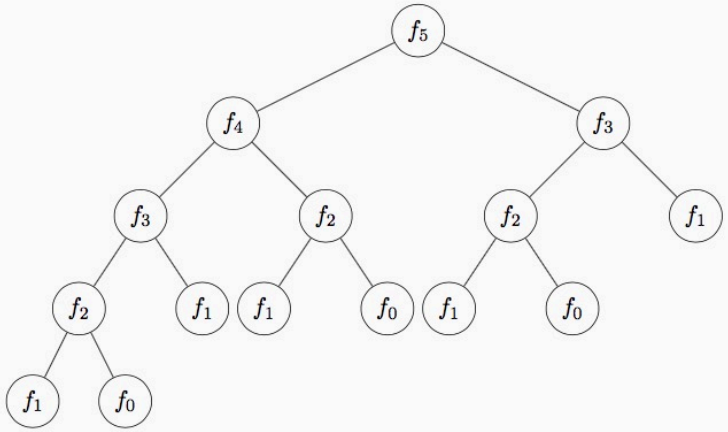
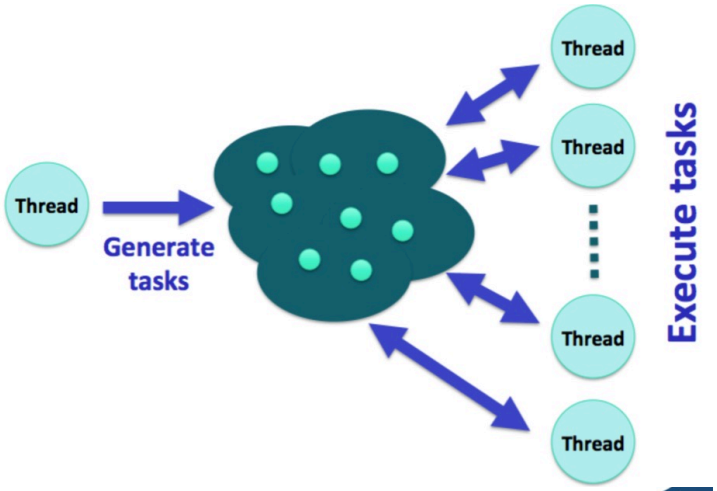
Extending the Scope of OpenMP with Task Parallelism

- **What**if my application was not written in a loop-based fashion?

```
1  int fib(int n) {
2      int i, j;
3      if (n < 2) {
4          return n;
5      } else {
6          i = fib(n - 1)
       ;
7          j = fib(n - 2)
       ;
8          return i + j;
9      }
10 }
```



- **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

- The application programmer specifies 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

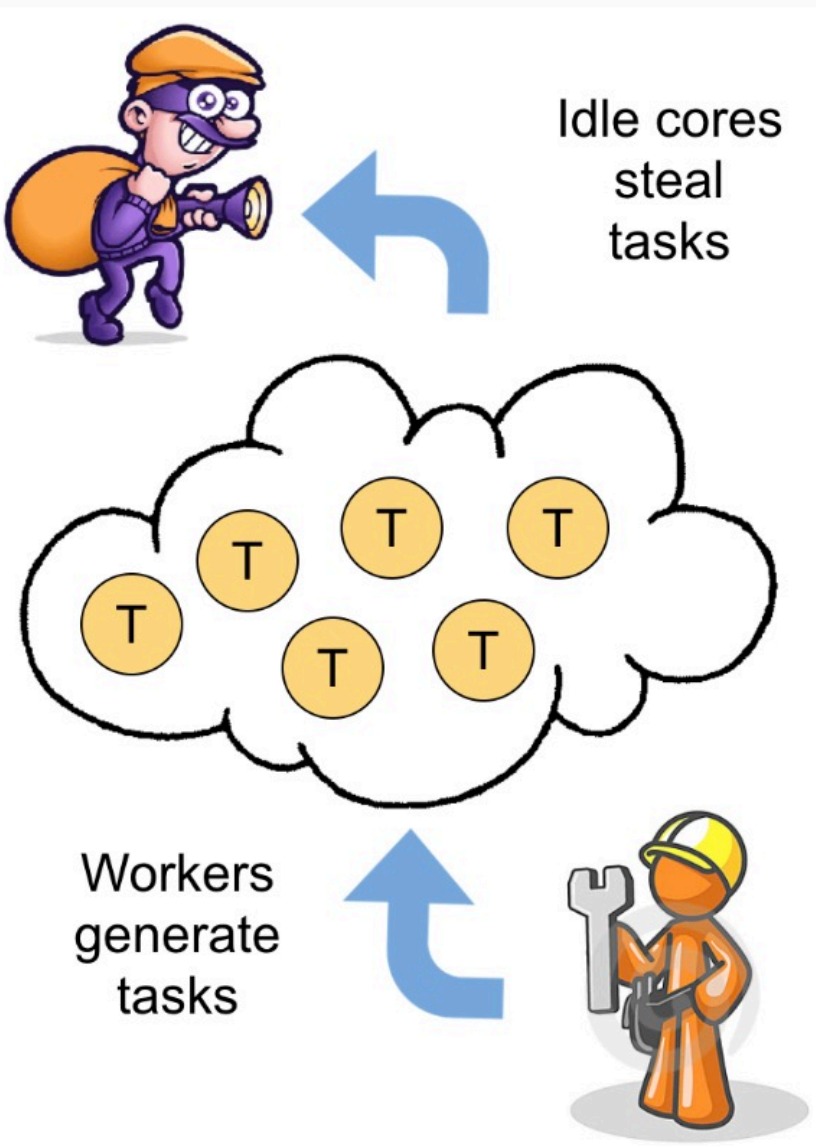
```
1  int main(void) {
2      ...
3      #pragma omp parallel
4      {
5          #pragma omp single
6          res = fib(50);
7      }
8      ...
9  }
10
11 int fib(int n) {
12     int i, j;
13     if (n < 2) {
14         return n;
15     } else {
16         #pragma omp task
17         i = fib(n - 1);
18         #pragma omp task
19         j = fib(n - 2);
20         #pragma omp task
       wait
21         return i + j;
22     }
23 }
```

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.



