RWTH AACHEN UNIVERSITY

Introduction to OpenMP

part II of III

Christian Terboven <terboven@rz.rwth-aachen.de>

30.07.2013 / Aachen, Germany

Stand: 22.07.2013

Version 2.3

Review of OpenMP Introduction

- **OpenMP is a parallel programming model for Shared-Memory machines. That is, all threads have access to a *shared* main memory. In addition to that, each thread may have *private* data.**

- **The parallelism has to be expressed explicitly by the programmer. The base construct is a *Parallel Region*:**
  A *Team* of threads is provided by the runtime system.

- **Using the *Worksharing* constructs, the work can be distributed among the threads of a team. The *Task* construct defines an explicit task along with it‘s data environment. Execution may be deferred.**

- **To control the parallelization, mutual exclusion as well as thread and task synchronization constructs are available.**

Agenda

- **More Examples**
  - PI
  - Fibonacci
  - Scoping with Tasks
  - Task Synchronization
- **Runtime Library and Environment Variables**
- **The *Schedule* Clause**
- **OpenMP and the Machine Architecture (Thread Binding)**
- **How to build a (simple) Performance Model**

More Examples

PI

Example: Pi (1/2)

- **Simple example: calculate Pi by integration**

```
double f(double x) {
   return (double)4.0 / ((double)1.0 + (x*x));
}

void computePi() {
   double h = (double)1.0 / (double)iNumIntervals;
   double sum = 0, x;

...
   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }

   myPi = h * sum;
}
```

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

Example: Pi (1/2)

- **Simple example: calculate Pi by integration**

```
double f(double x) {
   return (double)4.0 / ((double)1.0 + (x*x));
}

void computePi() {
   double h = (double)1.0 / (double)iNumIntervals;
   double sum = 0, x;

#pragma omp parallel for private(x) reduction(+:sum)
   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }

   myPi = h * sum;
}
```

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

Example: Pi (2/2)

- **Results (with C++ version):**

# Threads	Runtime [sec.]	Speedup
1	1.11	1.00
2		
4		
8	0.14	7.93

- **Scalability is pretty good:**
  - About 100% of the runtime has been parallelized.
  - As there is just one parallel region, there is virtually no overhead introduced by the parallelization.
  - Problem is parallelizable in a trival fashion ...

Fibonacci

Recursive approach to compute Fibonacci

```
int main(int argc,
         char* argv[])
{
   [...]
   fib(input);
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return x+y;
}
```

- **On the following slides we will discuss three approaches to parallelize this recursive code with Tasking.**

First version parallelized with Tasking (omp-v1)

