A Gentle Introduction to Parallel Programming using OpenMP

CEMRACS16 Summer School

François BROQUEDIS

July 20, 2016

CORSE INRIA team - Grenoble Institute of Technology

Acknowledgements

- Ruud Van der Pas (Oracle)
- Emmanuel Agullo (HIEPACS INRIA)
- Jean-François Méhaut (CORSE INRIA)
- Frédéric Desprez (CORSE INRIA)
- Laura Grigori (ALPINES INRIA)

Summary

Introduction to OpenMP: Basic Concepts and Syntax

Getting OpenMP Up to Speed

- Part I: Benefit from Cache Memory
- Part II: Control Thread and Data Placement
- Part III: Parallelism Grain and Runtime-related Overheads

Introduction to OpenMP: Basic Concepts and Syntax

What is OpenMP?

- A de-facto standard API to write shared memory parallel applications in C, C++ and Fortran
- Consists of compiler directives, runtime routines and environment variables
- Specification maintained by the *OpenMP Architecture Review Board* (`http://www.openmp.org`)
- Current version of the specification : 4.5 (November 2015)

Advantages of OpenMP

- A mature standard
  - *Speeding-up your applications since 1998*
- Portable
  - Supported by many compilers, ported on many architectures
- Allows **incremental parallelization**
- Imposes low to no overhead on the sequential execution of the program
  - Just tell your compiler to ignore the OpenMP pragmas and you get back to your sequential program
- Supported by a wide and active community
  - The specifications have been moving fast since revision 3.0 (2008) to support :
    - new kinds of parallelism (tasking)
    - new kinds of architectures (accelerators)

The OpenMP Execution Model

```
int main(void)
{
    some_statements();

    #pragma omp parallel
    {
        printf("Hello, world!\n");
    }

    other_statements();

    #pragma omp parallel
    {
        printf("This is definitely the\n");
        printf("worst motivating example\n");
        printf("ever...\n");
    }

    return EXIT_SUCCESS;
}
```

A fork-join execution model

- entering a parallel region will create some threads (*fork*)
- leaving a parallel region will terminate them (*join*)
- any statement executed outside parallel regions are executed sequentially

The OpenMP Memory Model

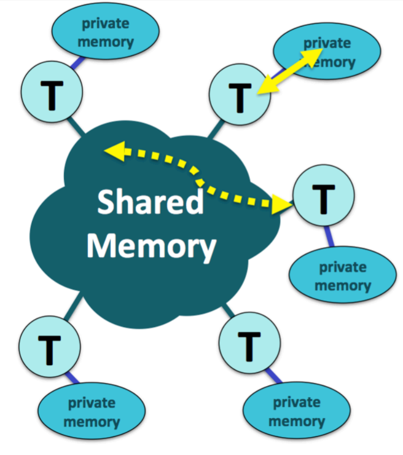


Figure 1: The way OpenMP threads handle memory

- All the threads have access to the same *globally shared* memory
- Each thread has access to *its own private memory area* that can't be accessed by other threads
- Data transfer is performed through shared memory and is *100% transparent to the application*
- The application programmer is responsible for providing the corresponding data-sharing attributes

Data-sharing Attributes

Need to set the visibility of each variable that appears inside an OpenMP parallel region using the following **data-sharing attributes** :

- **shared**: the data can be read and written by any thread of the team. All changes are visible to all threads.
- **private**: each thread is working on its own version of the data that cannot be accessed by other threads of the team.
- **firstprivate**: each thread is working on its own version of the variable. The data is initialized using the value it had before entering the parallel region.
- **lastprivate**: each thread is working on its own version of the variable. The value of the last thread leaving the region is copied back to the variable.

Putting Threads to Work: the Worksharing Constructs

```
void simple_loop(int N,
                 float *a,
                 float *b)
{
    int i;
    // i, N, a and b are shared by default
    #pragma omp parallel firstprivate(N)
    {
        // i is private by default
        #pragma omp for
        for (i = 1; i <= N; i++) {
            b[i] = (a[i] + a[i-1]) / 2.0;
        }
    }
}
```

- **omp for** : distribute the iterations of a loop over the threads of the parallel region.
- Here, assigns `N/P` iterations to each thread, `P` being the number of threads of the parallel region.
- **omp for** comes with an implicit **barrier synchronization** at the end of the loop one can remove with the **nowait** keyword.

OpenMP Loop Schedulers: Definitions

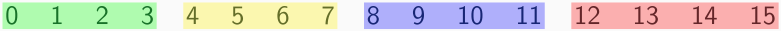
The **schedule** clause of the **for** construct specifies the way loop iterations are assigned to threads. The loop scheduler can be set to one of the following :

- **schedule(static, chunk_size)**: assign fixed chunks of iterations in a round robin fashion.
- **schedule(dynamic, chunk_size)**: fixed chunks of iterations are dynamically assigned to threads at runtime, depending on the threads availability.
- **schedule(guided, chunk_size)**: like dynamic, but with a chunk size that decreases over time.
- **runtime**: the loop scheduler is chosen at runtime thanks to the `OMP_SCHEDULE` environment variable.

OpenMP Loop Schedulers: Chunks

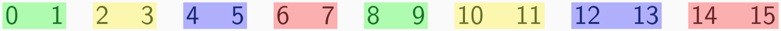
The **chunk_size** attribute determines the granularity of iterations chunks the loops schedulers are working with.

legend: 🟩 thread0 🟨 thread1 🟦 thread2 🟥 thread3



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

Figure 2: `static` scheduler, 16 iterations, 4 threads, **default** **`chunk_size`**



