Lecture 4: More on OpenMP

http://people.inf.ethz.ch/iyves/pnc11/

Peter Arbenz*, Andreas Adelmann**

*Computer Science Dept, ETH Zürich,
E-mail: arbenz@inf.ethz.ch

**Paul Scherrer Institut, Villigen
E-mail: andreas.adelmann@psi.ch
From last week

- **OpenMP**: An application programming interface (API) that provides a parallel programming model for shared memory multiprocessors. It *extends* the programming languages C/C++ and Fortran by
  - a set of *compiler directives* (called *pragmas*) to express shared memory parallelism.
  - a set of *runtime library routines* and *environment variables* to examine and modify execution parameters.
From last week (cont.)

- Creation of multiple threads with the `parallel` pragma:
  
  ```c
  #pragma omp parallel [clause [clause] ... ]
  ```
  
  - The **parallel directive** is used to create multiple threads of execution that execute concurrently.
  - The master thread and its execution context exist for the duration of the entire program.
  - If the master thread arrives at a parallel directive it spawns some new threads and forms a team of threads.
  - The parallel directive applies to a structured block, i.e. a block of code with one entry point at the top and one exit point at the bottom of the block (parallel region).
  - At the end of the parallel section the execution is joined again in the single master thread. An **implicit barrier** forces all threads to wait until all have completed their work.
From last week (cont.)

- Master thread in serial portion of code
- Master thread encounters parallel for directive
- Creates slave threads
- Master and slave threads divide iterations of parallel for loop and executes in parallel
- Implicit barrier: wait for threads to finish their iterations
- Master thread resumes execution after for loop. Slave threads disappear.

cf. Chandra et al.: Parallel Programming in OpenMP
From last week (cont.)

see http://en.wikipedia.org/wiki/OpenMP
From last week (cont.)

- Work-sharing constructs divide work among threads

  #pragma omp for [clause [clause] ... ]
  Distribute iterations of a for-loop over the threads.

  #pragma omp sections [clause [clause] ... ]
  Distribute independent work units.

  #pragma omp single
  Only one thread executes the code block.

- Work-sharing constructs do \textit{not} launch new threads.
- There is an implicit barrier at the \textit{end} of the construct. (Cf. nowait clause)
- If work-sharing constructs occur outside a parallel region, they are ignored. (Function called inside/outside parallel region.)
From last week (cont.)

- `#pragma omp for [clause [clause] ... ]`

  Most important work-sharing construct that implements **loop-level parallelism**;
  distributes iterations of a `for`-loop among threads.

**Schedule clauses** determine how loop iterations are mapped onto threads

- `schedule(static [,chunk])`
  Deal out blocks of iterations of size “chunk” to each thread.

- `schedule(dynamic [,chunk])`
  Each thread grabs “chunk” iterations off a queue until all iterations have been handled

- `schedule(guided [,chunk])`
  Dynamic schedule with variable (decreasing) chunk sizes.
Combined work-sharing loop constructs

```c
#pragma omp parallel for [clause [clause] ... ]
#pragma omp parallel sections [clause [clause] ... ]
```

Combined parallel work-sharing constructs are shortcuts that can be used when a parallel region comprises exactly one work-sharing construct, i.e., the work-sharing construct contains all the code in the parallel region.
Parallelizing simple loops

- In order for the compiler to successfully transform the sequential loop into a parallel loop, it must be able to determine the number of loop iterations.
- The control clause of the for loop

```c
#pragma omp parallel for
    for (idx=first; idx op last; increment){
        assignments;
    }
```

must be in canonical shape (see next page).
- Neither the loop counter `idx` nor the loop boundaries must be altered in the loop.
- There must not be a `break` in the loop.
- A `continue` statement is allowed.
Canonical shape

In order to be made parallel, the control clause of a for loop must have canonical shape. Above you see the legal variants. The identifiers start, end, and inc may be expressions.
Accessing variables / communicating among threads