First OpenMP Tasking Experience

```
1 int main(void)
2 {
3     printf("A ");
4     printf("race ");
5     printf("car ");
6
7     printf("\n");
8     return 0;
9 }
```

- 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
```

First OpenMP Tasking Experience, contd.

```
1  int main(void)
2  {
3      #pragma omp parallel
4      {
5          printf("A ");
6          printf("race ");
7          printf("car ");
8      }
9
10     printf("\n");
11     return 0;
12 }
```

- 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-2
$ A race A race car car
```

First OpenMP Tasking Experience, contd.

```
1  int main(void)
2  {
3      #pragma omp parallel
4      {
5          #pragma omp single
6          {
7              printf("A ");
8              #pragma omp
       task
9              printf("race ")
       ;
10             #pragma omp
       task
11             printf("car ");
12         }
13     }
14
15     printf("\n");
16     return 0;
17 }
```

- 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
```

First OpenMP Tasking Experience, contd.

```
1  int main(void)
2  {
3      #pragma omp parallel
4      {
5          #pragma omp single
6          {
7              printf("A ");
8              #pragma omp task
9              printf("race ");
10             #pragma omp task
11             printf("car ");
12
13             printf("is fun ");
14             printf("to watch "
       );
15         }
16     }
17
18     printf("\n");
19     return 0;
20 }
```

- 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
```

First OpenMP Tasking Experience, contd.

```
1  int main(void)
2  {
3      #pragma omp parallel
4      {
5          #pragma omp single
6          {
7              printf("A ");
8              #pragma omp task
9              printf("race ");
10             #pragma omp task
11             printf("car ");
12             #pragma omp task
       wait
13             printf("is fun ");
14             printf("to watch "
       );
15         }
16     }
17
18     printf("\n");
19     return 0;
20 }
```

- 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.

```
1  void data_flow_example (void)
2  {
3      type x, y;
4
5      #pragma omp parallel
6      #pragma omp single
7      {
8          #pragma omp task
9          write_data(&x);   // Task A
10         #pragma omp task
11         write_data(&y);   // Task B
12
13         #pragma omp taskwait
14
15         #pragma omp task
16         print_results(x); // Task
       C
17         #pragma omp task
18         print_results(y); // Task
       D
19     }
20 }
```

OpenMP Tasks Dependencies: Rationale

- The **depend** clause allows you to provide information on the way a task will access data.
- It 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

```
1  void data_flow_example (void)
2  {
3      type x, y;
4
5      #pragma omp parallel
6      #pragma omp single
7      {
8          #pragma omp task depend(out:
       x)
9          write_data(&x);   // Task A
10         #pragma omp task depend(out:
       y)
11         write_data(&y);   // Task B
12
13         #pragma omp task depend(in: x
       )
14         print_results(x); // Task C
15         #pragma omp task depend(in: y
       )
16         print_results(y); // Task D
17     }
18 }
```

- 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

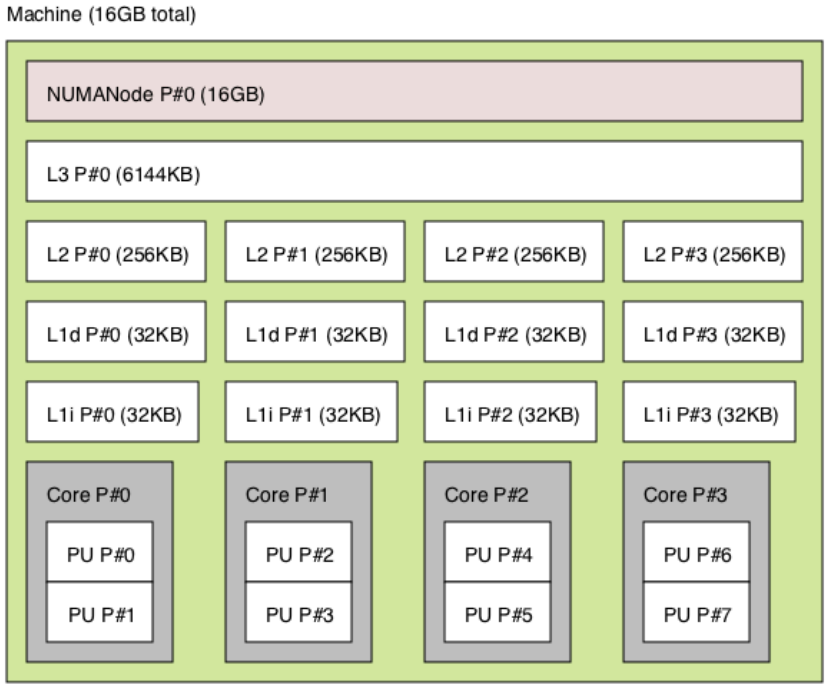
- Expressing dependencies sometimes helps unlocking more parallelism

Speeding up OpenMP applications

- **Preambule:** Have a closer look at your favorite/target 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?

Getting to Know Your Platform with hwloc

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

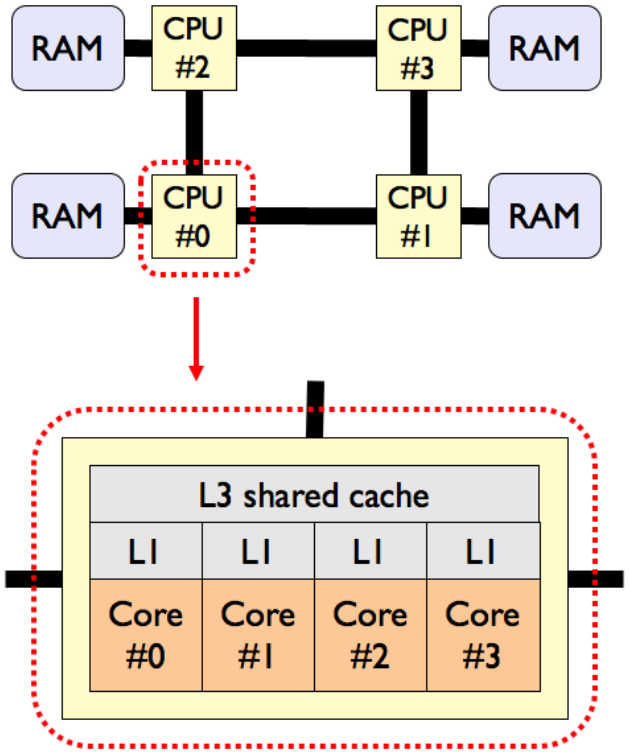


- Provides information about
  - the processing units (logical/physical cores)
  - the cache hierarchy
  - the memory hierarchy (NUMA nodes)
  - However, **hwloc does not provide the entire architecture topology** (the way processors are connected together).

