Parallel Computing

PD Dr. rer. nat. habil. Ralf-Peter Mundani
Computation in Engineering / BGU
Scientific Computing in Computer Science / INF

Winter Term 2016/17
Part 4: Programming Memory-Coupled Systems

Technology is dominated by two types of people: those who understand what they do not manage, and those who manage what they do not understand.

—Archibald Putt
overview

- cache coherence
- dependency analysis
- programming with OpenMP
Cache Coherence

- reminder: memory hierarchy
  - memory hierarchy
    - exploitation of program characteristics such as locality
    - compromise between costs and performance
    - components with different speeds and capacities
Cache Coherence

- reminder: cache
  - cache memory
    - fast access buffer between main memory and processor
    - provides copies of current (main) memory content for fast access during program execution
  - cache management
    - tries to provide always those data that processor needs for the next computation step
    - due to small capacity certain strategies for load and update operations of cache content necessary

\[
\begin{align*}
\text{main memory} & \quad \text{block } B_j \quad j = 0, \ldots, n - 1 \\
\text{cache memory } (m \ll n) & \\
\text{cache-line } L_i \quad i = 0, \ldots, m - 1 \\
\end{align*}
\]

- mapping \( B_j \) to \( L_i \)
Cache Coherence

- reminder: cache (cont’d)
  - for any memory access the cache controller checks if
    1. the respective memory content has a copy stored in cache
    2. this cache entry is labelled as valid
  - check-up leads to a
    - cache hit: (1) and (2) are fulfilled → access served by cache
    - cache miss: (1) and/or (2) are not fulfilled
      - read miss
        - data is read from memory and a copy stored in cache
        - cache entry is labelled as valid
      - write miss: update strategy decides whether
        - the respective block is loaded (from memory) into cache and becomes updated due to write access
        - only memory is updated and cache stays unmodified
Cache Coherence

- definitions
  - processors with local cache that have independent access to a shared memory cause validity problems, i.e. several copies of the same memory block exist that contain different values
  - cache management is called
    - coherent: a read access always provides a memory block’s value from its last write access
    - consistent: all copies of a memory block in main memory and local caches are identical (i.e. coherence implicitly given)
  - inconsistencies between cache and main memory occur when updates are only performed in cache but not in main memory (so called copy-back or write-back cache policy, in contrast to write-through cache policy)
  - drawback: consistency is very expensive
Cache Coherence

- definitions (cont’d)
  - hence, inconsistencies (to some extent) can be acceptable if at least cache coherence is assured (e.g. temporary variables)
    - write-update protocol
      - an update of a copy in one cache requires also the update of all other copies in other caches
      - update can be delayed, at the latest with next access
    - write-invalidate protocol
      - exclusive write access of a processor to shared data that should be updated has to be assured
      - before the update of a copy in one cache all other copies in other caches are labelled as invalid
  - in general, write-invalidate protocol together with copy-back cache policy used for SMP systems
Cache Coherence

- definitions (cont’d)

  - example: write-invalidate protocol / write-through cache policy

1. $P_1$ gets exclusive access for $A$
2. invalidation of other copies of $A$
3. $P_1$ writes to $A$
4. update of $A$ in main memory
Cache Coherence

- **bus snooping**
  - processors with local cache are attached to a shared main memory via a bus (e.g. SMP system)
  - each processor ‘listens’ to all addresses sent over the bus by other processors and compares them to its own cache-lines
  - in case one cache-line matches this address, bus logic executes the following steps dependent from the cache-line’s state
    - **unmodified cache-line**
      - in case of a write access the cache-line becomes invalid
    - **modified cache-line**
      - bus logic interrupts the transaction and writes the modified cache-line to the main memory
      - afterwards, the initial transaction is executed again
  - MESI protocol frequently used with bus snooping
Cache Coherence

- **MESI protocol**
  - cache coherence protocol (write-invalidate) for bus snooping
  - each cache-line is assigned one of the following states
    - **exclusive modified (M)**: cache-line is the only copy in any of the caches and was modified due to a write access
    - **exclusive unmodified (E)**: cache-line is the only copy in any of the caches and was transferred for read access
    - **shared unmodified (S)**: copies of this cache-line reside in more than one cache and were transferred for read access
    - **invalid (I)**: cache-line is invalid
  - for write-through cache policy only the states shared unmodified and invalid are relevant
Cache Coherence