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

Figure 3: `static` scheduler, 16 iterations, 4 threads, **`chunk_size`=2**

OpenMP Loop Schedulers: Example on 500 iterations, 4 threads

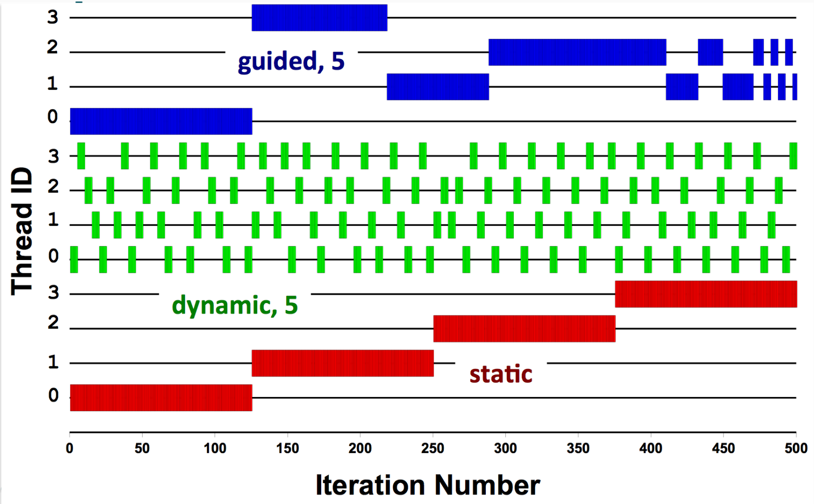
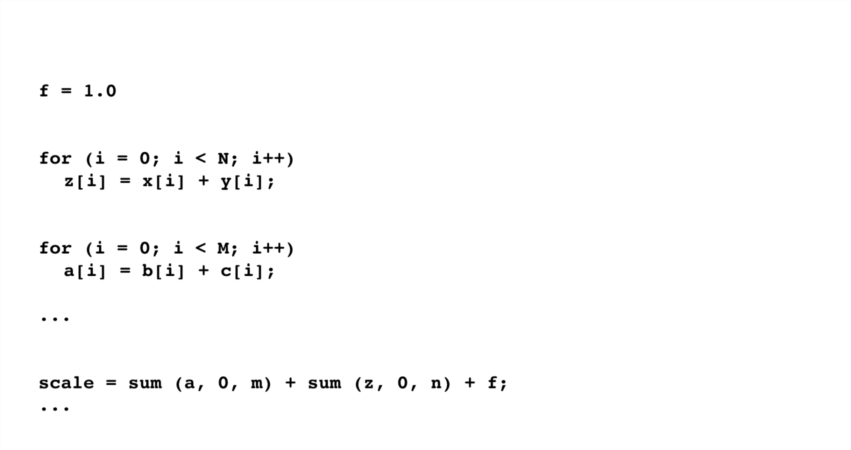


Figure 4: A parallel loop scheduled with different OpenMP schedulers

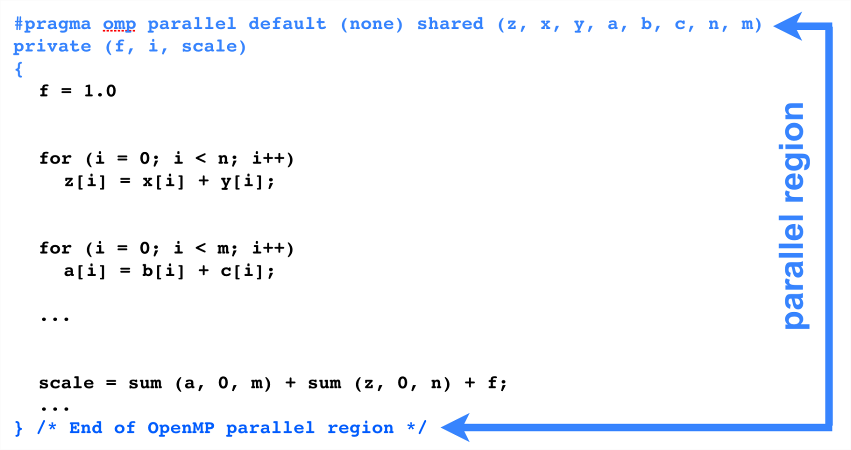
A First Example to Illustrate OpenMP Capabilities



```
f = 1.0

for (i = 0; i < N; i++)
  z[i] = x[i] + y[i];

for (i = 0; i < M; i++)
  a[i] = b[i] + c[i];

...

scale = sum (a, 0, m) + sum (z, 0, n) + f;
...
```

Figure 5: Our job for today: parallelize this using OpenMP

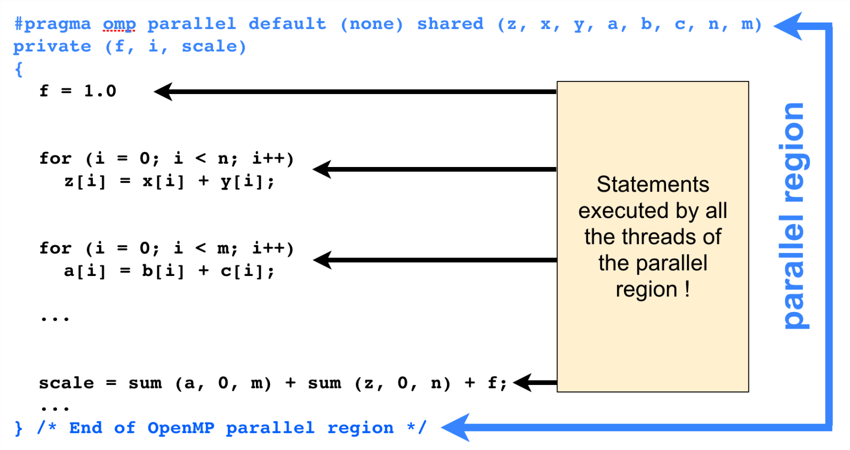
A First Example to Illustrate OpenMP Capabilities



```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
  f = 1.0


  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];


  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...


  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

Figure 6: First create the parallel region and define the data-sharing attributes

A First Example to Illustrate OpenMP Capabilities



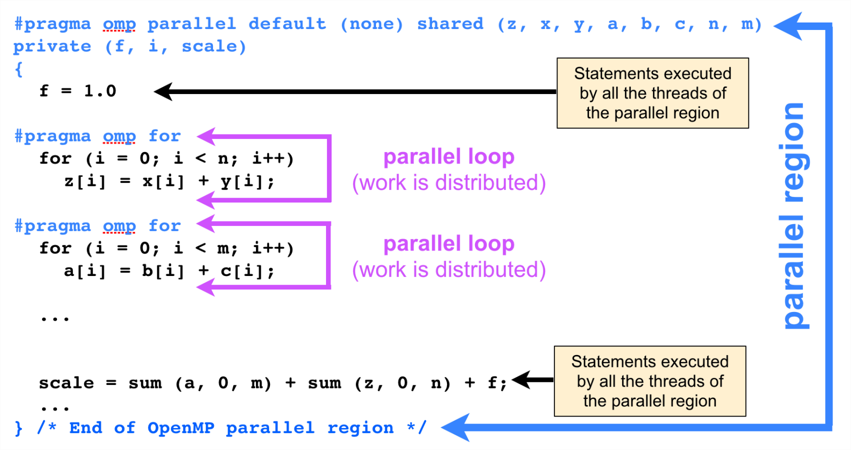
```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
   f = 1.0

