Simulation and Scientific Computing

(Simulation und Wissenschaftliches Rechnen - SiWiR)

Winter Term 2014/15

Florian Schornbaum
Chair for System Simulation

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Assignment 2: OpenMP-Parallel Red-Black Gauss-Seidel Method

November 3, 2014 – November 25, 2014

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

The Red-Black Gauss-Seidel Method

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Discretizing Elliptic PDEs

- Elliptic partial differential equation:

$$-\Delta u(x,y) + k^2 u(x,y) = f(x,y)$$

$$-\left(\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2}\right) + k^2 u(x,y) = f(x,y)$$

- Discretization using the **differential quotient** for $\Delta u(x,y)$:

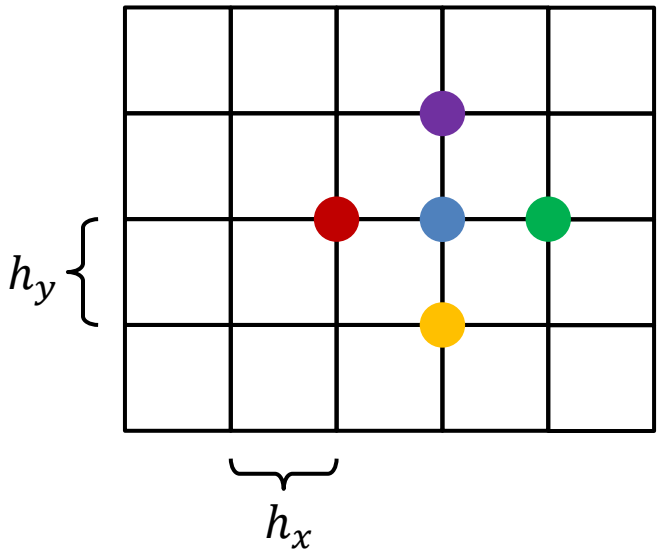
$$-\left(\frac{u(x-h_x,y) - 2u(x,y) + u(x+h_x,y)}{h_x^2} + \frac{u(x,y-h_y) - 2u(x,y) + u(x,y+h_y)}{h_y^2}\right) + k^2 u(x,y) = f(x,y)$$

Discretizing Elliptic PDEs

- Discretization using the differential quotient for $\Delta u(x,y)$:

$$-\frac{1}{h_x^2}\left[u(x-h_x,y)+u(x+h_x,y)\right]-\frac{1}{h_y^2}\left[u(x,y-h_y)+u(x,y+h_y)\right]$$
$$+\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}+k^2\right)u(x,y)=f(x,y)$$

- We are solving the given PDE on a **discretized domain** Ω:

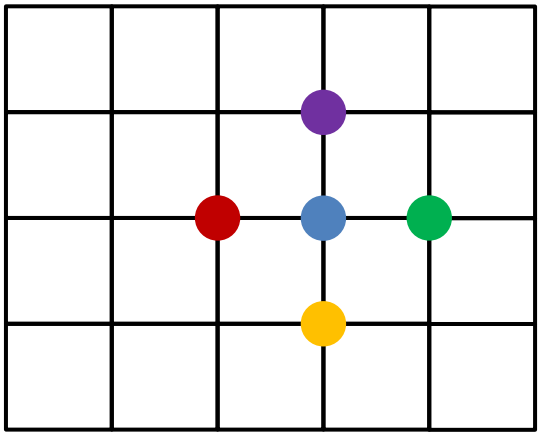


$u(x,y)$ / $f(x,y)$
$u(x-h_x,y)$
$u(x+h_x,y)$
$u(x,y-h_y)$
$u(x,y+h_y)$

Discretizing Elliptic PDEs

- For every point $u_{x,y}$ on the grid, we can formulate a linear equation:

$$-\frac{1}{h_x^2}\left[u_{x-1,y}+u_{x+1,y}\right]-\frac{1}{h_y^2}\left[u_{x,y-1}+u_{x,y+1}\right]+\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}+k^2\right)u_{x,y}=f_{x,y}$$



$u_{x,y}$ / $f_{x,y}$

$u_{x-1,y}$

$u_{x+1,y}$

$u_{x,y-1}$

$u_{x,y+1}$

- Formulating this equation for every point of the grid leads to a linear system of equations (LSE): $A\vec{u}=\vec{f}$

Jacobi Method

- The Jacobi method **iteratively** solves the LSE (k represents the number of the iteration) according to the following formula:

$$u_i^{k+1} = \frac{1}{a_{ii}} \left(f_i - \sum_{j \neq i} a_{ij} u_j^k \right)$$

  ⇒ This corresponds to solving the i-th equation of the LSE using the unknowns from the **previous** iteration.