- The threads of a team of threads are executed in a shared address space.
- In the parallel section of the saxpy example below all threads can access all variables.
  ```c
  #pragma omp parallel for schedule(static,chunk_size)
  for (i=0; i< N; i++)
      y[i] = alpha*x[i] + y[i];
  ```
- In this way, communication among threads is very easy: A thread simply writes in a shared variable where it can be accessed by other threads.
- This behavior is not always desired. Therefore, OpenMP allows to define so-called private variables that are assigned exclusively to a thread. The scope of such variables is limited to that thread.
Shared vs. private vs. reduction variables

- A **shared** variable has a single storage location in memory for the whole duration of the parallel construct. All threads that reference such a variable will access the same memory location. *Thus, reading and writing such a variable provides an easy mechanism for communicating between threads.*

- A **private** variable has multiple storage locations, one within the execution context of each thread.

This holds for pointers as well. Threads are not allowed to mess with the private memory of other threads. Therefore, do not assign to a private pointer the address of a private variable of another thread. The result is not defined.
Shared vs. private vs. reduction variables (cont.)

- The content of private variables at entry / at exit of its domain of definition is not defined. The clauses `firstprivate` / `lastprivate` allow to initialize / finalize their values.

- In C/C++, by default, all program variables except the loop index become shared variables in a parallel region.

- By the clause
  ```c
  default(shared) or default(none)
  ```
  this can be overwritten.

- Variables that are defined within the scope of a parallel pragma are private to this thread. (stack-allocated)

- There is a further type of variables: a reduction variable is used in reduction operations for accumulating values across threads, see a later example.
General properties of data scope clauses

- A data scope clause consists of the keyword identifying the clause, followed by a comma-separated list of variables within parenthesis.
  
  \texttt{private(x,y) shared(z) private(alpha,beta)}

- Any variable can be marked with a data scope clause (automatic, global, or formal parameters to subroutines)

- However, the directive with the scope clause must be in the lexical (static) extent of the declaration of each of the variables named in the scope clause.

- A data scoping clause applies to whole objects not to elements.

- An individual variable can appear at most once in a clause.

- The clause applies to precisely the directive it belongs to.
Private variable initialization and finalization

Private variables live only in parallel sections. If a variable named \textit{var} that appears in the private clause is used in the sequential portion of the code as well then the variable in the parallel region is stored in a different memory location. The private variable \textit{var} has no defined value at the beginning of the parallel region! If the value of the global variable \textit{var} should be inherited by the private variable \textit{var} the latter is initialized by this value by

\begin{verbatim}
#pragma omp parallel firstprivate(var)
\end{verbatim}

In a similar way

\begin{verbatim}
#pragma omp parallel lastprivate(var)
\end{verbatim}

transfers the value of the private copy \textit{var} in the \textit{sequentially last iteration} of the loop to the variable \textit{var} in the master thread.
void wrong()
{
    ...
    is = 0;
    #pragma omp parallel for private(is)
    for (j=0; j < 1000; j++)
        is = is + ... ;

    printf("%d",is);
}

Remarks:
- Inside the loop is is not initialized.
- Regardless of initialization, is is undefined in the print statement.
- Example from T. Mattson (Intel): Introduction to OpenMP.
void closer()
{
  ...

  is = 0;
  #pragma omp parallel for firstprivate(is)
  for (j=0; j < 1000; j++)
    is = is + ... ;

  printf("%d",is);
}

Remarks:
  ▶ In each thread, is is initialized to 0.
  ▶ is is still undefined in the print statement.
void even_closer()
{
...

is = 0;
#pragma omp parallel for firstprivate(is) lastprivate(is)
for (j=0; j < 1000; j++)
    is = is + ... ;

printf("%d",is);
}

Remark:
- is gets the value it had in the thread that executed the last iteration (j=999) of the loop.
- This may not be the desired value.
OpenMP synchronization

Threads can easily communicate with each other through reads and writes of shared variables. However, this has to be done with caution.