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

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)

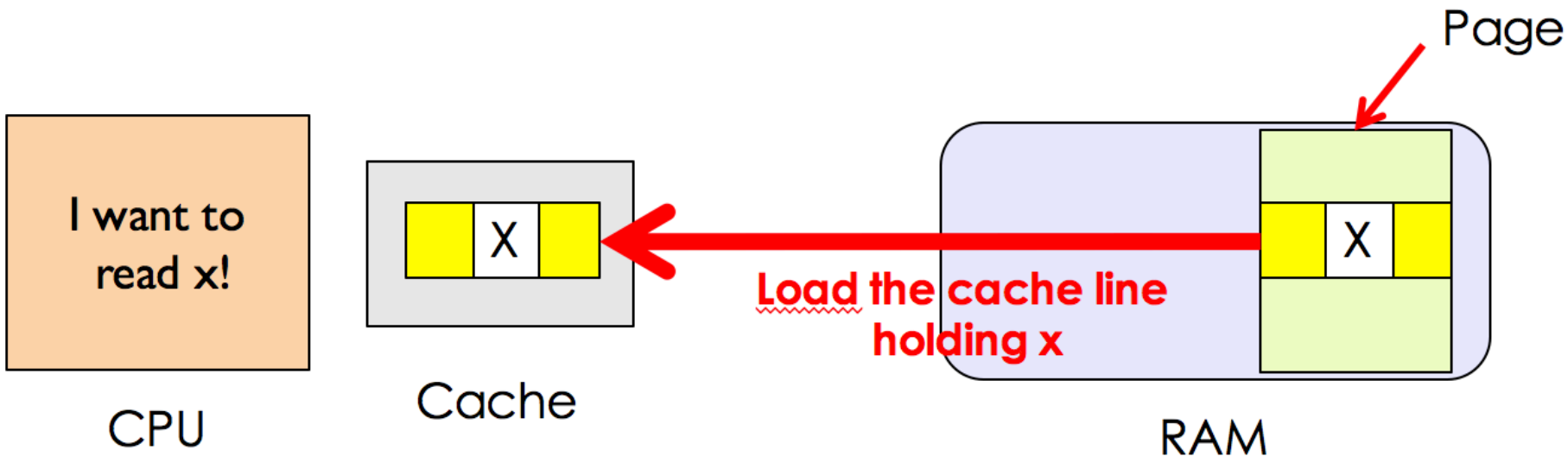


	N_0	N_1	N_2	N_3
N_0	0	10	10	20
N_1	10	0	20	10
N_2	10	20	0	10
N_3	20	10	10	0

A 4-nodes NUMA machine with the corresponding NUMA distance table

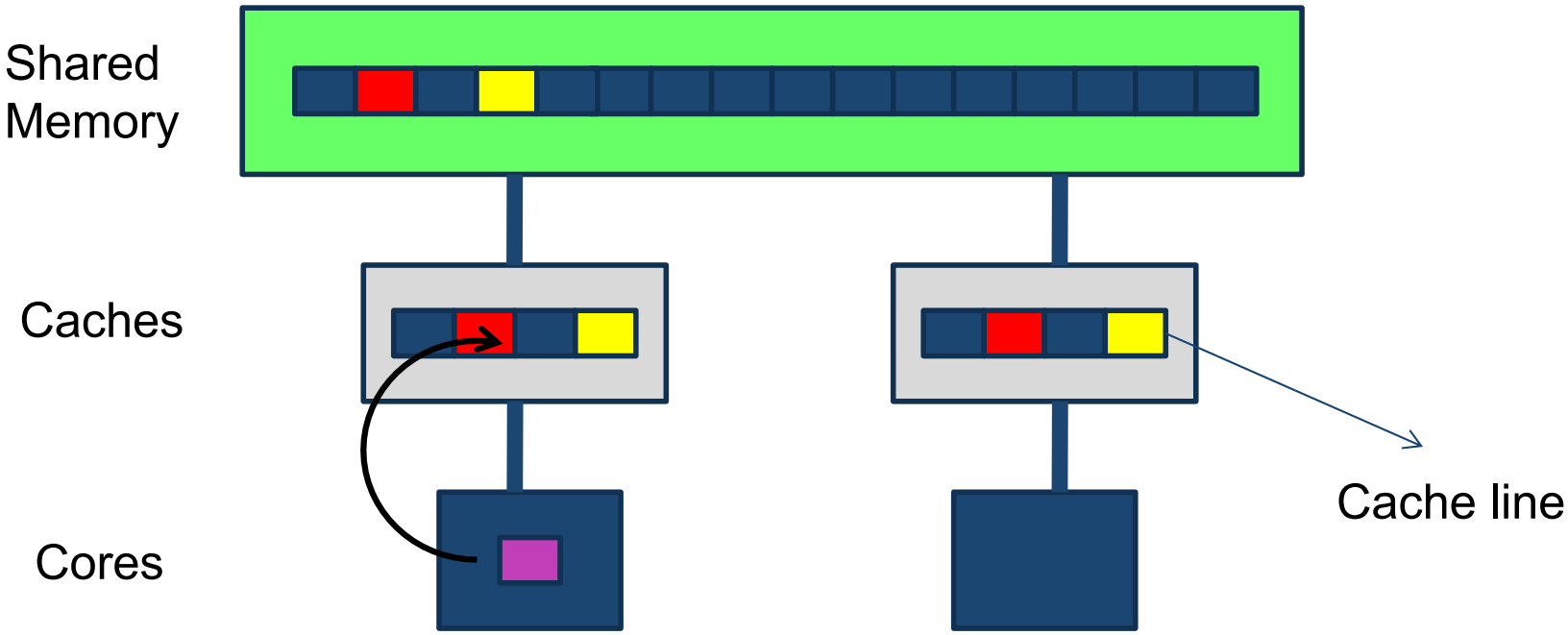
Cache Memory: Basic Concept

- 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**).