- MESI protocol (cont’d)
  - state: invalid
    - due to read / write access a valid copy is loaded into cache
    - other processes (snoop hit on a read) send signal SHARED if they have a valid copy
    - read miss: read miss shared \((RMS)\) or read miss exclusive \((RME)\) leads to state transition to \(S\) or \(E\), resp.
    - write miss \((WM)\): state transition to \(M\)
Cache Coherence

- MESI protocol (cont’d)
  - state: shared unmodified
    - read hit (RH) / snoop hit on a read (SHR): state is unchanged → process sends signal SHARED in case of SHR
    - write hit (WH): state transition to M
    - snoop hit on a write (SHW): state transition to I
Cache Coherence

- MESI protocol (cont’d)
  - state: exclusive unmodified
    - RH: state is unchanged ⇒ no bus usage necessary
    - SHR: process sends signal SHARED ⇒ state transition to S
    - SHW: state transition to I
    - WH: state transition to M ⇒ no bus usage necessary
Cache Coherence

- MESI protocol (cont’d)
  - state: exclusive modified
    - RH / WH: state is unchanged → no bus usage necessary
    - SHR / SHW: other process is notified via signal RETRY that a copy-back of this cache-line to main memory is necessary → state transition to I or S in case of SHW or SHR, resp.
Cache Coherence

- MESI protocol (cont’d)
  - putting it all together

**Diagam:**
- **I** (Intent)
- **M** (Modified)
- **S** (Shared)
- **E** (Exclusive)

- **RH/WH**: read hit
- **RMS**: read miss shared
- **RME**: read miss exclusive
- **WH**: write hit
- **WM**: write miss
- **SHR**: snoop hit on a read
- **SHW**: snoop hit on a write
Cache Coherence

- MESI protocol (cont’d)
  - example: SMP system with two processors
    - subsequent read / write access to same cache-line

*P*₁ wants to read cache-line...

### States

<table>
<thead>
<tr>
<th></th>
<th><em>P</em>₁</th>
<th><em>P</em>₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="State E" /></td>
<td><img src="image2" alt="State l" /></td>
</tr>
</tbody>
</table>

- read miss
- load valid copy from main memory
- state transition *I* → *E*

- snoop hit on a read
Cache Coherence

- MESI protocol (cont’d)
- example: SMP system with two processors
  - subsequent read / write access to same cache-line

$P_2$ wants to read cache-line...

$P_1$:

- snoop hit on a read
- send signal SHARED
- state transition $E \rightarrow S$

$P_2$:

- read miss
- load valid copy from main memory
- state transition $I \rightarrow S$
Cache Coherence

- MESI protocol (cont’d)
  - example: SMP system with two processors
    - subsequent read / write access to same cache-line

\( P_1 \) wants to write cache-line...

\begin{array}{c|c}
\hline
\text{P}_1 & \text{P}_2 \\
\hline
\text{M} & \text{I} \\
A = 7 & A = \emptyset \\
\hline
\end{array}

- write hit
- update cache-line
- state transition \( S \to M \)

- snoop hit on a write
- invalidate cache-line (i.e. state transition \( S \to I \))
Cache Coherence

- MESI protocol (cont’d)
  - example: SMP system with two processors
    - subsequent read / write access to same cache-line

\[ P_2 \text{ wants to read cache-line...} \]

\[ P_1: \]
\[ \text{snoop hit on a read} \]
\[ \text{send signal RETRY} \]
\[ \text{copy back cache-line and state transition } M \rightarrow E \]
\[ \text{snoop hit on a read} \]
\[ \text{send signal SHARED} \]
\[ \text{state transition } M \rightarrow S \]

\[ P_2: \]
\[ \text{read miss} \]
\[ \text{STOP} \]
\[ \text{read miss} \]
\[ \text{load valid copy from main memory} \]
\[ \text{state transition } I \rightarrow S \]
overview

- cache coherence ✓
- dependency analysis
- programming with OpenMP
Dependency Analysis

- types of dependencies
  - a program might have execution-order constraints between statements (i.e. instructions) due to dependencies
  - hence, dependence analysis should determine whether or not it is safe to reorder or parallelise these statements
- topics to be addressed by dependence analysis
  - control dependencies
  - data dependencies
  - loop dependencies
Dependency Analysis

