UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Shared memory: OpenMP optimizations

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Memory and thread allocation

Allocating memory

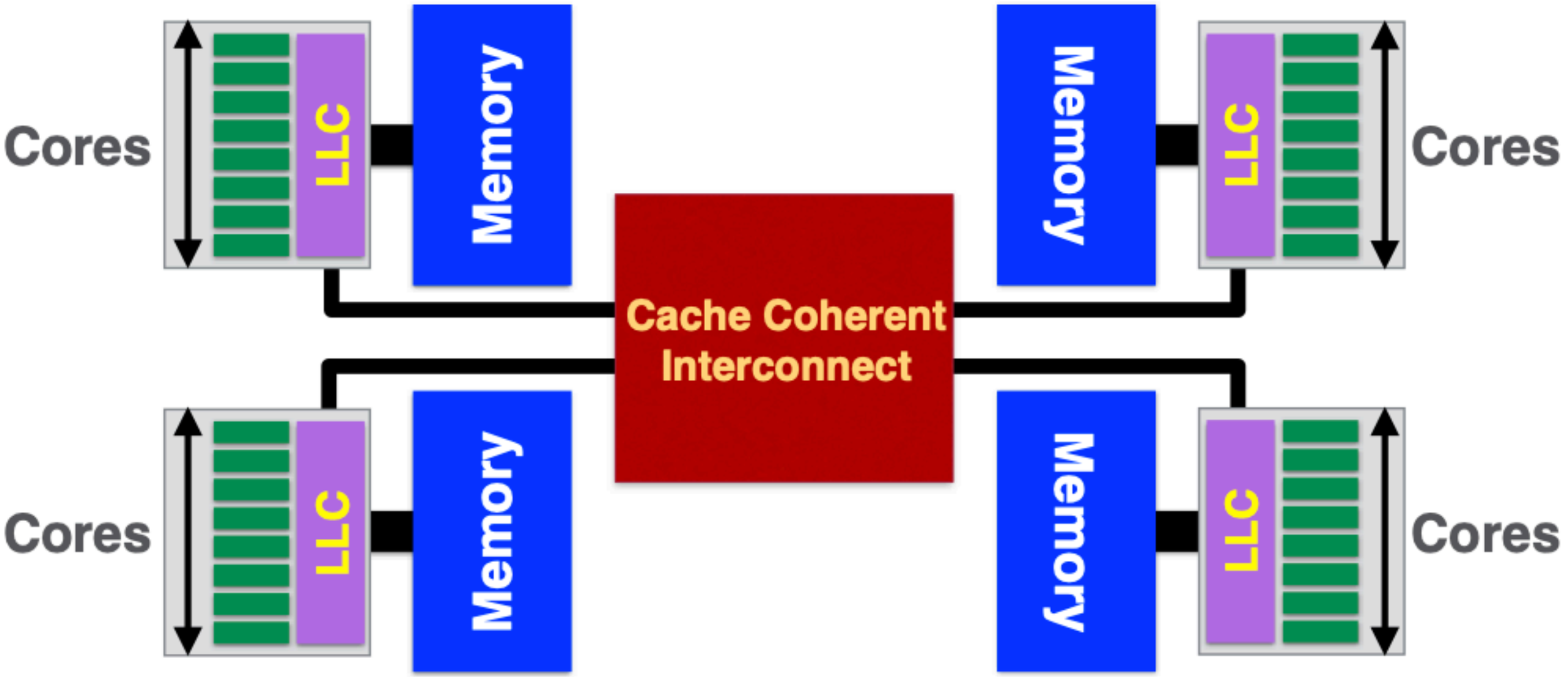
- The OS kernel can use several techniques to manage memory for OpenMP and threading.
  The most common technique is the **first touch** concept, where memory is allocated nearest to the thread where it is first touched.

- The first touch of an array causes the memory to be allocated. The memory is allocated near the thread location where the touch occurs. Prior to the first touch, the memory only exists as an entry in a virtual memory table. The physical memory that corresponds to the virtual memory is created when it is first accessed.

  - Policy used on Linux

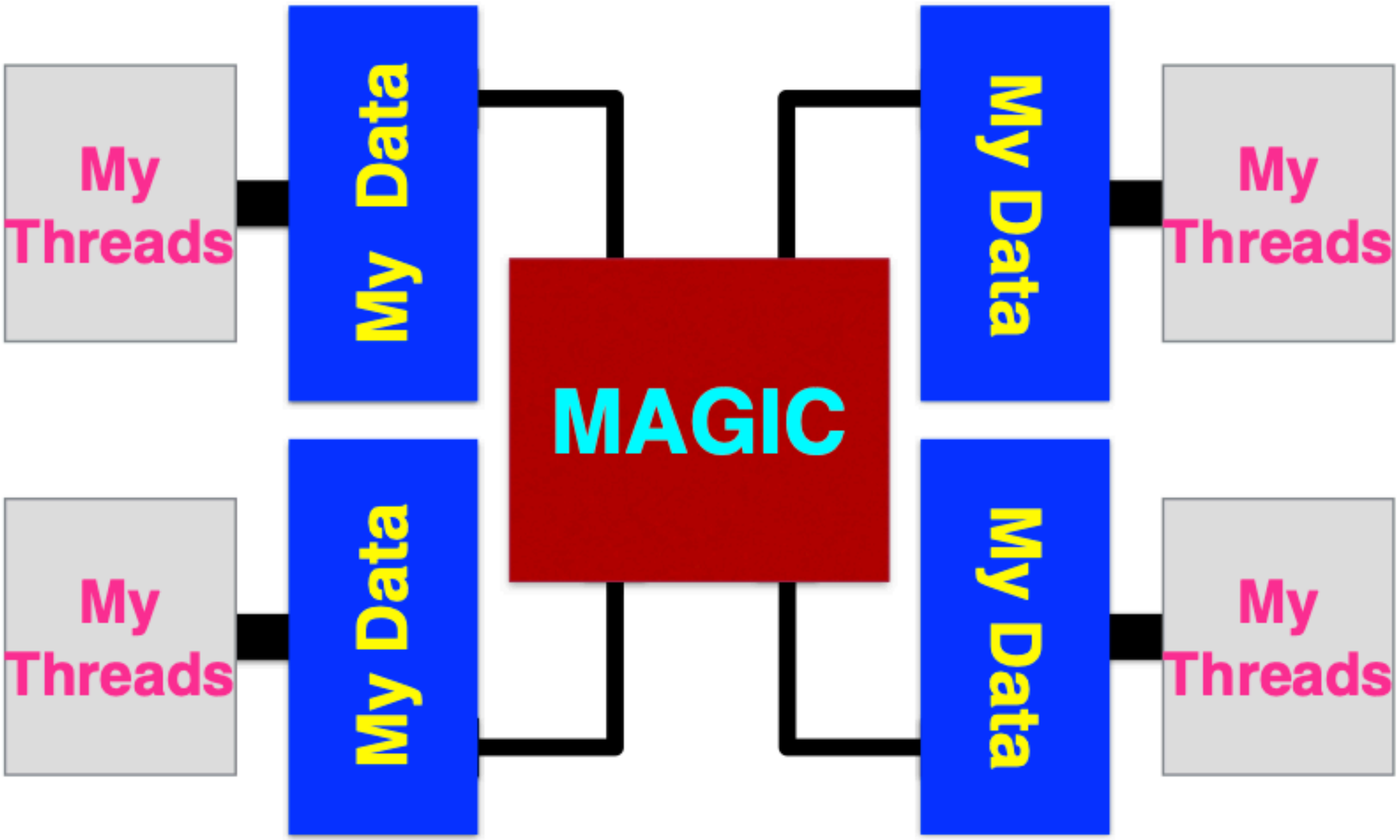
NUMA systems

- Modern parallel computers are based on NUMA

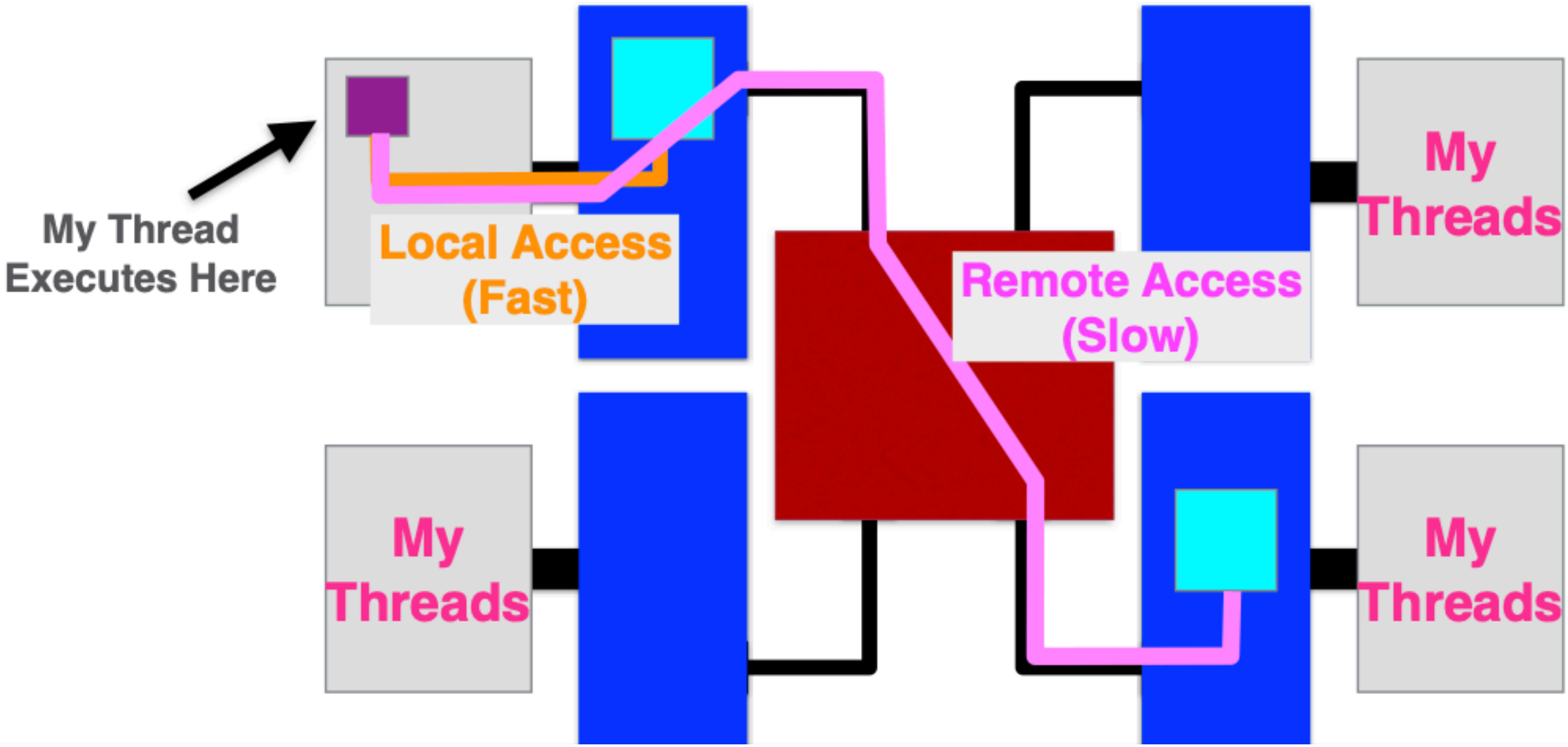


NUMA systems

- Modern parallel computers are based on NUMA



NUMA systems



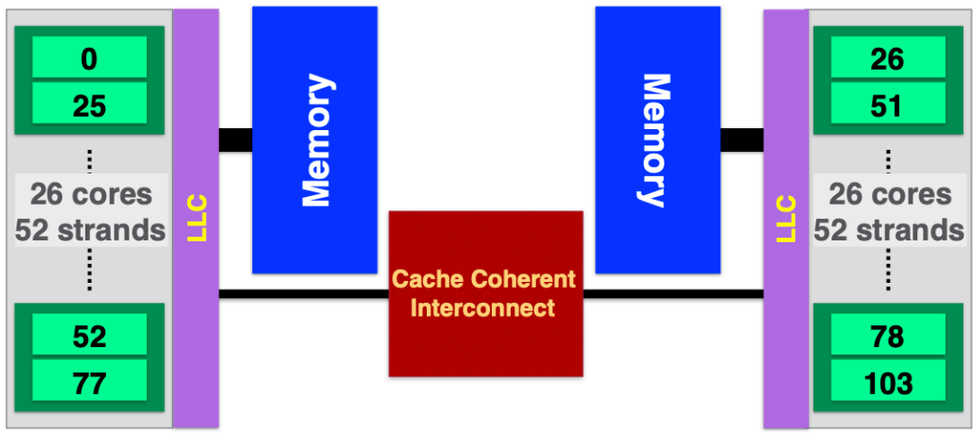
numactl

- Get NUMA details with `numactl` command, e.g. with `numactl -H`:

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77

node 0 size: 385386 MB
node 0 free: 384584 MB
node 1 cpus: 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
49 50 51 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
101 102 103

node 1 size: 387061 MB
node 1 free: 386796 MB

node distances:

node   0   1
  0:  10  21
  1:  21  10
```

Puoi vedere che succede nel caso di macchine a più cores/nodi

- The distances table show the relative latencies



LLC = last level cache

Memory allocation and first touch

- If a thread allocates all the memory then all the data will be stored on a single node. Other CPUs will have slow access to it.

- Solution: parallelize data initialization, e.g.:

```
#pragma omp parallel for  \\
  schedule(static)
for (int i=0; i<n; i++)
    a[i] = 0;
```

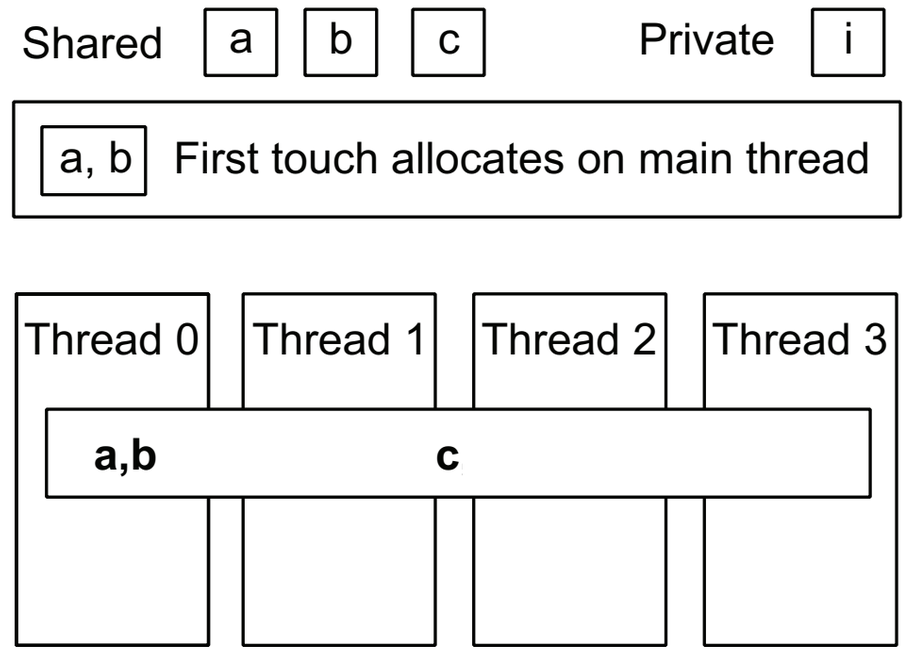
- each thread has a slice of “a” in its local memory

Example w/o first touch

```
#define ARRAY_SIZE 80000000
static double a[ARRAY_SIZE], b[ARRAY_SIZE], c[ARRAY_SIZE];
```

Notare lo static

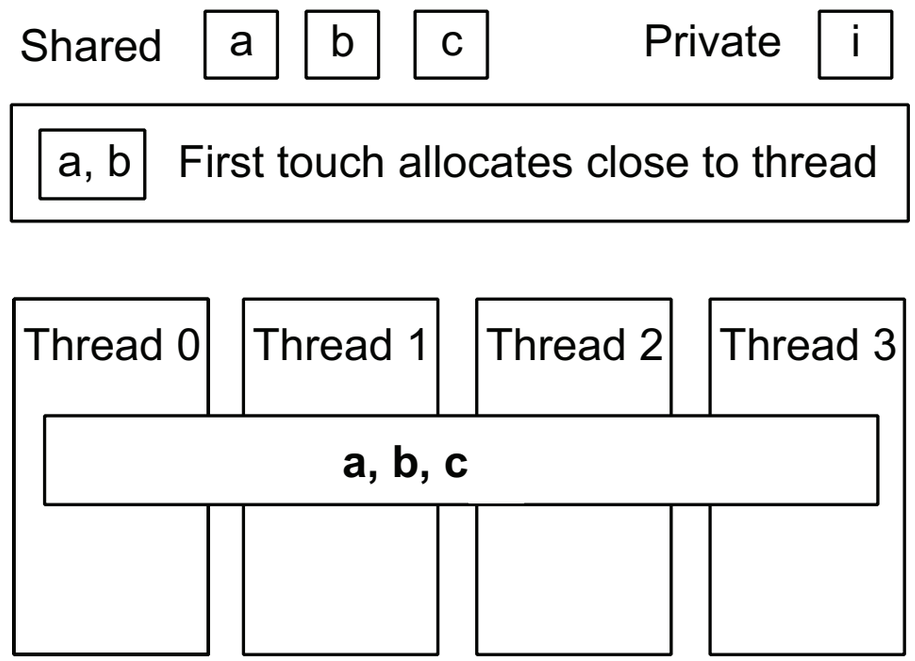
```
// …
for (int i=0; i<ARRAY_SIZE; i++) {
  a[i] = 1.0;
  b[i] = 2.0;
}

// …

#pragma omp parallel for
for (int i=0; i < n; i++) {
  c[i] = a[i] + b[i];
}
```



Thread 0 ha molto più facile accesso ad a e b rispetto agli altri thread a quali viene assegnato c

Example with first touch

```
#define ARRAY_SIZE 80000000
static double a[ARRAY_SIZE], b[ARRAY_SIZE], c[ARRAY_SIZE];


// …
#pragma omp parallel for
for (int i=0; i<ARRAY_SIZE; i++) {
  a[i] = 1.0;
  b[i] = 2.0;
}

// …

#pragma omp parallel for
for (int i=0; i < n; i++) {
  c[i] = a[i] + b[i];
}
```



Thread affinity

- Tying a thread to the location of the memory it uses is important to achieve good memory latency and bandwidth.

- Two OpenMP environment variables help to control how threads are allocated and moved

- `OMP_PLACES` - Defines where threads may run

- `OMP_PROC_BIND` - Defines how threads map onto the OpenMP places

Thread affinity

Rilevante per aumentare lo speedup… se thread rientra su zona di memoria simile (con simili dati) và più veloce!

- `export OMP_PLACES=cores`
  Threads are scheduled on the system cores

  `omp_get_num_places()` returns the number of available places

- `export OMP_PROC_BIND=spread`
  Threads should be placed on cores as far away from each other as possible

  `omp_get_proc_bind()` reports the policy used

OMP_PLACES values

- `sockets [<n>]` - Threads are scheduled on sockets

- `cores [<n>]` - Threads are scheduled on cores

- `threads [<n>]` - Threads are scheduled on strands (i.e. hardware threads, a CPU core with hyperthreading has two “strands”)

- “user defined set” - Use strand IDs to schedule threads, e.g. “`{0},{8},{16},{24}`”

OMP_PLACES values

- `sockets [<n>]` - Threads are scheduled on sockets

- `cores [<n>]` - Threads are scheduled on cores

- `threads [<n>]` - Threads are scheduled on strands (i.e. hardware threads, a CPU core with hyperthreading has two “strands”)

  In this context strand is used to differentiate between software threads of OpenMP from hardware threads of the CPU

- “user defined set” - Use strand IDs to schedule threads, e.g. “`{0},{8},{16},{24}`”

OMP_PROC_BIND values

- Since OpenMP 4.0:
- `master` - Schedule threads in the same place where the master thread is executing. Since OpenMP 5.1 it is renamed `primary`
- `close` - Keep threads “close” in terms of the places
- `spread` - Spread threads as far as possible in terms of the places

Spread può essere un idea nel caso di aggiornamenti dei valori… per evitare magari conflitti

OMP_PROC_BIND values

- Alternatively, and before OpenMP 4.0
- `true` - the kernel does not to move the thread once it gets scheduled.
- `false` - the kernel scheduler is free to move threads around

Rules of thumb

- Hyperthreading does not help with simple memory-bound kernels (i.e. functions), but it also doesn't hurt.

- For memory-bandwidth-limited kernels on multiple sockets (NUMA domains), getting the sockets busy helps (e.g. using `OMP_PROC_BIND=spread`).

- Consider the first-touch policy to allocate memory, it may greatly improve performance (e.g. 10x)

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Optimizing high-level OpenMP

Optimization process

- Base implementation — Implement simple loop-level OpenMP
- Step 1: Reduce thread start up — Merge the parallel regions and join all the loop-
  level parallel constructs into larger parallel regions
- Step 2: Synchronization — Add `nowait` clauses to for loops, where synchronization is not needed, and calculate and manually partition the loops across the threads, which allows for removal of barriers and required synchronization.
- Step 3: Optimize — Make arrays and variables private to each thread when possible.

Evito sicuramente race conditions e quindi eventuali controlli

- Step 4: Code correctness — Check thoroughly for race conditions (after every step). Use tools like Intel Inspector.

Step 1

- Go from

```
#pragma omp parallel for
```

to

```
#pragma omp parallel
#pragma omp for
```

- OpenMP overhead may account for 10-15% of execution time

Step 2

- Whenever possible add now`ait` to reduce synchronization costs due to the implicit barriers (e.g. in for loops)

- A further optimization is to avoid the work sharing of for loops by explicitly computing the work assignment based on thread ids, e.g. using something like:

```
tbegin = N *  threadID    / nthreads
tend   = N * (threadID+1) / nthreads
```

  - The impact of the manual partitioning of the arrays is that it reduces cache thrashing and race conditions by not allowing threads to share the same space in memory

Step 3

- Privatization of variables avoids synchronization
- Helps compiler optimization since it does not have to consider dependencies

Più aggiungo private e meglio è…
ridurre il più possibile variabili shared

Scoping rules

Specification		Shared	Private	Reduction
Parallel construct		Variables declared outside parallel construct or in a shared clause	Automatic variables within parallel construct or in a `private` or `firstprivate` clause	Reduction clause
Parallel region	Fortran routine	`save` attribute, initialized variables, common block, module variables	All local variables or declared `threadprivate`. Also dynamically allocated	
	C routine	File scope variables, `extern`, or `static`		
	Arguments	Inherited from calling environment		
Always		All loop indices will be private		

Optimization example

- Let's consider some code to implement a stencil operation.

```
int imax=2002, jmax = 2002;

double** xtmp;
double** x = malloc2D(jmax, imax);
double** xnew = malloc2D(jmax, imax);
int *flush = (int *)malloc(jmax*imax*sizeof(int)*4);

cpu_timer_start(&tstart_total);
cpu_timer_start(&tstart_init);
#pragma omp parallel for
for (int j = 0; j < jmax; j++){
   for (int i = 0; i < imax; i++){
      xnew[j][i] = 0.0;
      x[j][i] = 5.0;
   }
}

#pragma omp parallel for
for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
   for (int i = imax/2 - 5; i < imax/2 -1; i++){
      x[j][i] = 400.0;
   }
}
```

```
double **malloc2D(int jmax, int imax)
{
   // first allocate a block of memory for the row pointers and the 2D array
   double **x = (double **)malloc(jmax*sizeof(double *) + jmax*imax*sizeof(double));

   // Now assign the start of the block of memory for the 2D array after the row pointers
   x[0] = (double *)(x + jmax);

   // Last, assign the memory location to point to for each row pointer
   for (int j = 1; j < jmax; j++) {
      x[j] = x[j-1] + imax;
   }

   return(x);
}
```

Optimization example

- Let's consider some code to implement a stencil operation.

```
int imax=2002, jmax = 2002;

double** xtmp;
double** x = malloc2D(jmax, imax);
double** xnew = malloc2D(jmax, imax);
int *flush = (int *)malloc(jmax*imax*sizeof(int)*4);

cpu_timer_start(&tstart_total);
cpu_timer_start(&tstart_init);
#pragma omp parallel for
for (int j = 0; j < jmax; j++){
   for (int i = 0; i < imax; i++){
      xnew[j][i] = 0.0;
      x[j][i] = 5.0;
   }
}

#pragma omp parallel for
for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
   for (int i = imax/2 - 5; i < imax/2 -1; i++){
      x[j][i] = 400.0;
   }
}
```

```
double **malloc2D(int jmax, int imax)
{
   // first allocate a block of memory for the row pointers and the 2D array
   double **x = (double **)malloc(jmax*sizeof(double *) + jmax*imax*sizeof(double));

   // Now assign the start of the block of memory for the 2D array after the row pointers
   x[0] = (double *)(x + jmax);

   // Last, assign the memory location to point to for each row pointer
   for (int j = 1; j < jmax; j++) {
      x[j] = x[j-1] + imax;
   }

   return(x);
}
```

First-touch initialization

```
#pragma omp parallel for
for (int j = 0; j < jmax; j++){
   for (int i = 0; i < imax; i++){
      xnew[j][i] = 0.0;
      x[j][i] = 5.0;
   }
}

#pragma omp parallel for
for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
   for (int i = imax/2 - 5; i < imax/2 -1; i++){
      x[j][i] = 400.0;
   }
}
init_time += cpu_timer_stop(tstart_init);

for (int iter = 0; iter < 10000; iter++){
   cpu_timer_start(&tstart_flush);
   #pragma omp parallel for
   for (int l = 1; l < jmax*imax*4; l++){
       flush[l] = 1.0;
   }
   flush_time += cpu_timer_stop(tstart_flush);
   cpu_timer_start(&tstart_stencil);
   #pragma omp parallel for
   for (int j = 1; j < jmax-1; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
      }
   }
   stencil_time += cpu_timer_stop(tstart_stencil);

   SWAP_PTR(xnew, x, xtmp);
   if (iter%1000 == 0) printf("Iter %d\n",iter);
}
```

```
#pragma omp parallel for
for (int j = 0; j < jmax; j++){
   for (int i = 0; i < imax; i++){
      xnew[j][i] = 0.0;
      x[j][i] = 5.0;
   }
}

#pragma omp parallel for
for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
   for (int i = imax/2 - 5; i < imax/2 -1; i++){
      x[j][i] = 400.0;
   }
}
init_time += cpu_timer_stop(tstart_init);

for (int iter = 0; iter < 10000; iter++){
   cpu_timer_start(&tstart_flush);
   #pragma omp parallel for
   for (int l = 1; l < jmax*imax*4; l++){
       flush[l] = 1.0;
   }
   flush_time += cpu_timer_stop(tstart_flush);
   cpu_timer_start(&tstart_stencil);
   #pragma omp parallel for
   for (int j = 1; j < jmax-1; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
      }
   }
   stencil_time += cpu_timer_stop(tstart_stencil);

   SWAP_PTR(xnew, x, xtmp);
   if (iter%1000 == 0) printf("Iter %d\n",iter);
}
```

Spawn threads

```
#pragma omp parallel for
for (int j = 0; j < jmax; j++){
   for (int i = 0; i < imax; i++){
      xnew[j][i] = 0.0;
      x[j][i] = 5.0;
   }
}

#pragma omp parallel for
for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
   for (int i = imax/2 - 5; i < imax/2 -1; i++){
      x[j][i] = 400.0;
   }
}
init_time += cpu_timer_stop(tstart_init);

for (int iter = 0; iter < 10000; iter++){
   cpu_timer_start(&tstart_flush);
   #pragma omp parallel for
   for (int l = 1; l < jmax*imax*4; l++){
       flush[l] = 1.0;
   }
   flush_time += cpu_timer_stop(tstart_flush);
   cpu_timer_start(&tstart_stencil);
   #pragma omp parallel for
   for (int j = 1; j < imax-1; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
      }
   }
   stencil_time += cpu_timer_stop(tstart_stencil);

   SWAP_PTR(xnew, x, xtmp);
   if (iter%1000 == 0) printf("Iter %d\n",iter);
}
```

Spawn threads

Implicit barrier.
Join threads.

Un po' troppi fork joins…

```
#pragma omp parallel for
for (int j = 0; j < jmax; j++){
   for (int i = 0; i < imax; i++){
      xnew[j][i] = 0.0;
      x[j][i] = 5.0;
   }
}

#pragma omp parallel for
for (int j = jmax/2 - 5; j < jmax/2 + 5; j++){
   for (int i = imax/2 - 5; i < imax/2 -1; i++){
      x[j][i] = 400.0;
   }
}
init_time += cpu_timer_stop(tstart_init);

for (int iter = 0; iter < 10000; iter++){
   cpu_timer_start(&tstart_flush);
   #pragma omp parallel for
   for (int l = 1; l < jmax*imax*4; l++){
       flush[l] = 1.0;
   }
   flush_time += cpu_timer_stop(tstart_flush);
   cpu_timer_start(&tstart_stencil);
   #pragma omp parallel for
   for (int j = 1; j < jmax-1; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
      }
   }
   stencil_time += cpu_timer_stop(tstart_stencil);

   SWAP_PTR(xnew, x, xtmp);
   if (iter%1000 == 0) printf("Iter %d\n",iter);
}
```

Spawn threads

Implicit barrier.
Join threads.

What is it for? It's to evaluate the performance of the program in a cold cache scenario. It mimics the performance in the case of cache misses for **x** and **xnew**.
It is useful if you run the code within a loop to get better measurements

```
#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   for (int iter = 0; iter < 10000; iter++){
      if (thread_id ==0) cpu_timer_start(&tstart_flush);
      #pragma omp for nowait
      for (int l = 1; l < jmax*imax*4; l++){
         flush[l] = 1.0;
      }
      if (thread_id == 0){
         flush_time += cpu_timer_stop(tstart_flush);
         cpu_timer_start(&tstart_stencil);
      }
      #pragma omp for
      for (int j = 1; j < jmax-1; j++){
         for (int i = 1; i < imax-1; i++){
            xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
         }
      }
      if (thread_id == 0){
         stencil_time += cpu_timer_stop(tstart_stencil);

         SWAP_PTR(xnew, x, xtmp);
         if (iter%1000 == 0) printf("Iter %d\n",iter);
      }
   }
} // end omp parallel
```

Spawn threads

```
#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   for (int iter = 0; iter < 10000; iter++){
      if (thread_id ==0) cpu_timer_start(&tstart_flush);
      #pragma omp for nowait
      for (int l = 1; l < jmax*imax*4; l++){
         flush[l] = 1.0;
      }
      if (thread_id == 0){
         flush_time += cpu_timer_stop(tstart_flush);
         cpu_timer_start(&tstart_stencil);
      }
      #pragma omp for
      for (int j = 1; j < jmax-1; j++){
         for (int i = 1; i < imax-1; i++){
            xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
         }
      }
      if (thread_id == 0){
         stencil_time += cpu_timer_stop(tstart_stencil);

         SWAP_PTR(xnew, x, xtmp);
         if (iter%1000 == 0) printf("Iter %d\n",iter);
      }
   }
} // end omp parallel
```

```
#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   for (int iter = 0; iter < 10000; iter++){
      if (thread_id ==0) cpu_timer_start(&tstart_flush);
      #pragma omp for nowait
      for (int l = 1; l < jmax*imax*4; l++){
         flush[l] = 1.0;
      }
      if (thread_id == 0){
         flush_time += cpu_timer_stop(tstart_flush);
         cpu_timer_start(&tstart_stencil);
      }
      #pragma omp for
      for (int j = 1; j < jmax-1; j++){
         for (int i = 1; i < imax-1; i++){
            xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
         }
      }
      if (thread_id == 0){
         stencil_time += cpu_timer_stop(tstart_stencil);

         SWAP_PTR(xnew, x, xtmp);
         if (iter%1000 == 0) printf("Iter %d\n",iter);
      }
   }
} // end omp parallel
```

Spawn threads

Un solo fork - join!

Implicit barrier.
Join threads

```
#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   for (int iter = 0; iter < 10000; iter++){
      if (thread_id ==0) cpu_timer_start(&tstart_flush);
      #pragma omp for nowait
      for (int l = 1; l < jmax*imax*4; l++){
         flush[l] = 1.0;
      }
      if (thread_id == 0){
         flush_time += cpu_timer_stop(tstart_flush);
         cpu_timer_start(&tstart_stencil);
      }
      #pragma omp for
      for (int j = 1; j < jmax-1; j++){
         for (int i = 1; i < imax-1; i++){
            xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
         }
      }
      if (thread_id == 0){
         stencil_time += cpu_timer_stop(tstart_stencil);

         SWAP_PTR(xnew, x, xtmp);
         if (iter%1000 == 0) printf("Iter %d\n",iter);
      }
   }
} // end omp parallel
```

Spawn threads

No synchronization (not required)

Implicit barrier.
Join threads

```
#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   int nthreads = omp_get_num_threads();

   int jltb = 1 + (jmax-2) * ( thread_id     ) / nthreads;
   int jutb = 1 + (jmax-2) * ( thread_id + 1 ) / nthreads;

   int ifltb = (jmax*imax*4) * ( thread_id     ) / nthreads;
   int ifutb = (jmax*imax*4) * ( thread_id + 1 ) / nthreads;

   int jltb0 = jltb;
   if (thread_id == 0) jltb0--;
   int jutb0 = jutb;
   if (thread_id == nthreads-1) jutb0++;
```

Approccio alternativo…
scrivo a mano i thread da associare

```
int kmin = MAX(jmax/2-5,jltb);
int kmax = MIN(jmax/2+5,jutb);

if (thread_id == 0) cpu_timer_start(&tstart_init);
for (int j = jltb0; j < jutb0; j++){
   for (int i = 0; i < imax; i++){
      xnew[j][i] = 0.0;
      x[j][i] = 5.0;
   }
}

for (int j = kmin; j < kmax; j++){
   for (int i = imax/2 - 5; i < imax/2 -1; i++){
      x[j][i] = 400.0;
   }
}
#pragma omp barrier
if (thread_id == 0) init_time += cpu_timer_stop(tstart_init);
```

```
#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   int nthreads = omp_get_num_threads();

   int jltb = 1 + (jmax-2) * ( thread_id     ) / nthreads;
   int jutb = 1 + (jmax-2) * ( thread_id + 1 ) / nthreads;

   int ifltb = (jmax*imax*4) * ( thread_id     ) / nthreads;
   int ifutb = (jmax*imax*4) * ( thread_id + 1 ) / nthreads;

   int jltb0 = jltb;
   if (thread_id == 0) jltb0--;
   int jutb0 = jutb;
   if (thread_id == nthreads-1) jutb0++;

   int kmin = MAX(jmax/2-5,jltb);
   int kmax = MIN(jmax/2+5,jutb);

   if (thread_id == 0) cpu_timer_start(&tstart_init);
   for (int j = jltb0; j < jutb0; j++){
      for (int i = 0; i < imax; i++){
         xnew[j][i] = 0.0;
         x[j][i] = 5.0;
      }
   }

   for (int j = kmin; j < kmax; j++){
      for (int i = imax/2 - 5; i < imax/2 -1; i++){
         x[j][i] = 400.0;
      }
   }
   #pragma omp barrier
   if (thread_id == 0) init_time += cpu_timer_stop(tstart_init);
```

Computation of the loop bounds.
The input matrix is divided in horizontal stripes, each one is assigned to a thread (`jltb`/`jutb`).
Magic numbers:
2=half of the (stencil size-1)
5=stencil size.

Passo a variabili private...
no race conditions e quindi non hanno bisogno di sinc
Variabili scritte nei registri! … più ottimizzate

```
#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   int nthreads = omp_get_num_threads();

   int jltb = 1 + (jmax-2) * ( thread_id     ) / nthreads;
   int jutb = 1 + (jmax-2) * ( thread_id + 1 ) / nthreads;

   int ifltb = (jmax*imax*4) * ( thread_id     ) / nthreads;
   int ifutb = (jmax*imax*4) * ( thread_id + 1 ) / nthreads;

   int jltb0 = jltb;
   if (thread_id == 0) jltb0--;
   int jutb0 = jutb;
   if (thread_id == nthreads-1) jutb0++;

   int kmin = MAX(jmax/2-5,jltb);
   int kmax = MIN(jmax/2+5,jutb);

   if (thread_id == 0) cpu_timer_start(&tstart_init);
   for (int j = jltb0; j < jutb0; j++){
      for (int i = 0; i < imax; i++){
         xnew[j][i] = 0.0;
         x[j][i] = 5.0;
      }
   }

   for (int j = kmin; j < kmax; j++){
      for (int i = imax/2 - 5; i < imax/2 -1; i++){
         x[j][i] = 400.0;
      }
   }
   #pragma omp barrier
   if (thread_id == 0) init_time += cpu_timer_stop(tstart_init);
```

Computation of the loop bounds.
The input matrix is divided in horizontal stripes, each one is assigned to a thread (`jltb`/`jutb`).
Magic numbers:
2=half of the (stencil size-1)
5=stencil size.

Use loop bounds computed with thread ids instead of `for` loop directive

```
#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   int nthreads = omp_get_num_threads();

   int jltb = 1 + (jmax-2) * ( thread_id     ) / nthreads;
   int jutb = 1 + (jmax-2) * ( thread_id + 1 ) / nthreads;

   int ifltb = (jmax*imax*4) * ( thread_id     ) / nthreads;
   int ifutb = (jmax*imax*4) * ( thread_id + 1 ) / nthreads;

   int jltb0 = jltb;
   if (thread_id == 0) jltb0--;
   int jutb0 = jutb;
   if (thread_id == nthreads-1) jutb0++;

   int kmin = MAX(jmax/2-5,jltb);
   int kmax = MIN(jmax/2+5,jutb);

   if (thread_id == 0) cpu_timer_start(&tstart_init);
   for (int j = jltb0; j < jutb0; j++){
      for (int i = 0; i < imax; i++){
         xnew[j][i] = 0.0;
         x[j][i] = 5.0;
      }
   }

   for (int j = kmin; j < kmax; j++){
      for (int i = imax/2 - 5; i < imax/2 -1; i++){
         x[j][i] = 400.0;
      }
   }
   #pragma omp barrier
   if (thread_id == 0) init_time += cpu_timer_stop(tstart_init);
```

Computation of the loop bounds.
The input matrix is divided in horizontal stripes, each one is assigned to a thread (`jltb`/`jutb`).
Magic numbers:
2=half of the (stencil size-1)
5=stencil size.

Use loop bounds computed with thread ids instead of `for` loop directive

Add barrier to avoid race condition. This is not required with for loop thanks to the implied barrier

qui serve scriverlo perchè non lo fa da solo

```
for (int iter = 0; iter < 10000; iter++){
   if (thread_id == 0) cpu_timer_start(&tstart_flush);
   for (int l = ifltb; l < ifutb; l++){
      flush[l] = 1.0;
   }
   if (thread_id == 0){
      flush_time += cpu_timer_stop(tstart_flush);
      cpu_timer_start(&tstart_stencil);
   }
   for (int j = jltb; j < jutb; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
      }
   }
   #pragma omp barrier
   if (thread_id == 0){
      stencil_time += cpu_timer_stop(tstart_stencil);

      SWAP_PTR(xnew, x, xtmp);
      if (iter%1000 == 0) printf("Iter %d\n",iter);
   }
   #pragma omp barrier
  }
} // end omp parallel
```

```
for (int iter = 0; iter < 10000; iter++){
   if (thread_id == 0) cpu_timer_start(&tstart_flush);
   for (int l = ifltb; l < ifutb; l++){
      flush[l] = 1.0;
   }
   if (thread_id == 0){
      flush_time += cpu_timer_stop(tstart_flush);
      cpu_timer_start(&tstart_stencil);
   }
   for (int j = jltb; j < jutb; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
      }
   }
   #pragma omp barrier
   if (thread_id == 0){
      stencil_time += cpu_timer_stop(tstart_stencil);

      SWAP_PTR(xnew, x, xtmp);
      if (iter%1000 == 0) printf("Iter %d\n",iter);
   }
   #pragma omp barrier
  }
} // end omp parallel
```

Use loop bounds computed with thread ids instead of `for` loop directive

```
for (int iter = 0; iter < 10000; iter++){
   if (thread_id == 0) cpu_timer_start(&tstart_flush);
   for (int l = ifltb; l < ifutb; l++){
      flush[l] = 1.0;
   }
   if (thread_id == 0){
      flush_time += cpu_timer_stop(tstart_flush);
      cpu_timer_start(&tstart_stencil);
   }
   for (int j = jltb; j < jutb; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j
      }
   }
   #pragma omp barrier
   if (thread_id == 0){
      stencil_time += cpu_timer_stop(tstart_stencil);

      SWAP_PTR(xnew, x, xtmp);
      if (iter%1000 == 0) printf("Iter %d\n",iter);
   }
   #pragma omp barrier
  }
} // end omp parallel
```

Use loop bounds computed with thread ids instead of `for` loop directive

Use loop bounds computed with thread IDs instead of `for` loop directive

```
for (int iter = 0; iter < 10000; iter++){
   if (thread_id == 0) cpu_timer_start(&tstart_flush);
   for (int l = ifltb; l < ifutb; l++){
      flush[l] = 1.0;
   }
   if (thread_id == 0){
      flush_time += cpu_timer_stop(tstart_flush);
      cpu_timer_start(&tstart_stencil);
   }
   for (int j = jltb; j < jutb; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j
      }
   }
   #pragma omp barrier
   if (thread_id == 0){
      stencil_time += cpu_timer_stop(tstart_stencil);

      SWAP_PTR(xnew, x, xtmp);
      if (iter%1000 == 0) printf("Iter %d\n",iter);
   }
   #pragma omp barrier
  }
} // end omp parallel
```

Use loop bounds computed with thread ids instead of `for` loop directive

Use loop bounds computed with thread IDs instead of `for` loop directive

Add barrier to avoid race condition. This is not required with for loop thanks to the implied barrier

```
for (int iter = 0; iter < 10000; iter++){
   if (thread_id == 0) cpu_timer_start(&tstart_flush);
   for (int l = ifltb; l < ifutb; l++){
      flush[l] = 1.0;
   }
   if (thread_id == 0){
      flush_time += cpu_timer_stop(tstart_flush);
      cpu_timer_start(&tstart_stencil);
   }
   for (int j = jltb; j < jutb; j++){
      for (int i = 1; i < imax-1; i++){
         xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j
      }
   }
   #pragma omp barrier
   if (thread_id == 0){
      stencil_time += cpu_timer_stop(tstart_stencil);

      SWAP_PTR(xnew, x, xtmp);
      if (iter%1000 == 0) printf("Iter %d\n",iter);
   }
   #pragma omp barrier
  }
} // end omp parallel
```

Use loop bounds computed with thread ids instead of `for` loop directive

Use loop bounds computed with thread IDs instead of `for` loop directive

Add barrier to avoid race condition. This is not required with for loop thanks to the implied barrier

Uses thread ID instead of master/ masked to reduce synchronization

For loop and SIMD

- OpenMP threaded loops can be combined with SIMD instructions to further improve the performance:

- `#pragma omp [parallel] for simd`
  `#pragma omp simd`

```
#pragma omp for
for (int j = 1; j < jmax-1; j++){
   for (int i = 1; i < imax-1; i++){
      xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
   }
}
```

```
#pragma omp for
for (int j = 1; j < jmax-1; j++){
   #ifdef OMP_SIMD
   #pragma omp simd
   #endif
   for (int i = 1; i < imax-1; i++){
      xnew[j][i] = ( x[j][i] + x[j][i-1] + x[j][i+1] + x[j-1][i] + x[j+1][i] )/5.0;
   }
}
```

A more complex example

- Let us consider a more complex example using a stencil that can be computed separately along its x and y axis

- E.g. a cross shaped kernel

- E.g. a Gaussian kernel is separable and we could compute it using two 1D kernels

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

Memory requirements

- Considering this case where we need to compute along two different faces we notice that we have different requirements to memory access.

- Considering a distribution of the data along horizontal stripes, the computations along x follow the data distribution
  - Each thread can use private and local data

- Instead, computations along y need data distributed among different threads,
  - Need shared data

- Let’s use as much as possible private and local data

Data locality

- The first touch principle of most kernels says that memory will most likely be local to the thread (except at the edges between threads on page boundaries).

- We can improve the memory locality by making the array sections completely private to the thread where possible, such as the x-oriented data.

- Increasing the data locality is essential in minimizing the increasing speed gap between processors and memory.

Sequential implementation

```
void SplitStencil(double **a, int imax, int jmax)
{
   double** xface = malloc2D(jmax, imax);
   double** yface = malloc2D(jmax, imax);
   for (int j = 1; j < jmax-1; j++){
      for (int i = 0; i < imax-1; i++){
         xface[j][i] = (a[j][i+1]+a[j][i])/2.0;
      }
   }
   for (int j = 0; j < jmax-1; j++){
      for (int i = 1; i < imax-1; i++){
         yface[j][i] = (a[j+1][i]+a[j][i])/2.0;
      }
   }
   for (int j = 1; j < jmax-1; j++){
      for (int i = 1; i < imax-1; i++){
         a[j][i] = (a[j][i]+xface[j][i]+xface[j][i-1]+yface[j][i]+yface[j-1][i])/5.0;
      }
   }
   free(xface);
   free(yface);
}
```

- Let's consider a cross shaped stencil that computes values on the faces of the cells.
- X-face calc needs adjacent cells in the x direction
- Y-face calc needs adjacent cells in the y direction
- We add contributions from x and y faces for the final result

OpenMP and memory

- Each thread computes x-face data in a local variable
- Y-face data is computed in a shared scope variable, since threads operating on border elements need to access data from other threads
  - Race condition: it must have been already computed !
- There’s no race condition in writing the final results, each thread operates on its cells
  - Perhaps there’s risk of cache false sharing
- Remind the thread scoping of OpenMP:

Scoping rules

<table>
<tr><th colspan="2">Specification</th><th>Shared</th><th>Private</th><th>Reduction</th></tr>
<tr><td colspan="2">Parallel construct</td><td>Variables declared outside parallel construct or in a shared clause</td><td>Automatic variables within parallel construct or in a <code>private</code> or <code>firstprivate</code> clause</td><td rowspan="4">Reduction clause</td></tr>
<tr><td rowspan="3">Parallel region</td><td>Fortran routine</td><td><code>save</code> attribute, initialized variables, common block, module variables</td><td rowspan="2">All local variables or declared <code>threadprivate</code>. Also dynamically allocated</td></tr>
<tr><td>C routine</td><td>File scope variables, <code>extern</code>, or <code>static</code></td></tr>
<tr><td>Arguments</td><td colspan="2">Inherited from calling environment</td></tr>
<tr><td colspan="2">Always</td><td colspan="3">All loop indices will be private</td></tr>
</table>

Parallelization

- Similar to the previous example initialize the input matrix in a `#pragma omp parallel` or `#pragma omp parallel for` section
  - Prefer the first one… compute the indices to allocate matrix parts next to the cores
- Execute the function computing the stencil within a `#pragma omp parallel` section
  - Could be the section used to allocate the input matrix

Function parallelization

```
void SplitStencil(double **a, int imax, int jmax)
{
   int thread_id = omp_get_thread_num();
   int nthreads = omp_get_num_threads();

   int jltb = 1 + (jmax-2) * ( thread_id     ) / nthreads;
   int jutb = 1 + (jmax-2) * ( thread_id + 1 ) / nthreads;

   int jfltb = jltb;
   int jfutb = jutb;
   if (thread_id == 0) jfltb--;

   double** xface = (double **)malloc2D(jutb-jltb, imax-1);
   static double** yface;
   if (thread_id == 0) yface = (double **)malloc2D(jmax+2, imax);
#pragma omp barrier
   for (int j = jltb; j < jutb; j++){
      for (int i = 0; i < imax-1; i++){
         xface[j-jltb][i] = (a[j][i+1]+a[j][i])/2.0;
      }
   }
   for (int j = jfltb; j < jfutb; j++){
      for (int i = 1; i < imax-1; i++){
         yface[j][i] = (a[j+1][i]+a[j][i])/2.0;
      }
   }
#pragma omp barrier
   for (int j = jltb; j < jutb; j++){
      for (int i = 1; i < imax-1; i++){
         a[j][i] = (a[j][i]+xface[j-jltb][i]+xface[j-jltb][i-1]+
                           yface[j][i]+yface[j-1][i])/5.0;
      }
   }
   free(xface);
#pragma omp barrier
   if (thread_id == 0) free(yface);
}
```

- Compute the indexes to assign the chunks of input data to the threads
- Initialize local memory to store x-face data
- Initialize shared memory to store y-face data
  - Static variable
  - Synchronize with a barrier or threads may start before the shared memory is ready !
- Compute x-face data
- Compute y-face data
  - Synchronize with a barrier or threads may start computing the final value w/o y-face data !
- Use of local memory may lead to super-linear speedup !

Function parallelization: details

```
void SplitStencil(double **a, int imax, int jmax)
{
   int thread_id = omp_get_thread_num();
   int nthreads = omp_get_num_threads();

   int jltb = 1 + (jmax-2) * ( thread_id     ) / nthreads;
   int jutb = 1 + (jmax-2) * ( thread_id + 1 ) / nthreads;

   int jfltb = jltb;
   int jfutb = jutb;
   if (thread_id == 0) jfltb--;

   double** xface = (double **)malloc2D(jutb-jltb, imax-1);
   static double** yface;
   if (thread_id == 0) yface = (double **)malloc2D(jmax+2, imax);
#pragma omp barrier
```

- `xface` is allocated on the thread stack. It's local and private. It's small: each thread needs to store only the data required for it's computation.
- `yface` is `static` therefore shared. Only one thread needs to allocate it. It's allocation MUST be synchronized with the barrier. It's large to contain all values.

Function parallelization: details

```
#pragma omp barrier
   for (int j = jltb; j < jutb; j++){
      for (int i = 0; i < imax-1; i++){
         xface[j-jltb][i] = (a[j][i+1]+a[j][i])/2.0;
      }
   }
   for (int j = jfltb; j < jfutb; j++){
      for (int i = 1; i < imax-1; i++){
         yface[j][i] = (a[j+1][i]+a[j][i])/2.0;
      }
   }
#pragma omp barrier
```

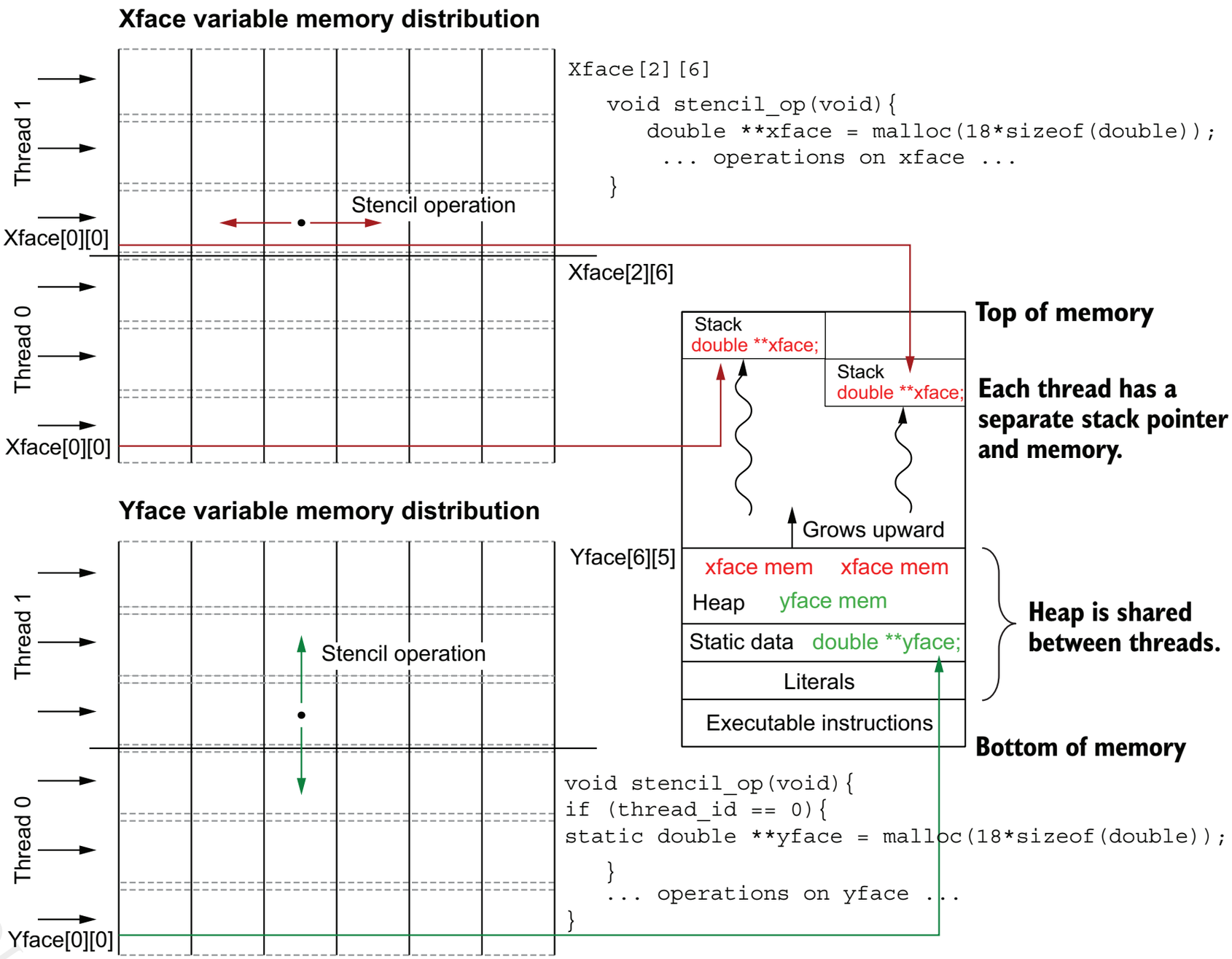
- Computing `xface` does not require synchronization. It's a local and private variable
- `yface` must be ready before it's used in the following stage where the final value is computed. This requires synchronization, or we'll have a race condition in the following loop.

Function parallelization: details

```
#pragma omp barrier
   for (int j = jltb; j < jutb; j++){
      for (int i = 1; i < imax-1; i++){
         a[j][i] = (a[j][i]+xface[j-jltb][i]+xface[j-jltb][i-1]+
                              yface[j][i]+yface[j-1][i])/5.0;
      }
   }
   free(xface);
#pragma omp barrier
   if (thread_id == 0) free(yface);
}
```

- Since `yface` is surely fully computed we compute the final values.
- Access to `xface` is fast, it's a local memory, access to `yface` may not be local.
- Before freeing `yface` we must be sure that all threads have completed their computation, thus we need a final barrier.

Memory view


Xface variable memory distribution
Xface[2][6]
void stencil_op(void){
double **xface = malloc(18*sizeof(double));
... operations on xface ...
}
Thread 1
Thread 0
Stencil operation
Xface[0][0]
Xface[2][6]
Top of memory
Stack
double **xface;
Stack
double **xface;
Each thread has a separate stack pointer and memory.
Yface variable memory distribution
Grows upward
Yface[6][5]
xface mem
xface mem
Heap
yface mem
Static data
double **yface;
Literals
Executable instructions
Heap is shared between threads.
Bottom of memory
Thread 1
Thread 0
Stencil operation
void stencil_op(void){
if (thread_id == 0){
static double **yface = malloc(18*sizeof(double));
}
... operations on yface ...
}
Yface[0][0]

Credits

- These slides report material from:
  - Ruud van der Pas (Oracle)

UNIVERSITÀ DEGLI STUDI FIRENZE

Books

- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 7 and 14