Performance: False Sharing effect

Cache coherency and the negative effect of false sharing can have a big impact on performance



Charging a line of a shared cache invalids the other copies of this line

Performance: False Sharing effect, contd.

Using data structures in memory may lead to a decrease of performance and a lack of scalability

- To get performance, use the cache
- If several cores manipulate different data items close in memory, the update of individual elements leads a load of a line of cache (to keep coherency with the main memory)

False sharing leads to bad performance when the following conditions are met

- Shared data a modified on different cores
- Several threads on different cores update data which are located on the same cache line
- These updates appear simultaneously and frequently

Performance: False Sharing effect, contd.

When data are only read, we don't get false sharing

It can be avoided (or reduced) by

- Privatizing variables
- Increasing the array size or by using "padding"
- By increasing the packet size (modifying the way loop iterations are shared between threads)

Performance: False Sharing effect, contd.

```
Integer, dimension(n) :: a
…
!$OMP PARALLEL DO SHARED(nthreads,a) SCHEDULE(static,1)
            DO i=0, nthreads-1
                        a(i) = i
    END DO
!$OMP END PARALLEL DO
```

Nthreads : # threads executing the loop
If we suppose that every thread possess a copy of a in his local cache
Packet size of 1 leads to a false sharing phenomenon for each update

If a cache line can contain **C** elements of vector **a**, we can sole the problem by extending artificially the array dimensions (array padding)
We declare an array **a(C,n)** and remplace **a(i)** by **a(1,i)**

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 $C_0 = A_0B_0 + A_1B_4 + A_2B_8 + A_3B_{12}$

Let's implement this using what we've learned about OpenMP!

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

```
1  void gemm_omp(double *A, double *B, double *C, int n) {
2      #pragma omp parallel
3      {
4          int i, j, k;
5          #pragma omp for
6          for (i=0; i<n; i++) {
7              for (j=0; j<n; j++) {
8                  for (k=0; k<n; k++) {
9                      C[i*n+j] += A[i*n+k]*B[k*n+j];
10                 }
11             }
12         }
13     }
14 }
```

On the Intel192 machine

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?

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

$$C_0 = A_0B_0 + A_1B_4 + A_2B_8 + A_3B_{12}$$

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?

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

$$C_0 = A_0B_0 + A_1B_4 + A_2B_8 + A_3B_{12}$$

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?

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

$$C_0 = A_0B_0 + A_1B_4 + A_2B_8 + A_3B_{12}$$

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?

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

$$C_0 = A_0B_0 + A_1B_4 + A_2B_8 + A_3B_{12}$$

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?

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

$$C_0 = A_0B_0 + A_1B_4 + A_2B_8 + A_3B_{12}$$

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

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

$C_0 = A_0B_0$
$+ A_1B_4$
$+ A_2B_8$
$+ A_3B_{12}$

Performance evaluation on Intel192:

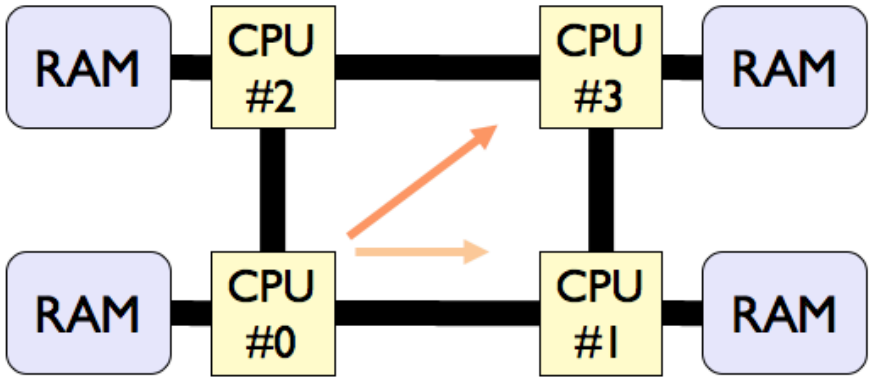
Serial time: 5.188652s (prev: 40.3018250s)

Parallel time: 0.067657s (prev: 0.270773s)

Achieved speed-up : 77 (rel to prev: 595(!))

Control Data Placement on NUMA Systems

Non-Uniform Memory Accesses (NUMA)



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**

STREAM Benchmark

```
1  void STREAM_Triad(double *a,
2                    double *b,
3                    double *c,
4                    double
       scalar)
5  {
6    int j;
7  #pragma omp parallel for
8    for (j=0; j<N; j++)
9        a[j] = b[j]+scalar*c[j];
10 }
```

- 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 specification 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:

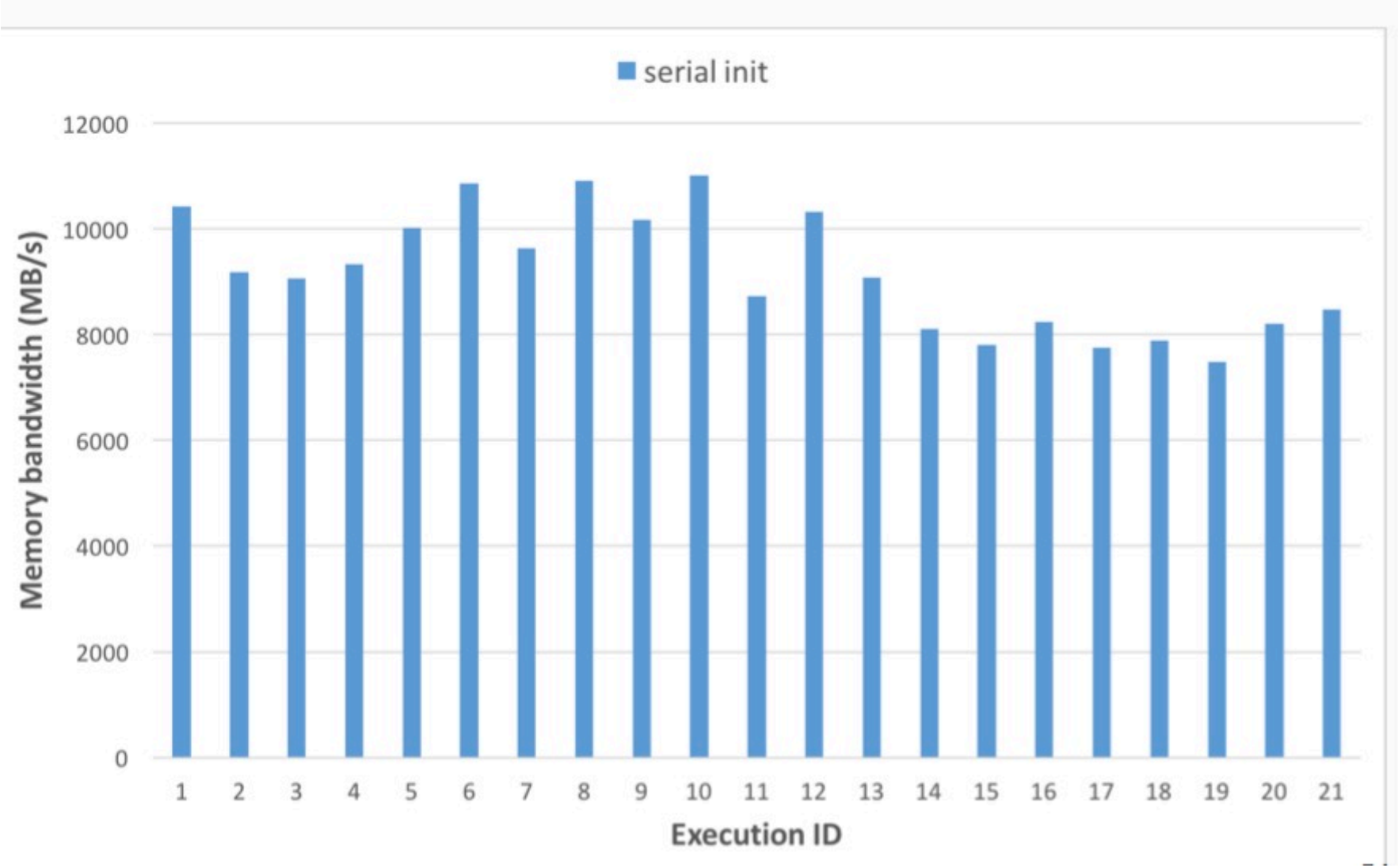
     `OMP PLACES="0,1,2,3"`

   - 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)

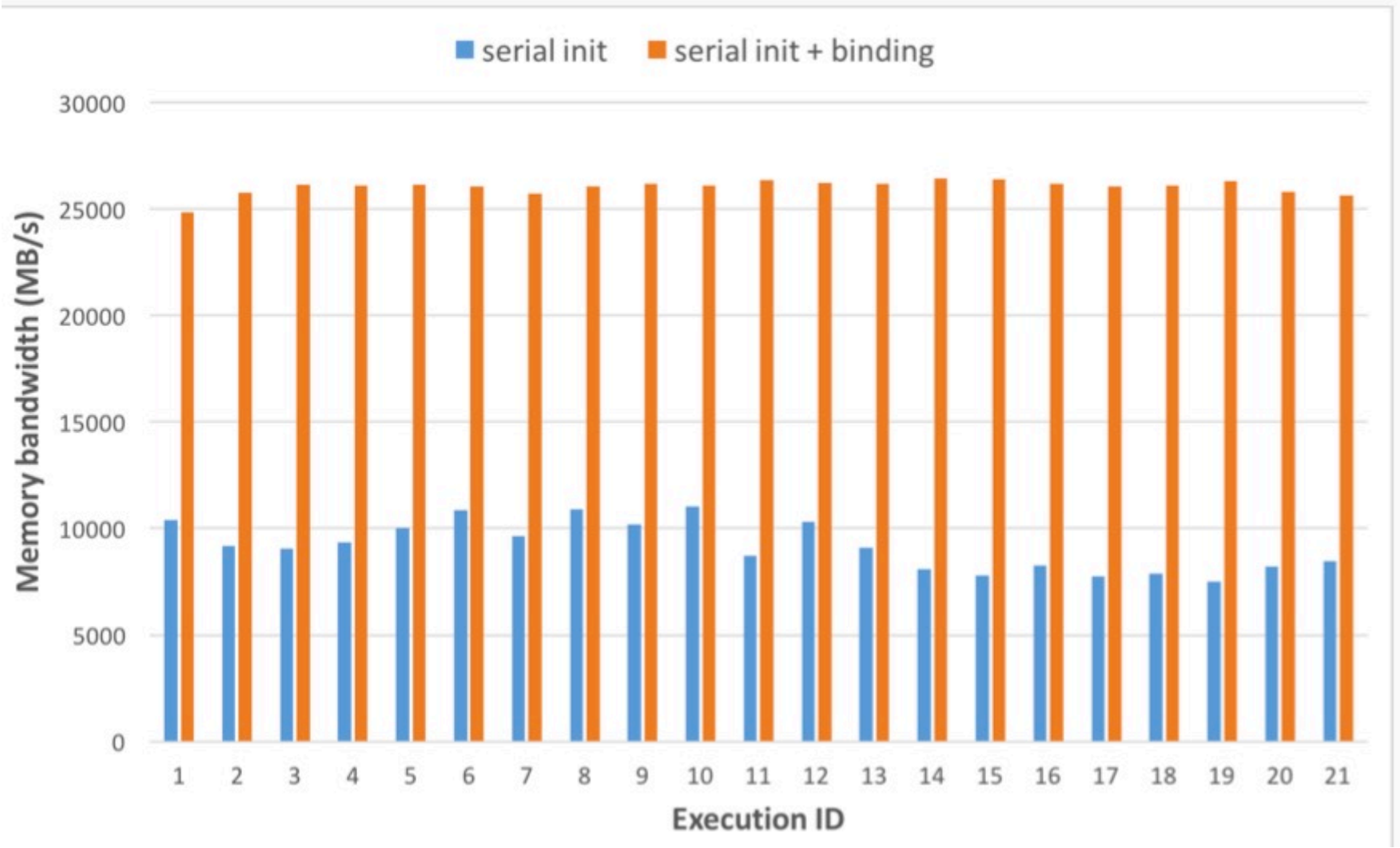
STREAM: Evaluation

- We ran the STREAM benchmark 20 times on Intel192 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