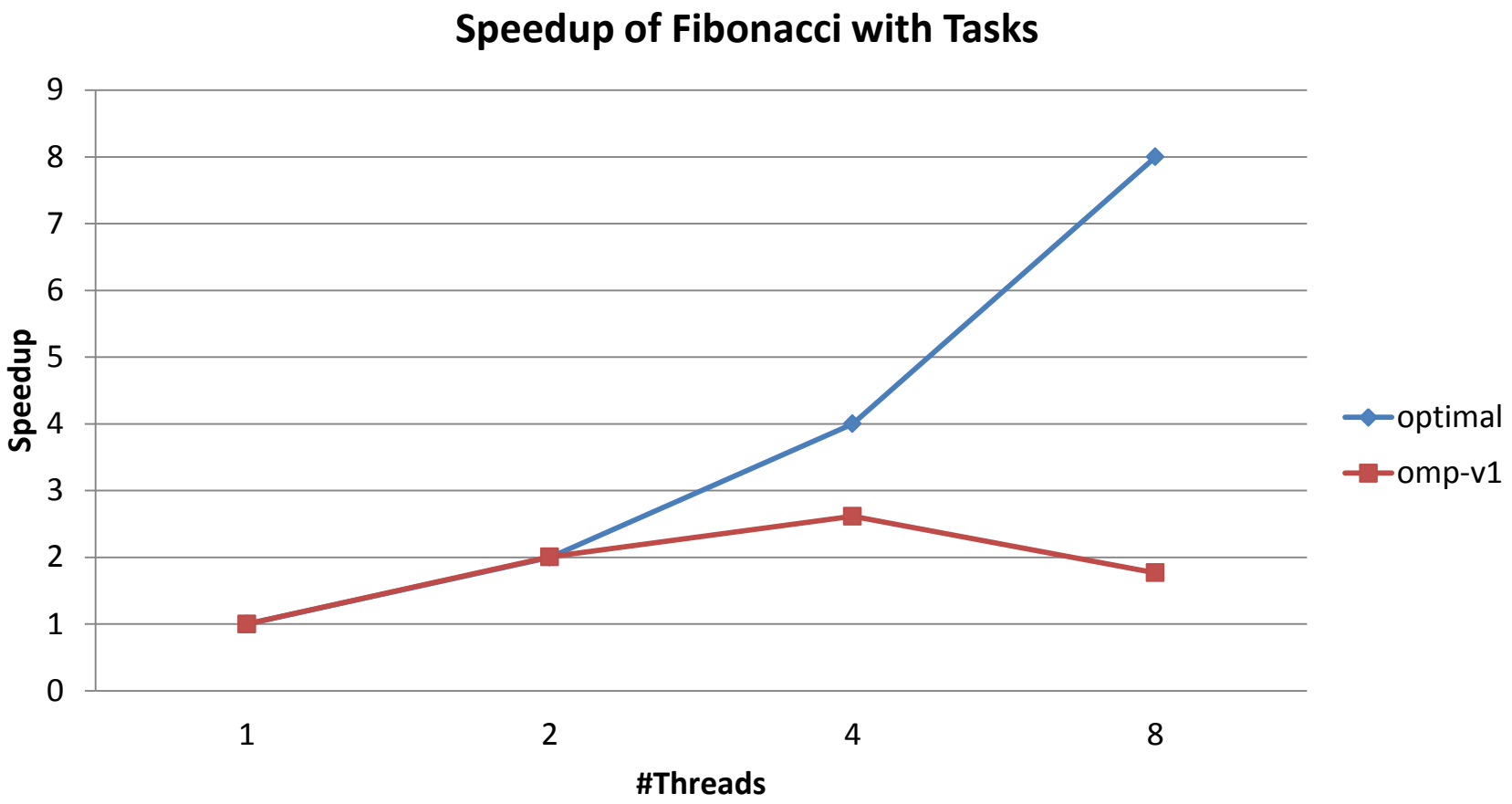
```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

- Only one Task / Thread enters `fib()` from `main()`, it is responsable for creating the two initial work tasks
- Taskwait is required, as otherwise `x` and `y` would be lost

Scalability measurements (1/3)

- Overhead of task creation prevents better scalability!



Improved parallelization with Tasking (omp-v2)

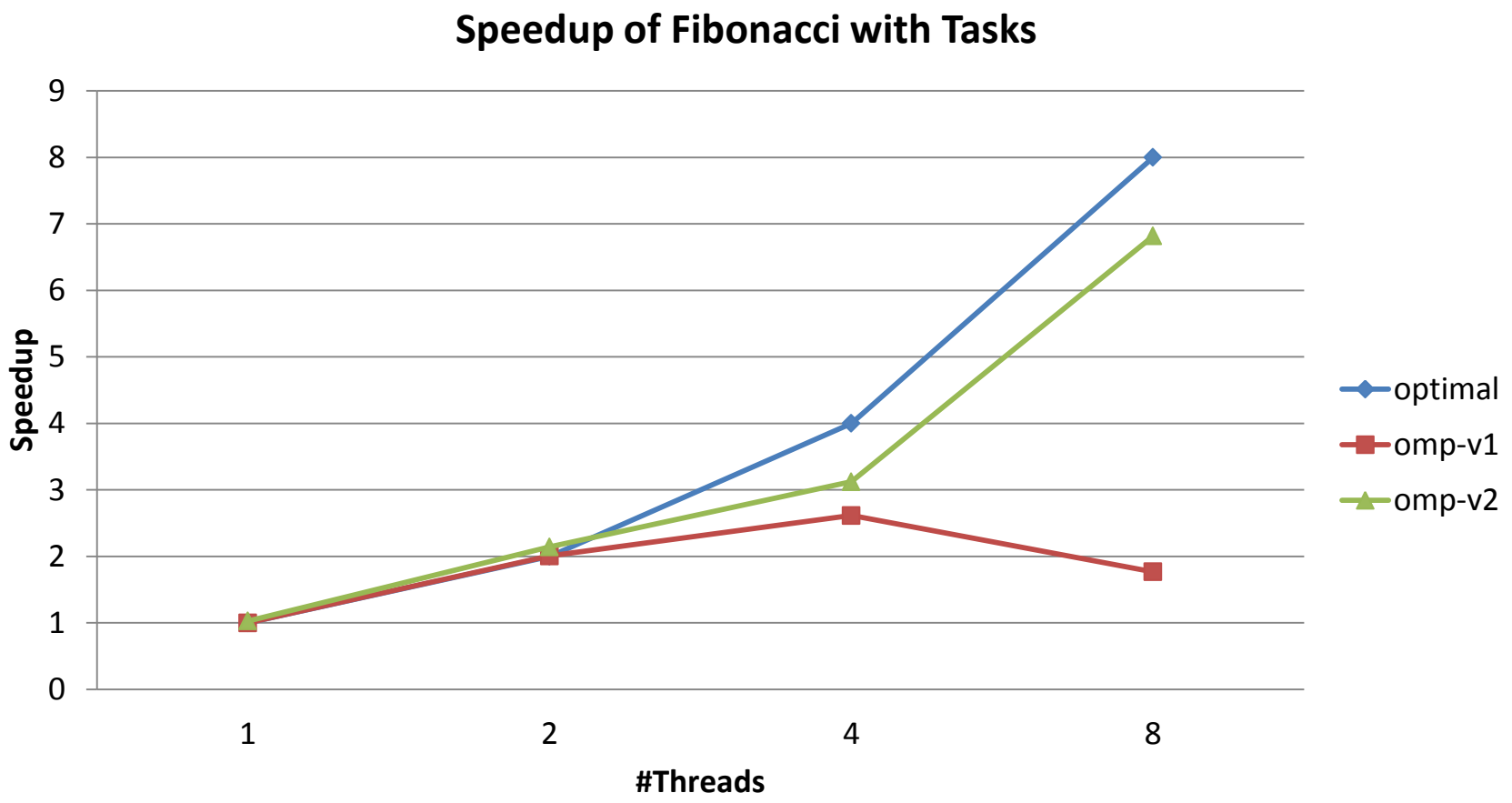
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
int x, y;
#pragma omp task shared(x) \
   if(n > 30)
{
   x = fib(n - 1);
}
#pragma omp task shared(y) \
   if(n > 30)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

Scalability measurements (2/3)

- Speedup is ok, but we still have some overhead when running with 4 or 8 threads



Improved parallelization with Tasking (omp-v3)

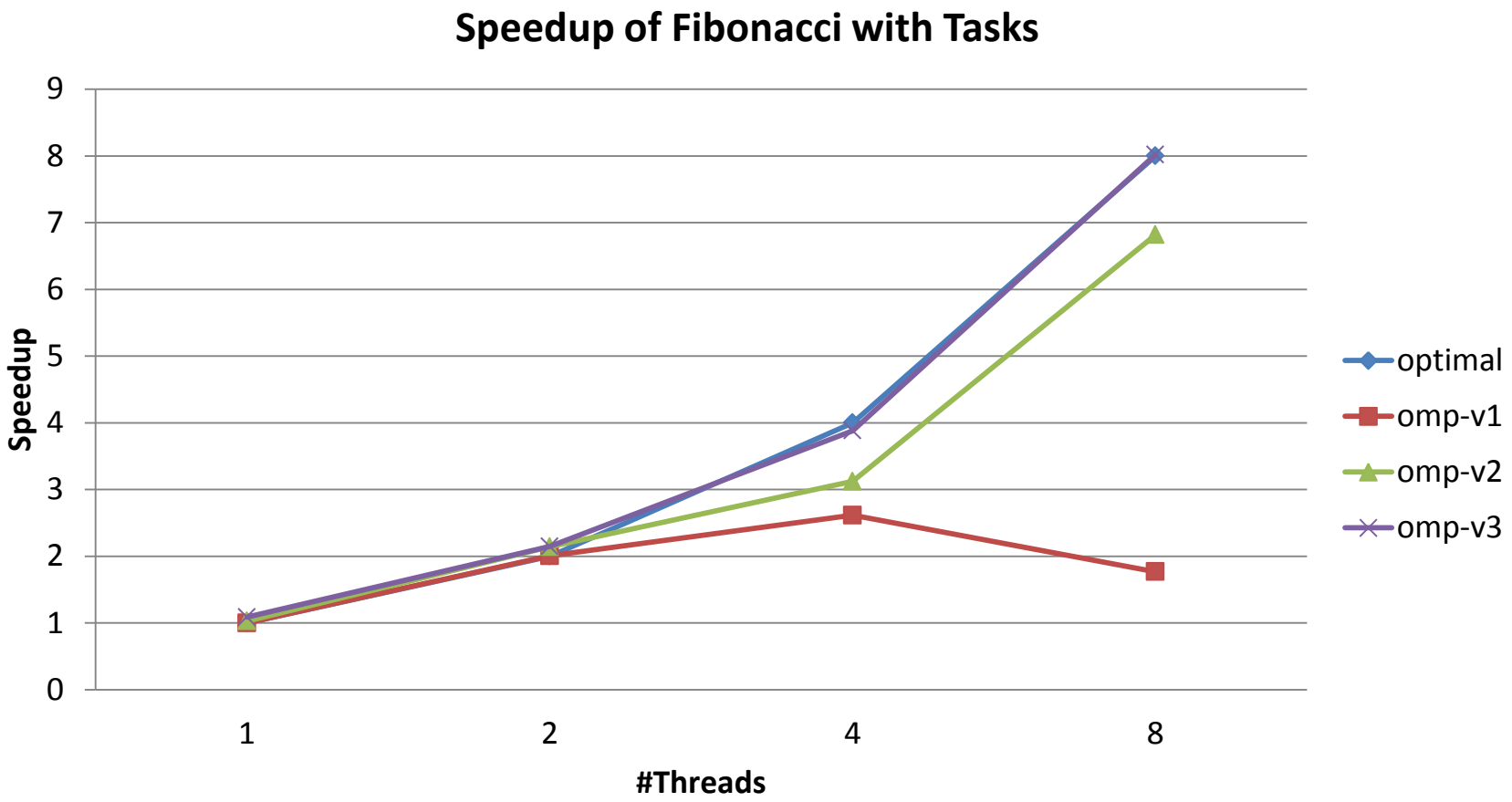
- **Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)**

```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
   if (n <= 30)
      return serfib(n);
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

Scalability measurements (3/3)

- Everything ok now ☺



Scoping with Tasks

Data Scoping Example (1/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a:
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (2/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a: shared
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (3/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (4/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (5/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d: firstprivate
               // Scope of e:
} } }
```

Data Scoping Example (6/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d: firstprivate
               // Scope of e: private
} } }
```