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];

   for (i = 0; i < m; i++)
      a[i] = b[i] + c[i];

   ...

   scale = sum (a, 0, m) + sum (z, 0, n) + f;
   ...
} /* End of OpenMP parallel region */
```

Statements executed by all the threads of the parallel region !

parallel region

Figure 7: At this point, all the threads execute the whole program (you won't get any speed-up from this!)

A First Example to Illustrate OpenMP Capabilities

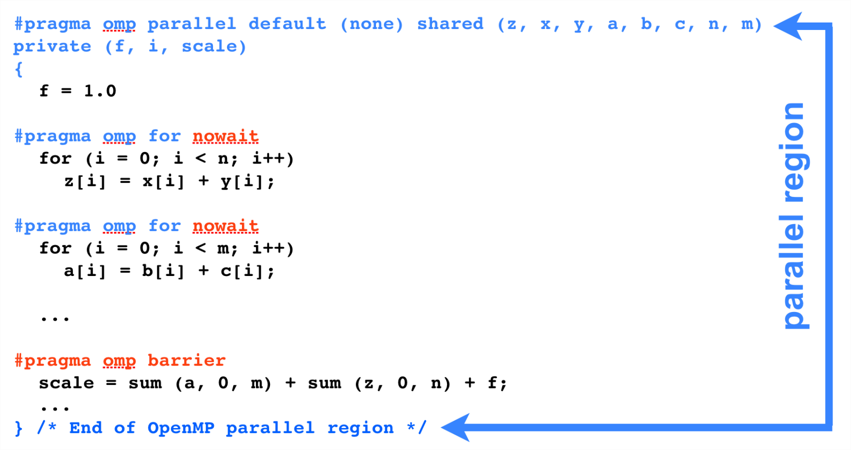


```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
  f = 1.0

#pragma omp for
  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];

#pragma omp for
  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...


  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

Statements executed by all the threads of the parallel region
parallel loop (work is distributed)
parallel loop (work is distributed)
parallel region
Statements executed by all the threads of the parallel region

Figure 8: Now distribute the loop iterations over the threads using **omp for**.

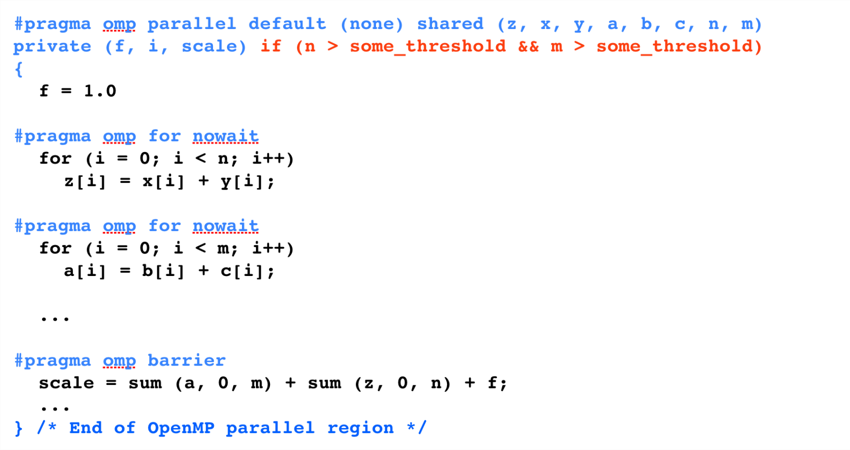
Optimization #1: Remove Unnecessary Synchronizations



```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
  f = 1.0

#pragma omp for nowait
  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];

#pragma omp for nowait
  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...

#pragma omp barrier
  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

Figure 9: There are no dependencies between the two parallel loops, we remove the implicit barrier between the two.

Optimization #2: Don’t Go Parallel if the Workload is Small



```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale) if (n > some_threshold && m > some_threshold)
{
  f = 1.0

#pragma omp for nowait
  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];

#pragma omp for nowait
  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...

#pragma omp barrier
  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

Figure 10: We don’t want to pay the price of thread management if the workload is too small to be computed in parallel.

Extending the Scope of OpenMP with Task Parallelism

omp for has made OpenMP popular and remains for most users its central feature. But what if my application was not written in a loop-based fashion?

```c
int fib(int n) {
    int i, j;
    if (n < 2) {
        return n;
    } else {
        i = fib(n - 1);
        j = fib(n - 2);
        return i + j;
    }
}
```

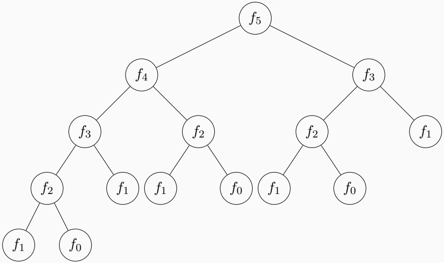


Figure 11: Call graph of fib(5)

Tasking in OpenMP: Basic Concept

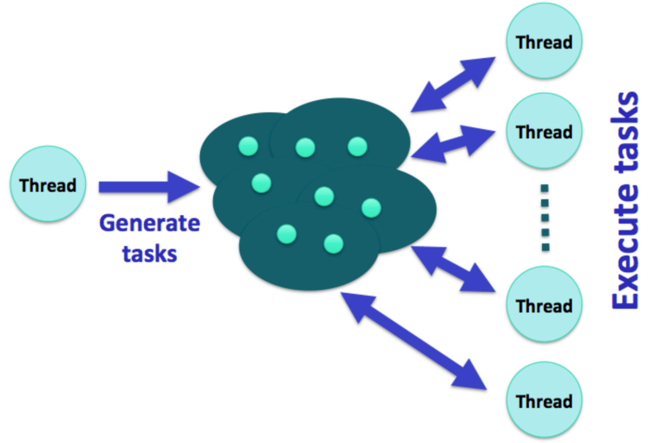


Figure 12: The OpenMP tasking concept : tasks generated by one OpenMP thread can be executed **by any of the threads** of the parallel region.

