.

- Secondary diagonals are redefined to a n size by adding zeroes at their beginning for the upper triangle and at their end for the lower one.



Band column format (2)

- The initial matrix element a_{ij} is stored in an element of the array $A[i-j+q+1, j]$, where is the upper band width of the matrix A.
- Example of a matrix stored in the band column format:

Matrix A

$(n = 6, q = 2, p = 1, m = 4)$

1	0	2			
3	4	5	0		
	6	7	8	9	
		10	11	0	12
			0	13	14
				15	16

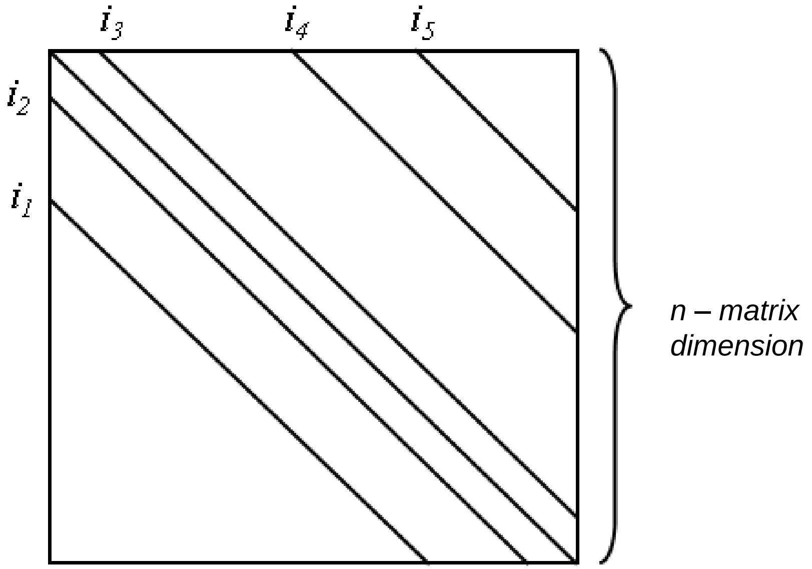
Storage structure:

`Matrix`

0	0	2	0	9	12
0	0	5	8	0	1
1	4	7	11	13	16
3	6	10	0	16	0

Diagonal format (1)

- *Diagonal storage format* is used when all matrix nonzeroes ar located on different diagonals that are not densely spaced.

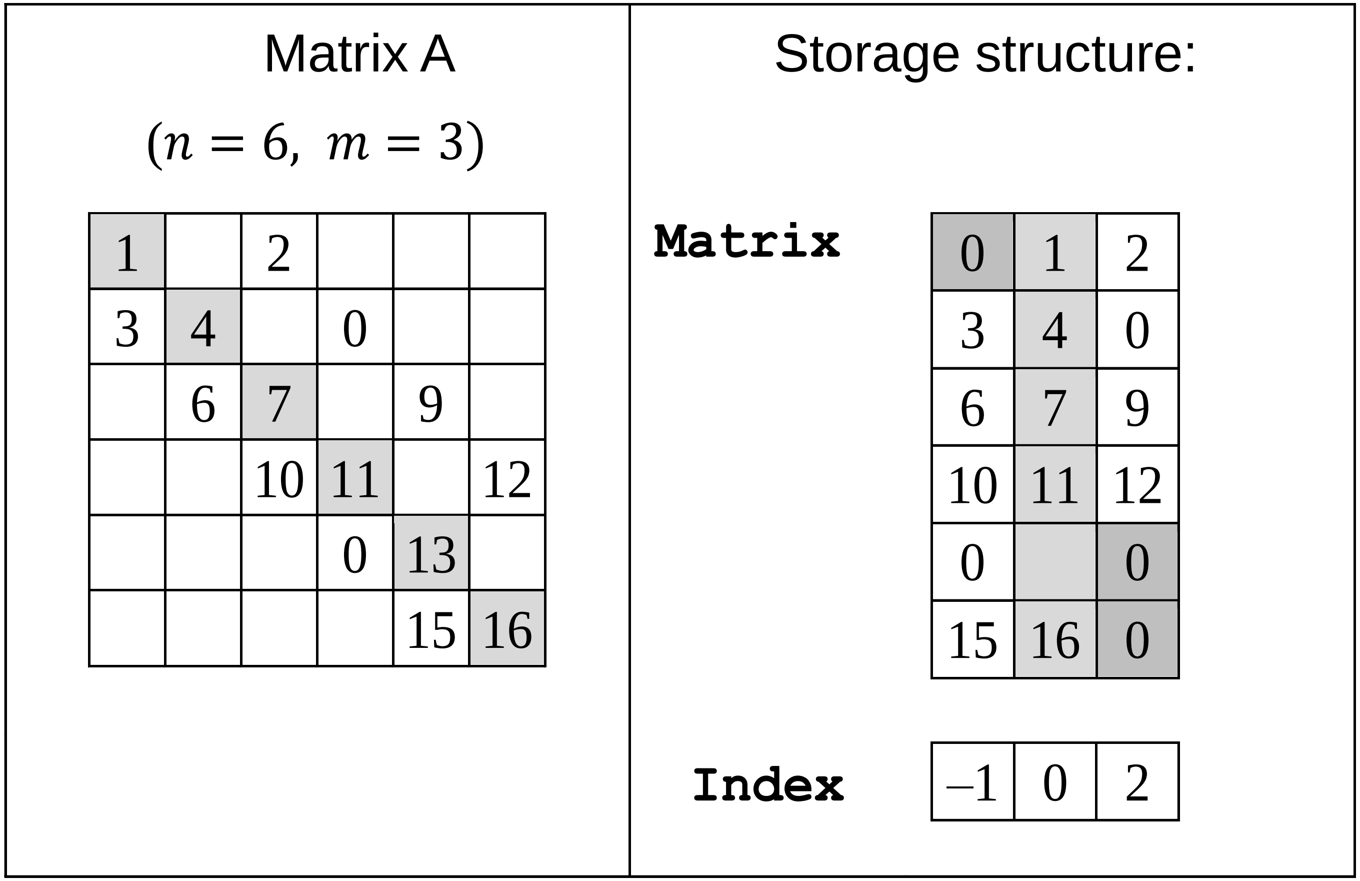


Diagonal format (2)

- To implement the matrix storage, two arrays are used.
- Matrix nonzeroes are stored in the $n \times m$ **Matrix** array, where n is the initial matrix dimension and m is the number of nonzero diagonals.
- Secondary diagonals are redefined to the common size by adding zeroes like in case with the band format.
- In addition, a m **Index** array of integers will be stored to indicate for each diagonal the values of shift from the main diagonal, positive indices for the upper triangle and negative indices for the lower one.

Diagonal format (3)

- Example of a matrix stored in the diagonal format:



Matrix A

$(n = 6,\ m = 3)$

1		2			
3	4		0		
	6	7		9	
		10	11		12
			0	13	
				15	16

Storage structure:

`Matrix`

0	1	2
3	4	0
6	7	9
10	11	12
0		0
15	16	0

`Index`

–1	0	2

Profile format (1)

- *Profile format* is used to store a matrix when the matrix has a wide band with a great number of zeroes inside. The matrix does not have a pronounced structure, but its nonzeroes are concentrated close to its main diagonal.
- Let us see how the profile format is applied to the $n \times n$ symmetric matrix A.
- For each row i of the matrix A, determine the first nonzero shift from the main diagonal.

$$\beta_i = i - j_{min}(i),$$

where $j_{min}(i)$ is the minimum number of column or row i for which $a_{ij} \neq 0$.

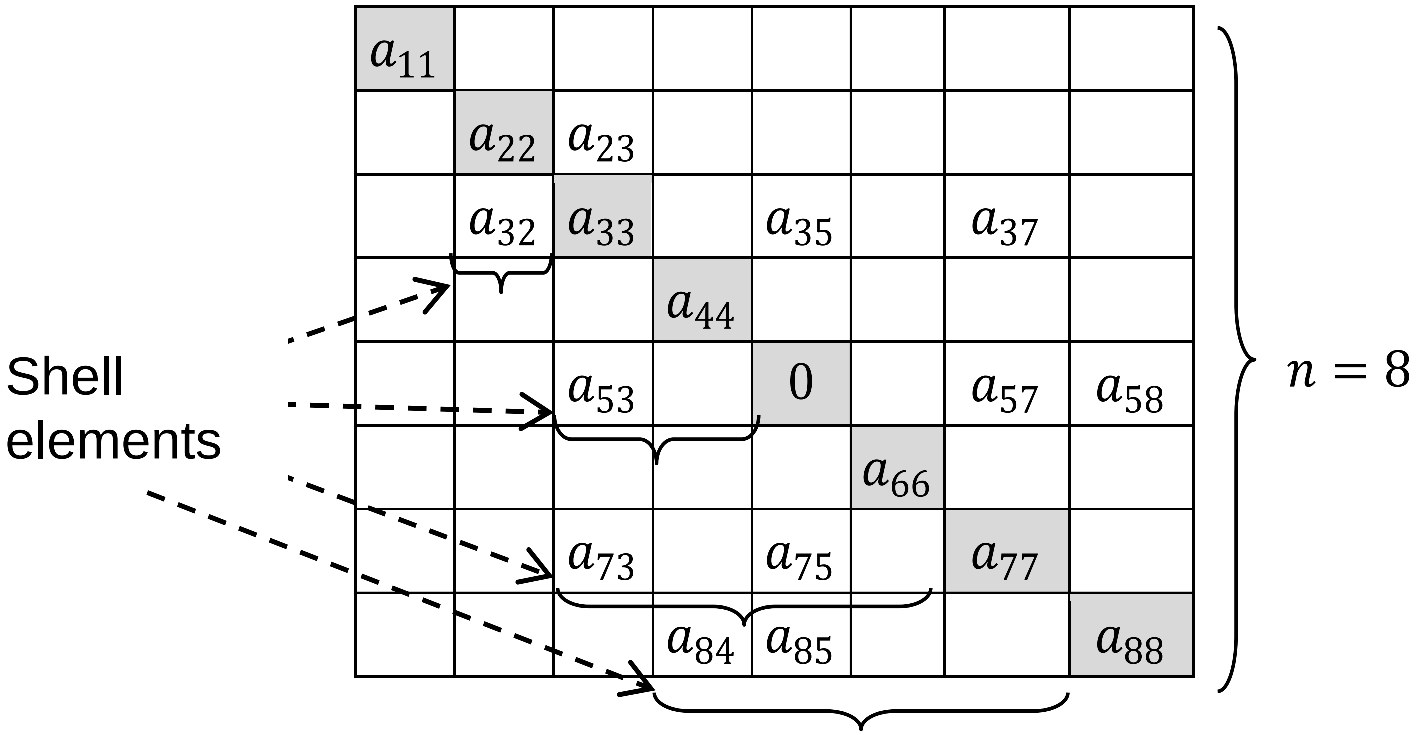
Profile format (2)

