# Easy-going parallel C-programming with OpenMP

(a very basic intro)

Sven Reissmann (sven@0x80.io)

# Intro

- OpenMP - Open Multi-Processing
  - API for explicitly direct multi-threaded, shared memory parallelism
  - Jointly defined by a group of hardware and software vendors
  - Thought to provide a portable model for developers of shared memory parallel applications
  - Supports C/C+ and Fortran on a wide variety of architectures (only C will be a topic in this presentation)
  - Is comprised of
    - Compiler directives
    - Runtime library
    - Environmental variables

# Development timeline

| | | |
|---|---|---|
| Oct 1997 | Version 1.0 (Fortran) | |
| Oct 1998 | Version 1.0 (C/C++) | |
| Nov 1999 | Version 1.1 (Fortran) | |
| Nov 2000 | Version 2.0 (Fortran) | |
| Mar 2002 | Version 2.0 (C/C++) | |
| May 2005 | OpenMP 2.5 | - Combining both specifications into one |
| May 2008 | OpenMP 3.0 | - Support for tasking added |
| Jul 2011 | OpenMP 3.1 | - Support for tasking improved |
| Jul 2013 | OpenMP 4.0 | - Support for SIMD, teams, affinity and offloading |
| Nov 2015 | OpenMP 4.5 | - Improved support |
| Nov 2016 | OpenMP 5.0 Preview 1 | - Release planned for 2018 |

# Compiler directives

- Appear as comments in the source code
- Ignored by compilers, unless activated by an appropriate compiler flag
- Used for
  - Creating parallel regions
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization
- Compiler directives syntax
  - **sentinel directive-name [clause [[,] clause], ...] <newline>**
    **{structured-block}**
  - #pragma omp parallel default(shared) private(beta,pi)

# Runtime library routines

- The run-time library provides routines for a variety of purposes
  - Setting and querying the number of threads
  - Querying a thread's unique identifier, a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- To use the runtime library routines in C, you need to include the <*omp.h*> header file.

# Environment variables

- Environment variables allow to control the execution of parallel code at run-time
- They allow things, such as:
  - Setting the number of threads
  - Specifying how loop interations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy

# A basic OpenMP program

```
#include <omp.h>

int main (void)
{
  int var1, var2, var3;

  /* Serial code */

  /* Beginning of parallel region. Fork a team of threads.
   * Specify variable scoping
   */
  #pragma omp parallel
  {
    /* Parallel region executed by all threads.
     * Other OpenMP directives.
     * Run-time Library calls.
     * All threads join master thread and disband.
     */
  }

  /* Resume serial code */
}
```

# Some important clauses (I)

- **Data sharing**
  - **shared**
    The data within a parallel region is shared. Default, except for loop iteration counters.
  - **private**
    The data within a parallel region is private. A private variable is not initialized and the value is not maintained for use outside the parallel region.
  - **default**
    Allows the programmer to state that the default data scoping within a parallel region will be either shared or none.
- **Initialization**
  - **firstprivate**
    Like private except initialized to original value.
  - **lastprivate**
    Like private except original value is updated after construct.

# Some important clauses (II)

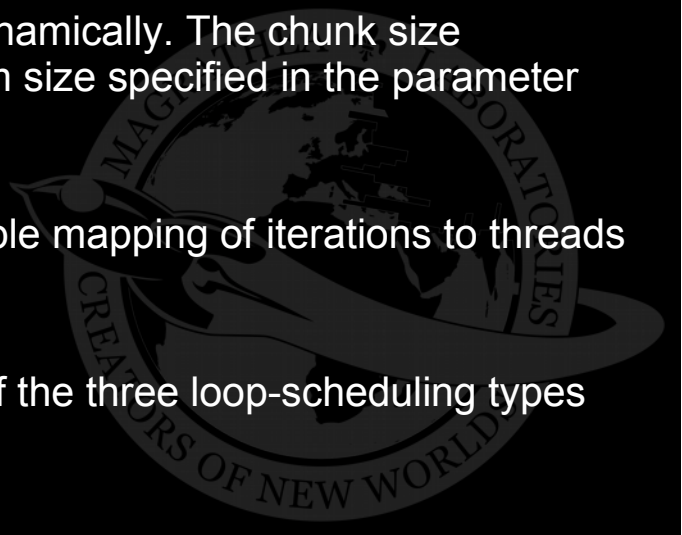
- **Synchronization**
  - **critical**
    The enclosed code block will be executed by only one thread at a time (not simultaneously by multiple threads).
  - **atomic**
    The memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic.
  - **ordered**
    The structured block is executed in the order in which iterations would be executed in a sequential loop.
  - **barrier**
    Each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
  - **nowait**
    Specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish.