Tasking in OpenMP: Basic Concept (cont'd)

- The application programmer specify regions of code to be executed in a task with the **#pragma omp task** construct
- All tasks can be executed *independently*
- When any thread encounters a task construct, a task is generated
- Tasks are executed **asynchronously** by any thread of the parallel region
- Completion of the tasks can be guaranteed using the **taskwait** synchronization construct

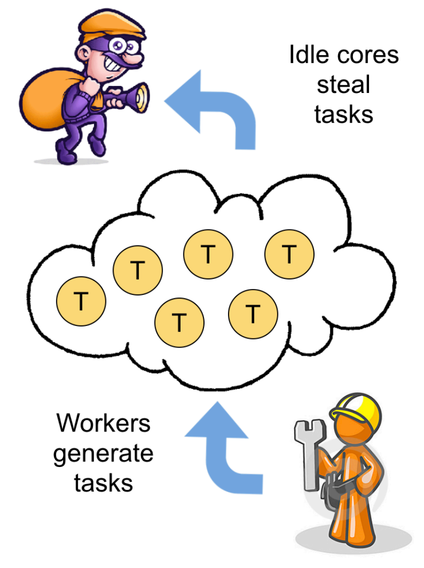
```c
int main(void) {
    ...
    #pragma omp parallel
    {
        #pragma omp single
        res = fib(50);
    }
    ...
}

int fib(int n) {
    int i, j;
    if (n < 2) {
        return n;
    } else {
        #pragma omp task
        i = fib(n - 1);
        #pragma omp task
        j = fib(n - 2);
        #pragma omp taskwait
        return i + j;
    }
}
```

Tasking in OpenMP: Execution Model

The Work-Stealing execution model

- Each thread has its own *task queue*
- Entering an `omp task` construct pushes a task to the thread's local queue
- When a thread's local queue is empty, it steals tasks from other queues

Tasks are well suited to applications with irregular workload.



Your Very First OpenMP Tasking Experience (1/5)

```c
int main(void)
{
    printf("A ");
    printf("race ");
    printf("car ");

    printf("\n");
    return 0;
}
```

- We want to use OpenMP to make this program print either `A race car` or `A car race` using tasks.
- Here is a battle plan :
  1. Create the threads that will execute the tasks
  2. Create the tasks and make one of the thread generate them

Program output:

```
$ OMP_NUM_THREADS=2 ./task-1
$ A race car
```

Your Very First OpenMP Tasking Experience (2/5)

```
int main(void)
{
    #pragma omp parallel
    {
        printf("A ");
        printf("race ");
        printf("car ");
    }

    printf("\n");
    return 0;
}
```

Program output:

```
$ OMP_NUM_THREADS=2 ./task-2
$ A race A race car car
```

- We want to use OpenMP to make this program print either `A race car` or `A car race` using tasks.
- Here is a battle plan :
  1. Create the threads that will execute the tasks
  2. Create the tasks and make one of the thread generate them

Your Very First OpenMP Tasking Experience (3/5)

```c
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            printf("race ");
            #pragma omp task
            printf("car ");
        }
    }

    printf("\n");
    return 0;
}
```

- We want to use OpenMP to make this program print either `A race car` or `A car race` using tasks.
- Here is a battle plan :
  1. Create the threads that will execute the tasks
  2. Create the tasks and make one of the thread generate them

Program output:

```
$ OMP_NUM_THREADS=2 ./task-3
$ A race car
$ OMP_NUM_THREADS=2 ./task-3
$ A car race
```

Your Very First OpenMP Tasking Experience (4/5)

```c
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            printf("race ");
            #pragma omp task
            printf("car ");

            printf("is fun ");
            printf("to watch ");
        }
    }

    printf("\n");
    return 0;
}
```

- Now that everything is working as intended, we would like to print `is fun to watch` at the end of the output string.
- This example illustrates the **asynchronous execution** of tasks.

Program output:

```
$ OMP_NUM_THREADS=2 ./task-4
$ A is fun to watch race car
$ OMP_NUM_THREADS=2 ./task-4
$ A is fun to watch car race
```

Your Very First OpenMP Tasking Experience (5/5)

```c
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            printf("race ");
            #pragma omp task
            printf("car ");
            #pragma omp taskwait
            printf("is fun ");
            printf("to watch ");
        }
    }

    printf("\n");
    return 0;
}
```

- Now that everything is working as intended, we would like to print `is fun to watch` at the end of the output string.
- This example illustrates the **asynchronous execution** of tasks.
- To fix this, you need to explicitly wait for the completion of the tasks with **taskwait** before printing "is fun to watch"

Program output:

```
$ OMP_NUM_THREADS=2 ./task-5
$ A race car is fun to watch
$ OMP_NUM_THREADS=2 ./task-5
$ A car race is fun to watch
```

What About Tasks with Dependencies on Other Tasks?

- Here, task A is writing some data that will be processed by task C. The same goes for task B and task D.
- The **taskwait** construct here makes sure task C won't execute before task A and task D before task B.
- As a side effect, task C won't execute until the execution of task B is over, creating some kind of *fake dependency* between task B and C.

```
void data_flow_example (void)
{
    type x, y;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        write_data(&x);  // Task A
        #pragma omp task
        write_data(&y);  // Task B

        #pragma omp taskwait

        #pragma omp task
        print_results(x); // Task C
        #pragma omp task
        print_results(y); // Task D
    }
}
```

OpenMP Tasks Dependencies : Rationale

The **depend** clause allows you to provide information on the way a task will access data. The depend clause is followed by an access mode that can be `in`, `out` or `inout`. Here are some examples of use for the `depend` clause:

- **depend(in: x, y, z)**: the task will read variables x, y and z
- **depend(out: res)**: the task will write variable res, any previous value of res will be ignored and overwritten
- **depend(inout: k, buffer[0:n])**: the task will both read and write variable `k` and the content of `n` elements of `buffer` starting from index 0

The OpenMP runtime system dynamically decides whether a task is ready for execution or not considering its dependencies (there is no need for further user intervention here).

OpenMP Tasks Dependencies : Some Trivial Example

```
void data_flow_example (void)
{
    type x, y;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task depend(out: x)
        write_data(&x);  // Task A
        #pragma omp task depend(out: y)
        write_data(&y);  // Task B

        #pragma omp task depend(in: x)
        print_results(x); // Task C
        #pragma omp task depend(in: y)
        print_results(y); // Task D
    }
}
```

- Here is the previous example program written with tasks dependencies.
- The **taskwait** construct is gone : the runtime system will rely on data dependencies to choose a ready task to execute.
- In this version, task C could be executed before task B, as long as the execution of task A is over.