Two forms of process synchronization or coordination are

- **mutual exclusion**
  OpenMP provides a critical directive to provide a thread-exclusive access to a shared variable.

- **event synchronization**
  The barrier directive is a simple means to synchronize threads. A barrier defines a point where each thread waits for all other threads to arrive.

Remember that there is an implicit barrier at the end of each parallel and work-sharing construct.
Example: dot product

```c
#include <stdio.h>
#include <omp.h>
#define N 1000
#define NUM_THREADS 4

main() {
    int i;
    float sum, x[N], y[N];

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for
    for (i=0; i< N; i++){
        y[i] = 1.0;
        x[i] = (float)i;
    }
};
```
/* Now comes the interesting part of the code */

sum = 0.0;
#pragma omp parallel for
   for (i=0; i< N; i++){
      sum = sum + x[i]*y[i];
   }

printf("%f\n", sum);
}

What is the problem here? Let this code run several (many) times. Unintended sharing of variables can lead to race conditions: the results (may) change as the threads are scheduled differently.
So, the problem is the uncontrolled reading and writing of the shared variable `sum`. So, let's protect this operation.

Version II:

```c
sum = 0.0;
#pragma omp parallel for
#pragma omp critical
    { sum = sum + x[i]*y[i]; }
```

Have things improved? What about the cost of synchronization?
Example: dot product

<table>
<thead>
<tr>
<th>version</th>
<th>$p = 1$</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1875</td>
<td>4321</td>
<td>8799</td>
<td>8105</td>
<td>6348</td>
<td>7339</td>
<td>6538</td>
</tr>
<tr>
<td>II</td>
<td>155'547</td>
<td>121'387</td>
<td>139'795</td>
<td>140'576</td>
<td>171'973</td>
<td>1'052'832</td>
<td>3'541'113</td>
</tr>
<tr>
<td>III</td>
<td>1387</td>
<td>728</td>
<td>381</td>
<td>264</td>
<td>225</td>
<td>176</td>
<td>93</td>
</tr>
<tr>
<td>IV</td>
<td>1392</td>
<td>732</td>
<td>381</td>
<td>269</td>
<td>220</td>
<td>176</td>
<td>88</td>
</tr>
</tbody>
</table>

**Table 1:** Some execution times in $\mu$sec for the dot product

Numbers from Peterson & Arbenz.
Version III:

main()
{
    int i;
    float sum, local_sum, x[N], y[N];
    ...
    sum = 0.0;
    #pragma omp parallel private(local_sum)
    {
        local_sum = 0.0;
        #pragma omp for
        for (i=0; i< N; i++)
            local_sum = local_sum + x[i]*y[i];
        #pragma omp critical
        sum = sum + local_sum;
    }

This is much better. There are only NUM_THREADS sync points.
Parallel reduction operations

The most elegant way how to deal with the above problem is to give the variable \( \text{sum} \) the reduction attribute. Version IV:

\[
\text{sum} = 0.0; \\
\#pragma omp parallel for reduction(+ : sum) \\
\text{for (i=0; i< N; i++)} \\
\quad \text{sum} = \text{sum} + x[i] \times y[i]; \\
\} \\
\]

Here, the compiler implicitly creates a local copy of \( \text{sum} \). At the end of the loop, the local copies are added (in an optimal way) to yield the desired result.
Parallel reduction operations (cont.)

- Reductions are so common that OpenMP allows us to add a reduction clause to the parallel for pragma.
- OpenMP takes care of the details, such as storing partial sums in private variables and then adding the partial sums to the shared variable after the loop.
- Syntax:
  ```
  reduction (red_op : var_list)
  ```
- Reduction operations are possible with the operators 
  `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`.
- The initial values are the most meaningful ones.
- Admitted variable types are the basic data types in C/C++. 
Critical sections in OpenMP

The syntax of critical sections in OpenMP is:

```plaintext
#pragma omp critical [(name)]
{
    ...
}
```

The critical directive marks the subsequent block of code as critical.