# Some important clauses (III)

- **schedule(type[, chunk])**
  The iterations in the work sharing construct are assigned to threads according to the scheduling method.

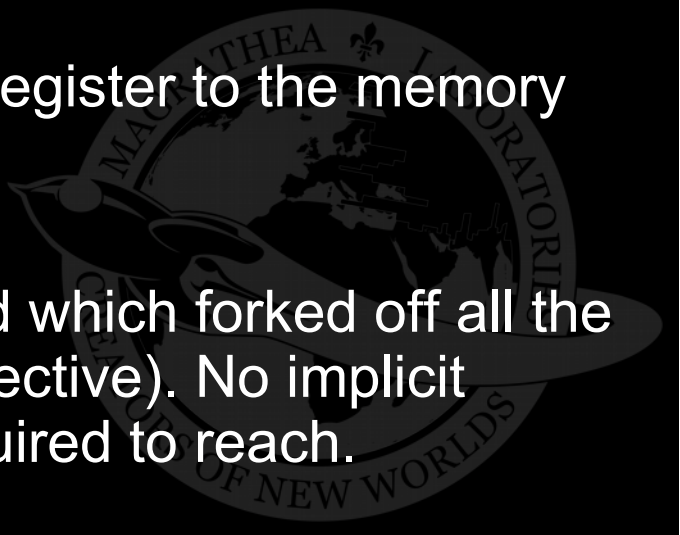
  The types of scheduling are:

  - **static**
    All threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. Specifying an integer for the parameter chunk will allocate chunk number of contiguous iterations to each thread.
  - **dynamic**
    Some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter chunk defines the number of contiguous iterations that are allocated to a thread at a time.
  - **guided**
    A large chunk of contiguous iterations are allocated to each thread dynamically. The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter chunk.
  - **auto**
    The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
  - **runtime**
    Uses the OMP_schedule environment variable to specify which one of the three loop-scheduling types should be used.



# Some important clauses (IV)

- **Reduction**
  - **reduction(operator | intrinsic : list)**
    Joins the work from all threads at the end of a parallel section.
- **If clause**
  - **if**
    This will cause the threads to parallelize the task only if a condition is met. Otherwise the code block executes serially.
- **Other**
  - **flush**
    The value of this variable is restored from the register to the memory for using this value outside of a parallel part
  - **master**
    Executed only by the master thread (the thread which forked off all the others during the execution of the OpenMP directive). No implicit barrier; other team members (threads) not required to reach.



# Parallel regions

Syntax

```
#pragma omp parallel [clause[[,] clause] ...] new-line
structured-block
```

Clauses

```
if, num_threads, default, private, firstprivate, shared, copyin, reduction,
proc_bind
```

Example

```
#pragma omp parallel if(MULTI == 1)
{
  printf (“Hello, world!\n“);
}
```

# Work sharing (for)

## Syntax

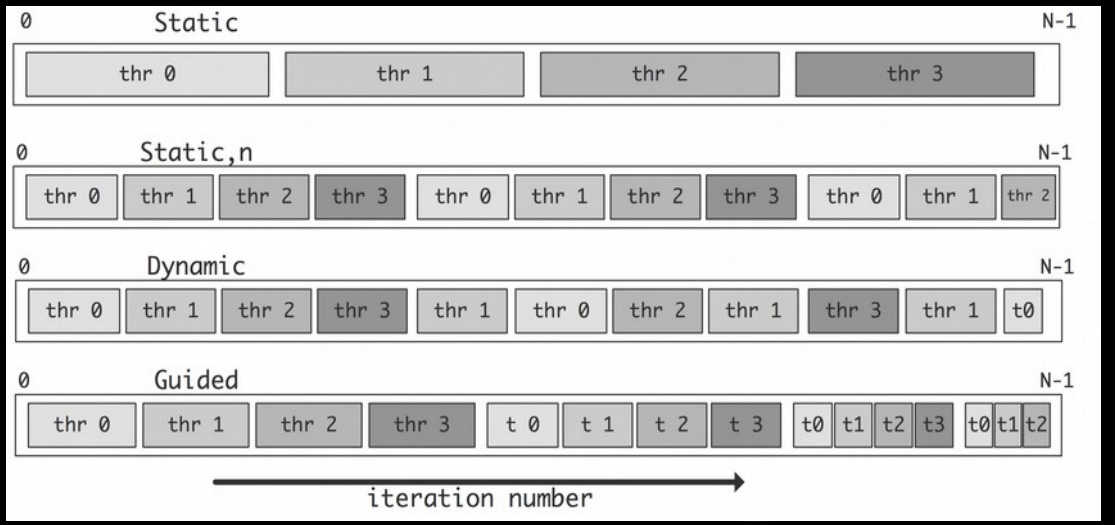
```
#pragma omp for [clause[[,] clause] ...] new-line
for-loops
```