- *Shell* of the matrix A is a set of elements a_{ij}, for which $0 < \ i - j \leq \beta_i$. In the ith matrix row, elements with row indices from $jmin_i$ to $i{-}1$, a total of β_i elements, belong to the shell. Diagonal elements are not included into the shell.
- *Profile* of the matrix A is the number of elements in the shell:

$$profile(A) = \sum_{i=1}^{n} \beta_i$$

Profile format (3)

- Matrix example:



$$
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
a_{11} & & & & & & & \\ \hline
 & a_{22} & a_{23} & & & & & \\ \hline
 & a_{32} & a_{33} & & a_{35} & & a_{37} & \\ \hline
 & & & a_{44} & & & & \\ \hline
 & & a_{53} & & 0 & & a_{57} & a_{58} \\ \hline
 & & & & & a_{66} & & \\ \hline
 & & a_{73} & & a_{75} & & a_{77} & \\ \hline
 & & & a_{84} & a_{85} & & & a_{88} \\ \hline
\end{array}
$$

Shell elements

$n = 8$

$$\beta_1 = \beta_2 = \beta_4 = \beta_6 = 0;\ \beta_3 = 1;\ \beta_5 = 2;\ \beta_7 = \beta_8 = 4$$

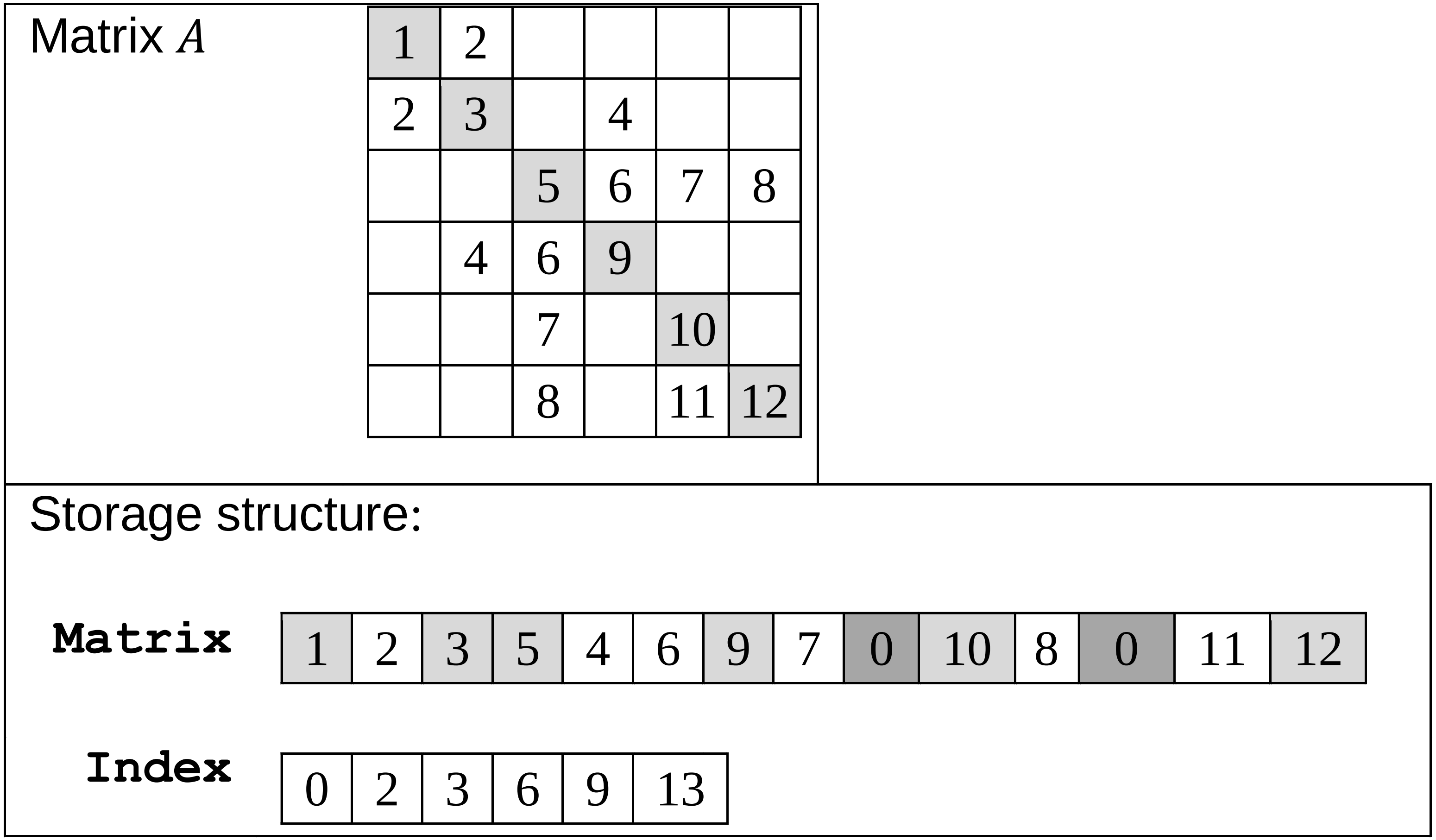
$$profile(A) = 11$$

Profile format (4)

- To store a symmetric matrix in a profile format, two arrays are required.
- All elements of the matrix shell including zeroes arranged in rows are stored in the **Matrix** array size $(profile)A + n$. The diagonal element for this row is placed at its end.
- In addition, the n **Index** array will be stored to contain indices of diagonal matrix elements in the **Matrix** array. Thus, if $i \geq 1$ elements of the i th row of the matrix A are stored in the **Matrix** array from **Index[i – 1]** + 1 to **Index[i].**

Profile format (5)

- Example of a matrix stored in the profile format:



Matrix *A*

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

Storage structure:

Matrix

1	2	3	5	4	6	9	7	0	10	8	0	11	12

Index

0	2	3	6	9	13

Profile format (6)

- Profile storage scheme modification for a non-symmetric matrix with a symmetric pattern requires four arrays:
  - The n **di** array contains diagonal elements.
  - The $profile\ (A)$ **au** and **al** arrays contain off-diagonal elements of the upper triangle in the column-wise manner and lower triangle elements in the row-wise manner, respectively.
  - The auxiliary n **Index** array contains row indices in the **au** and **al** arrays.

Format used to solve the problem (1)

- For the purposes of work, the row format of the symmetrical matrix storage $(-A)$ will be used.
- Both the upper and lower triangles will be stored, as one of the basic SOR operations is multiplication of a matrix row by the vector of unknowns$U$.
- Additional array **Index** will contain elements $-n+1, -1, 0, 1, n-1$.
- Add $n-1$ zeroes to the vector of unknowns at the beginning and the end:

$$U = \left(U_{1,0}, U_{2,0}, \dots, U_{n-1,0}, U_{1,1}, \dots, U_{n-1,m-1}, U_{1,m}, U_{2,m}, \dots, U_{n-1,m}\right),$$

$$U_{i,0} = U_{i,m} = 0, i = \overline{1, n-1}$$

Format used to solve the problem (2)

- Example of use of the selected storage format:

Matrix A

$$(n = 4,\ m = 3)$$

50	-16		-9		
-16	50	-16		-9	
	-16	50			-9
-9			50	-16	
	-9		-16	50	-16
		-9		-16	50

Storage structure:

Matrix

0	0	50	-16	-9
0	-16	50	-16	-9
0	-16	50	0	-9
-9	0	50	-16	0
-9	-16	50	-16	0
-9	-16	50	0	0

Index

-3	-1	0	1	3

SOFTWARE IMPLEMENTATION

Consecutive version

Project creation (1)

- Run **Microsoft Visual Studio 2008**
- From the **File** menu, select **New→Project….**
- From the **New Project**, select **Win32** from the Project types pane and **Win32 Console Application** from the Templates pane; enter **BandOverRelaxation** in the **Solution** field, enter **01_BandOR_seqv** in the **Name** field, enter **c:\ParallelCalculus\** (path to the folder with laboratory works). Press **OK.**
- From the **Win32 Application Wizard** dialog, press **Next** and click **Empty Project**. Press **Finish.**

Project creation (2)

- From the **Solution Explorer**, execute **Add→New Item** in the **Source Files** folder. In the selection tree, select **Code**; select **C++ File (.cpp)** in the templates on the right, enter **main** in the **Name** field. Press **Add.**
- In a similar way, add **BandOverRelax.cpp**, PoissonDecision.cpp and Utilities.cpp.
- From the **Solution Explorer**, execute **Add→New Item** in the **Header Files** folder. In the selection tree, select **Code**; select **Header File (.h)** in the templates on the right, enter **BandOverRelax** in the **Name** field. Press **Add.**

Project creation (3)

- In a similar way, add **PoissonDecision.h** and **Utilities.h**.