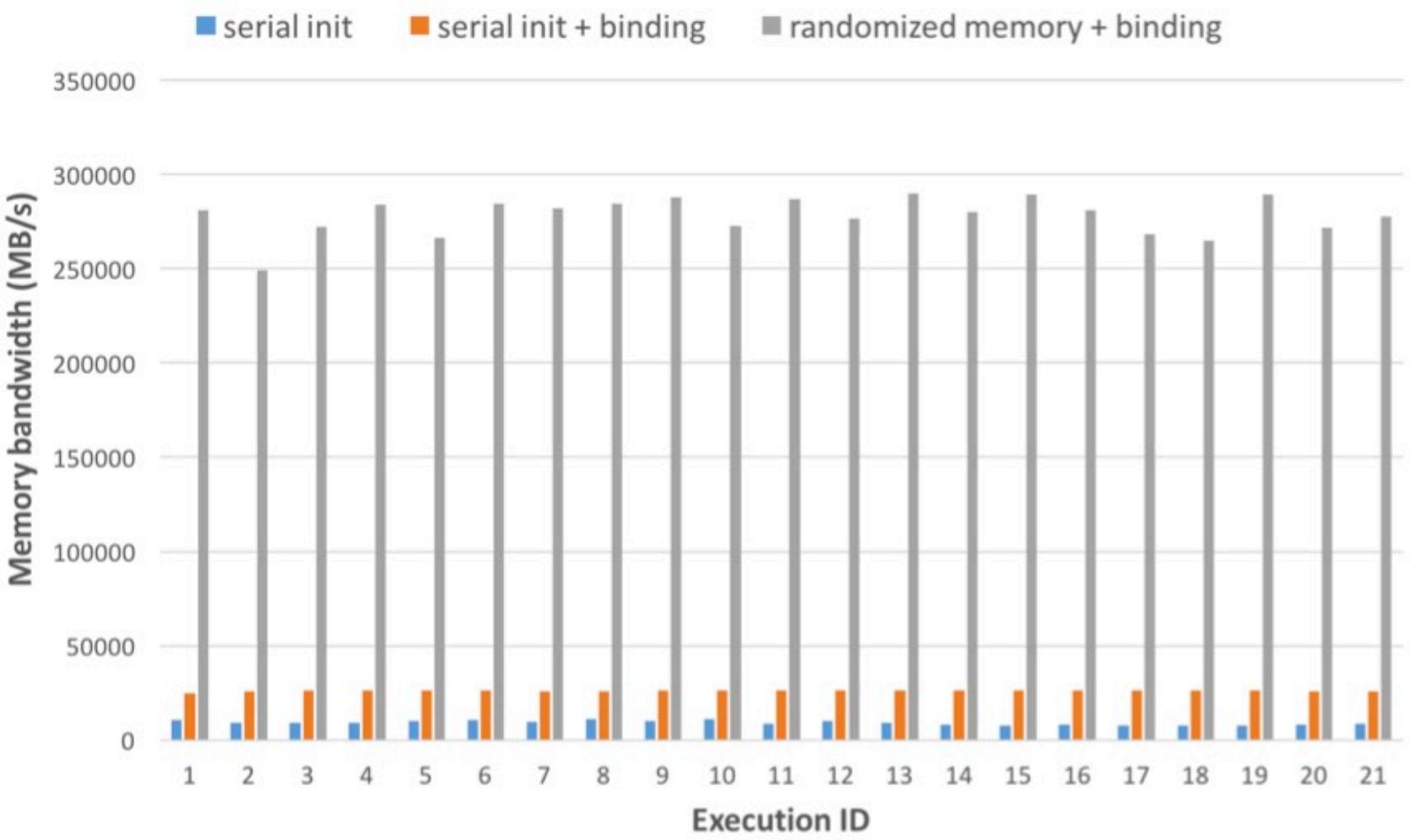
serial init


serial init
12000
10000
8000
6000
4000
2000
0
Memory bandwidth (MB/s)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Execution ID