Data Scoping Example (7/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d: firstprivate
               // Scope of e: private
} } }
```

Hint: Use default(none) to be forced to think about every variable if you do not see clear.

Task Synchronization

RWTH AACHEN UNIVERSITY

Simple example of Task synchronization in OpenMP 3.0:

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
   function_A();
#pragma omp barrier
#pragma omp single
   {
#pragma omp task
      function_B();
   }
}
```

np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

1 Task created here

B-Task guaranteed to be completed here

Runtime Library and Environment Variables

- **C and C++:**
  - If OpenMP is enabled during compilation, the preprocessor symbol `_OPENMP` is defined. To use the OpenMP runtime library, the header `omp.h` has to be included.
  - `omp_set_num_threads(int)`: The specified number of threads will be used for the parallel region encountered next.
  - `int omp_get_num_threads`: Returns the number of threads in the current team.
  - `int omp_get_thread_num()`: Returns the number of the calling thread in the team, the Master has always the id 0.
- **Additional functions are available, e.g. to provide locking functionality.**

Environment Variables

Name	Possible Values	Most Common Default
OMP_NUM_THREADS	Non-negative Integer	1 or #cores
OMP_SCHEDULE	„schedule [, chunk]“	„static, (N/P)“
OMP_DYNAMIC	{TRUE | FALSE}	TRUE
OMP_NESTED	{TRUE | FALSE}	FALSE
OMP_STACKSIZE	„size [B | K | M | G]“	-
OMP_WAIT_POLICY	{ACTIVE | PASSIVE}	PASSIVE
OMP_MAX_ACTIVE_LEVELS	Non-negative Integer	-
OMP_THREAD_LIMIT	Non-negative Integer	1024
OMP_PROC_BIND	{TRUE | FALSE}	FALSE
OMP_PLACES	Place List	-
OMP_CANCELLATION	{TRUE | FALSE}	FALSE
OMP_DISPLAY_ENV	{TRUE | FALSE}	FALSE
OMP_DEFAULT_DEVICE	Non-negative Integer	-

Schedule Clause

Load Imbalance

Influencing the For Loop Scheduling

- ***for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:**
  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- **Default on most implementations is `schedule(static)`.**

OpenMP and the Machine Architecture

Step 1: Get Info on the System Topology

- **Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:**
  - Intel MPI's `cpuinfo` tool
    - `cpuinfo`
    - Delivers information about the number of sockets (= packages) and the mapping of processor ids used by the operating system to cpu cores.
  - hwlocs' `hwloc-ls` tool (comes with Open-MPI)
    - `hwloc-ls`
    - Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to cpu cores and additional info on caches.

Step 2: Decide for Binding Strategy

- **Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.**
  - Putting threads far apart, i.e. on different sockets
    - May improve the aggregated memory bandwidth available to your application
    - May improve the combined cache size available to your application
    - May decrease performance of synchronization constructs
  - Putting threads close together, i.e. on two adjacent cores which possibly shared some caches
    - May improve performance of synchronization constructs
    - May decrease the available memory bandwidth and cache size
- **If you are unsure, just try a few options and then select the best one.**

Step 3: Implement Binding Strategy

Intel C/C++/Fortran Compiler

- Use environment variable `KMP_AFFINTIY`
  - `KMP_AFFINITY=scatter`: Put threads far apart
  - `KMP_AFFINITY=compact`: Put threads close together
  - `KMP_AFFINITY=<core_list>`: Bind threads in the order in which they are started to the cores given in the list, one thread per core.
  - Add “`,verbose`“ to print out binding information to stdout.

GNU C/C++/Fortran Compiler

- use environment variable GOMP_CPU_AFFINITY
  - `GOMP_CPU_AFFINTIY=<core_list>`: Bind threads in the order in which they are started to the cores given in the list, one thread per core.

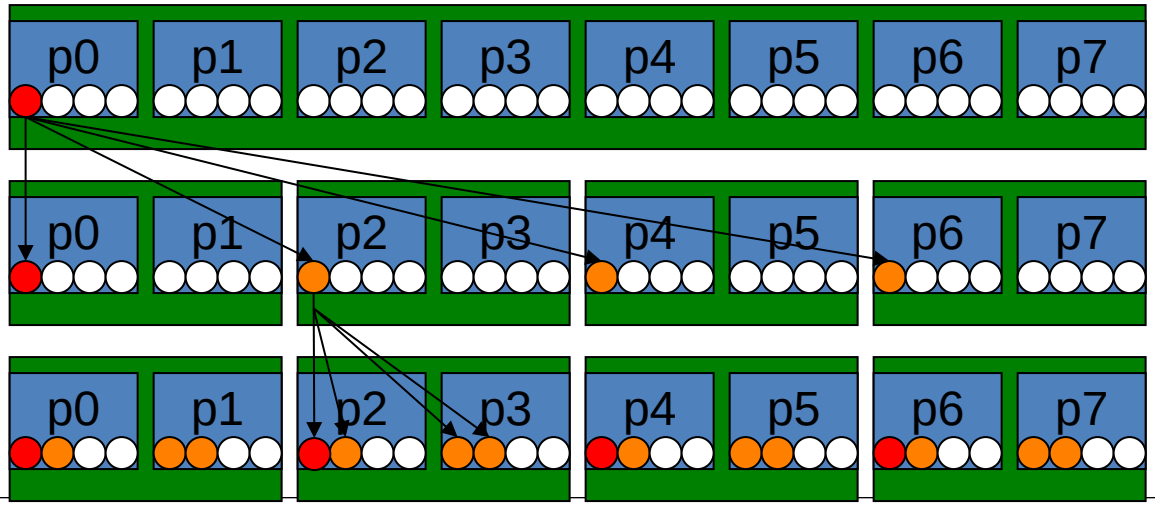
OpenMP 4.0: Places + Policies (1)

- **Define OpenMP Places**
  - set of OpenMP threads running on one or more processors
  - can be defined by the user
  - pre-defined places available:
    - *threads*: one place per hyper-thread
    - *cores* :one place exists per physical core
    - *sockets*: one place per processor package
- **Define a set of OpenMP Thread Affinity Policies**
  - SPREAD: spread OpenMP threads evenly among the places
  - CLOSE: pack OpenMP threads near master thread
  - MASTER: collocate OpenMP thread with master thread
- **Goals**
  - user has a way to specify where to execute OpenMP threads
  - locality between OpenMP threads / less false sharing / memory bandwidth

- **Example's Objective:**
  - separate cores for outer loop and near cores for inner loop
- **Outer Parallel Region: proc_bind(spread)**
  Inner Parallel Region: proc_bind(close)
  - spread creates partition, compact binds threads within respective partition