- control dependencies
  - definition: an instruction executes if the previous instruction evaluates in a way that allows its execution
  - hence, a statement $S_2$ is control dependent on $S_1$ iff the execution of $S_2$ is conditionally guarded by $S_1$
  - example
    1:  \textbf{if} \hspace{1em} u > 2 \textbf{ then } // \text{ branch: if } u \leq 2 \text{ goto 3}
    2:  \hspace{1em} u \leftarrow u - z
    fi
    3:  v \leftarrow x \times y
    4:  w \leftarrow u + v

  - however, lines 3 and 4 will execute regardless of how the branch at line 1 executes \Rightarrow lines 3 and 4 are not control dependent on line 1 and may execute concurrently
  - essential for exploitation of instruction-level parallelism
Dependency Analysis

- data dependencies
  - arise due to competitive access to shared data
  - to be distinguished
    - flow dependence: read after write (RAW)
    - antidependence: write after read (WAR)
    - output dependence: write after write (WAW)
    - input dependence: read after read (RAR)

- data dependencies might lead to inefficiencies and bottlenecks, hence preventing optimisations such as out-of-order execution or parallelisation

- modern tools use dependence graphs, for instance, to find potential problem areas (= cycles within graphs) and examine to see if they can be broken

- example: KAP preprocessors for C, F77, and F90
Dependency Analysis

- data dependencies (cont’d)
  - flow dependence a.k.a. true dependence (RAW)
    - a statement $S_2$ is flow dependent on $S_1$ iff $S_1$ modifies a resource that $S_2$ reads and $S_1$ precedes $S_2$ in execution
    - example (sequence in a loop)
      1: $a[i] \leftarrow x[i] - 3$
      2: $b[i] \leftarrow a[i] / c[i]$
  - general problem: flow dependence cannot be avoided
    - here, $a[i]$ has to be calculated first in line 1 before using it in line 2 → lines 1 and 2 cannot be processed in parallel
Dependency Analysis

- data dependencies (cont’d)
  - antidependence (WAR)
    - a statement $S_2$ is antidependent on $S_1$ iff $S_2$ modifies a resource that $S_1$ reads and $S_1$ precedes $S_2$ in execution
    - example (sequence in a loop)
      1: $a[i] \leftarrow x[i] - 3$
      2: $b[i] \leftarrow a[i+1] / c[i]$

    - $a[i+1]$ is first used with its former value in line 2 and only then computed at the next execution of the loop in line 1 ➔ several iterations of the loop cannot be processed in parallel
    - in general, **antidependence can be avoided**
Dependency Analysis

- data dependencies (cont’d)
  - output dependence (WAW)
    - a statement $S_2$ is output dependent on $S_1$ iff $S_1$ and $S_2$ modify the same resource and $S_1$ precedes $S_2$ in execution
  - example (sequence in a loop)

1: $c[i+4] \leftarrow b[i] + a[i+1]$
2: $c[i+1] \leftarrow x[i]$

- some value is first assigned to $c[i+4]$ in line 1 and after three executions of the loop a new value is assigned to the same element again in line 2 $\Rightarrow$ several iterations of the loop cannot be processed in parallel
- nevertheless, output dependence can also be avoided
Dependency Analysis

- data dependencies (cont’d)
  - input dependence (RAR)
    - a statement $S_2$ is input ‘dependent’ on $S_1$ iff $S_1$ and $S_2$ read the same resource and $S_1$ precedes $S_2$ in execution
  - example (sequence in a loop)

1: $d[i] \leftarrow a[i] + 3$
2: $b[i] \leftarrow a[i+1] / c[i]$

- $a[i+1]$ is first used in line 2 and afterwards used again at the next execution of the loop in line 1 $\Rightarrow$ not a dependence in the same sense as the others, hence it does not prohibit reordering instructions or parallel execution of lines 1 and 2
Dependency Analysis

- data dependencies (cont’d)
  - removing of name dependencies
    - antidependence and output dependence may be removed through renaming of variables
  - example:
    
    1: \( a \leftarrow 2 \times x \)  
    2: \( b \leftarrow a / 3 \)  
    3: \( a \leftarrow 9 \times y \)
    
    renaming ➔
    
    1: \( c \leftarrow 2 \times x \)  
    2: \( b \leftarrow c / 3 \)  
    3: \( a \leftarrow 9 \times y \)
    
  - problem: line 3 (in variable \( a \)) is both antidependent on line 2 and output dependent on line 1
  - after renaming, both dependencies have been removed, but line 2 (in variable \( c \)) is still flow dependent on line 1
Dependency Analysis