- Since there are **no data dependencies**, every u_i^{k+1} can be computed in parallel:

```
#pragma omp parallel for
for( int i = 0; i < n; ++i )
   u[i] = ( f[i] - a_west  * u_previous[i-1] - a_east  * u_previous[i+1]
                 - a_south * u_previous[i-m] - a_north * u_previous[i+m]
          ) / a_center;
```

Gauss-Seidel Method

- Solving the i-th equation using the Gauss-Seidel method:

$$u_i^{k+1} = \frac{1}{a_{ii}}\left(f_i - \underbrace{\sum_{j<i} a_{ij} u_j^{k+1}}_{\text{same iteration}} - \underbrace{\sum_{j>i} a_{ij} u_j^{k}}_{\text{previous iteration}}\right)$$

  ⇒ Some unknowns come from the previous iteration, some have just been computed in the same iteration $k+1$.

- Generally **better convergence** compared to the Jacobi method
- There are definitely **data dependencies**: Updating u_i requires some other u_j to be already computed.

Solving Elliptic PDEs

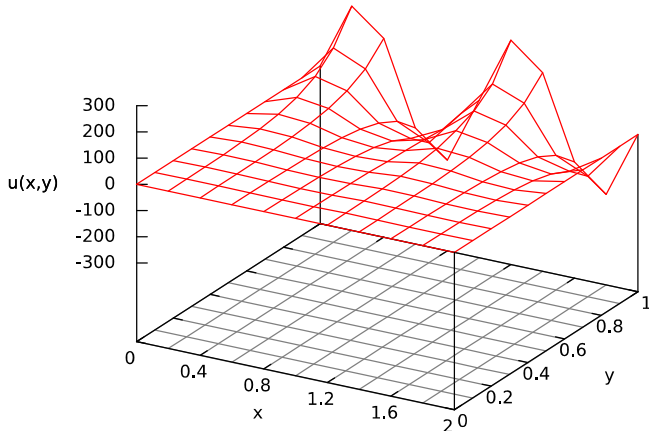
- In order to solve the elliptic partial differential equation …

$$-\Delta u(x,y) + k^2 u(x,y) = f(x,y)$$

… no Matrix A must be assembled.

- We just need to know the "stencil":

$$\begin{bmatrix} & -\frac{1}{h_y^2} & \\ -\frac{1}{h_x^2} & \frac{2}{h_x^2}+\frac{2}{h_y^2}+k^2 & -\frac{1}{h_x^2} \\ & -\frac{1}{h_y^2} & \end{bmatrix}$$



A solution of the PDE

- Every grid point $u_{x,y}$ can be computed as follows:

$$u_{x,y} = \frac{1}{\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}+k^2\right)}\left(f_{x,y} + \frac{1}{h_x^2}\left[u_{x-1,y} + u_{x+1,y}\right] + \frac{1}{h_y^2}\left[u_{x,y-1} + u_{x,y+1}\right]\right)$$

Solving Elliptic PDEs

- Every grid point $u_{x,y}$ is computed as follows (over and over again, iteration for iteration):

$$u_{x,y} = \frac{1}{\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}+k^2\right)}\left(f_{x,y} + \frac{1}{h_x^2}\left[u_{x-1,y} + u_{x+1,y}\right] + \frac{1}{h_y^2}\left[u_{x,y-1} + u_{x,y+1}\right]\right)$$