The OpenMP runtime environment guarantees that at any given time the block of code is executed by at most one thread.
Critical sections in OpenMP (cont.)

All code segments in a program that are indicated critical are treated as one single critical section! They cannot be accessed by more than one thread.

This leads to the situation that threads at different locations in a program have to wait for each other although they need not be synchronized at all.

To improve this situation and get finer granularity for synchronization, critical sections can be named.

If critical sections are named then threads wait for entering a critical section until no other thread is inside the critical section with the same name.
Deadlocks

A **deadlock** is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does.

From Wikipedia:

*When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.* (Kansas law)

Famous example of Dijkstra:

The five dining philosophers problem.
Deadlocks (cont.)

With threads: a deadlock is a situation where one or multiple threads have to wait infinitely long for an event to complete or a resource to become available and only the involved threads could complete the event or make available the resource.

Example:
A function is called in a critical section $C_0$. Inside the routine there is another critical section $C_1$ with the same name (or both unnamed).

A thread that as entered $C_0$ will never be able to enter $C_1$. It waits entering $C_1$ until all threads (including itself) have left $C_0$ which is not possible since it would have to complete $C_1$ before returning into $C_0$. 
Data dependences

Whenever a statement in a program reads or writes a memory location and another statement reads or writes the same memory location, and at least one of the two statements writes the location, then there is a data dependence on that memory location between the two statements.

Example:

```c
for (i=1; i<N; i++) {a[i] = a[i] + a[i-1];}
```

Here, a[i] is written in loop iteration i and read in loop iteration i + 1. This loop cannot be executed in parallel. The results may not be correct.

(Assume a[i]=1 for all i. The correct (serial) result would be a[i]=i+1. However, if the second iteration step reads a[1] before it is written in the first iteration step then a[1]=a[2]=2.)
How to detect data dependences

We have to analyze each variable:

- Is the variable only read and never assigned within the loop body? If so, there are no dependences involving it.
- Otherwise, consider the memory locations that make up the variable and that are assigned within the loop. For each such location, is there exactly one iteration that accesses the location? If so, there are no dependences involving the variable. If not, there is a dependence.

In other words, if there is a dependence then there are two indices $i \neq j$ (in the bounds of the loop) such that iteration $i$ assigns to some element of an array $a$, say, and iteration $j$ reads or writes to that same element of $a$.
Such a data dependence is called loop-carried.
Classification of data dependences

Let the statement executed earlier in the sequential execution be loop \( S_1 \), and let the later statement be \( S_2 \).

- **flow dependence**: The memory location is written in \( S_1 \) and read in \( S_2 \). \( S_1 \) must execute before \( S_2 \) to produce the value that is consumed in \( S_2 \).

- **anti-dependence**: The memory location is read in \( S_1 \) and written in \( S_2 \).

- **output dependence**: The memory location is written in both statements \( S_1 \) and \( S_2 \).
Anti-dependences and output dependences can be removed by auxiliary variables (arrays) or reductions

Example, sequential version:

```c
for (i=0; i<N-1; i++) {
    x = (b[i] + c[i])/2;
    a[i] = a[i+1] + x;
}
```

Parallel version with dependences removed:

```c
#pragma omp parallel for shared(a, a2)
for (i=0; i<N-1; i++)
    a2[i] = a[i+1];

#pragma omp parallel for shared(a, a2) lastprivate(x)
for (i=0; i<N-1; i++)
    x = (b[i] + c[i])/2;
    a[i] = a2[i] + x;
```
Flow dependences can in general not be removed.