- loop dependencies
  - statements (almost always w.r.t. array access and modification) within a loop body might form a dependence
  - problem: finding dependencies throughout different iterations
  - prototype of a ‘normalised’ nested loop with $N$ levels

```latex
for i_1 \leftarrow 1 \text{ to } n_1 \text{ do} \\
\text{// loop #1}
for i_2 \leftarrow 1 \text{ to } n_2 \text{ do} \\
\text{// loop #2}

\text{for } i_N \leftarrow 1 \text{ to } n_N \text{ do} \\
\text{// loop #N}
… \leftarrow … \\
\text{// statements}
```

- nesting level $K$ ($1 \leq K \leq N$): number of surrounding loops + 1
- iteration number $I_K$: value of iteration variable at nesting level $K$
Dependency Analysis

- loop dependencies (cont’d)
  - iteration vector $I$: vector of integers containing the iteration numbers $I_K$ of a particular iteration for each of the loops in order of the nesting levels
    
    $$I = (I_1, I_2, \ldots, I_N)^T$$
    
    with iteration numbers $I_K$, $1 \leq K \leq N$

  - iteration space: set of all possible iteration vectors (for a statement)

  - precedence $I < J$: iteration $I$ precedes iteration $J$ iff

    $$\exists K: I_R = J_R, \ \forall R: 1 \leq R < K, \text{ and } I_K < J_K$$

  - statement $S(I)$: statement $S$ under iteration vector $I$
Dependency Analysis

- loop dependencies (cont’d)
  - a statement $S_2(J)$ is loop dependent on $S_1(I)$ iff
    1) $I < J$ or
    $I = J$ and there exists a path from $S_1$ to $S_2$ in the loop body
    2) a memory location is accessed by $S_1$ on iteration $I$ and by $S_2$ on iteration $J$
    3) one of these accesses is a write

- theorem of loop dependence

There exists a dependence graph from statement $S_1$ to statement $S_2$ in a common nested loop if and only if there exist two iteration vectors $I$ and $J$ (for the nested loop), such that $S_2(J)$ is loop dependent on $S_1(I)$. 
Dependency Analysis

- **loop dependencies (cont’d)**
  - **distance vector** $D(I, J)$: if statement $S_2(J)$ is loop dependent on $S_1(I)$ then the dependence distance vector is computed as follows
    \[
    D(I, J)_k = J_k - I_k, \ 1 \leq K \leq N
    \]
  - **direction vector** $R(I, J)$: if statement $S_2(J)$ is loop dependent on $S_1(I)$ then the dependence direction vector is computed as follows
    \[
    R(I, J)_k = \begin{cases} 
    ‘<’ & \text{if } D(I, J)_k > 0 \\
    ‘=’ & \text{if } D(I, J)_k = 0, \ 1 \leq K \leq N \\
    ‘>’ & \text{if } D(I, J)_k < 0 
    \end{cases}
    \]
Dependency Analysis

- loop dependencies (cont’d)
  - types of different loop dependencies
    - loop-carried dependence
      - dependence from statement $S_1(I)$ to statement $S_2(J)$ iff $R(I, J)$ contains a ‘<’ as its leftmost component which is not equal to ‘=’
      - level of a loop-carried dependence conforms to the index of the leftmost component of $R(I, J)$ that is not equal to ‘=’
    - loop-independent dependence
      - dependence from statement $S_1(I)$ to statement $S_2(J)$ iff $I = J$
Dependency Analysis

- loop dependencies (cont’d)
  - example

```plaintext
for i ← 1 to N do
  for j ← 1 to M do
    1: a[i, j] ← b[i, j]
    2: c[i, j] ← 2*c[i, j] + a[i-1, j]
  od
od
```

- again, loop dependence iff
  - 1) \( I \leq J \)
  - 2) \( S_1(I) \) and \( S_2(J) \) access the same resource
  - 3) one of these accesses is a write
Dependency Analysis

- loop dependencies (cont’d)
  - example
    - flow dependence (RAW) in variable $a$
      1: $a[i, j] \leftarrow \ldots$
      2: $\ldots \leftarrow \ldots + a[i-1, j]$
      
      - $D(I, J) = (1, 0)^T$ and $R(I, J) = (‘<’, ‘=’)^T$
      - hence, a loop-carried dependence of level 1
  