  $\Rightarrow$ stencil $\approx$ update rule

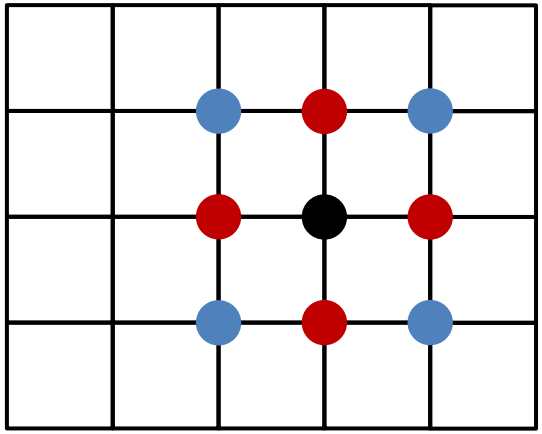
- Difference between Jacobi and Gauss-Seidel:
  - Jacobi: uses **two grids**, one for storing the values of the previous iteration, one for storing the values of the current iteration
  - Gauss-Seidel: uses only **one grid**, calculates new values "**in place**" (thereby reading some old and some already updated, new values)

Red-Black Gauss-Seidel Method

- Data dependencies (take a look at the “update rule”/stencil):

$$u_{x,y} = \frac{1}{\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}+k^2\right)}\left(f_{x,y} + \frac{1}{h_x^2}\left[u_{x-1,y} + u_{x+1,y}\right] + \frac{1}{h_y^2}\left[u_{x,y-1} + u_{x,y+1}\right]\right)$$

$$\begin{bmatrix} & -\frac{1}{h_y^2} & \\ -\frac{1}{h_x^2} & \frac{2}{h_x^2}+\frac{2}{h_y^2}+k^2 & -\frac{1}{h_x^2} \\ & -\frac{1}{h_y^2} & \end{bmatrix}$$



North (N)
↕
South (S)

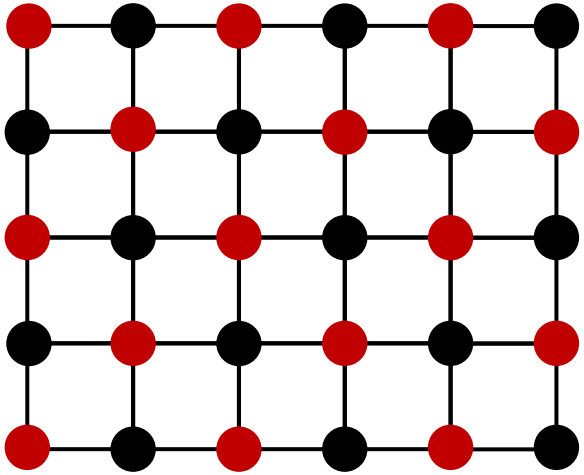
West (W) ↔ East (E)

no data dependencies in NW, NE, SW, and SE direction

data dependencies only in W, E, N, and S direction

Red-Black Gauss-Seidel Method

- Parallelization → **checkerboard reordering**:



$$\begin{bmatrix} & NORTH & \\ WEST & CENTER & EAST \\ & SOUTH & \end{bmatrix}$$

A general stencil for the PDE

⇒ Idea: Partition the unknowns into two groups (red and black) so that there are no data dependencies within each group.

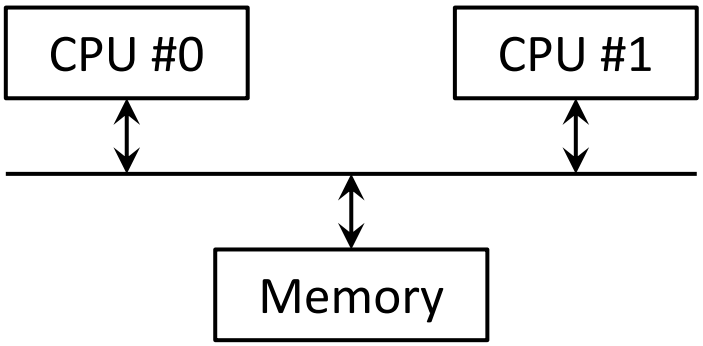
⇒ Consequence: The unknowns within each group can be updated independently, i.e. in parallel!