## Clauses

```
private, firstprivate, lastprivate, reduction, schedule, collapse, ordered, nowait
```

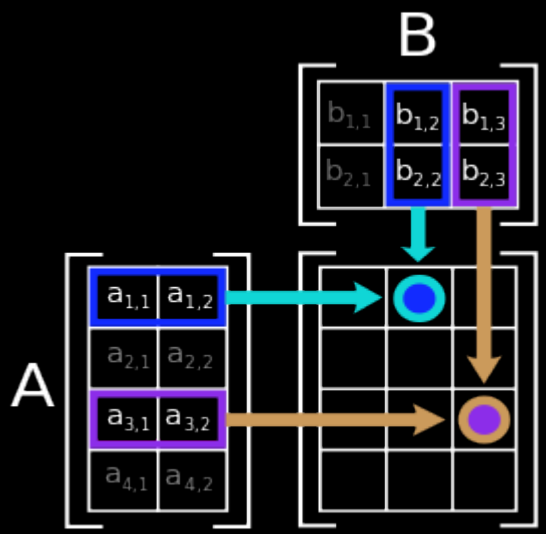
## Example

```
#pragma omp parallel for schedule(static, 2)
for (i = 0; i < P; i++)
  {
    for (j = 0; j < Q; j++)
    {
      a[i][j] = rand() % 10;
    }
  }
```



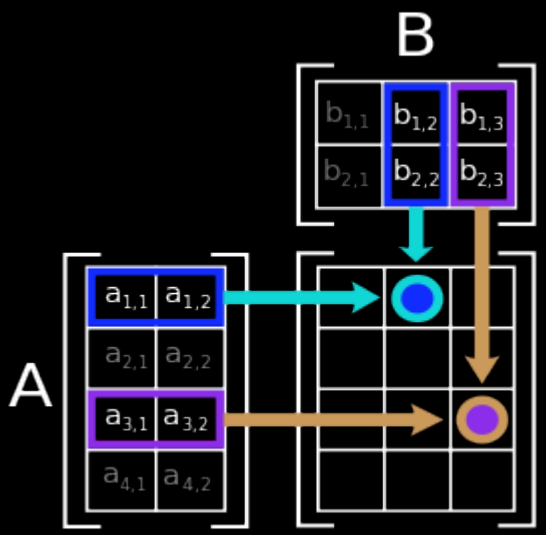
# Matrix multiplication

```
for (i = 0; i < P; i++)
{
  for (j = 0; j < Q; j++)
  {
    a[i][j] = rand() % 10;
  }
}


for (i = 0; i < Q; i++)
{
  for (j = 0; j < R; j++)
  {
    b[i][j] = rand() % 10;
  }
}


for (i = 0; i < P; i++) {

  for (j = 0; j < R; j++) {
    c[i][j] = 0;
    for (k = 0; k < Q; k++) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```




# Matrix multiplication (OpenMP)

```
#pragma omp parallel default(none) private(i, j, tid) shared(a, b, c)
{

  tid = omp_get_thread_num ();

  #pragma omp for
  for (i = 0; i < P; i++)
  {
    for (j = 0; j < Q; j++)
    {
      a[i][j] = rand() % 10;
    }
  }

  #pragma omp for
  for (i = 0; i < Q; i++)
  {
    for (j = 0; j < R; j++)
    {
      b[i][j] = rand() % 10;
    }
  }

  #pragma omp for
  for (i = 0; i < P; i++) {
    printf ("Thread %d did row %d", tid, i);
    for (j = 0; j < R; j++) {
      c[i][j] = 0;
      for (k = 0; k < Q; k++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }

}
```




# Work sharing (sections)

Syntax

```
#pragma omp sections [clause[[,] clause] ...] new-line
{
  [#pragma omp section new-line]
    structured-block

  [#pragma omp section new-line
    structured-block]
}
```

Clauses

```
private, firstprivate, lastprivate, reduction, nowait
```

# Tasks

## Syntax

```
#pragma omp task [clause[[,] clause] ...] new-line
```

## Clauses

```
if, final, untied, default, mergable, private, firstprivate, shared, depend
```

## Example

```
#pragma omp parallel
{
  #pragma omp task
  foo ();
  #pragma omp barrier

  #pragma omp single
  {
    #pragma omp task
    bar ();
  }
}
```

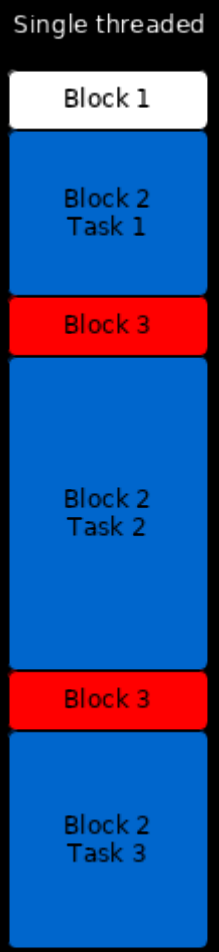
# Tasks: Example (I)

```
list *root;    /* Pointer to beginning of list  */
list *el;      /* An arbitrary list element     */


#pragma omp parallel
{
  #pragma omp single
  {
    el = root;

    while (el)
    {
      #pragma omp task
      process (el);

      el = el->next;
    }
  }
}
```

# Tasks: Example (II)

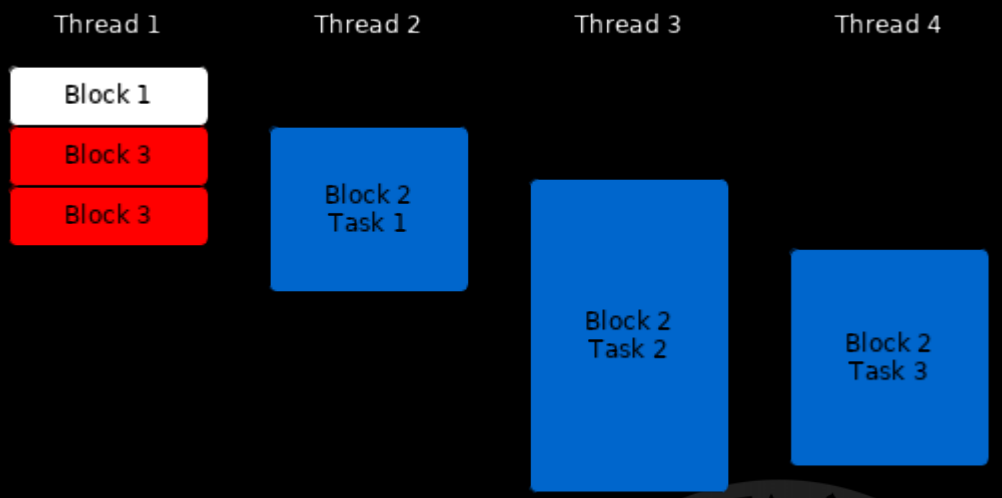
```
list *root;    /* Pointer to beginning of list  */
list *el;      /* An arbitrary list element     */
```

```
#pragma omp parallel
{
  #pragma omp single
  {
    el = root;

    while (el)
    {
      #pragma omp task
      process (el);

      el = el->next;
    }
  }
}
```




# Tasks: Example (III)

```
list *root;    /* Pointer to beginning of list  */
list *el;      /* An arbitrary list element     */


#pragma omp parallel
{
  #pragma omp single
  {
    el = root;

    while (el)
    {
      #pragma omp task
      process (el);

      el = el->next;
    }
  }
}
```




# Fragen?

- Slides und examples

  https://github.com/reissmann/openmp_talk

- OpenMP specifications

  http://www.openmp.org/specifications/

- Contact

  IRC: irc.hackint.org:6697 #mag.lab (user: major)
  Mail: sven@0x80.io