- **BandOverRelax.h** and **BandOverRelax.cpp** will store prototypes and implementations of functions necessary for the SOR method.
- **PoissonDecision.h** and **PoissonDecision.cpp** will contain prototypes and implementations of functions determining the right-hand part and boundary conditions of the differential equation and the functions of solving differential equations.
- **Utilities.h** and **Utilities.cpp** will contain prototypes and implementations of auxiliary functions.

Connection to the Intel® Math Kernel Library (1)

- To check for correctness the solution obtained using the SOR method, use the linear system solution functions from the MKL library.
- Library connection:
  - Open **Tools** → **Options** and select **Projects and Solutions→VC++ Directories**.
  - In the drop-down menu, first select **Include Files**, add a new entry containing the path to MLK library header files (e. g. **C:\Program Files (x86)\Intel\ComposerXE-2011\mkl\incl**ude),

Connection to the Intel® Math Kernel Library (2)

- Then select **Library Files** and add the path to the library files:
  - To assemble a 32-bit application, enter the path to the static library for the ia-32 platform (e. g. **C:\Program Files (x86)\Intel\ComposerXE-2011\mkl\lib\ia32**)
  - To assemble a 64-bit application, enter the path to the static library for the 64-bit platform (e. g. **C:\Program Files (x86)\Intel\ComposerXE-2011\mkl\lib\intel64**).

Connection to the Intel® Math Kernel Library (3)

- From **Configuration Properties** in the **Linker→Input→Additional Dependencies tab**, enter the following static libraries:
  - for a 32-bit application, they are **mkl_core.lib, mkl_intel_c.lib, mkl_Consecutive.lib.**
  - for z 64-bit application, they are **mkl_core.lib, mkl_Consecutive.lib, mkl_intel_lp64.lib, mkl_blas95_lp64.lib.**

Elementary function (1)

```
//SOR method accuracy
#define EPSILON 0.00001

int main(int argc, char* argv[]) {
  int n, m;      //grid parameters
  int StepCount; //number of steps performed by the SOR method
  int size;      //linear system dimension
  // variables to store the computing function runtimes
  double time, MKLtime;
  // set accuracy of the SOR method as a stop criterion
  double Accuracy = EPSILON;
  double ORAccuracy;   //attainable SOR accuracy
  double* Decision;    //solution found using the SOR method
  double *DecisionMKL; //exact system solution found using MKL
  // difference between the exact solution and the solution
  // found using the SOR method (in norm)
  double ExcAccuracy;
  // variables to save the results in a file
  char* FileName = "sparseOR_res.csv"; FILE* file;
  // continued in the following slide
```

Elementary function (2)

```
//1. Reading the command line parameters
  if ((argc > 2) && (argc < 6)) {
    n = atoi(argv[1]);
    m = atoi(argv[2]);
    if (argc >= 4) {
      Accuracy = atof(argv[3]);
      if (argc == 5)
        FileName = argv[4];
    }
  }
  else {
    printf("Invalid input parameters\n");
    return 1;
  }
  if ((n < 0) || (m < 0) || (Accuracy < 0)) {
    printf("Incorrect arguments of main\n");
    return 1;
  }
// continued in the following slide
```

Elementary function (3)

```
   size = (n - 1)*(m - 1);

   //2. Memory allocation to arrays
   //Decision and DecisionMKL of the size dimension

   //3. Calling the function of solving linear systems by the SOR 
method,
   printf("OverRelaxation:\ntime = %.15f\n", time);
   printf("Accuracy = %.15f, stepCount = %d\n", ORAccuracy, StepCount);

   //4. Solution verification:
   //  4.1. Finding the exact system solution using MKL
   //  4.2. Comparison of Decision and DecisionMKL
   MKLtime = ComputeDecisionMKL(n, m, DecisionMKL);
   ExcAccuracy = CompareDecisions(Decision, DecisionMKL, size);
   printf("MKL:\ntime = %.15f\n", MKLtime);
   printf("OR and MKL comparison = %.15f\n", ExcAccuracy);

   // continued in the following slide
```

Elementary function (4)

```
//5. Results filing
  file = fopen(FileName, "a+");
  if (file) {
    fprintf(file, "%d;%d;%.15f;%.15f;%.15f;%d;%.15f\n",
                   n, m, Accuracy, ORAccuracy, ExcAccuracy,
                   StepCount, time);
  }
  fclose(file);


  //6. Memory release for Decision and DecisionMKL arrays

  return 0;
}
```

Auxiliary functions

- **Utilities.h** will contain prototypes and **Utilities.cpp** will contain implementation of memory allocation and release functions.

```cpp
//Memory allocation
void InitializeVector(double** Vector, int size);
void InitializeVector(int ** Matrix, int size);

// memory release
void FreeVector(double** Vector);
void FreeVector(int** Vector);
```

Auxiliary functions. Differential equation description (1)

- **PoissonDecision.h** will contain prototypes and **PoissonDecision.cpp** will contain implementation of functions describing the right-hand part and boundary conditions of the differential equation:

```
#define _USE_MATH_DEFINES
#include "math.h"

#define LEFT_BOUND  10.0
#define RIGHT_BOUND 10.0


// Function of right-hand part computation for partial differential
equations
double f(double x, double y) {
  return  10*sin(M_PI*x/LEFT_BOUND)*sin(M_PI*y/RIGHT_BOUND);
}
```

Auxiliary functions. Differential equation description (2)

```
// Function of BC computation at the left side of the rectangle
double mu1(double y) {
  return  y*(RIGHT_BOUND - y)*cos(M_PI*(RIGHT_BOUND - y)/
                RIGHT_BOUND)*cos(M_PI*y/RIGHT_BOUND);
}

// Function of BC computation at the right side of the rectangle
double mu2(double y) {
  return  -y*(RIGHT_BOUND - y)*sin(M_PI*(RIGHT_BOUND - y)/RIGHT_BOUND);
}

// Function of BC computation at the lower side of the rectangle
double mu3(double x) {
  return x*(LEFT_BOUND - x)*cos(M_PI*(LEFT_BOUND - x)/
                LEFT_BOUND)*cos(M_PI*x/ LEFT_BOUND);
}

// Function of BC computation at the upper side of the rectangle
double mu4(double x) {
  return -x*(LEFT_BOUND - x)*sin(M_PI*(LEFT_BOUND - x)/ LEFT_BOUND);
}
```

Auxiliary functions. Linear system initialization (1)

- **Utilities.h** will contain prototypes and **Utilities.cpp** will contain implementation of functions forming the right-hand matrix and vector of the linear system based on a difference scheme in the selected format.

```
// matrix initialization for a grid (n, m)
void CreateDUMatrix(int n, int m, double** Matrix, int** Index) {
  // matrix dimension, band width
  int size = (n - 1)*(m - 1), bandWidth = 5;
  // matrix elements
  double hsqr = (double)n*n/LEFT_BOUND/ LEFT_BOUND; // 1/h
  double ksqr = (double)m*m/RIGHT_BOUND/RIGHT_BOUND;// 1/k
  double A = 2*(hsqr + ksqr);

  //1. Memory allocation
  InitializeVector(Matrix, size*bandWidth);
  InitializeVector(Index, bandWidth);
  // continued in the following slide
```

Auxiliary functions. Linear system initialization (2)

```
  //2. Index array initialization
  (*Index)[0] = -n + 1; (*Index)[1] = -1; (*Index)[2] = 0;
  (*Index)[3] = 1; (*Index)[4] = n - 1;

  //3. Initialization of the matrix (-A) based on a differential scheme
  for (int i = 0; i < size; i++) {
    if (i >= n - 1) (*Matrix)[i*bandWidth] = -ksqr;
    else  (*Matrix)[i*bandWidth] = 0.0;
    i
    else  (*Matrix)[i*bandWidth + 1] = 0.0;
    (*Matrix)[i*bandWidth + 2] = A;
    if ((i + 1) % (n - 1) != 0) (*Matrix)[i*bandWidth + 3] = -hsqr;
    else  (*Matrix)[i*bandWidth + 3] = 0.0;
    if (i < (n - 1)*(m - 2)) (*Matrix)[i*bandWidth + 4] = -ksqr;
    else  (*Matrix)[i*bandWidth + 4] = 0.0;
  }
}
```