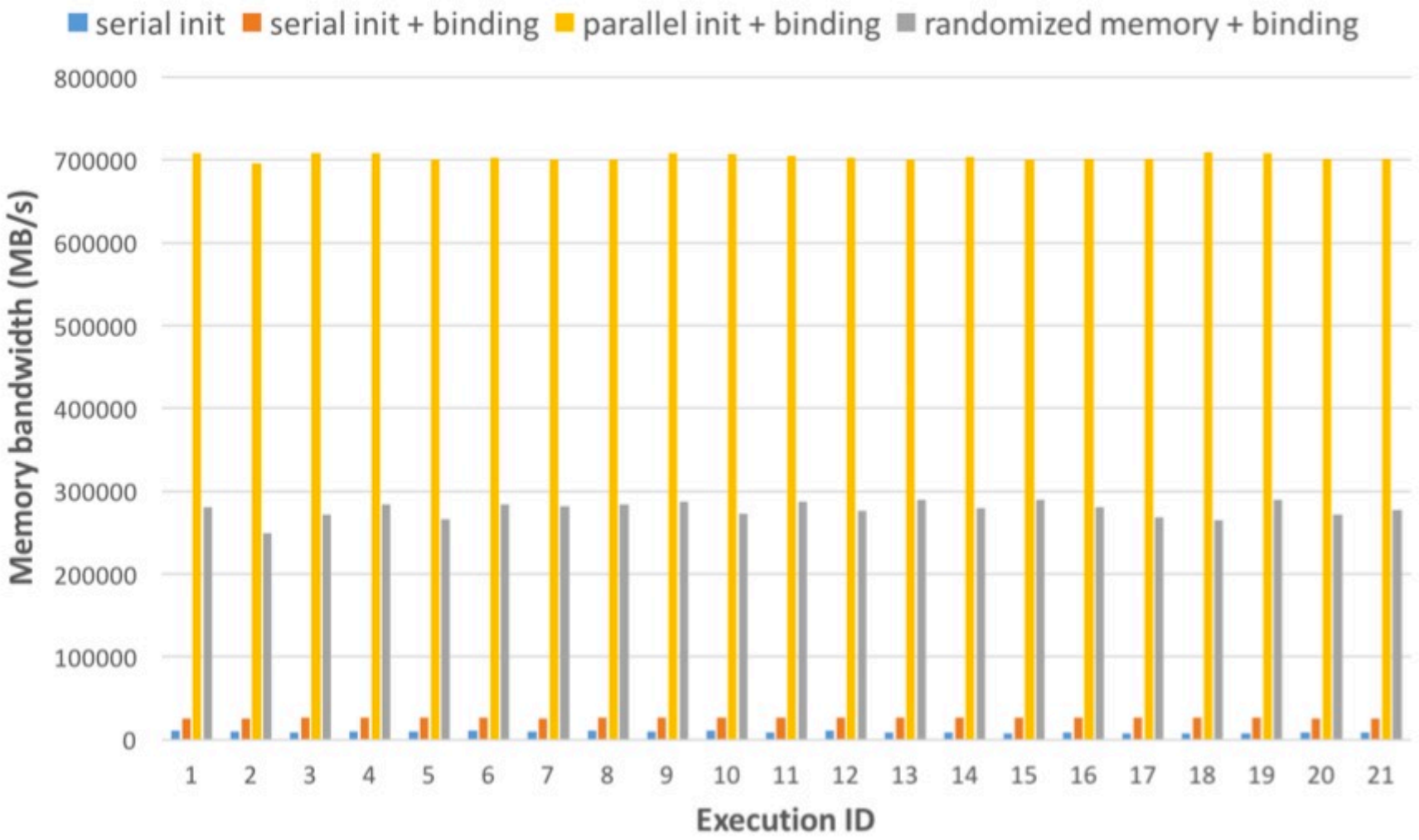
serial init + binding



randomized memory + binding



parallel init + binding



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

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

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

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

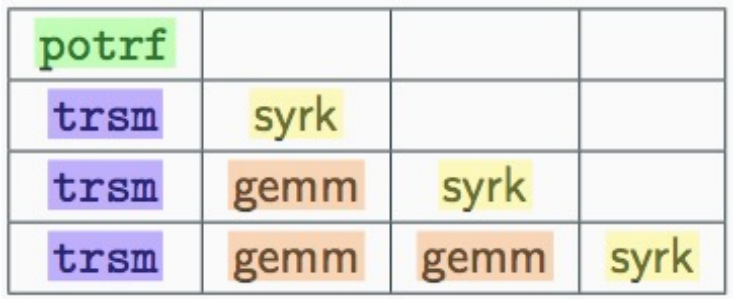
A Task-Based Cholesky Decomposition

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 multiplication

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)