Improving Cache Efficiency

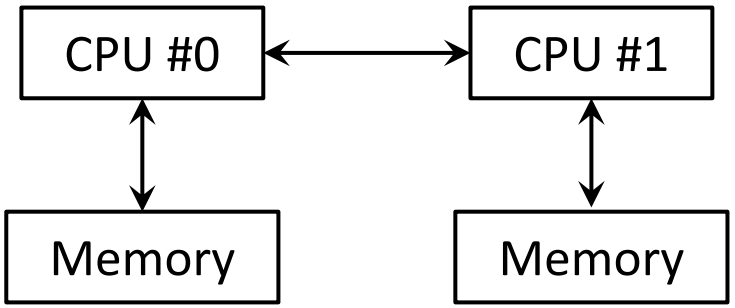
- If the red unknowns are updated, only the data of the right-hand side (RHS, $f(x, y)$) that corresponds to those red points is read.
- Splitting the data of the RHS into two separate chunks of memory **improves spatial locality**!
- While the red unknowns are updated, only black unknowns are read, and the red unknowns are only written to memory.
- Many processors have instructions to directly write to memory without first loading the data into the cache: **non-temporal writes**.
- Separating red and black unknowns (just like splitting the data of the RHS) allows for non-temporal writes.
- When splitting red and black data, don't mix up the indices!

Shared Memory Access

- Differences between **SMP** and **NUMA**:



conventional symmetric multiprocessing (SMP)



non-uniform memory access (NUMA)

- Memory pages are assigned to the processor that **first touches** the memory – which can be a problem on NUMA systems!
- To avoid non-local memory access, initialize the data by the thread that will later access it and prevent migration of threads to different processors by explicitly **binding/pinning** them.

Finite Differences

- Additional material on finite differences by Pascal Frey:

  http://www.ann.jussieu.fr/~frey/cours/UPMC/finite-differences.pdf

  ⇒ get it and read it!

- Another well written article on solving linear systems by the same author which might also be worth reading:

  http://www.ann.jussieu.fr/~frey/cours/UPMC/linear%20systems.pdf

OpenMP

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

The OpenMP Programming Model

- OpenMP …
  - … is an **API** that can be used with FORTRAN, C, and C++.
  - … is a **standard programming model** for shared address space machines.
  - … allows incremental parallelization.
  - … provides support for concurrency, synchronization, and data handling.
  - … obviates the need for explicitly setting up mutexes and condition variables.

- The workload is distributed among threads:
  - Variables can be …
    - … shared among all threads or …
    - … duplicated for each thread.
  - **Threads communicate by sharing variables**.

Runtime Library Functions

- Setting the total number of threads:
  - At runtime via the function “omp_set_num_threads”:

```
#include <omp.h>
void omp_set_num_threads( int num_threads )
```

  - Via the environment variable OMP_NUM_THREADS

```
export OMP_NUM_THREADS=4
```
(if you are using bash)

- Getting the total number of threads:

```
#include <omp.h>
int omp_get_num_threads( void )
```

- Getting the ID of the thread that executes this command:

```
#include <omp.h>
int omp_get_thread_num( void )
```

The OpenMP Programming Model

- OpenMP directives in C and C++ are based on **`#pragma`** compiler directives:

```
#pragma omp directive [clause list]
```