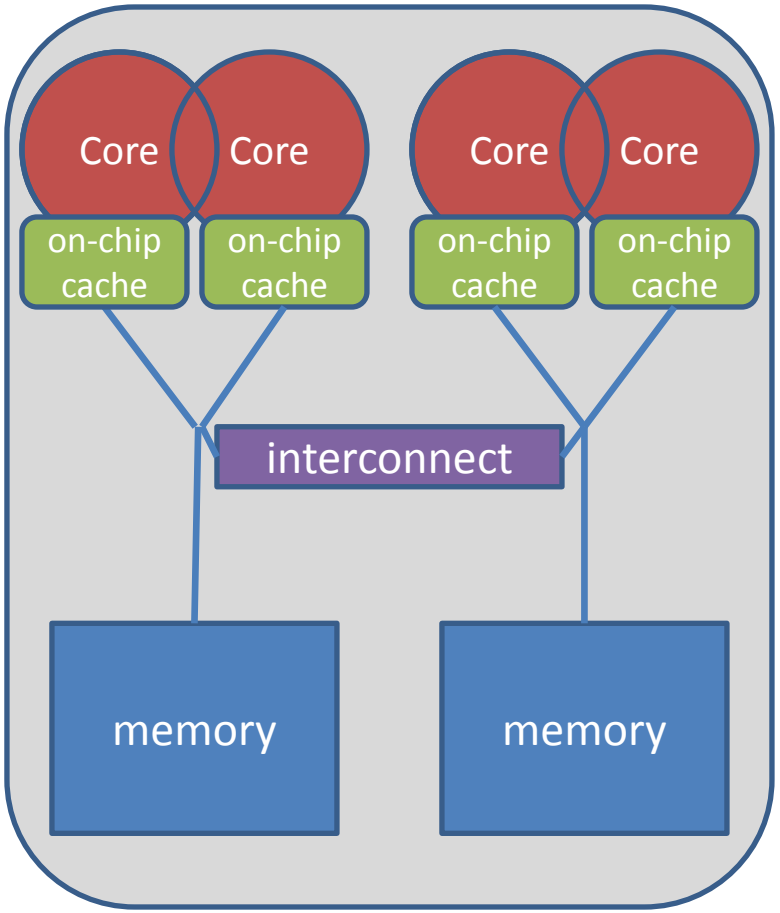
```
OMP_PLACES={0,1,2,3}, {4,5,6,7}, ... = {0-4}:4:8

#pragma omp parallel proc_bind(spread)

#pragma omp parallel proc_bind(close)
```

- **Example**
  - initial
  - spread 4
  - close 4

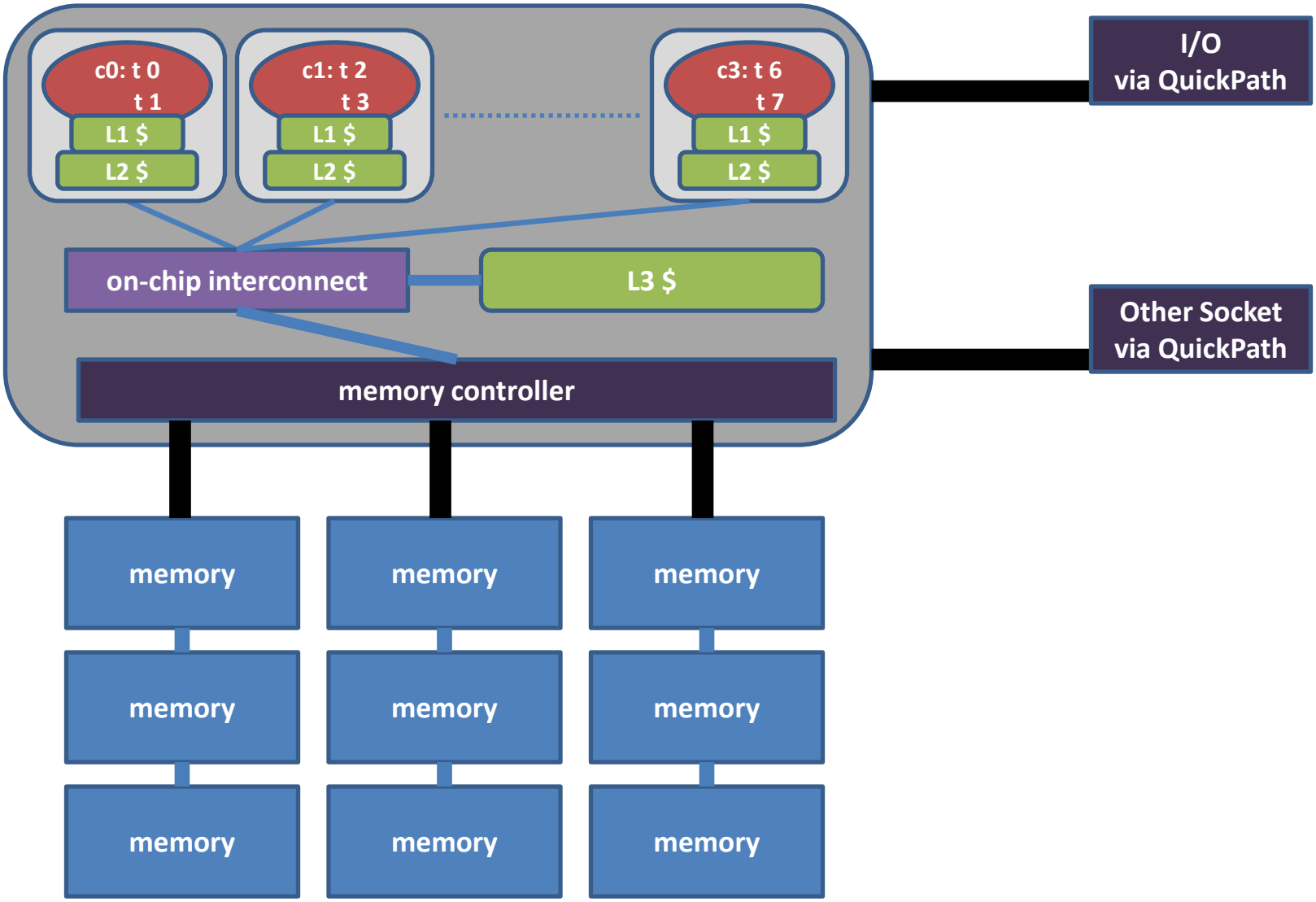


Example for a cc-NUMA system

- **Dual-socket AMD Opteron (dual-core) system**
  - Two cores per chip, 2.4 GHz
  - Each core has separate 1 MB of L2 cache on-chip
  - No off-chip cache
  - Interconnect: HyperTransport
- **cc-NUMA:**
  - Memory access time is non-uniform
  - Scalable (only if you do it right, as we will see)