Let's consider the LU factorization of tridiagonal diagonally dominant matrix:

\[
\begin{pmatrix}
  a_0 & c_0 \\
  b_1 & a_1 & c_1 \\
  b_2 & a_2 & c_2 \\
  b_3 & a_3 & c_3 \\
  b_4 & a_4 & c_4 \\
  b_5 & a_5
\end{pmatrix}
\]

\[
= \begin{pmatrix}
  1 \\
  \ell_1 & 1 \\
  \ell_2 & 1 \\
  \ell_3 & 1 \\
  \ell_4 & 1 \\
  \ell_5 & 1
\end{pmatrix}
\begin{pmatrix}
  d_0 & c_0 & c_1 & c_2 & c_3 & c_4 \\
  d_1 & d_2 & c_2 & d_3 & c_3 & c_4 \\
  d_3 & d_4 & c_3 & d_4 & c_4 & c_5 \\
\end{pmatrix}
\]

\[T = LU\]
For solving
\[ T \mathbf{x} = LU \mathbf{x} = L \mathbf{z} = \mathbf{b}, \quad \mathbf{z} = U \mathbf{x}, \]
we proceed as follows (forward / backward substitution)
\[ L \mathbf{z} = \mathbf{b}, \quad U \mathbf{x} = \mathbf{z}. \]

In C the forward substitution \( L \mathbf{z} = \mathbf{b} \) is
\[
\begin{align*}
\mathbf{z}[0] &= \mathbf{b}[0]; \\
\text{for}(i=1;i<n;i++)
\quad &\mathbf{z}[i] = \mathbf{b}[i] - l[i]*\mathbf{z}[i-1]; \\
\end{align*}
\]
This is a very tight recurrence! Loop iteration \( i (S_2) \) reads \( \mathbf{z}[i-1] \) which was written in loop iteration \( i-1 (S_1) \).

To remove such dependences the algorithm must be changed.
Case study: Matrix-vector multiplication

Let us multiply a $M$-by-$N$ matrix $A$ with a $N$-vector $x$ and store the result in the $M$-vector $y$

$$y = Ax.$$ 

Component-wise this is

$$y_j = \sum_{i=1}^{N} a_{ji}x_i.$$
Version I: j,i, outer

```c
#pragma omp parallel for private(i)
for (j=0; j<M; j++){
    y[j] = 0.0;
    for (i=0; i<N; i++)
        y[j] = y[j] + A[j+i*M]*x[i];
}
```

- The picture shows the case where 5 threads are used.
- Each thread accesses another portion of the matrix/result vector (grey scale).
- The outer loop is parallelized.
- Note that the matrix is stored by columns.
Version II: j, i / inner. with reduction variable

```c
for (j=0; j<M; j++){
    tmp = 0.0;
    #pragma omp parallel for reduction(+ : tmp)
    for (i=0; i<N; i++)
        tmp = tmp + A[j+i*M]*x[i];
    y[j] = tmp;
}
```

Version III: i,j / outer. with 'vector reduction'/critical section

```c
#pragma omp parallel private(j,z)
{
    for (j=0; j<M; j++)
        z[j] = 0.0;

#pragma omp for
    for (i=0; i<N; i++)
        for (j=0; j<M; j++)
            z[j] = z[j] + A[j+i*M]*x[i];

#pragma omp critical
    for (j=0; j<M; j++)
        y[j] = y[j] + z[j];
}
```
Version IV: i,j / inner

for (i=0; i<N; i++){
    #pragma omp parallel for
    for (j=0; j<M; j++)
        y[j] = y[j] + A[j+i*M]*x[i];
}
Comparison / Timings

<table>
<thead>
<tr>
<th>Case</th>
<th>p</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = M = 500$</td>
<td>4</td>
<td>1.2</td>
<td>20</td>
<td>0.1</td>
<td>20</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.6</td>
<td>38</td>
<td>0.1</td>
<td>27</td>
<td>0.35</td>
</tr>
<tr>
<td>$N = M = 1000$</td>
<td>4</td>
<td>5.0</td>
<td>43</td>
<td>0.50</td>
<td>46</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>2.5</td>
<td>73</td>
<td>0.25</td>
<td>80</td>
<td>0.75</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Some execution times in millisec for $y = Ax$