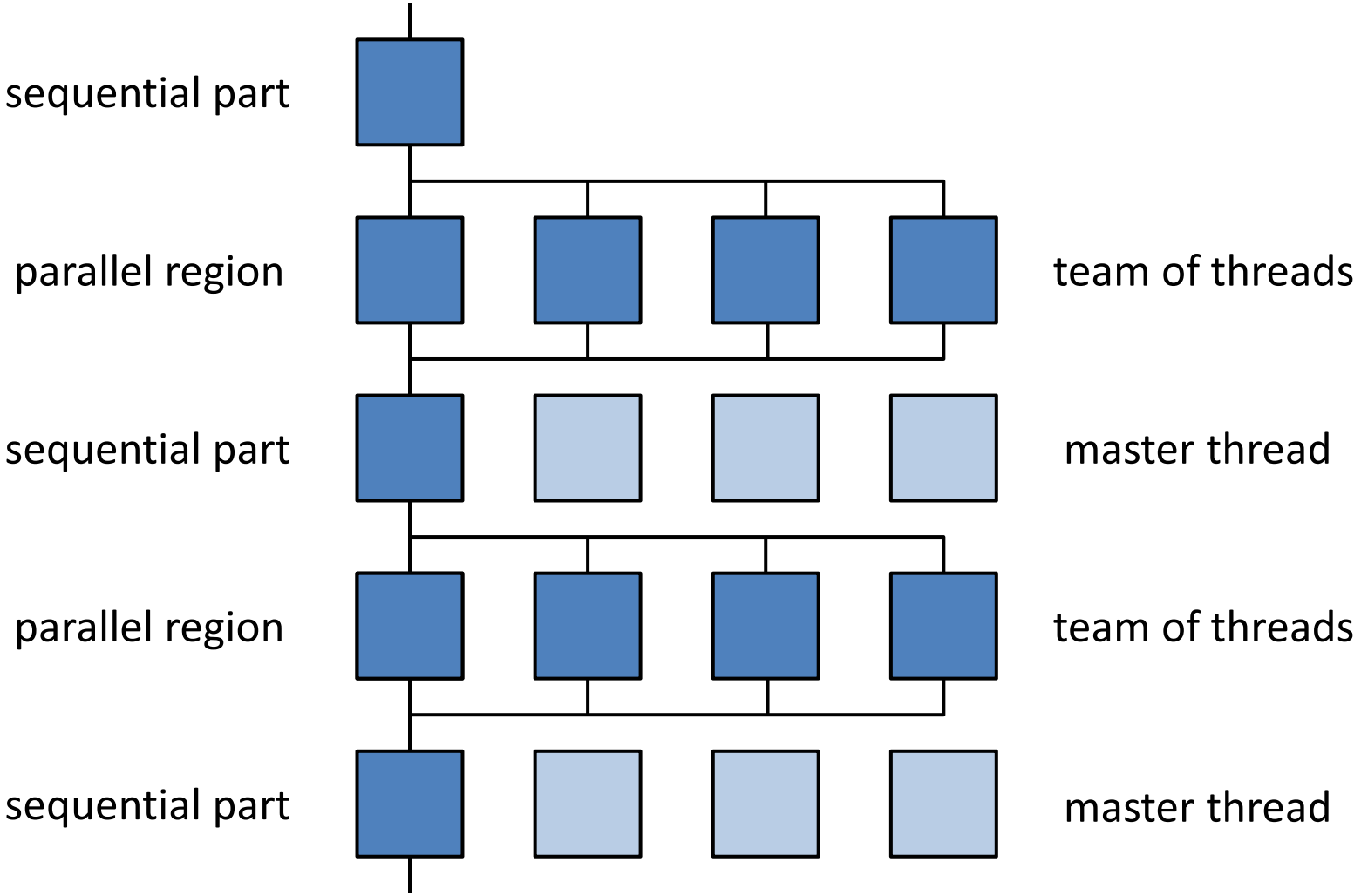
- OpenMP programs execute serially until they encounter the **`parallel`** directive:

```
#pragma omp parallel [clause list]
/* structured block */
```

- Each thread executes the structured block specified by the **`parallel`** directive.

- The **`clause list`** specifies the conditional parallelization, the number of threads, and data handling.

The OpenMP Execution Model



OpenMP Example


```
#include <omp.h>
#include <iostream>

int main ()
{
   /* Fork a team of threads */
   #pragma omp parallel
   {
      /* Obtain and print the thread id */
      int tid = omp_get_thread_num();
      std::cout << "Hello world from thread " << tid << std::endl;

      /* Only the master thread does this */
      if( tid == 0 ) {
         int nthreads = omp_get_num_threads();
         std::cout << "Number of threads = " << nthreads << std::endl;
      }

   } /* All threads join the master thread and terminate */

   return 0;
}
```

The OpenMP Parallel Directive

- The clause **`if( scalar expression )`** determines whether the parallel construct results in the creation of threads (only one if clause can be used with a parallel directive).

```
#pragma omp parallel if( is_parallel == 1 )
/* structured block */
```

- The clause **`num_threads( integer expression )`** specifies the number of threads that are created by the parallel directive.

```
#pragma omp parallel num_threads( 8 )
/* structured block */
```

Data Scope Attribute Clauses

- **`private( variable list )`**: indicates that the set of variables is local to each thread – i.e., each thread gets its own copy of each variable in the list. Variables declared **`private`** should be assumed to be uninitialized for each thread.

- **`firstprivate( variable list )`**: same as **`private`** , however, all variables are initialized with copies of the values of the original variables.

- **`shared( variable list )`**: indicates that all variables in the list are shared across all the threads.
  - Shared variables exist in only one memory location – all threads read and write to that address (that's how threads can communicate).
  - It's your responsibility that multiple threads properly access these variables.