Expressing dependencies sometimes helps unlocking more parallelism.

Getting OpenMP Up to Speed

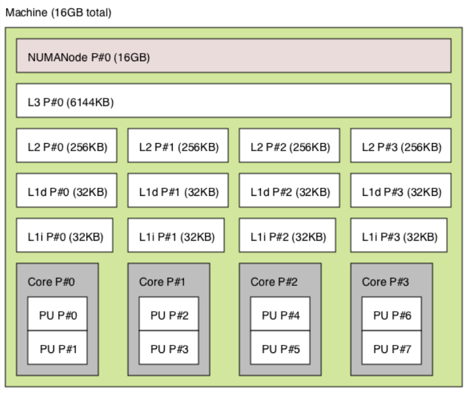
Preambule: A Closer Look at Your Favorite Platform

Improving the execution of a parallel application **requires a good understanding of the target platform architecture**. In particular, knowing about the following items is always useful:

- The multicore processor: how many cores are available? Which of them are physical/logical cores (HyperThreading and friends)?
- The memory hierarchy: what kind of memory is available? How is it organized?
- The architecture topology: how (multicore) processors are connected together and how do they access memory?

Preambule: Getting to Know Your Platform with hwloc

The **hwloc** library gathers valuable information about your platform and synthesize it into a generic representation.



Provides information about:

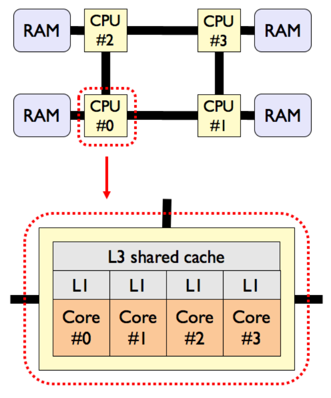
- the processing units (logical/physical cores)
- the cache hierarchy
- the memory hierarchy (NUMA nodes)

However, hwloc does not provide the entire architecture topology (the way processors are connected together).

`https://www.open-mpi.org/projects/hwloc/`

Preambule: Understanding the Architecture Topology

The operating system knows about the way processors are connected together **to some extent**. It can provide a *distance table* that roughly represent how many crossbars you need to cross to access a specific NUMA node (see the hwloc-distances program).



	N_0	N_1	N_2	N_3
N_0	0	10	10	20
N_1	10	0	20	10
N_2	10	20	0	10
N_3	20	10	10	0

Figure 13: A 4-nodes NUMA machine with the corresponding NUMA distance table.

Our Target Platform for Today : the Intel192 Machine

The Intel192 machine:

- 12 pairs of NUMA nodes made of a two 8-core Xeon processors each and 32GB of RAM
- a two-level NUMA topology (pairs can be up to two hops away from each other)

Table 1: NUMA distances from node 0 advertised by the hwloc library on Intel192.

NUMA nodes location	*local*	*peer*	*one hop away*	*two hops away*
hwloc distances	1.0	5.0	6.5	7.9

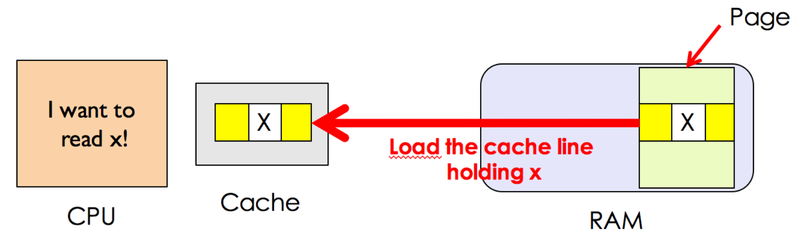
Part I: Benefit from Cache Memory

Cache Memory: Basic Concept

A **cache** can be seen as a table of **cache lines** holding a predefined amount of memory (64B on most processors).

Accessing a variable results in a cache hit if the corresponding cache line has already been cached (**fast memory access**).

It can also result in a cache miss if the corresponding cache line is not cached yet. The hardware has to load the cache line to the cache before the processor can access it (**longer memory access)**.



Case Study #1: a Naive Square Matrix Multiplication Algorithm

We consider a simple matrix multiplication algorithm involving square matrices of double precision floats.

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

where $\mathrm{C}_0 = \mathrm{A}_0\mathrm{B}_0 + \mathrm{A}_1\mathrm{B}_4 + \mathrm{A}_2\mathrm{B}_8 + \mathrm{A}_3\mathrm{B}_{12}$
Let's implement this using what we've learned about OpenMP!

Speeding-Up OpenMP: Benefit from Cache Memory (2)

```c
void gemm_omp(double *A, double *B, double *C, int n) {
    #pragma omp parallel
    {
        int i, j, k;
        #pragma omp for
        for (i=0; i<n; i++) {
            for (j=0; j<n; j++) {
                for (k=0; k<n; k++) {
                    C[i*n+j] += A[i*n+k]*B[k*n+j];
                }
            }
        }
    }
}
```

Let's run this on the Intel192 machine!

```
$ OMP_PLACES=cores ./mxm-v1 1536
```

Speeding-Up OpenMP: Benefit from Cache Memory (2)

```c
void gemm_omp(double *A, double *B, double *C, int n) {
    #pragma omp parallel
    {
        int i, j, k;
        #pragma omp for
        for (i=0; i<n; i++) {
            for (j=0; j<n; j++) {
                for (k=0; k<n; k++) {
                    C[i*n+j] += A[i*n+k]*B[k*n+j];
                }
            }
        }
    }
}
```

Let's run this on the Intel192 machine!

```
$ OMP_PLACES=cores ./mxm-v1 1536
```

Serial time : 40.3018250s

Parallel time : 0.270773s

Achieved speed-up : 148

We Can Do Better : It’s All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of `C`?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$\mathrm{C}_0 = \mathrm{A}_0\mathrm{B}_0$