Auxiliary functions. Linear system initialization (3)

```
// vector initialization for a grid (n, m)
void CreateDUVector(int n, int m, double** Vector) {
  // auxiliary variables
  double h = LEFT_BOUND/(double)n;
  double k = RIGHT_BOUND/(double)m;
  double hsqr = (double)n*n/LEFT_BOUND/LEFT_BOUND;
  double ksqr = (double)m*m/RIGHT_BOUND/RIGHT_BOUND;

  //1. Memory allocation
  InitializeVector(Vector, (n - 1)*(m - 1));

  //2. Initialization of the linear system left part based on a differential
scheme
  for(int j = 0; j < m - 1; j++) {
    for(int i = 0; i < n - 1; i++)
      (*Vector)[j*(n - 1) + i] = f((double)(i + 1)*h,
                                   (double)(j + 1)*k);
    (*Vector)[j*(n - 1)]           += hsqr*mu1((double)(j + 1)*k);
    (*Vector)[j*(n - 1) + n - 2] += hsqr*mu2((double)(j + 1)*k);
  }
  // continued in the following slide
```

Auxiliary functions. Linear system initialization (4)

```
  for (int i =0; i < n - 1; i++) {
    (*Vector)[i] += ksqr*mu3((double)(i + 1)*h);
    (*Vector)[(m - 2)*(n - 1) + i] += ksqr*mu4((double)(i + 1)*h);
  }
}
```

- ☐ Implement **CreateMKLMatrix()** that enables initialization of the matrix $(-A)$ based on a differential scheme (8) from the (n,m) grid in the format used by the MKL library (the upper triangle will be stored in a column-wise way).

```
// matrix initialization for MKL
void CreateMKLMatrix(int n, int m, double** Matrix);
```

Auxiliary functions. Obtaining the exact system solution (1)

```
// finding the exact system solution using MKL
#include "mkl_lapack.h"

double ComputeDecisionMKL(int n, int m, double* Decision) {
  double *Matrix, *Vector; //system matrix and vector
  int size = (n - 1) * (m - 1);
  // auxiliary variables for MKL functions
  char uplo = 'U'; //consider the matrix as an upper triangular one
  int kd = n - 1;  //number of top diagonals
  int ldab = n;    //first matrix dimension
  int info = 0;    //output parameter, error code
  int nrhs = 1;    //number of right-hand parts
  double time;     //runtime

  //1. Matrix and right-hand vector initialization
  CreateMKLMatrix(n, m, &Matrix);
  CreateDUVector(n, m, &Vector);

  // continued in the following slide
```

Auxiliary functions. Obtaining the exact system solution (2)

```
 //2. Solving the system
  clock_t start = clock();
  dpbtrf(&uplo, &size, &kd, Matrix, &ldab, &info);
  dpbtrs(&uplo, &size, &kd, &nrhs, Matrix, &ldab, Vector, &size,
&info);
  time = (double)(clock() - start) / CLOCKS_PER_SEC;

  //3. Memorization
  memcpy(Decision, Vector, sizeof(double)*size);

  //4. Memory release
  FreeVector(&Matrix);
  FreeVector(&Vector);

  return time;
}
```

Auxiliary functions. Obtaining the exact system solution (3)

- Implement **CompareDecisions()** that enable finding the norm of difference between the solution found using the SOR method and the exact solution obtained by the MKL (the vector norm can be defined as $\|x\|_{\infty} = \max_{i} |x_i|$).

```
// comparison of solutions
double CompareDecisions(double* ORResult, double* MKLDecision,
                                                    int size);
```

Auxiliary functions. Calling the Successive Over Relaxation method (1)

- ComputeDecision() will compute an approximated decision of the differential equation using the SOR method within a (n, m) grid. Place the function implementation in **PoissonDecision.cpp** and declare the respective prototype in **PoissonDecision.h.**

```
// approximated differential equation solution computation for a grid (n, m)
// obtained solution is stored in the Decision vector
// function returns the method runtime
double ComputeDecision(int n, int m, double* Decision,
                       double Accuracy, double &ORAccuracy,
                       int &StepCount) {
  // matrix, vector, solution
  double* Matrix, *Vector, *Result;
  int* Index;
  int size = (n - 1)*(m - 1);     //system dimension
  int ResSize = size + 2*(n - 1); // augmented vector dimension
  // continued in the following slide
```

Auxiliary functions. Calling the Successive Over Relaxation method (2)

```
int bandWidth = 5; // band width

 // variables to measure time
 clock_t start, finish;
 double time;
 // SOR method parameters
 double WParam;
 double step = n/LEFT_BOUND > m/RIGHT_BOUND) ?
                    (double) LEFT_BOUND /n : (double) RIGHT_BOUND /m;
 //1. System initialization
 CreateDUMatrix(n, m, &Matrix, &Index);
 CreateDUVector(n, m, &Vector);

 //2. Method initialization
 InitializeVector(&Result, ResSize);
 GetFirstApproximation(&Result, ResSize);
 WParam = GetWParam(step);

 // continued in the following slide
```

Auxiliary functions. Calling the Successive Over Relaxation method (3)

```
 //3. Approximated solution computation using the successive over
  // relaxation method
  start = clock();
  ORAccuracy = BandOverRelaxation(Matrix, Vector, &Result, Index, size,
                                  bandWidth, WParam, Accuracy,
                                  &stepCount);
  finish = clock();
  time = (double)(finish - start)/CLOCKS_PER_SEC;
  // solution saving
  memcpy(Decision, Result + n - 1, sizeof(double)*size);

  //4. Memory release
  FreeVector(&Matrix);
  FreeVector(&Index);
  FreeVector(&Vector);
  FreeVector(&Vector);

  return time;
}
```

Implementation of the Successive Over Relaxation method (1)

- Implement the SOR method as applicable to the block five-diagonal matrix. Place prototypes of the respective functions in **BandOverRelax.h** and their implementation - in **BandOverRelax.cpp.**

```
#define N_MAX 50000 //maximum allowable number of steps

// setting zero approximation for the SOR method
void GetFirstApproximation(double** Result, int size) {
  for(int i = 0; i < size; i++)
    Result[i] = 0.0;
}

// setting the SOR method parameter depending on the grid size
double GetWParam(double Step) {
  return 2 / (1 + 2*sin(M_PI*Step/2));
}
```

Implementation of the Successive Over Relaxation method (2)

```
//SOR method for band matrices
// function returns the attainable system solution accuracy
double BandOverRelaxation(double* Matrix, double* Vector,
                          double** Result, int* Index, int size,
                          int bandWidth, double WParam,
                          double Accuracy, int &StepCount) {
  // auxiliary variables
  double CurrError; //attainable accuracy for the iteration
  double sum, TempError;
  int ii, index = Index[bandWidth - 1], bandHalf = (bandWidth - 1)/2;
  StepCount = 0;

  do {
    CurrError = -1.0;
    for(int i = index; i < size + index; i++) {
      ii = i - index;
      TempError = (*Result)[i];
      sum = 0.0;

  // continued in the following slide
```

Implementation of the Successive Over Relaxation method (3)

```
for (int j = 0; j < bandWidth; j++)
        sum += Matrix[ii*bandWidth + j] * (*Result)[i + Index[j]];
      (*Result)[i] = (Vector[ii] - sum) * WParam /
                      Matrix[ii*bandWidth + bandHalf] + (*Result)[i];
      TempError = fabs((*Result)[i] - TempError);
      if (TempError > CurrError) CurrError = TempError;
    }
    StepCount++;
  }
  while ((CurrError > Accuracy)&&(StepCount < N_MAX));
  return CurrError;
}
```

Project compilation and application run

- Add missing functions to the software implementation; include necessary header files.
- Having developed the software implementation, build the project by executing **Build→Rebuild 01_BandOR_seqv** and check the application for consistent running.

Method convergence analysis (1)

- To analyze the implemented SOR method, let us consider a system resulting from a differential equation with a predefined known solution and boundary conditions.
- Study temperature variation in a plate with lateral lengths $l1_= l2 = 1_.$ Let the function $u(x,y) = x^2y + y^2x$ be the heat diffusion equation solution Then, the equation (1) will look as follows:

$$\Delta U = -f(x,y) = 2(x+y) \qquad (19)$$

Method convergence analysis (2)

- Boundary conditions (2) will satisfy the following equations:

$$U(0,y) = \mu_1(y) = 0 \qquad (20)$$
$$U(l_1,y) = \mu_2(y) = l_1(l_1 + y^2)$$
$$U(x,0) = \mu_3(x) = 0$$
$$U(x,l_2) = \mu_4(x) = l_2(l_2 + x^2)$$

- Replace implementation of functions corresponding to setting functions f, μ_1, μ_2, μ_3, μ_4 in the program code.

Method convergence analysis (3)

- In **PoissonDecision.cpp**, implement the `FunkU()` function that returns the solution fucntion value at a certain point:

```
// exact solution of a differential equation
double FuncU(double x, double y);
```

- In **Utilities.cpp**, implement the **CheckDecision()** function that makes it possible to find the residual norm of a solution obtained using a certain method with a predetermined solution.

```
// solution accuracy verification
double CheckDecision(double* Decision, int n, int m)
```

Method convergence analysis (4)

- Experimental results with various accuracy:

n, m	ε	Number of steps	Attainable accuracy	Allowed difference from the exact solution
10	0,001	20	0,000467	0,000383
50	0,001	101	0,000723	0,000380
100	0,001	201	0,000703	0,000455
500	0,001	1001	0,000461	0,000671
1000	0,001	2001	0,000380	0,000844
10	0,0001	22	0,000078	0,000107
50	0,0001	104	0,000075	0,000268
100	0,0001	204	0,000084	0,000360
500	0,0001	1004	0,000080	0,000535
1000	0,0001	2004	0,000085	0,000624
10	0,00001	26	0,0000099	0,000013
50	0,00001	115	0,0000095	0,000058
100	0,00001	219	0,0000098	0,000131
500	0,00001	1016	0,0000097	0,000393
1000	0,00001	2036	0,0000099	0,000382

Method convergence analysis (5)

- The method quickly converges to a solution. This effect may be partially explained by simplicity of the right-hand function and boundary conditions.
- The norm of difference from the exact solution for all the above cases has an order of at least 10^{-4}.
- For the selected functions, a single-order decrease of the required accuracy ε given a fixed grid size has an insignificant influence on the number of method iterations.