OpenMP Example


```
// The parallel section is executed by 8 threads
// depending on the value of 'is_parallel'. Each of
// the threads gets a local copy of a and c, and
// shares the value of b. Furthermore, the value of
// each copy of c is initialized to the value of c
// before the parallel directive.

#pragma omp parallel if( is_parallel == 1 ) \
                     num_threads( 8 ) private( a ) \
                     shared( b ) firstprivate( c )
{
   // parallel section
}
```

Data Reduction

- **`reduction`**: specifies how multiple "local copies" of a shared variable are combined into a single copy for the master thread when the parallel section exits.
  - possible operators: +, -, *, &, |, ^, &&, and ||.

```
double sum = 0.0;
#pragma omp parallel shared( sum ) reduction(+: sum) \
                    num_threads( 8 )
{
   // parallel section, executed by 8 threads
   /* thread local sum is computed here */
}
/* here, sum contains the sum of all local instances of sum */
```

Work-Sharing Constructs

- Work-sharing constructs …
  - … must be enclosed within a parallel region.
  - … divide the execution of the enclosed code region among all currently existing threads.
  - … do not launch new threads.
  - … have no implicit barrier on entry.

- Work-sharing constructs are:
  - The **`for`** directive
  - The **`sections`** directive

The `for` Directive

- The **`for`** directive specifies that the iterations of the loop immediately following it must be executed in parallel by all currently existing threads.
- The general form looks like as follows:

```
#pragma omp for [clause list]
/* for loop */
```

- The **`for`** directive can be fused with the **`parallel`** directive:

```
#pragma omp parallel for [clause list]
/* for loop */
```

- Possible clauses: **`private`**, **`firstprivate`**, **`shared`**, **`reduction`** (The loop control variable is private by default)

The `for` Directive - Example


```
int a[20];
int b[20];
int f = 23;

/* b is initialized */

#pragma omp parallel for schedule( static )
for( int i = 0; i < 20; ++i )
   a[i] = b[i] + f*(i+1)
```

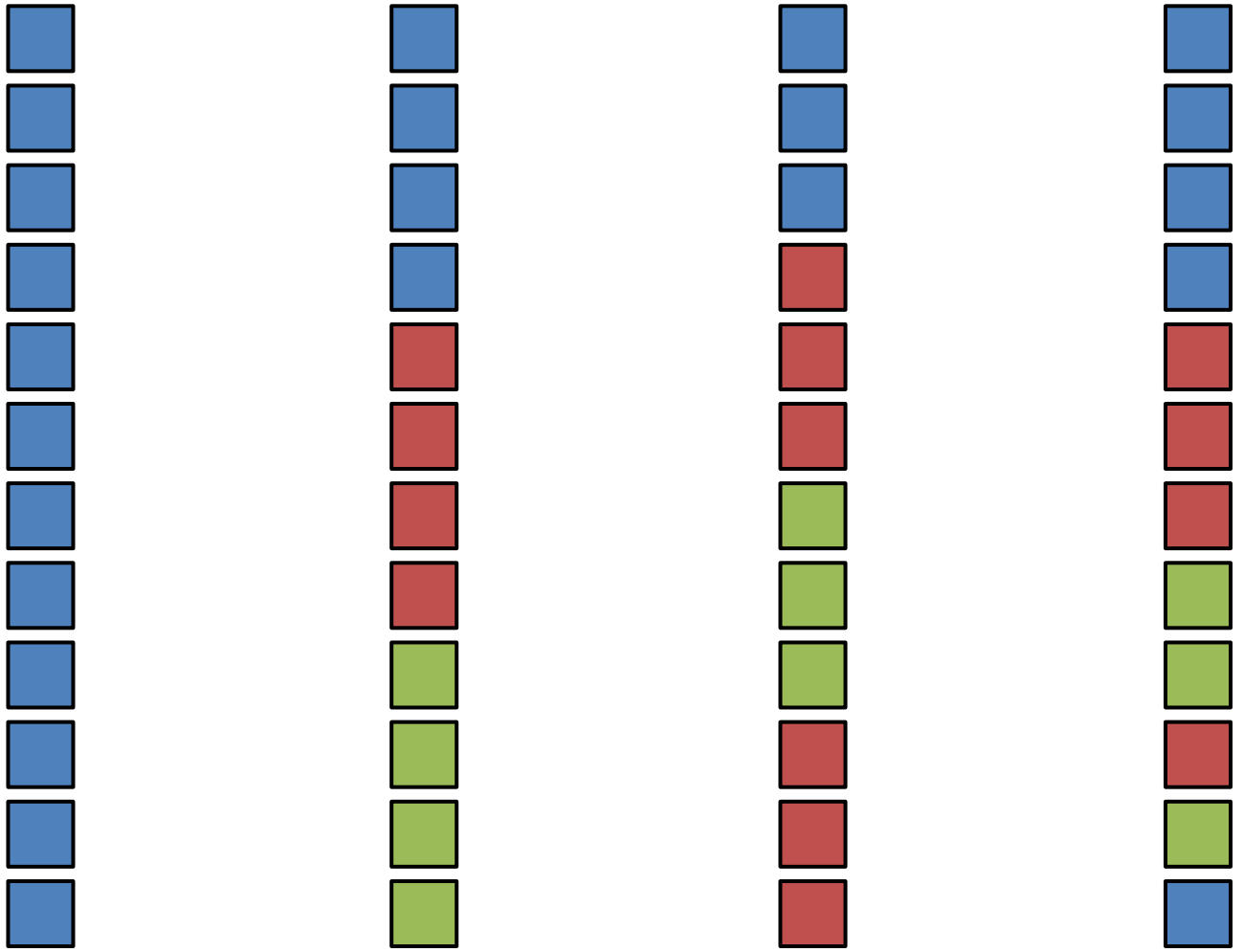
If this program is compiled, **`OMP_NUM_THREADS`** is set to 4, and then the program is executed, each thread calculates exactly 5 values in the array **`a`**. The first thread computes **`a[0]`** to **`a[4]`**, the second thread **`a[5]`** to **`a[9]`**, …