  - antidependence (WAR) in variable $c$
    
    2: $\ldots \leftarrow 2^*c[i, j] + \ldots$
    2: $c[i, j] \leftarrow \ldots$

      - $D(I, J) = (0, 0)^T$ and $R(I, J) = (‘=’, ‘=’)^T$
      - hence, a loop-independent dependence
overview

- cache coherence ✓
- dependency analysis ✓
- programming with OpenMP
Programming with OpenMP

brief overview

- OpenMP is an application programming interface (API) for writing multithreaded programs, consisting of
  - a set of compiler directives
  - (runtime) library routines
  - environment variables

- available for C, C++, and Fortran
- suited for programming
  - UMA and NUMA systems
  - hybrid systems (i.e. NORMA systems with shared-memory nodes) in combination with message passing (e.g. MPI)

- further information: http://www.openmp.org
Programming with OpenMP

- compiler directives
  - prototypical form of compiler directives (C and C++)

```
#pragma omp directive-name [clause, …] newline
```

- `directive-name`: a valid OpenMP directive such as
  - `parallel`
  - `for, sections, single`
  - `master, critical, barrier`
  - `…`

- `clause`: optional statements such as
  - `if`
  - `private, firstprivate, lastprivate, shared`
  - `reduction`
  - `…`
Programming with OpenMP

- compiler directives (cont’d)
  - parallel region construct

```c
#pragma omp parallel [clause, …] newline
```

- precedes a parallel region (i.e. structured block of code) that will be executed by multiple threads
- when a thread reaches a ‘parallel’ directive, it creates a team of threads and becomes the master of that team
- code is duplicated and all threads will execute that code
- implicit barrier at the end of parallel region
- it is illegal to branch into or out of a parallel region
- number of threads set via `omp_set_num_threads()` library function or `OMP_NUM_THREADS` environment variable
- threads numbered from 0 (master thread) to \(N-1\)
Programming with OpenMP

- compiler directives (cont’d)
  - parallel region construct
    - some clauses
      - if (condition): must evaluate to TRUE in order for a team of threads to be created; only a single ‘if’ clause is permitted
      - private (list): listed variables are private to each thread; variables are uninitialised and not persistent (i.e. they do not longer exist when the parallel region is left)
      - shared (list): listed variables are shared among all threads
      - default (shared | none): default value for all variables in a parallel region
      - firstprivate (list): like private, but listed variables are initialised according to the value of their original objects
Programming with OpenMP

- compiler directives (cont’d)
  - parallel region construct
    - example

```c
#include <omp.h>

main ()
{
    int nthreads, tid;
    #pragma omp parallel private (tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf ("%d threads running\n", nthreads);
        } else
        {
            printf ("thread %d: Hello World!\n", tid);
        }
    }
}
```
Programming with OpenMP

- compiler directives (cont’d)
  - work-sharing constructs
    - divides the execution of the enclosed code region among the members of the team that encounter it
    - work-sharing constructs do not launch new threads
    - there is no implied barrier upon entry of a work-sharing constructs, only at the end
  - different types of work-sharing constructs
    - for: shares iterations of a loop (data parallelism)
    - sections: work is broken down into separate sections, each to be executed by a thread (function parallelism)
    - single: serialises a section of code
  - must be encountered by all members of a team or none at all
Programming with OpenMP

- compiler directives (cont’d)
  - work-sharing constructs: for

```plaintext
#pragma omp for [clause, …] newline
```

- iterations of the loop immediately following the ‘for’ directive to be executed in parallel (only in case a parallel region has already been initiated)
- branching out of a loop (e.g. break, return, exit) associated with a ‘for’ directive is illegal
- program correctness must not depend upon which thread executes a particular iteration
- some clauses
  - `lastprivate (list)`: like private, but values of listed variables are copied back at the end into their original variables
  - `nowait`: threads do not synchronise at the end of loop
Programming with OpenMP

- compiler directives (cont’d)
  - work-sharing constructs: for
    - clause schedule (type [,chunk]): describes how iterations of the loop are divided among the threads; the default schedule is implementation dependent
      - static: iterations are divided into pieces of size chunk and statically assigned to threads (if chunk is omitted, the iterations are evenly distributed)
      - dynamic: when a thread finishes one chunk, it is dynamically assigned another (default chunk size is 1)
      - runtime: the scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE
Programming with OpenMP

- compiler directives (cont’d)
  - work-sharing constructs: for
    - example

```c
int i;
float a[N], b[N], c[N];

#pragma omp parallel shared (a, b, c) private (i)
{
    #pragma omp for schedule (dynamic, 10) nowait
    for (i = 0; i < N; ++i)
        c[i] = a[i] + b[i];
}
Programming with OpenMP

- compiler directives (cont’d)
  - work-sharing constructs: sections

```c
#pragma omp sections [clause, …] newline
{
  #pragma omp section newline
  structured_block #1

  #pragma omp section newline
  structured_block #2

  ...
}
```

- independent ‘section’ directives are nested within a ‘sections’ directive; each section is executed once by a thread, different sections may be executed by different threads
- there is an implied barrier at the end of a ‘sections’ directive
- branching into or out of section blocks is illegal
Programming with OpenMP

- compiler directives (cont’d)
  - work-sharing constructs: sections
    - example

```c
int i;
float a[N], b[N], c[N];

#pragma omp parallel shared (a, b, c) private (i)
{
  #pragma omp sections nowait
  {
    #pragma omp section
    for (i = 0; i < N/2; ++i) { c[i] = a[i] + b[i]; }
    
    #pragma omp section
    for (i = N/2; i < N; ++i) { c[i] = a[i] + b[i]; }
  }
}
```
Programming with OpenMP

- compiler directives (cont’d)
  - work-sharing constructs: single

```c
#pragma omp single [clause, …] newline
```

- the enclosed code block is to be executed by only one thread (the thread that reaches the code block first)
- threads that do not execute the ‘single’ directive wait at the end of the enclosed code block
- might be useful when dealing with sections of code that are not thread safe (such as I/O)
- branching into or out of a single block is illegal
Programming with OpenMP

- compiler directives (cont’d)
  - synchronisation constructs

  `#pragma omp master newline`

  - specifies a region that is only to be executed by the master
  - there is no implied barrier associated with this directive
  - branching into or out of a master block is illegal

`#pragma omp critical [name] newline`

  - specifies a region of code that must be executed by only one thread at a time; threads trying to enter critical region are blocked until they get permission
  - optional name enables multiple critical regions to exist
  - branching into or out of a critical region is illegal
Programming with OpenMP

- compiler directives (cont’d)
  - synchronisation constructs

    #pragma omp barrier  
    
    - synchronises all threads, i.e. before resuming execution a thread has to wait here until all other threads have reached that barrier, too

    #pragma omp atomic  
    
    - specifies the atomic update of a specific memory location
    - applies only to a single, immediately following statement
    - example

    #pragma omp atomic
    x ← x + 1;
Programming with OpenMP

- compiler directives (cont’d)
  - reduction (operator: list)
    - performs a reduction on the listed variables, i.e. several values are reduced to a single scalar value combined via the named operation operator (e.g. sum, product)
    - listed variables must be of scalar type (no arrays and structs) and be declared shared in the enclosing context
    - the final result is written to the global shared variable
    - operator can be one of the following types
      - numerical: +, −, *, /
      - logical: AND (‘&amp;’), OR (‘|’)
      - bitwise: AND (‘&’), OR (‘|’), XOR (‘^’)
Programming with OpenMP

- compiler directives (cont’d)
  - reduction
    - example

```c
int i;
int a[MAX], b[MAX];
int res = 0;

#pragma omp parallel default (shared) private (i)
{
    #pragma omp for reduction (+: res) nowait
    for (i = 0; i < MAX; ++i)
        res = res + a[i]*b[i];
}

printf ("result = %d\n", res);
```
Programming with OpenMP

- runtime library

```c
void omp_set_num_threads (int num_threads)
```

- sets the number of threads that will be used in the next parallel region; it has precedence over the OMP_NUM_THREADS environment variable
- can only be called from serial portions of the code

```c
int omp_get_num_threads (void)
int omp_get_max_threads (void)
```

- returns
  - the number of threads that are currently executed in the parallel region from which it is called
  - the maximum number of threads that can be active
Programming with OpenMP

- runtime library (cont’d)

  int omp_get_thread_num (void)

  returns the number \(0 \leq TID \leq N-1\) of the thread making this call, the master thread has number 0

  int omp_in_parallel (void)

  may be called to determine if the section of code which is executed is parallel or not \(\Rightarrow\) returns a non-zero integer if parallel, and zero otherwise

  further runtime library routines available, see OpenMP specification (http://www.openmp.org) for details