SOFTWARE IMPLEMENTATION

Parallel Intel® Cilk Plus-based version

Project creation

- From **BandOverRelaxation**, create a new project entitled **02_BandOR_cilk**.
- Create empty files **main.cpp, PoissonDecision.h, PoissonDecision.cpp, BandOverRelax.h, BandOverRelax.cpp, Utilities.h, Utilities.cpp** and copy to these files the code from the respective files of **01_BandOR_seq**.
- Connect Cilk Plus. For this purpose, open **Configuration Properties** and select **C\C++→Language** to make sure that the value of the **Disable Intel Cilk Plus Keywords For Serial Semantics** field is “No”.

main() function modification (1)

```
int main(int argc, char* argv[]) {
  ...
  int NumThreads; //number of threads

  //1. Reading the command line parameters
  if ((argc > 2) && (argc < 7)) {
    n = atoi(argv[1]);
    m = atoi(argv[2]);
    NumThreads = atoi(argv[3]);
    if (argc >= 5) {
      Accuracy = atof(argv[4]);
      if (argc == 6)
        FileName = argv[6];
    }
  }

  ...
  // continued in the following slide
```

main() function modification (2)

```
  ...
  //3. Calling the function of solving linear systems by the SOR method
  time = ComputeDecision(n, m, Decision, NumThreads,
                                Accuracy, ORAccuracy, StepCount);
  //5. Results filing
  file = fopen(FileName, "a+");
  if (file)
  {
    fprintf(file, "%d;%d;%.15f;%.15f;%.15f;%d;%.15f;%d\n",
                      n, m, Accuracy, ORAccuracy, ExcAccuracy,
                      StepCount, time, NumThreads);
  }
  fclose(file);
  ...
}
```

ComputeDecision() function modification

```
#include "cilk/cilk_api.h"

double ComputeDecision(...) {
  ...
  WParam = GetWParam(step);

  // set the number of threads
  char nt[3];
  itoa(NumThreads, nt, 10);
  __cilkrts_set_param("nworkers", nt);

  //calling the successive over relaxation method
  start = clock();
  ORAccuracy = BandOverRelaxationCilk(Matrix, Vector, Result, Index,
                        size, bandWidth, WParam, Accuracy, StepCount);
  finish = clock();
  time = (double)(finish - start)/CLOCKS_PER_SEC;
  ...
}
```

BandOverRelaxation() function modification (1)

- We will obtain the SOR method modification for cycle parallelization within the iteration. Computation will result in mixed approximations whose elements have been obtained using mixed new and old components without keeping strictly to the method formula (14).
- Introduce simple changes to the **BandOverRelaxation()** code and name it **BandOverRelaxationCilk().**
- Parallelize the cycle by vector elements using **cilk_for**. In this case, the variables assuming different values for each cycle iteration must be declared locally.

BandOverRelaxation() function modification (2)

- Declare the **currError** variable as a reducer for the maximization operation. This will ensure safe use of the shared variable, reduce synchronization costs and enable its parallel computation.
- To work with **currError**, use the following functions:
  - **set_value()** to initialize the accuracy value at the beginning of the iteration;
  - **get_value()** to obtain the value;
- The reduction operation itself is effected by means of **cilk::max_of().**

BandOverRelaxation() function modification (3)

```
#include "cilk/cilk.h"
#include "cilk/reducer_max.h"

double BandOverRelaxationCilk(...) {
  cilk::reducer_max<double> CurrError;
  ...
  StepCount = 0;
  do {
    CurrError.set_value(-1.0);
    cilk_for (int i = index; i < size + index; i++) {
    int ii = i - index;
    double TempError = Result[i];
    double sum = 0.0;
    for (int j = 0; j < bandWidth; j++)
      sum += Matrix[ii*bandWidth + j] * Result[i + Index[j]];
    Result[i] = (Vector[ii] - sum) * WParam /
                       Matrix[ii*bandWidth + bandHalf] + Result[i];
    TempError = fabs(Result[i] - TempError);
    // continued in the following slide
```

BandOverRelaxation() function modification (4)

```
      CurrError = cilk::max_of(TempError, CurrError);
    }
    StepCount++;
  }
  while ((CurrError.get_value() > Accuracy)&&
         (StepCount < N_MAX));

  return CurrError.get_value();
}
```

Project compilation and application run

- Modify the program code as required.
- Having developed the software implementation, build the project by executing **Build→Rebuild 02_BandOR_seqv** and check the application for consistent running.

Scalability analysis (1)

- To analyze the pipelined scheme efficiency, perform an experiment using functions based on formulas (3) – (7) with an accuracy of $\varepsilon = 10^{-5}$.

- See the next slide for the table showing dependence of the number of method iterations on the number of application threads.

Scalability analysis (2)

Grid size	Consecutive version	Parallel Intel® Cilk Plus-based version				
		1 thread	2 threads	4 threads	6 threads	8 threads
100	2917	2917	2917	2995	3114	3136
200	5409	5409	5409	5657	6049	6172
300	7731	7731	7732	8243	8844	9023
400	9944	9944	9945	10611	11766	12126
500	12076	12076	12076	12993	14366	14896
600	14145	14145	14145	15237	17219	18120
700	16160	16160	16160	17573	19713	20836
800	18129	18129	18131	19808	22536	23848
900	20059	20059	20060	21738	25065	26494
1000	21953	21953	22031	23862	27662	29155

Scalability analysis (3)

- The greater is the number of threads, the more iterations are performed. The reason is that the parallel scheme does not take into account the strict sequence of the method approximations but builds intermediate approximations.
- See the next slide for results of a multi-thread Cilk version. *T* is the runtime (in seconds), *S* is the acceleration as compared to single thread operation.

Scalability analysis (4)

System dimension	1 thread	2 threads		4 threads		6 threads		8 threads	
	T	*T*	*S*	*T*	*S*	*T*	*S*	*T*	*S*
100	0,68	0,53	1,27	0,32	2,12	0,33	2,04	0,34	2,01
200	5,03	2,75	1,83	1,68	2,99	1,47	3,41	1,31	3,85
300	16,23	8,61	1,89	4,98	3,26	4,15	3,91	3,46	4,69
400	38,03	20,31	1,87	11,09	3,43	9,08	4,19	7,49	5,08
500	73,03	38,45	1,90	21,65	3,37	17,21	4,24	14,54	5,02
600	123,74	63,88	1,94	36,31	3,41	30,20	4,10	29,28	4,23
700	192,64	98,88	1,95	57,14	3,37	47,92	4,02	48,70	3,96
800	283,27	145,97	1,94	83,73	3,38	71,78	3,95	74,52	3,80
900	397,73	205,74	1,93	115,16	3,45	101,53	3,92	105,10	3,78
1000	538,16	278,87	1,93	155,83	3,45	137,41	3,92	143,02	3,76

Scalability analysis (6)

- The maximum acceleration of 5 was obtained for 8 threads when $n = 400, n = 500$. A greater grid size reduces the acceleration to 4. The reason is an increase in the number of iterations as the number of threads grows and an increase in contingencies and thread synchronization costs.
- Upon the whole, the approach showed satisfactory acceleration results, however, it requires parallel version modification to reduce unnecessary computations.

SOFTWARE IMPLEMENTATION

Parallel implementation of the Intel® TBB-based pipelined scheme

Project creation

- From **BandOverRelaxation**, create a new project entitled **03_BandOR_tbb**.
- Create empty files **main.cpp, PoissonDecision.h, PoissonDecision.cpp, BandOverRelax.h, BandOverRelax.cpp, Utilities.h, Utilities.cpp** and copy to these files the code from the respective files of **02_BandOR_seq**.
- Create **TaskImplementation.h** and **TaskImplementation.cpp** to store declaration and implementation of problem classes that will ensure the pipelined scheme operation.

TBB library connection

- Indicate the path to the library header files **(Configuration Properties→C/C++→General→Additional Include Directories)**,
- Indicate the path to the library **.lib** files **(Configuration Properties→Linker→General→Additional Library Directories)**,
- Indicate the **tbb.lib** library (**Configuration Properties→Linker→Input→Additional Dependencies**), to assemble the project.

TBB library initialization

- To benefit from parallelization capabilities offered by TBB, one must have at least one active (initialized) **tbb::task_scheduler_init** class instance.
- This class is intended for creation of threads and internal structures for the thread planner.