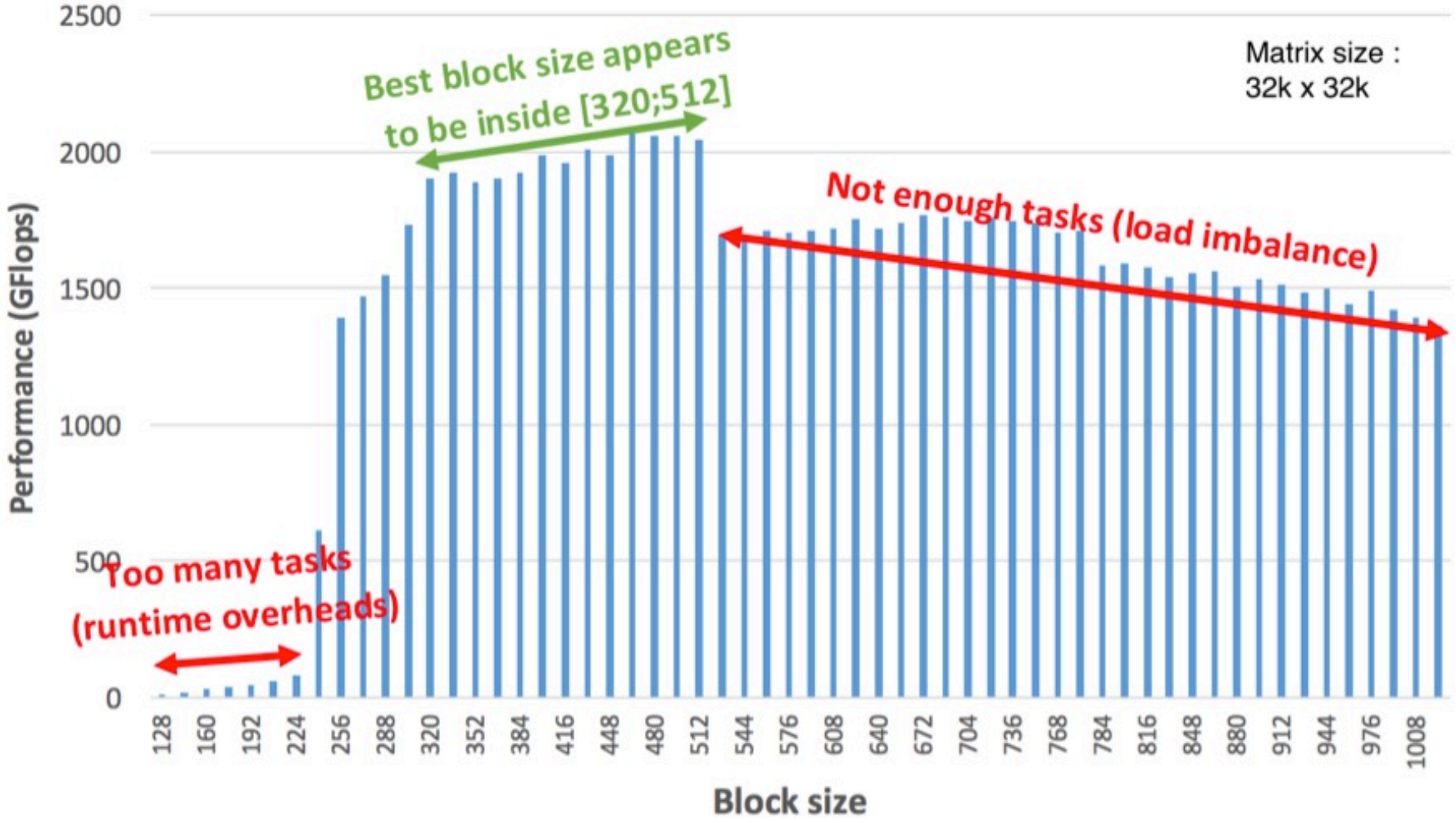


Cholesky Implementation using OpenMP 4.0 Dependent Tasks

```
1  for (int k = 0; k < NB; ++k) {
2      #pragma omp task shared(A) depend(inout: A[k][k
       ])
3      dpotrf(NB, &A[k][k]);
4
5      for (int m = k; m < NB; ++m) {
6          #pragma omp task shared(A)                \
7              depend(in: A[k][k])                   \
8              depend(inout: A[m][k])
9          dtrsm(NB, &A[k][k], &A[m][k]);
10     }
11
12     for (int m = k; m < NB; ++m) {
13         #pragma omp task shared(A)                \
14             depend(in: A[m][k])                   \
15             depend(inout: A[m][m])
16         dsyrk(NB, &A[m][k], &A[m][m]);
17
18         for (int n = k; n < m; ++n)
19             #pragma omp task shared(A)            \
20                 depend(in: A[m][k],A[n][k])       \
21                 depend(inout: A[m][n])
22             dgemm(NB, &A[m][k], &A[n][k], &A[m][n])
       ;
23     }
```

- **NB** is the number 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



OpenMP

- **It 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+)

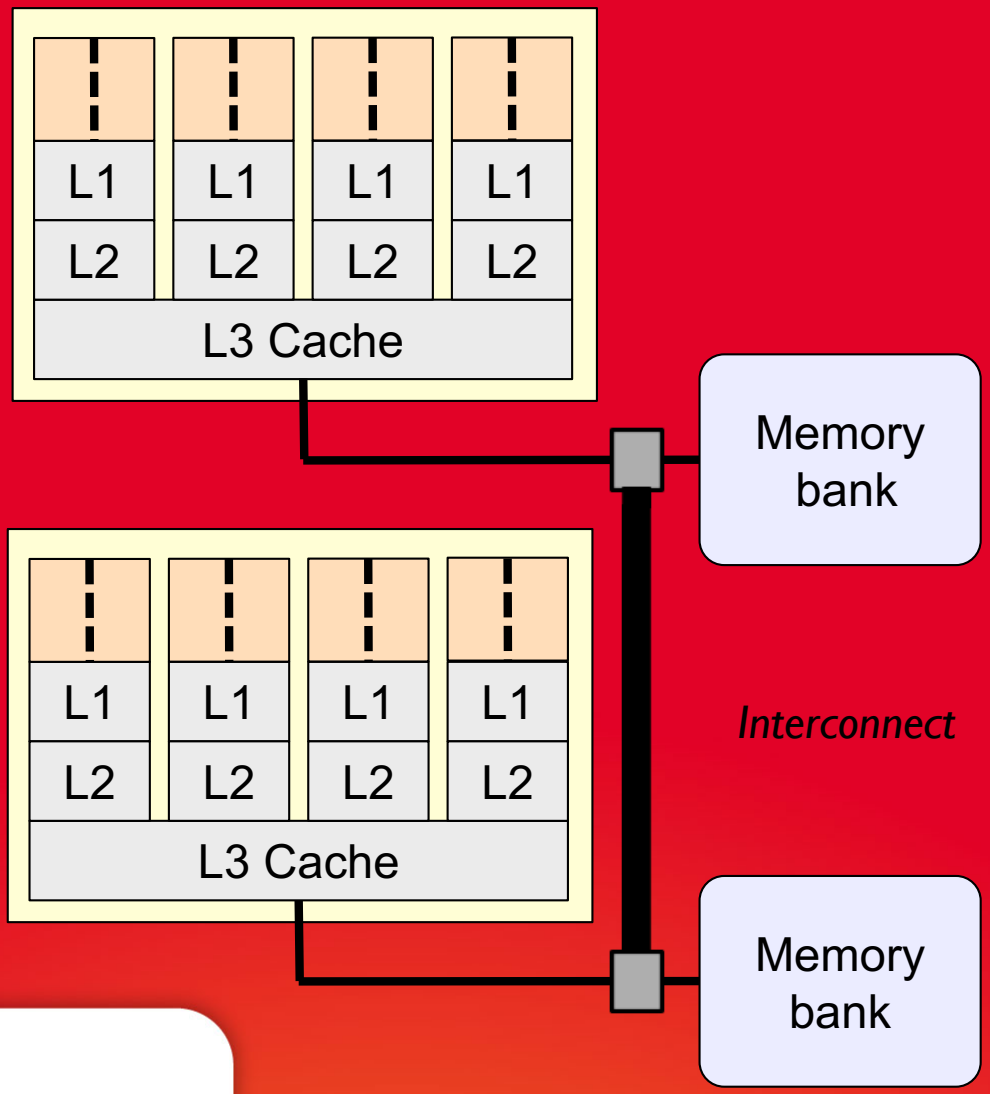
- **Getting parallel applications up to speed still require a good understanding of both software and hardware layers, in order to**
  - make your code cache-friendly, when possible
  - control data placement to avoid NUMA-related penalties
  - keep the runtime-related overheads at bay

OpenMP

- Requires a shared memory multi-processor
- Relatively easy implementation, even in a sequential program
- Allows the progressive parallelization of a sequential program
- The potential for parallel performance lies in parallel regions
- Within these parallel regions, work can be shared through loops and parallel sections
- But it is also possible to singularise a task for a particular job
- Explicit global or point-to-point synchronizations are sometimes required in parallel regions
- Particular care must be taken in defining the status of variables used in a construction
- Task parallelism for irregular computations

OpenMP versus MPI

- OpenMP uses memory common to all processes
- All communication is done by reading and writing in this memory (and using synchronization mechanisms)
- MPI is a library of routines allowing the communication between different processes (located on different machines or not), the communications are made by explicit sends or receives of messages
- For the programmer:
  - rewrite the code with MPI,
  - simple addition of directives in the sequential code with OpenMP
- Possibility of mixing the two approaches in the case of cluster of machines with multi-core nodes (hybrid programming)
- Choice strongly dependent: the machine, the code, the time that the developer wants to dedicate and the expected gain
- Superior scalability with MPI



INVENTEURS DU MONDE NUMÉRIQUE