$+ \mathrm{A}_1\mathrm{B}_4$

$+ \mathrm{A}_2\mathrm{B}_8$

$+ \mathrm{A}_3\mathrm{B}_{12}$

We Can Do Better : It's All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of `C`?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$\mathrm{C}_0 = \mathrm{A}_0\mathrm{B}_0$
$+ \mathrm{A}_1\mathrm{B}_4$
$+ \mathrm{A}_2\mathrm{B}_8$
$+ \mathrm{A}_3\mathrm{B}_{12}$

We Can Do Better : It's All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of `C`?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$\mathrm{C}_0 = \mathrm{A}_0\mathrm{B}_0$

$+ \mathrm{A}_1\mathrm{B}_4$

$+ \mathrm{A}_2\mathrm{B}_8$

$+ \mathrm{A}_3\mathrm{B}_{12}$

We Can Do Better : It's All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of `C`?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$\mathrm{C}_0 = \mathrm{A}_0\mathrm{B}_0$
$+ \mathrm{A}_1\mathrm{B}_4$
$+ \mathrm{A}_2\mathrm{B}_8$
$+ \mathrm{A}_3\mathrm{B}_{12}$

We Can Do Better : It's All About Being (Cache) Friendly

Assuming we work with 32 bytes-long cache lines and each element of the matrix is 8 bytes long, how many cache lines do I need to compute one element of C?

$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_1 & B_2 & B_3 \\ B_4 & B_5 & B_6 & B_7 \\ B_8 & B_9 & B_{10} & B_{11} \\ B_{12} & B_{13} & B_{14} & B_{15} \end{bmatrix}$$

$\mathrm{C}_0 = \mathrm{A}_0\mathrm{B}_0$
$+ \mathrm{A}_1\mathrm{B}_4$
$+ \mathrm{A}_2\mathrm{B}_8$
$+ \mathrm{A}_3\mathrm{B}_{12}$

Conclusion:

- Every access to A_i use the same cache line => **cache friendly**
- Every access to B_j use a different cache line => **poor cache utilization**

Deal with the B_j Situation

To improve cache utilization, we can transpose matrix B to make sure B_0, B_4, B_8 and B_{12} are stored on the same cache line

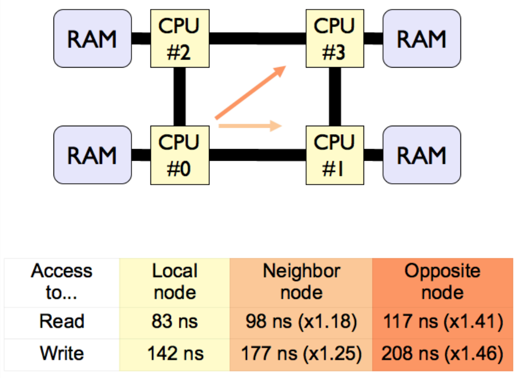
$$\begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix} = \begin{bmatrix} A_0 & A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 & A_7 \\ A_8 & A_9 & A_{10} & A_{11} \\ A_{12} & A_{13} & A_{14} & A_{15} \end{bmatrix} \times \begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix}$$

$\mathrm{C}_0 = \mathrm{A}_0\mathrm{B}_0$
$+ \mathrm{A}_1\mathrm{B}_4$
$+ \mathrm{A}_2\mathrm{B}_8$
$+ \mathrm{A}_3\mathrm{B}_{12}$

Performance evaluation on Intel192:
Serial time: 5.188652s (prev: 40.3018250s)
Parallel time: 0.067657s (prev: 0.270773s)
Achieved speed-up : 77 (rel to prev: 595(!))

Part II: Control Data Placement on NUMA Systems

Non-Uniform Memory Accesses



Access to...	Local node	Neighbor node	Opposite node
Read	83 ns	98 ns (x1.18)	117 ns (x1.41)
Write	142 ns	177 ns (x1.25)	208 ns (x1.46)

Software support to deal with data locality

- The **First-Touch** allocation policy (default behavior of most memory allocators)
- Some external libraries like *libNUMA* or *hwloc*

Allocating Memory on First Touch

On most UNIX-like operating systems, when allocating some memory using malloc and friends, the corresponding memory pages are physically allocated :

- when they are accessed for the first time (lazy allocation)
- next to the thread that performs this first access

In other words, **it's crucial to make sure a data is first touched by the thread that will access it during the computation phase**.

Case Study #2: the STREAM Benchmark

```
void STREAM_Triad(double *a,
                  double *b,
                  double *c,
                  double scalar)
{
  int j;
#pragma omp parallel for
  for (j=0; j<N; j++)
      a[j] = b[j]+scalar*c[j];
}
```

- STREAM is a memory benchmark written in C+OpenMP performing simple operations on vectors.
- It was designed to evaluate the **aggregated memory bandwidth** of a shared memory platform.
- Vectors are large enough not to fit into cache memory.

How to initialize vectors *a*, *b* and *c* to make sure the corresponding memory pages will be accessed locally when executing `STREAM_Triad`?

STREAM: Initializing Data the Right Way

```
#pragma omp parallel for
    for (j=0; j<N; j++) {
        a[j] = 1.0;
        b[j] = 2.0;
        c[j] = 0.0;
    }
```

```
void STREAM_Triad(double *a,
                  double *b,
                  double *c,
                  double scalar)
{
  int j;
#pragma omp parallel for
  for (j=0; j<N; j++)
      a[j] = b[j]+scalar*c[j];
}
```

- Here, we perform a parallel initialization of the data being accessed by `STREAM_Triad`.
- The OpenMP spec guarantees that the same iterations will be executed by the same threads as long as both parallel loops :
    - involve the same number of threads and iterations
    - involve the static loop scheduler with the same parameters (chunk_size)
- We then only need to **make sure a thread will be assigned to the same core in both regions.**

Thread Affinity in OpenMP

The proper way to bind OpenMP threads is as follows:

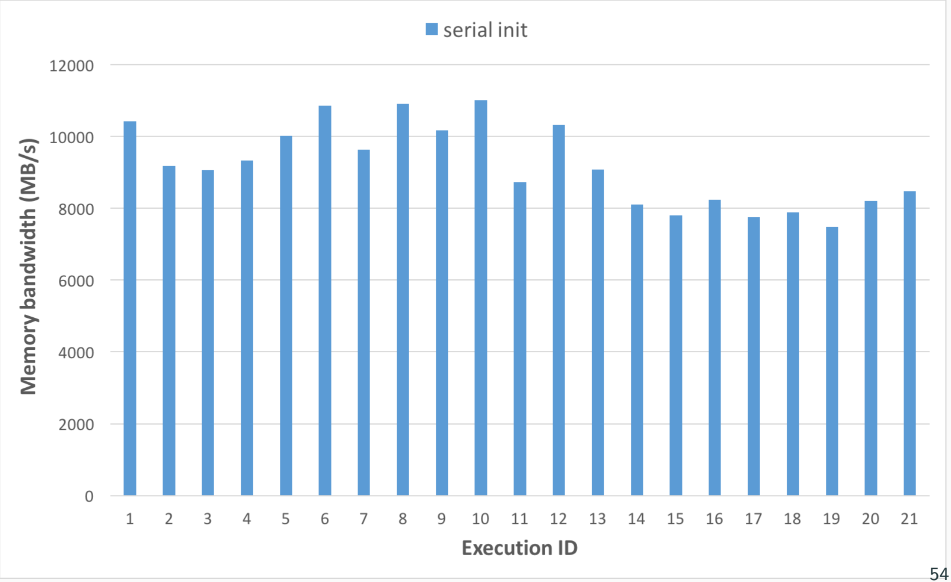
1. The programmer first defines a list of places on which the threads will be pinned. A place can be seen as a set of processing units (hardware threads, most of the time).
   - It can be explicit, refering to the processing unit OS numbering: `OMP_PLACES="0,1,2,3"`
   - Or you can use one of the predefined abstractions (threads, cores, sockets): `OMP_PLACES=cores` to refer to physical cores
2. The programmer then specifies the way threads will be distributed over the list of places.
   - Ex: `OMP_PROC_BIND=close` to perform a compact distribution over the places list (default behavior).

STREAM: Evaluation

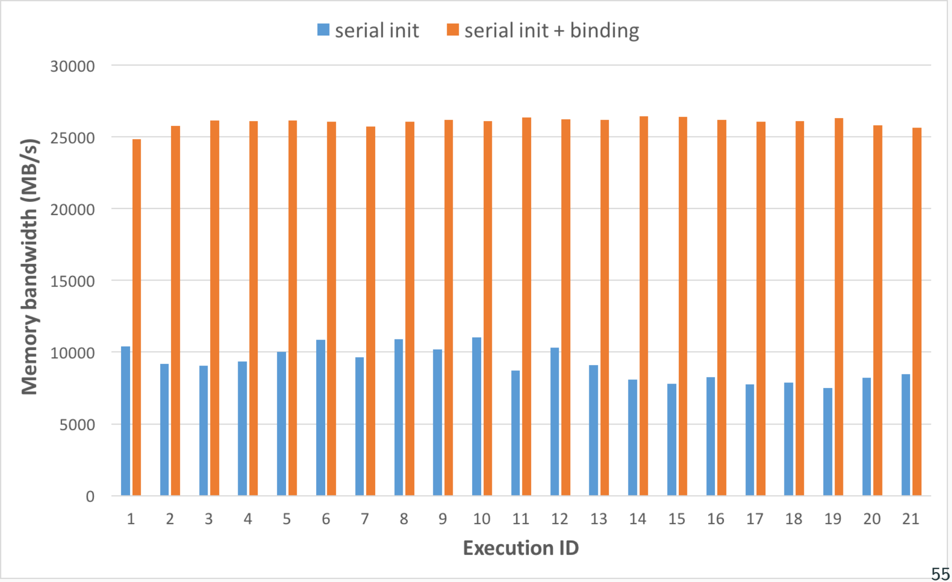
We ran the STREAM benchmark **20 times** and experimented with different strategies for thread and data placement :

- **serial init**: STREAM vectors are initialized sequentially and no thread binding is applied
- **serial init + binding**: same, except we bind the threads using `OMP_PLACES=cores`
- **randomized memory + binding**: we allocate the memory pages in a round robin fashion over the NUMA nodes using the `hwloc-bind --mempolicy interleave` tool
- **parallel init + binding**: we initialize the vectors in parallel and we make sure they don't move between initialization and computation phase

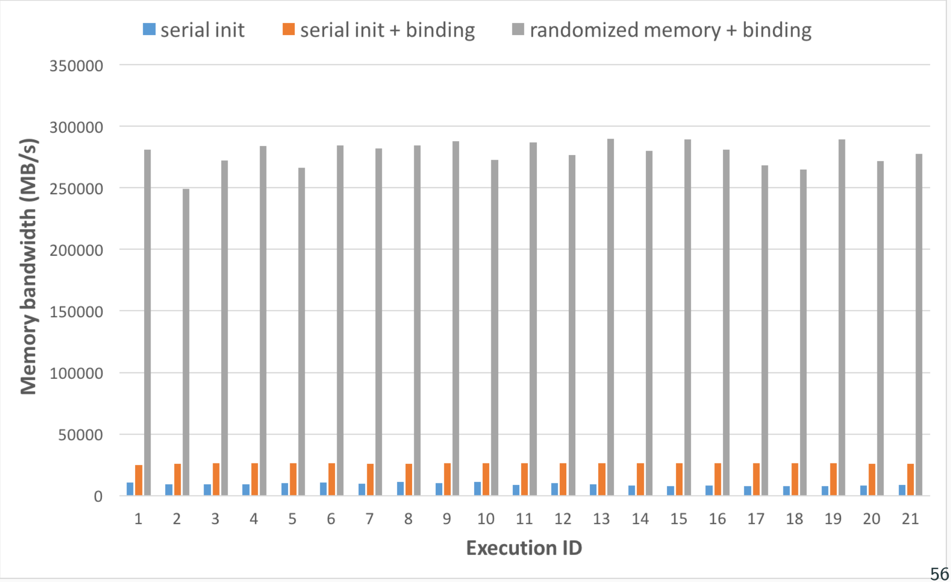
STREAM: Performance on Intel192 (1/4)