ComputeDecision() function modification

```
#include "tbb/task_scheduler_init.h"

double ComputeDecision(...) {
  ...
  GetFirstApproximation(Result, ResSize);
  WParam = GetWParam(step);

  // set the number of threads
  tbb::task_scheduler_init init(NumThreads);

  start = clock();
  ORAccuracy = BandOverRelaxationTBB(Matrix, Vector, &Result, Index,
          size, bandWidth, WParam, Accuracy, StepCount, n, NumThreads);
  finish = clock();
  ...
}
```

Description of a pipelined parallelization scheme (1)

- ❑ As you can see from (14), computing the next element ${x_i}^{(s+1)}$ requires the elements of $x^{(s)}$ approximation with numbers greater than i.
- ❑ For each grid node, its upper and right neighbours in the cross stencil will be taken from the previous approximation, while the left and bottom ones - from the current one.
- ❑ The number of nodes to be computed at each line is equal to $n-1$ for the (n,m) grid. Thus, computing ${x_i}^{(s+1)}$ requires computation of the previous approximation ${x_j}^{(s)}$ elements, $j=\overline{1,i+n-1}$.

Description of a pipelined parallelization scheme (2)

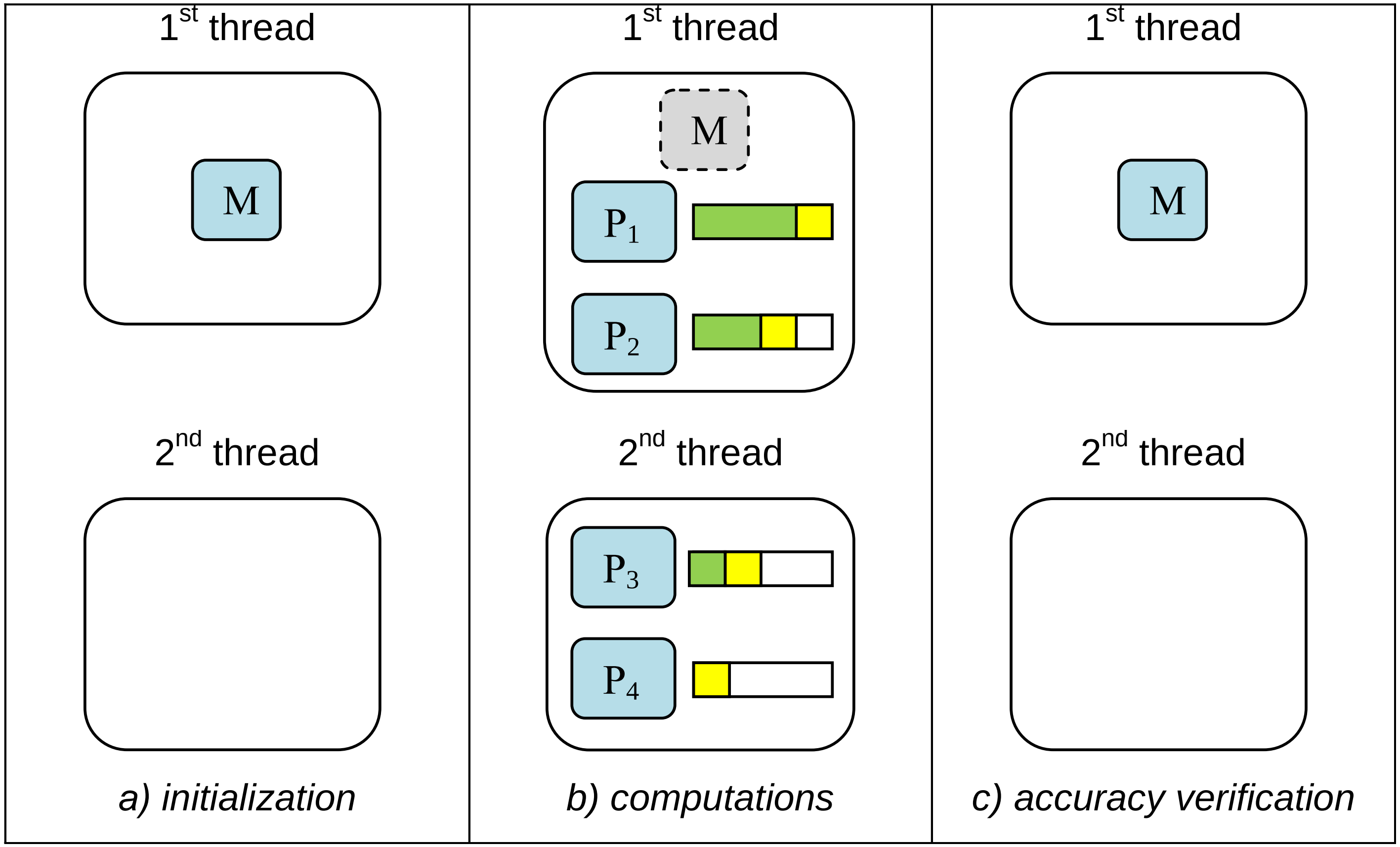
- If for several approximations the previous approximation has at least $n - 1$ more computed elements, further approximations may be computed in parallel and in an in-sync manner with a difference of $n - 1$ element.
- Perform parallel computations based on the Master/Worker scheme.
  - The master will coordinate computation of approximations, distribute load among the Workers and check the method stop criterion.
  - Approximated solutions will be found directly by Workers.

Description of a pipelined parallelization scheme (3)

- The pipelined scheme will be operated iteratively. Each iteration has three stages:
    - The Master will initialize data for the current computation step.
    - Having distributed the load between the Workers, the Master will wait for completion. The Workers will compute a certain number of elements, each for the respective approximation.
    - The Master will verify the method stop criterion.
- The number of workers and the quantity of physical threads may not be the same. To better balance the load, each physical thread must take the load from several Workers.

Description of a pipelined parallelization scheme (4)

- Example of pipelined scheme organization with two physical threads and four workers:



a) initialization *b) computations* *c) accuracy verification*

Description of a pipelined parallelization scheme (5)

- Let **Chunk** stand for a portion of $n - 1$ elements**.** The system dimension is a multiple of **Chunk** and is equal to $(n - 1) * (m - 1)$. Therefore, the approximation vector can be computed stepwise by computing a multiple of **Chunk** elements at a time.
- Use **Portion** to indicate the maximum number of portions of **Chunk** elements to be computed at a time. The last portion number, $m - 1,$ will be **maxChunk**.
- Each Worker has a pointer to the respective current and previous approximations. For the first Worker, the previous approximation will be the one computed by the last Worker.

Description of a pipelined parallelization scheme (6)

- Let there be **numWorkers** Workers. Their computed approximations are stored in the **WorkerResults** array (sized **numWorkers*ResSize**, where **ResSize** is the approximation vector dimension).
- The number of already computed portions of **Chunk** elements are stored in the **PrevPos** array (sized **numWorkers**). The maximum number of portion whose elements are to be found at the current stage, is stored in the **CurrPos** array (sized **numWorkers**).

Pipelined parallelization scheme procedure (1)

- Initialize **WorkerResults** approximations by the initial approximation. For each **i** Worker set **CurrPos[i] = 0, PrevPos[i] = 0.** Set the number of method steps equal to zero.
- Until the required accuracy is obtained:
  1. Determine the number of the current SOR method iteration **currIter.** This is the approximation with the least number whose computation was not completed.
  2. For all Workers, find the boundaries of elements to be computed at the current stage.

Pipelined parallelization scheme procedure (2)

- For worker numbered **k** determining the **currIter** approximation:

```
PrevPos[k] = CurrPos[k];
CurrPos[k] = min(CurrPos[k] + Portion, maxChunk);
```

- For remaining workers except for the one preceding the **k**th one:

```
PrevPos[i] = CurrPos[i];
CurrPos[i] = max(min(CurrPos[i] + Portion,
                 PrevPos[j] - 1), 0);
```

***** From this point on, **j** is the worker computing the previous approximation

Pipelined parallelization scheme procedure (3)

- For worker number **l** directly preceding the worker with the **currIter** approximation:
  - If its approximation has not been computed completely, the worker will compute the **currIter + numWorkers - 1**th approximation. Computation boundary:

```
PrevPos[l] = CurrPos[l];
CurrPos[l] = max(PrevPos[j] – 1, 0);
```

  - Otherwise, worker **l** has finished computing the **currIter - 1**th approximation and can proceed to computation of the **currIter + numWorkers**th approximation. Computation boundary:

```
PrevPos[l] = 0;
CurrPos[l] = min(Portion, PrevPos[j] - 1);
```