Version V: outer, calling BLAS-2 `dgemv` routine.
Case study 2: Sparse matrix-vector multiplication

The same problem as before, but now the $M$-by-$N$ matrix $A$ is sparse.

$$ y = Ax. $$

Component-wise this is

$$ y_j = \sum_{i=1}^{N} a_{ji} x_i. $$
Storing of sparse matrices

```matlab
>> full(A)
ans =
  1   0   0   0   0
  0   2   5   0   0
  0   3   6   0   9
  0   4   0   8   0
  0   0   7   0  10

>> A
A =
   (1,1)    1
   (2,2)    2
   (3,2)    3
   (4,2)    4
   (2,3)    5
   (3,3)    6
   (5,3)    7
   (4,4)    8
   (3,5)    9
   (5,5)   10
```
Compressed row storage (CRS) format

Store the **nonzero** elements row-wise.
Three vectors are needed:

- float vector storing the matrix elements:
  \[
  \text{val} = [1 \ 2 \ 5 \ 3 \ 6 \ 9 \ 4 \ 8 \ 7 \ 10]
  \]

- int vector storing the column indices of the matrix elements
  \[
  \text{col\_ind} = [1 \ 2 \ 3 \ 2 \ 3 \ 5 \ 2 \ 4 \ 3 \ 5]
  \]

- int vector pointing to the first elements of each row
  \[
  \text{row\_ptr} = [1 \ 2 \ 4 \ 7 \ 9]
  \]

**Modified CRS** format: The diagonal is stored separately.

*Remark:* If \( A \) is symmetric then it suffices to store lower (or upper) triangle!
\[ y = Ax \]

With the CRS format we compute \( y \) element-by-element:

\[
y(k) = \sum_{i=row\_ptr(k)}^{row\_ptr(k+1)-1} a(i) \cdot x(col\_ind(i)) \tag{\ast}
\]

This is a ‘sparse’ dot product. One vector is stored contiguously (stride one), the other vector is scattered in memory.

Notice:

- In (\ast) we assume that \( row\_ptr(N) \) points to the memory location right after the last element of \( a \), i.e. right after the last element of the last row of \( A \).
- \( col\_ind(i_1) \neq col\_ind(i_2) \) for \( row\_ptr(k) \leq i_1 < i_2 < row\_ptr(k+1) \).
y = Ax

With CRS parallelization over rows (index \( k \)) is straightforward

```c
#pragma omp parallel for private(i)
    for (k=0; k<M; k++){
        y[k] = 0;
        for (i=row_ptr[k]; i<row_ptr[k+1]; i++)
            y[k] += val[i]*x[col_ind[i]];
    }
```

What about load balance?
$y = A^T x$

If we multiply with the transpose of $A$, then we apply a sequence of ‘sparse’ axpy operation:

$y = 0$

for $i = 1, \ldots, n$

$y = y + (i\text{-th column of } A) \cdot x_i$

endfor

In C this becomes ($A^T$ is $N \times M$).

for (i=0; i<N; i++)

for (k=row_ptr[i]; k<row_ptr[i+1]; k++)

$y[col\text{\_}ind[k]] += val[k] * x[i]$;
for (i=0; i<N; i++)
#pragma omp parallel for
    for (k=row_ptr[i]; k<row_ptr[i+1]; k++)
        y[col_ind[k]] += val[k]*x[i];

Typically too short loops.

Notice that we know that col_ind[k1] ≠ col_ind[k2] in the k-loop.
#pragma omp parallel private(j,z)
{
    #pragma omp for
    for (j=0; j<N; j++) z[j] = 0.0;
    #pragma omp for
    for (i=0; i<M; i++)
        for (k=row_ptr[i]; k<row_ptr[i+1]; k++)
            z[col_ind[k]] += val[k]*x[i];
    #pragma omp critical
    for (j=0; j<N; j++)
        y[j] = y[j] + z[j];
}

Does not take into account that the vectors z are sparse.