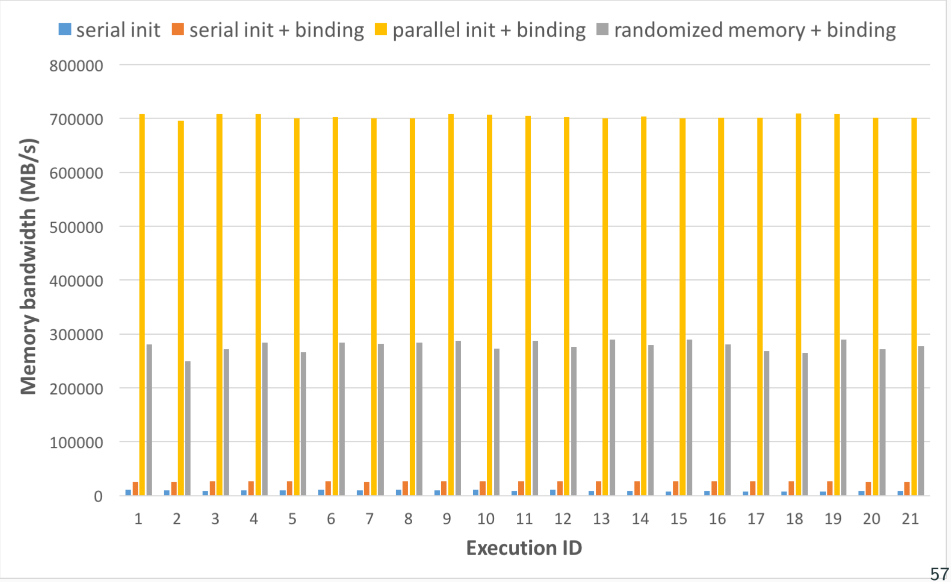
STREAM: Performance on Intel192 (2/4)



STREAM: Performance on Intel192 (3/4)



STREAM: Performance on Intel192 (4/4)



Part III: Parallelism Grain and Runtime-related Overheads

Some Raw Performance of OpenMP Implementations

The OpenMP **runtime system** is responsible for the low-level implementation of the OpenMP constructs, like managing the threads or executing the tasks for example.

Remember that nothing comes for free and it's up to you to keep these overheads under control.

Construct	GNU OMP	Intel OMP
parallel	372	26
for	121	19
parallel for	370	26
barrier	121	19
single	227	35
critical	6	1.5
lock/unlock	7.5	1.5
atomic	0.5	0.3
reduction	435	47

Figure 14: Overheads, in μs, of various OpenMP constructs (EPCC microbenchmark) on the Intel192 machine

Case Study #3: A Task-Based Cholesky Decomposition

The algorithm works on tiles (also called *blocks*) and involves the following BLAS kernels:

- `potrf`: Cholesky decomposition
- `trsm`: solving triangular matrix with multiple right hand sides
- `syrk`: symmetric rank-k update to a matrix
- `gemm`: matrix matrix multiply

`potrf`			
`trsm`	syrk		
`trsm`	gemm	syrk	
`trsm`	gemm	gemm	syrk

Figure 15: Cholesky workflow on a 4x4 tiled matrix

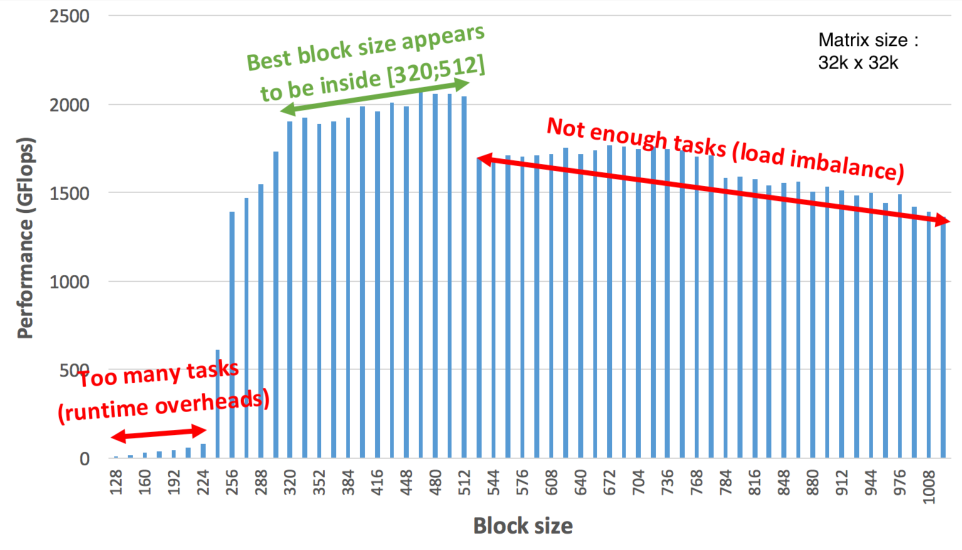
1. execute `potrf` on tile (1,1). This unlocks some `trsm` tasks to be executed ;
2. each `trsm` unlocks one `syrk` and some `gemm` tasks
3. repeat these steps on the lower 3x3 matrix starting with tile (2,2)

Cholesky Implementation using OpenMP 4.0 Dependent Tasks

```
for (int k = 0; k < NB; ++k) {
    #pragma omp task shared(A) depend(inout: A[k][k])
    dpotrf(NB, &A[k][k]);

    for (int m = k; m < NB; ++m) {
        #pragma omp task shared(A)               \
            depend(in: A[k][k])                  \
            depend(inout: A[m][k])
        dtrsm(NB, &A[k][k], &A[m][k]);
    }

    for (int m = k; m < NB; ++m) {
        #pragma omp task shared(A)               \
            depend(in: A[m][k])                  \
            depend(inout: A[m][m])
        dsyrk(NB, &A[m][k], &A[m][m]);

        for (int n = k; n < m; ++n)
            #pragma omp task shared(A)           \
                depend(in: A[m][k],A[n][k])      \
                depend(inout: A[m][n])
            dgemm(NB, &A[m][k], &A[n][k], &A[m][n]);
    }
}
```

- **NB** is the number of tiles of the matrix
- **A** is a matrix of pointers, each element of A points to a different tile of the matrix

Cholesky Performance Depending on the Block Size



Matrix size : 32k x 32k

Best block size appears to be inside [320;512]

Not enough tasks (load imbalance)

Too many tasks (runtime overheads)

Conclusion

Conclusion

OpenMP provides some high-level constructs to :

- run loops in parallel (OpenMP 2.5)
- express task-based parallelism (OpenMP 3.0)
- express task dependencies (OpenMP 4.0)
- offload computations on accelerators (OpenMP 4+)

Getting parallel applications up to speed still require a good understanding of both software and hardware layers, in order to

- make your code cache-friendly, when possible
- control data placement to avoid NUMA-related penalties
- keep the runtime-related overheads at bay

Thank you!