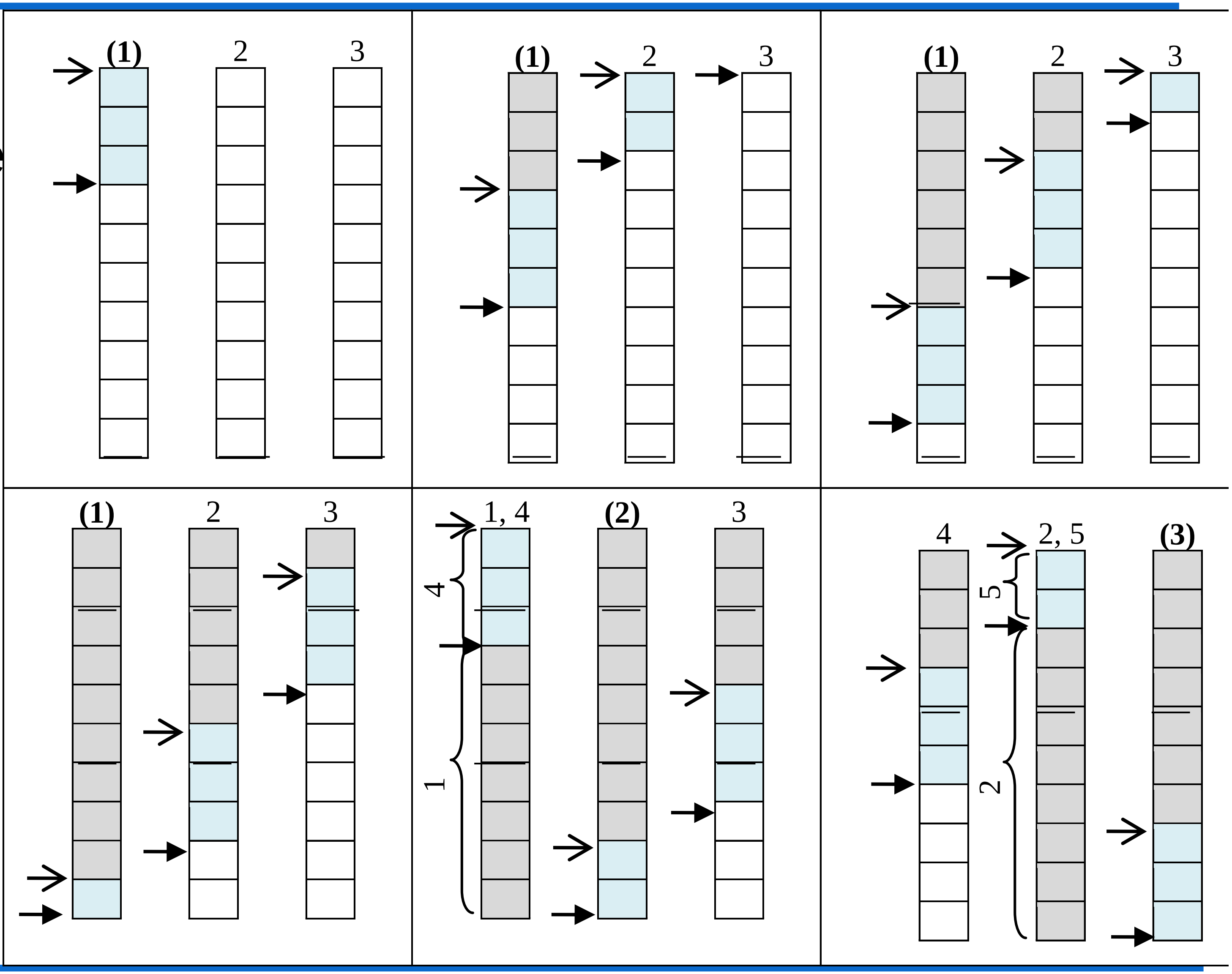
Pipelined parallelization scheme procedure (4)

3. In parallel, run computation of elements for each **i** Worker from **Chunk*PrevPos[i]** to **Chunk*CurrPos[i] – 1**, wait for completion.
4. If the current approximation has been computed completely, check the stop criterion. If the required accuracy has been attained, save the result and complete the work.
5. Initialize the new computation stage, go to step 1.

Pipelined parallelization scheme procedure (5)

Example of a pipelined scheme operation.

if $n = m = 11$, Portion = 3, numWorkers = 3.



Pipelined parallelization scheme procedure (6)

- ❑ The scheme restricts the correlation between the single portion Portion, grid size and the number of worker threads.
- ❑ Correctness of the pipelined scheme run will be guaranteed if:

$$\textbf{numWorkers*Portion} < \textbf{n} - \textbf{1} - \textbf{Portion} \qquad (21)$$

- ❑ For certain grids,one can select a parameter correlation so that the pipelined scheme will operate under a less strict condition:

$$\textbf{numWorkers*Portion} < \textbf{n} - \textbf{1} \qquad (22)$$

Software implementation

- ❑ Implement the described pipelined scheme using the Intel® TBB task mechanism.
- ❑ Let us suppose that the master and worker functionalities are implemented as separate classes of problems.
- ❑ Declare the classes of TBB-problems in **TaskImplementation.h** having mapped the required header file. Set the pipelined scheme parameters in the same file.

```
#include "tbb\task.h"
#define THREADS_PER_WORKER 2 //number of workers per thread
#define FIRST_PORTION 2       //first portion of computations
```

- ❑ Implement the **execute()** methods of the declared classes in **TaskImplementation.cpp.**

Class ensuring functionality of the worker (1)

```
// tbb class - worker problem
// computations at a certain method iteration
class OverRelaxWorker : public tbb::task {
public:
  double* PrevResult;  //approximation at the previous iteration
  double* CurrResult;  //approximation at the current iteration
  static  double* tMatrix;    //matrix
  static  int* tIndex;        //diagonal offset index
  static  double* tVector;    //right-hand vector
  static  double tWParam;     //method parameter
  static  int tSize;          //system dimension
  static  int tBandWidth;     //band width
  int start;    // initial position of the portion of computations
  int finish;      //final portion of computations
  double CurrError;//solution accuracy at the current approximation

  tbb::task* execute();
};
```

Class ensuring functionality of the master (2)

```
tbb::task* OverRelaxWorker::execute() {
  int ii, j;
  int index = tIndex[tBandWidth - 1], bandHalf = (tBandWidth - 1)/2;
  double sum, TempError;

  // computing the new approximation elements [start, finish)
  for (int i = index + start; i < index + finish; i++) {
    ii = i - index;
    sum = 0.0;
    for (j = 0; j < bandHalf; j++)
      sum += tMatrix[ii*tBandWidth + j] * CurrResult[i + tIndex[j]];
    for (j = bandHalf; j < tBandWidth; j++)
      sum += tMatrix[ii*tBandWidth + j] * PrevResult[i + tIndex[j]];
    CurrResult[i] = (tVector[ii] - sum) * tWParam /
                     tMatrix[ii*tBandWidth + bandHalf] + PrevResult[i];
    TempError = fabs(CurrResult[i] - PrevResult[i]);
    CurrError = max(CurrError, TempError);
  }
  return NULL;
}
```

Class ensuring functionality of the master (1)

```
// tbb class - master problem
// organizing parallel computations, method start and stop
class OverRelaxMaster : public tbb::task {
public:
  static  double* tDecision;  //obtained solution
  static  double tORAccuracy; //attainable accuracy
  static  double tAccuracy;   //set accuracy of solution
  static  int tSize;          //system dimension
  static  int tResSize;       //solution vector dimension
  static  int NumSteps;       //number of performed iterations
  static  int MaxNumSteps;    //maximum number of iterations
  static  int numWorkers;     //number of worker problems
  static  int Chunk;    //dimension of a single portion of computations

public:
  tbb::task* execute();
};
```

Class ensuring functionality of the master (2)

```
#ifndef min
#define min(a, b) ((a)<(b))?(a):(b)
#endif
#ifndef max
#define max(a, b) ((a)>(b))?(a):(b)
#endif

tbb::task* OverRelaxMaster::execute() {
  OverRelaxWorker **Workers;  //worker threads
  tbb::task_list tasks;       //list of generated problems
  double** workerResults;     //approximations found by workers
  double CurrError = -1.0;    //current method accuracy
  int *prevPos; //number of elements computed at the previous stage
  int *currPos; //number of elements computed at the current stage
  double *currErrorBuff;      //current error of workers
  int portion = FIRST_PORTION;//first portion of computations
  int currIter; //current iteration number
  // the number of maximum portion of computations (number of layers)
  int maxPortion = tSize / Chunk;
  // continued in the following slide
```

Class ensuring functionality of the master (3)

```
// auxiliary variables
int i;
int refCount;

//1. Memory initialization for operation:
// 1.1 - auxiliary arrays
InitializeVector(&currPos, numWorkers);
InitializeVector(&prevPos, numWorkers);
InitializeVector(&currErrorBuff, numWorkers);

// 1.2 - workers and the array of corresponding approximations
Workers       = new OverRelaxWorker* [numWorkers];
workerResults = new  double* [numWorkers];
for(i = 0; i < numWorkers; i++) {
  Workers[i] = new(tbb::task::allocate_child()) OverRelaxWorker;
  InitializeVector(&(workerResults[i]), tResSize);
  memset(workerResults[i], 0, sizeof(double) * tResSize);
}

// continued in the following slide
```

Class ensuring functionality of the master (4)

```
// 1.3 - initialization of approximations for the worker threads
for(i = 0; i < numWorkers; i++) {
  currPos[i] = 0;
  prevPos[i] = 0;
  currErrorBuff[i] = -1.0;
  Workers[i]->CurrResult = workerResults[i];
  Workers[i]->PrevResult = workerResults[(i+ numWorkers - 1) %
                                                   numWorkers];
}

//2. SOR method launch
NumSteps = 0;
while(true) {
  // 2.1 - determining the current approximation number
  currIter = NumSteps % numWorkers;
  // 2.2 - determining the boundary of the new portion of computations
  // for the current approximation
  prevPos[currIter] = currPos[currIter];
  currPos[currIter] = min(currPos[currIter] + portion, maxPortion);
  // continued in the following slide
```