The `for` Directive - Scheduling

- **`schedule( scheduling_class[, optional parameters] ):`**
  - **`static[,chunk_size]`**: Loop iterations are divided into pieces of size chunk and then assigned to all threads in a round robin fashion. If no chunk size is specified, the iterations are evenly divided among all threads.
  - **`dynamic[,chunk_size]`**: Loop iterations are divided into pieces of size chunk and assigned to threads as they become idle (= dynamic scheduling). The default chunk size is 1.
  - **`guided[,chunk_size]`**: Similar to dynamic except that the block size decreases each time work is given to a thread. Initial block size is equal to $\frac{\#iterations}{\#threads}$, subsequent blocks are proportional to $\frac{\#remaining\_iterations}{\#threads}$. The chunk size defines the minimum block size (default is 1).
  - **`runtime`**: strategy determined by environment variable **`OMP_SCHEDULE`**
- Next slide: example with 12 iterations and 3 threads

The `for` Directive - Scheduling

w/o OpenMP | static | dynamic, 3 | guided, 1



The `for` Directive

- There is an implicit barrier at the end of the loop – specify **`nowait`** in order omit this barrier.
- It is illegal to branch/**`break`** out of a parallel for loop.
- The loop control variable must be …
  - … a signed integer.
  - … an unsigned integer (OpenMP 3.0)
  - … a pointer type (OpenMP 3.0)
  - … a random access iterator (C++ only, OpenMP 3.0)
- The number of iterations must be computable at loop entry.
- The logical expression must be one of <, <=, >, >=.
- The increment expression must have integer increments or decrements only.

The `sections` Directive

- The **`sections`** directive is used for non-iterative parallel task assignment. In general it looks like as follows:

```
#pragma omp sections [clause list]
{
   #pragma omp section
   {
      /* first section */
   }
   #pragma omp section
   {
      /* second section */
   }
   …
}
```