Intel‘s Nehalem

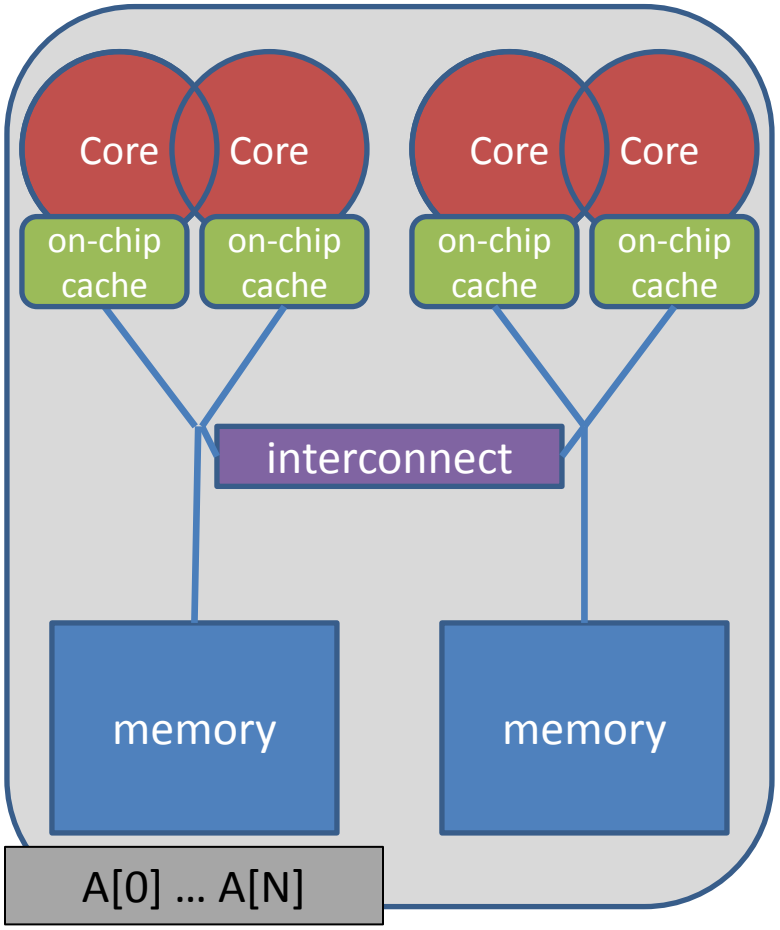


Non-uniform Memory

- **Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread**

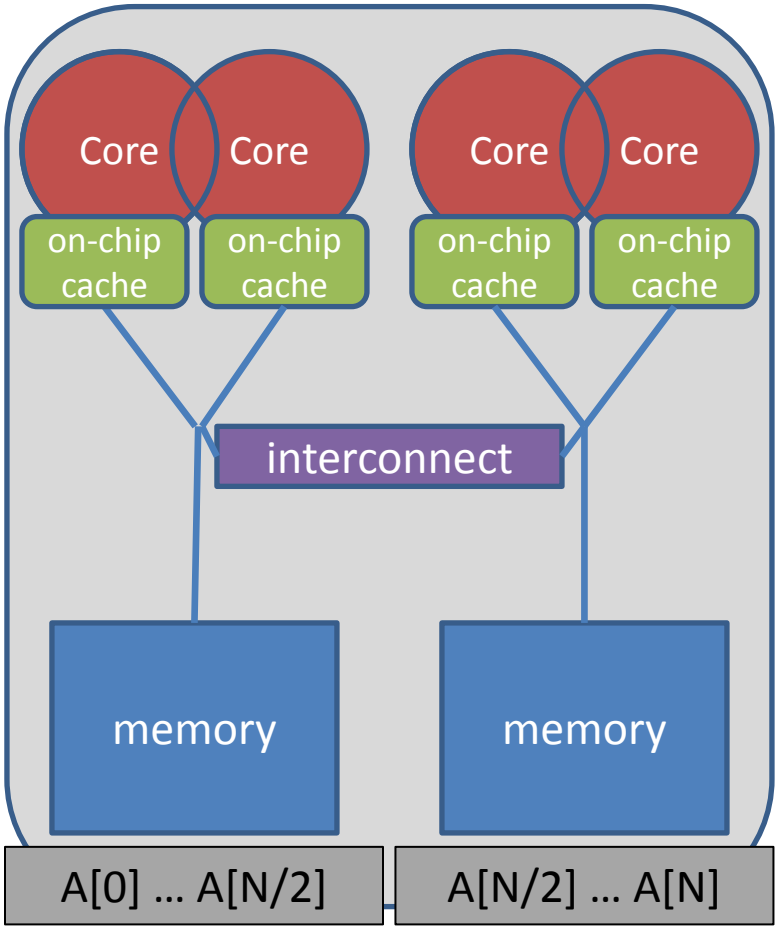
```
double* A;
A = (double*)
    malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



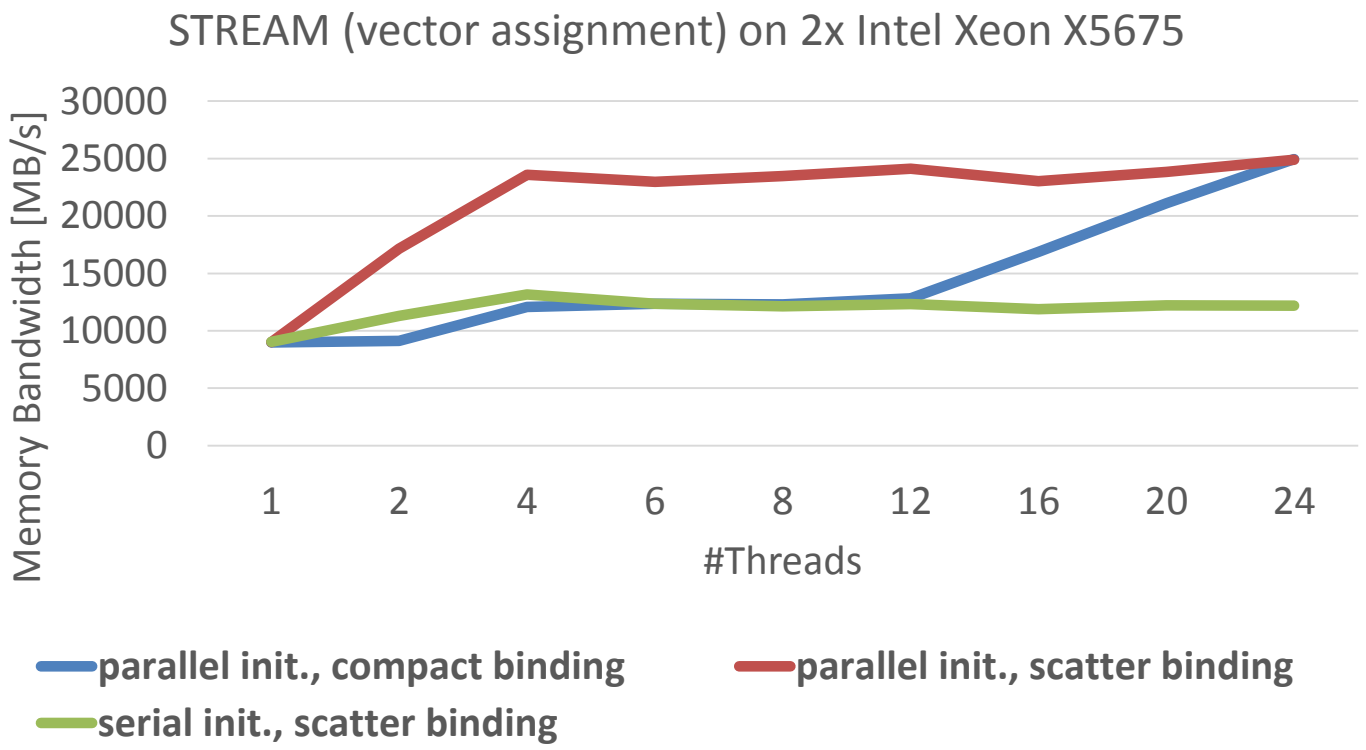
- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition

```
double* A;
A = (double*)
     malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



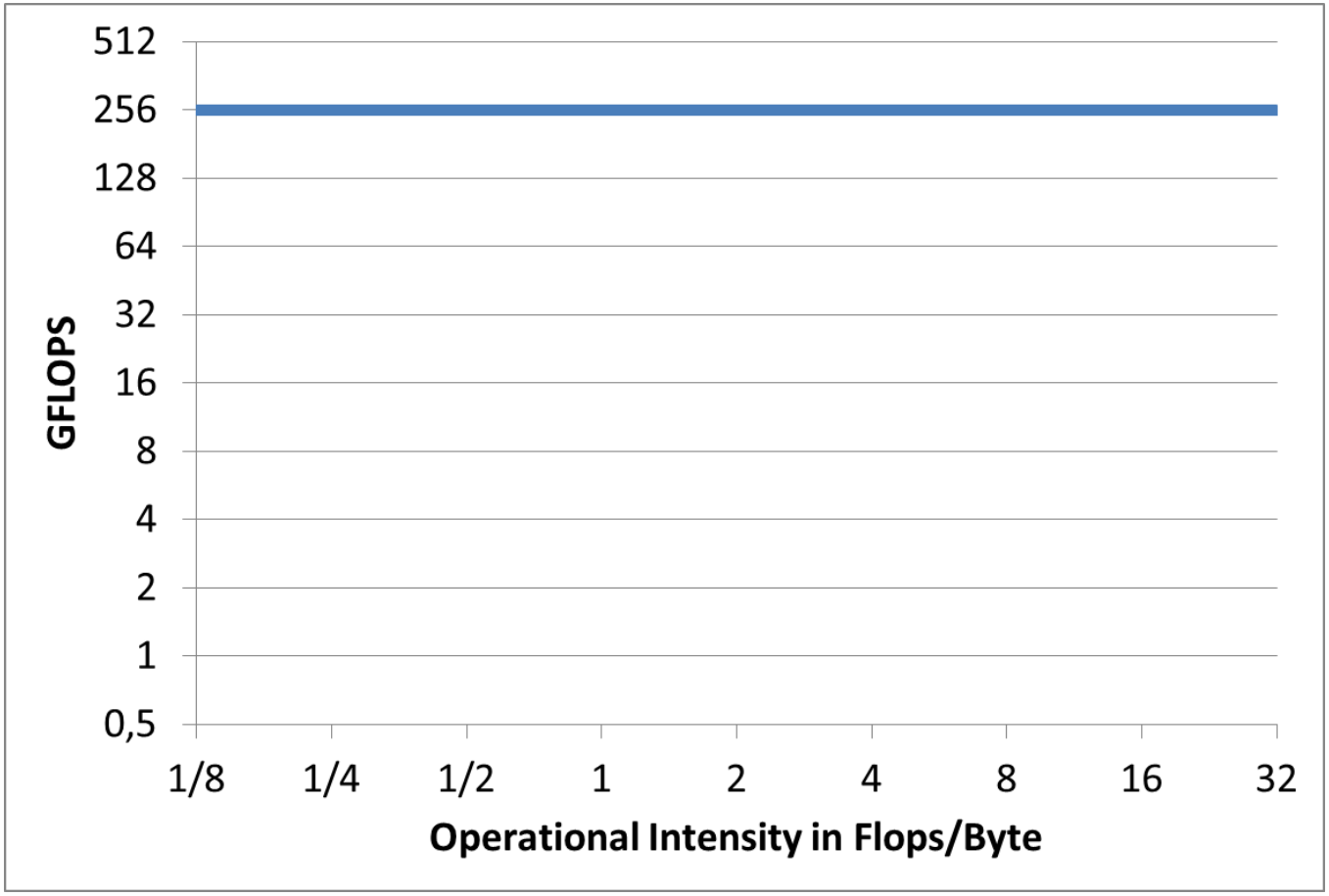
Serial vs. Parallel Initialization

- **Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 („Westmere“) using Intel® Composer XE 2013 compiler with different thread binding options:**



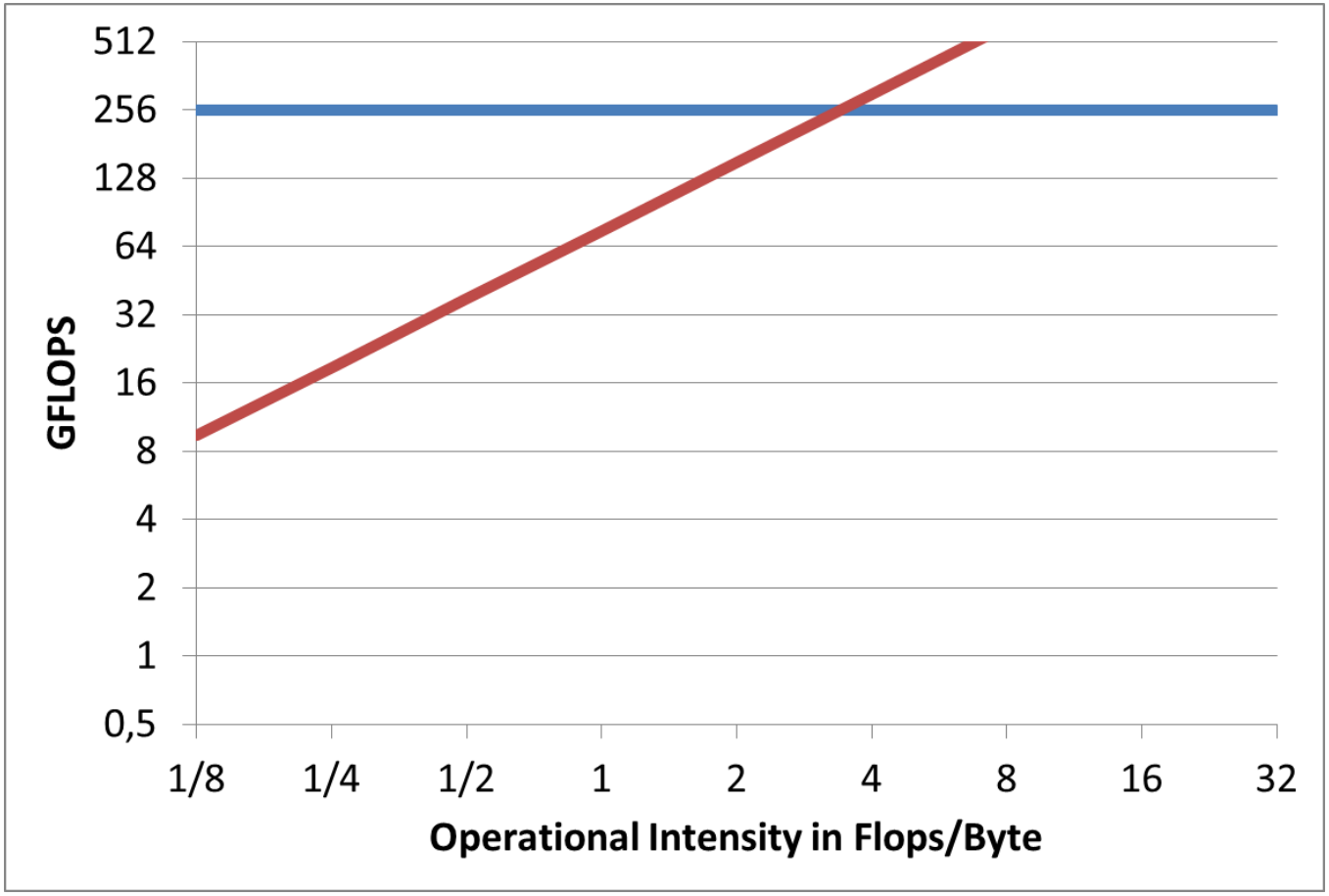
How to build a (simple) Performance Model

- Peak performance of a 4 socket Intel Nehalem-EX (2.0 GHz with 4.8 GT/s) server is 256 GFLOPS.



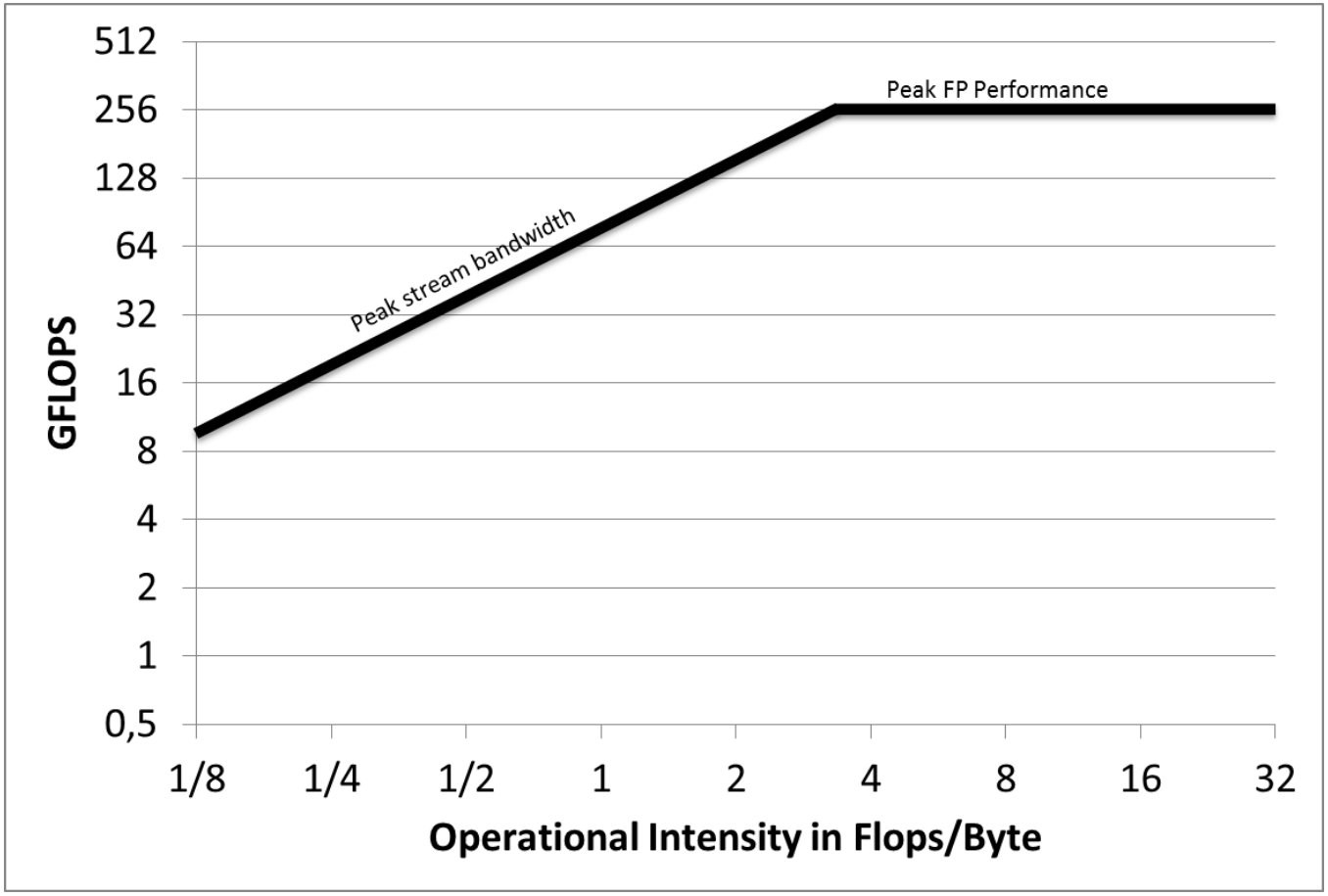
Roofline Model

- **Memory bandwidth measured with the STREAM benchmark is about 75 GB/s.**



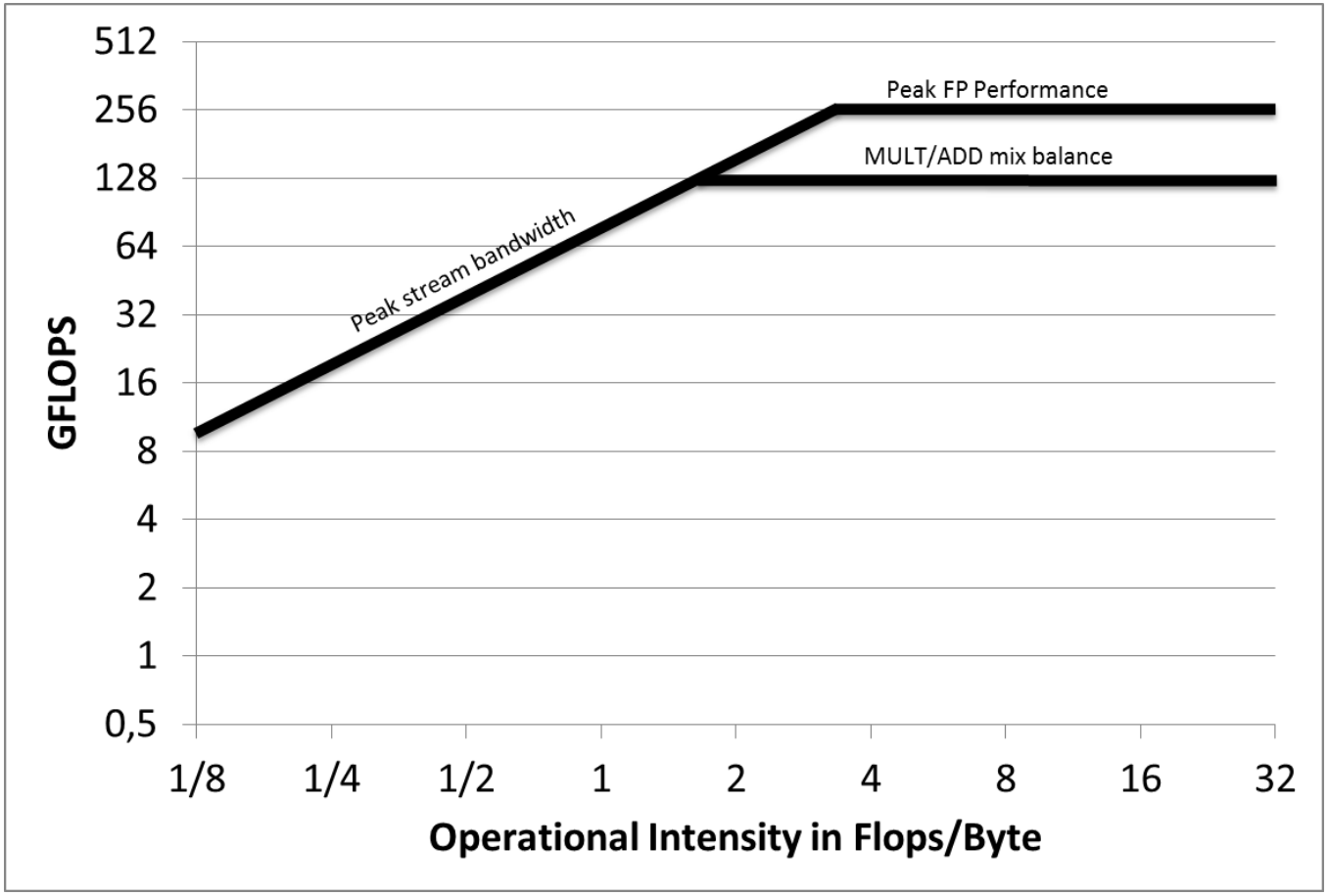
Roofline Model

- The “Roofline” is the peak performance depending on the algorithm’s “operational intensity”.



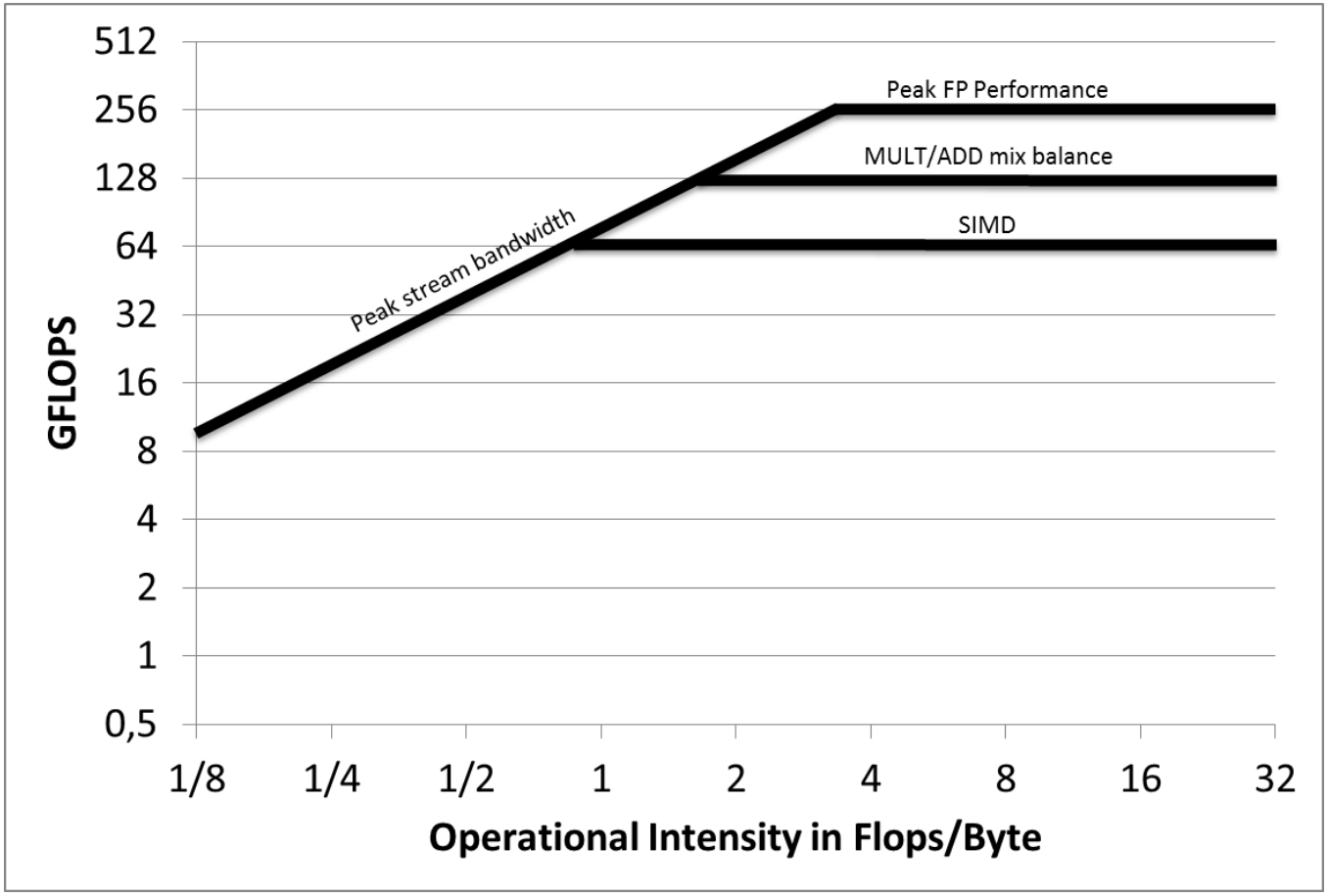
Roofline Model

- To reach the peak performance an even mix of multiply and add operations is needed.



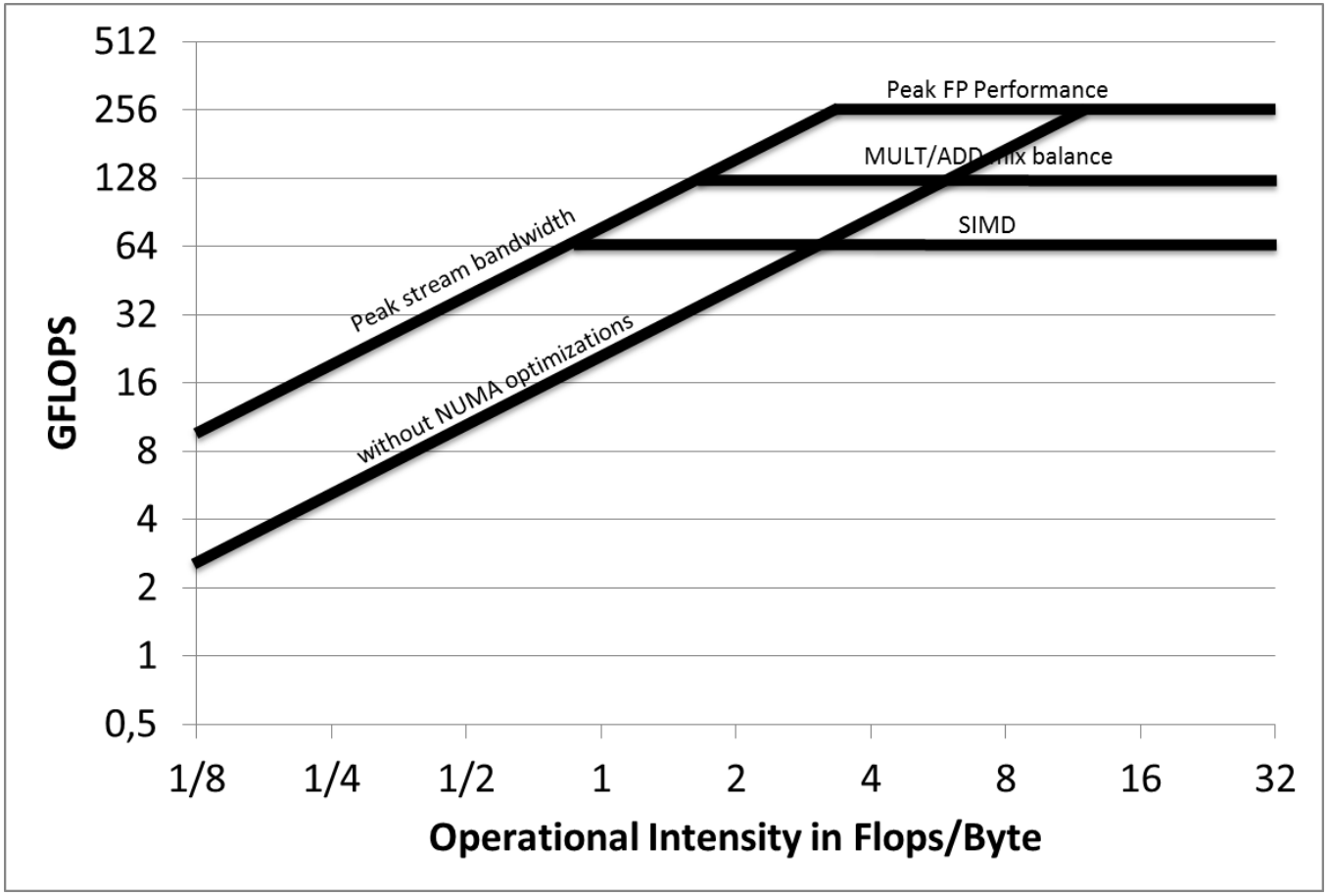
Roofline Model

- **Without vectorization only ¼ of the peak performance is achievable.**



Roofline Model

- **Peak STREAM Performance is only achievable if data is distributed optimally across NUMA nodes.**



Given

- x and y are in the cache, A is too large for the cache
- measured performance was 12 GFLOPS

- 1 ADD and 1 MULT per element
- load of value (double) and index (int) per element

-> 2 Flops / 12 Byte = 1/6 Flops/Byte