Class ensuring functionality of the master (5)

```
    Workers [currIter]->start  = prevPos[currIter] * Chunk;
    Workers [currIter]->finish = currPos[currIter] * Chunk;

    // recording the new problem in the list of problems
    refCount = 1;
    tasks.push_back(*(Workers[currIter]));
    refCount ++;

    // 2.3 - determining the boundary of the new portion of computations
for the following
    // computations, placing the workers in the list of problems
    for(i = 1; i < numWorkers - 1; i++) {
      prevPos[(currIter + i) % numWorkers] =
                      currPos[(currIter + i) % numWorkers];
      // If the previous approximation resulted in less that 1 portion
(layer)
      // the current position will be 0. Else - next portion
      currPos[(currIter + i) % numWorkers] = max(
        min(currPos[(currIter + i) % numWorkers] + portion,
            prevPos[(currIter + i - 1) % numWorkers] - 1), 0);
         // continued in the following slide
```

Class ensuring functionality of the master (6)

```
      Workers [(currIter + i) % numWorkers]->start  =
              prevPos[(currIter + i) % numWorkers] * Chunk;
      Workers [(currIter + i) % numWorkers]->finish =
              currPos[(currIter + i) % numWorkers] * Chunk;
      tasks.push_back(*(Workers[(currIter + i) % numWorkers]));
      refCount ++;
    }

    // 2.4 - determining the boundary of the new portion of
computations for the last
    // worker (the previous one as regards the current approximation)
    // a) if the approximation vector has not been computed completely
    if(currPos[(currIter + i) % numWorkers] != maxPortion) {
      prevPos[(currIter + i) % numWorkers] =
                       currPos[(currIter + i) % numWorkers];
      currPos[(currIter + i) % numWorkers] =
        max(prevPos[(currIter + i - 1) % numWorkers] - 1, 0);
    }

    // continued in the following slide
```

Class ensuring functionality of the master (7)

```
// b) if the approximation vector has been computed completely
else {
  prevPos[(currIter + i) % numWorkers] = 0;
  currPos[(currIter + i) % numWorkers] = min(portion,
         prevPos[(currIter + i - 1) % numWorkers] - 1);
  Workers [(currIter + i) % numWorkers]->CurrError = -1.0;
}
Workers [(currIter + i) % numWorkers]->start  =
          prevPos[(currIter + i) % numWorkers] * Chunk;
Workers [(currIter + i) % numWorkers]->finish =
          currPos[(currIter + i) % numWorkers] * Chunk;
tasks.push_back(*(Workers[(currIter + i) % numWorkers]));
refCount ++;

// 2.5 - Placing the problems in the pool and solving them
set_ref_count(refCount);
spawn_and_wait_for_all(tasks);

// continued in the following slide
```

Class ensuring functionality of the master (8)

```
// 2.6 - If the current approximation has been computed completely,
//       check the stop criterion
if(currPos[currIter] == maxPortion) {
  CurrError = Workers[currIter]->CurrError;
  Workers[currIter]->CurrError = -1.0;
  currErrorBuff[currIter] = -1.0;
  // if the solution is found, store it in tDecision
  if ((CurrError < tAccuracy) || (NumSteps > MaxNumSteps)) {
    tORAccuracy = CurrError;
    this->tDecision = Workers[currIter]->CurrResult;
    NumSteps++;
    return NULL;
  }
  NumSteps++;
}
// 2.7 - Identification of new problems
// a) remember current error of each approximation
for(i = 0; i < numWorkers; i++)
  currErrorBuff[i] = Workers[i]->CurrError;
// continued in the following slide
```

Class ensuring functionality of the master (9)

```
  // b) identify new problems
  delete [] Workers;
  Workers = new OverRelaxWorker * [numWorkers];
  for(i = 0; i < numWorkers; i++)
    Workers[i] = new(tbb::task::allocate_child())  OverRelaxWorker;
  // c) initialize approximations for the problems
  for(i = 0; i < numWorkers; i++) {
    Workers[i]->CurrError = currErrorBuff[i];
    Workers[i]->CurrResult = workerResults[i];
    Workers[i]->PrevResult =
         workerResults[(i + numWorkers - 1) % numWorkers];
  }
}

//3. Memory release
FreeVector(&currPos);
FreeVector(&prevPos);
FreeVector(&currErrorBuff);

// continued in the following slide
```

Class ensuring functionality of the master (10)

```
  for (i = 0; i < numWorkers; i++)
    if (workerResults[i] != tDecision)
      FreeVector(&workerResults[i]);
  delete [] workerResults;

  return NULL;
}
```

BandOverRelaxation() function modification (1)

- Modify the **BandOverRelaxation()** function and name it **BandOverRelaxationTBB()** .
- Before the **BandOverRelaxationTBB()** function, declare the static member variables of **OverRelaxWorker** and **OverRelaxMaster** classes.

BandOverRelaxation() function modification (2)

```
double BandOverRelaxationTBB(...) {
  // OverRelaxMaster static member variables
  OverRelaxMaster::tAccuracy = Accuracy;
  OverRelaxMaster::tSize = size;
  OverRelaxMaster::tResSize = size + 2*Index[bandWidth - 1];
  OverRelaxMaster::MaxNumSteps = N_MAX;
  OverRelaxMaster::tORAccuracy = -1.0;
  OverRelaxMaster::NumSteps = 0;
  OverRelaxMaster::Chunk = n - 1;
  OverRelaxMaster::numWorkers = THREADS_PER_WORKER*NumThreads;

  // OverRelaxWorker static member variables
  OverRelaxWorker::tMatrix = Matrix;
  OverRelaxWorker::tVector = Vector;
  OverRelaxWorker::tWParam = WParam;
  OverRelaxWorker::tIndex = Index;
  OverRelaxWorker::tSize = size;
  OverRelaxWorker::tBandWidth = bandWidth;

  // continued in the following slide
```

BandOverRelaxation() function modification (3)

```
  // SOR method start
  OverRelaxMaster& FirstIter =
       *new (tbb::task::allocate_root()) OverRelaxMaster();
  tbb::task::spawn_root_and_wait(FirstIter);
  (*Result) = OverRelaxMaster::tDecision;
  StepCount = OverRelaxMaster::NumSteps;

  return OverRelaxMaster::tORAccuracy;
}
```

Project compilation and application run

- Modify the program code as required.
- Having developed the software implementation, build the project by executing **Build→Rebuild 03_BandOR_cilk** and check the application for consistent running.

Scalability analysis (1)

- To analyze the pipelined scheme efficiency, perform experiments using functions based on formulas (3) – (7) with the accuracy of $\varepsilon = 10^{-5}$.

- See the next slide for the results of running a multiple-thread pipelined scheme. *T* is the runtime (in seconds), *S* is the acceleration as compared to single thread operation.

Scalability analysis (2)

System dimension	1 thread	2 threads		4 threads		6 threads		8 threads	
	T	*T*	*S*	*T*	*S*	*T*	*S*	*T*	*S*
100	0,67	0,35	1,90	0,21	3,20	0,17	3,94	0,14	4,65
200	4,98	2,57	1,94	1,45	3,43	1,08	4,59	0,90	5,52
300	16,03	8,25	1,94	4,60	3,48	3,34	4,80	2,71	5,91
400	37,87	19,61	1,93	10,50	3,61	7,46	5,08	6,01	6,31
500	73,47	36,98	1,99	19,75	3,72	13,91	5,28	11,17	6,58
600	123,42	62,83	1,96	32,96	3,74	23,10	5,34	18,53	6,66
700	190,60	96,76	1,97	51,26	3,72	35,63	5,35	28,56	6,67
800	280,10	141,79	1,98	74,73	3,75	51,91	5,40	41,88	6,69
900	390,15	199,02	1,96	104,32	3,74	72,27	5,40	58,73	6,64
1000	529,07	265,81	1,99	141,18	3,75	97,46	5,43	80,50	6,57

Scalability analysis (4)

- The maximum acceleration of 6.7 was obtained for 8 threads when $n = 800$. If the grid size $n > 300$ acceleration will exceed 6.
- Acceleration growth depending on the number of threads and grid size indicates good scalability of the proposed pipelined scheme.
- The pipelined scheme requires the same number of the SOR method iterations as the consecutive version.
- Additional acceleration may be ensured by fitting scheme parameters to a specific problem.

Test questions (1)

- Deduce a linear system resulting from grid approximation of the heat transfer equation. What structural peculiarities does this matrix have?
- Give the canonical SOR method form and approximation component computation formula.
- Substantiate the SOR method convergence. Demonstrate dependence of the convergence rate on the method parameters selection.
- What band matrix storage formats do you know? When is each of them used?

Test questions (2)

- Why is the number of the SOR method iterations different in case of a multi-thread Intel® Cilk Plus-based parallel implementation?
- Substantiate the necessary restrictions to the pipelined parallelization scheme parameters. What changes must be introduced to the system to relax these restrictions?

Added tasks (1)

- Prove that the block five-diagonal matrix used for solving linear systems in the course of this laboratory work is negative definite.
- Show that if the matrix A is a negative definite, the matrix $(-A)$ will be positive definite.
- Show that the eigenvalues of the matrix in question (part of the linear system) are computed using formulas (17) and (18). It is assumed that $h = k$.

Added tasks (2)

- Implement the Jacobi method as applied to a block five-diagonal matrix mentioned in this laboratory work. Think about a possible parallelization scheme.
- Implement the Seidel method as applied to a block five-diagonal matrix mentioned in this laboratory work. Think about a possible parallelization scheme.
- Conduct a computational experiment having found the best pipelined scheme parameter values using Intel® TBB for test grid dimensions.

Questions

- ❑ ???