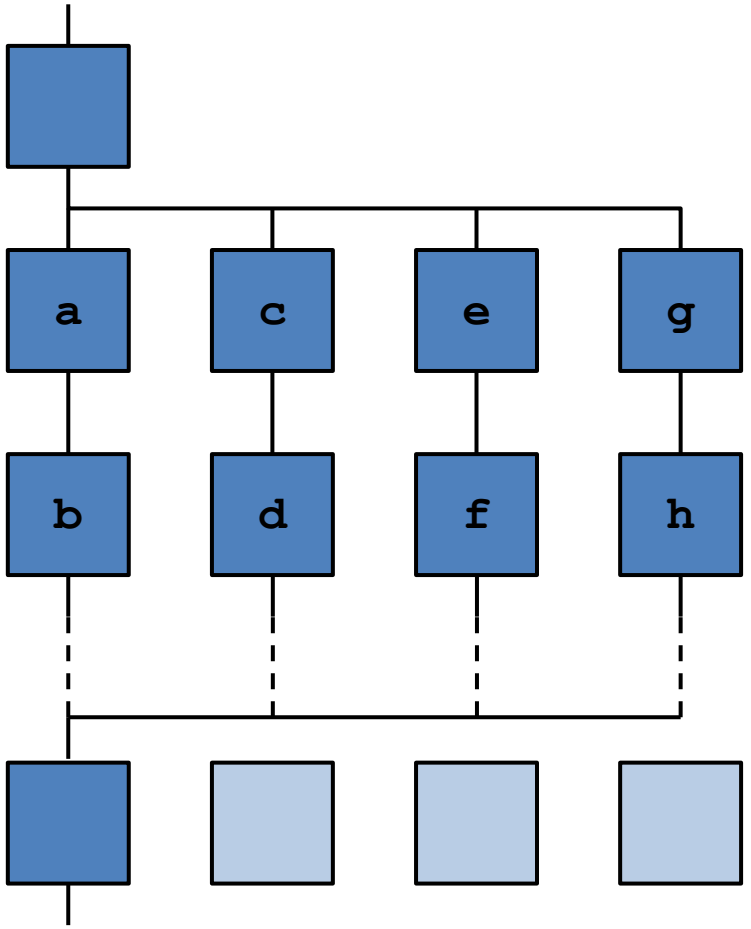
The `sections` Directive – Example

```
#pragma omp parallel {
    #pragma omp sections
    {
        #pragma omp section
        {   a = …;
            b = …; …;   }

        #pragma omp section
        {   c = …;
            d = …; …;   }

        #pragma omp section
        {   e = …;
            f = …; …;   }

        #pragma omp section
        {   g = …;
            h = …; …;   }
}   }
```



The `sections` Directive – Example

```
#include <omp.h>

int main ()
{
   int a[20], b[20], c[20], d[20];

   /* initialization of array a and b */

   #pragma omp parallel sections
   {
      #pragma omp section
      {
         for( int i = 0; i != 20; ++i )
            c[i] = a[i] + b[i];
      }
      #pragma omp section
      {
         for( int i = 0; i != 20; ++i )
            d[i] = a[i] * b[i];
      }
   }

   return 0;
}
```

The **`sections`** directive can be combined with the **`parallel`** directive.

Synchronization Constructs

- The **`barrier`** directive synchronizes all currently active threads:

```
#pragma omp barrier
```

- The **`critical`** directive specifies a region of code that …
  - … is executed by all threads.
  - … must be executed by only one thread at a time.

```
#pragma omp critical [(optional name)]
/* structured block */
```

A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name. All unnamed critical directives map to the same unspecified name.

Synchronization Constructs

- The **`master`** directive specifies a region of code that is only executed by the master thread – all other threads skip this region.

```
#pragma omp master
/* structured block */
```

  There are no implicit barriers!

- The **`single`** directive specifies a region of code that is only executed by one thread (not necessarily the master) – all other threads skip this region.

```
#pragma omp single [clause list]
/* structured block */
```

  There is an implicit barrier at the end of the block – unless a **`nowait`** clause is specified.

Synchronization Constructs – Example

```
int a[20], b[20];
int c = 0, f = 23;

/* b is initialized */

#pragma omp parallel
{
   #pragma omp for schedule( static )
   for( int i = 0; i < 20; ++i )
   {
      if( b[i] == 0 )
      {
         #pragma omp critical
         ++c;
      }
      a[i] = b[i] + f*(i+1)
   }

   #pragma omp single
   std::cout << "zeros in b: " << c << std::endl;
}
```

OpenMP – Further Information

- A well written and quite comprehensive overview on OpenMP:

  https://computing.llnl.gov/tutorials/openMP/

  ⇒ definitely worth reading!

- The official OpenMP API specification:

  http://openmp.org/wp/openmp-specifications/

The End

Questions ?

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT