

Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

Recap

- Linked-lists in C
- Read-Write Locks
- Thread Safety

Questions?

Misc

Threads Safety in MPI

Are MPI calls thread-safe? (e.g., can I call MPI_Send from multiple threads at the same time?)

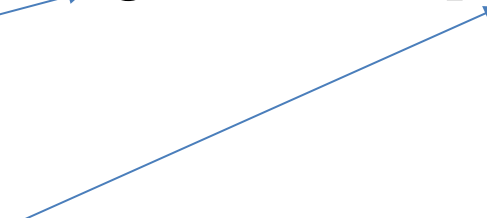
Instead of MPI_Init, use:

```
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
```

Same as MPI_Init



Required «threading level»
(IN)



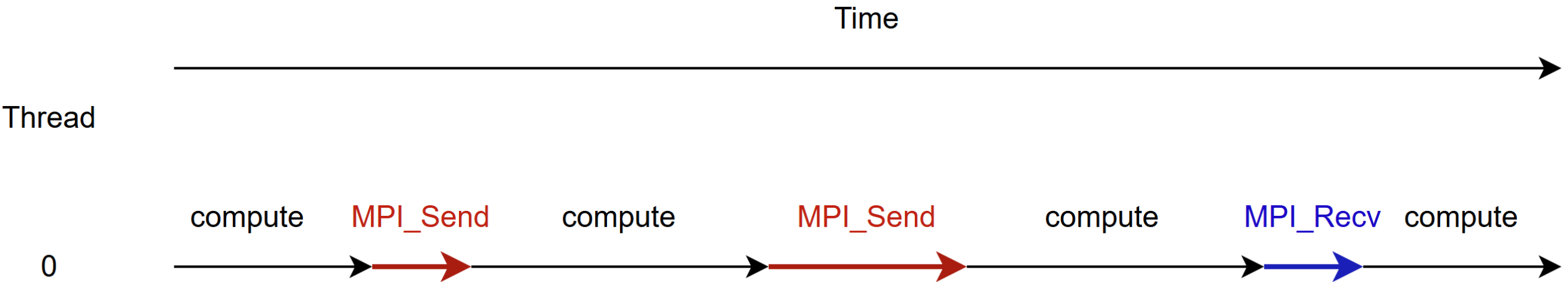
Supported «threading level»
(OUT)



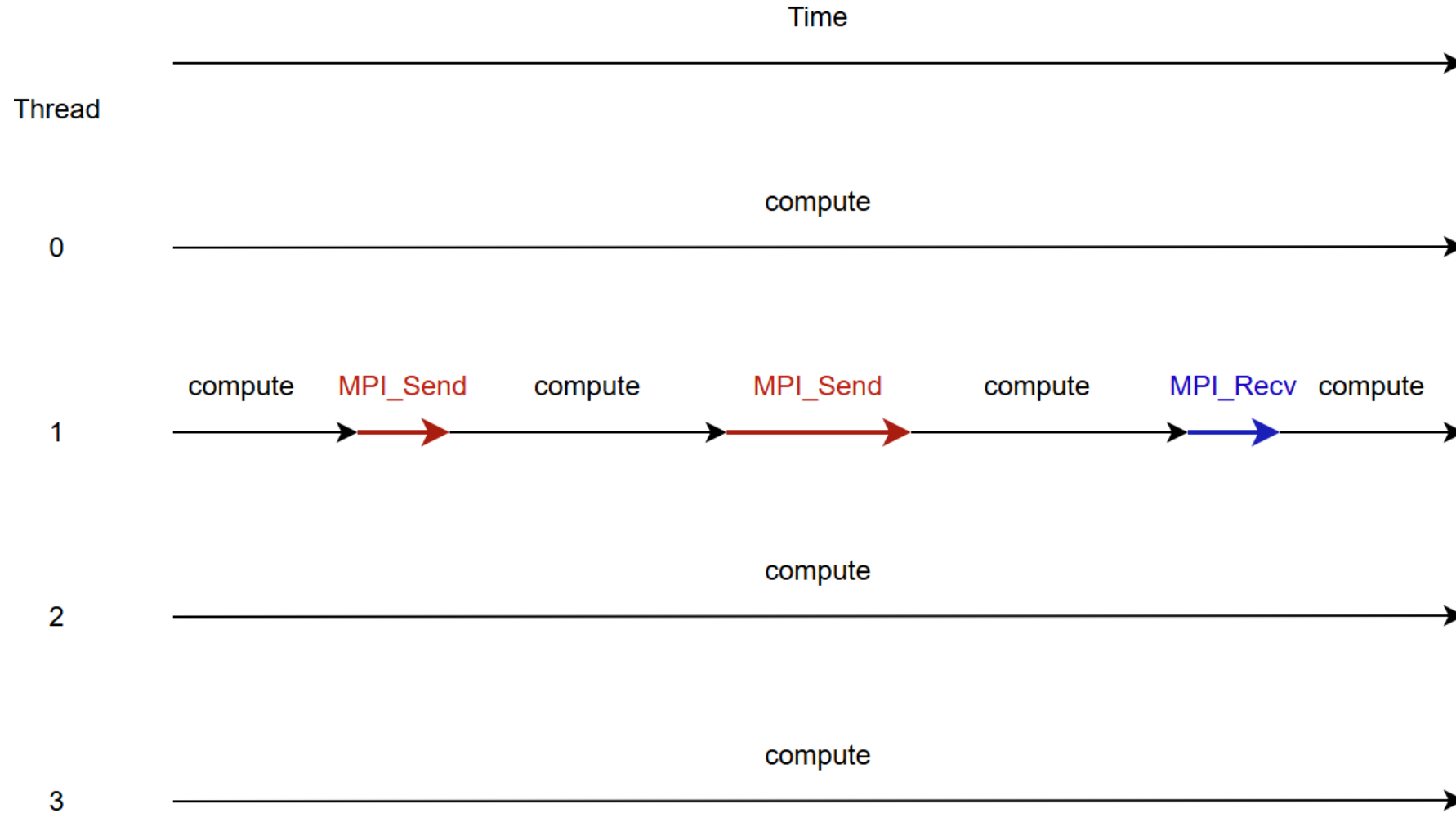
Threading Levels in MPI

- **MPI_THREAD_SINGLE:** rank is not allowed to use threads, which is basically equivalent to calling `MPI_Init`.
- **MPI_THREAD_FUNNELED:** rank can be multi-threaded but only the main thread may call MPI functions. Ideal for fork-join parallelism such as used in `#pragma omp parallel`, where all MPI calls are outside the OpenMP regions.
- **MPI_THREAD_SERIALIZED:** rank can be multi-threaded but only one thread at a time may call MPI functions. The rank must ensure that MPI is used in a thread-safe way. One approach is to ensure that MPI usage is mutually excluded by all the threads, eg. with a mutex.
- **MPI_THREAD_MULTIPLE:** rank can be multi-threaded and any thread may call MPI functions. The MPI library ensures that this access is safe across threads. Note that this makes all MPI operations less efficient, even if only one thread makes MPI calls, so should be used only where necessary.
- **ATTENTION:** Not all the threading levels are supported by all the MPI implementations (e.g., some implementations might not support `MPI_THREAD_MULTIPLE`)

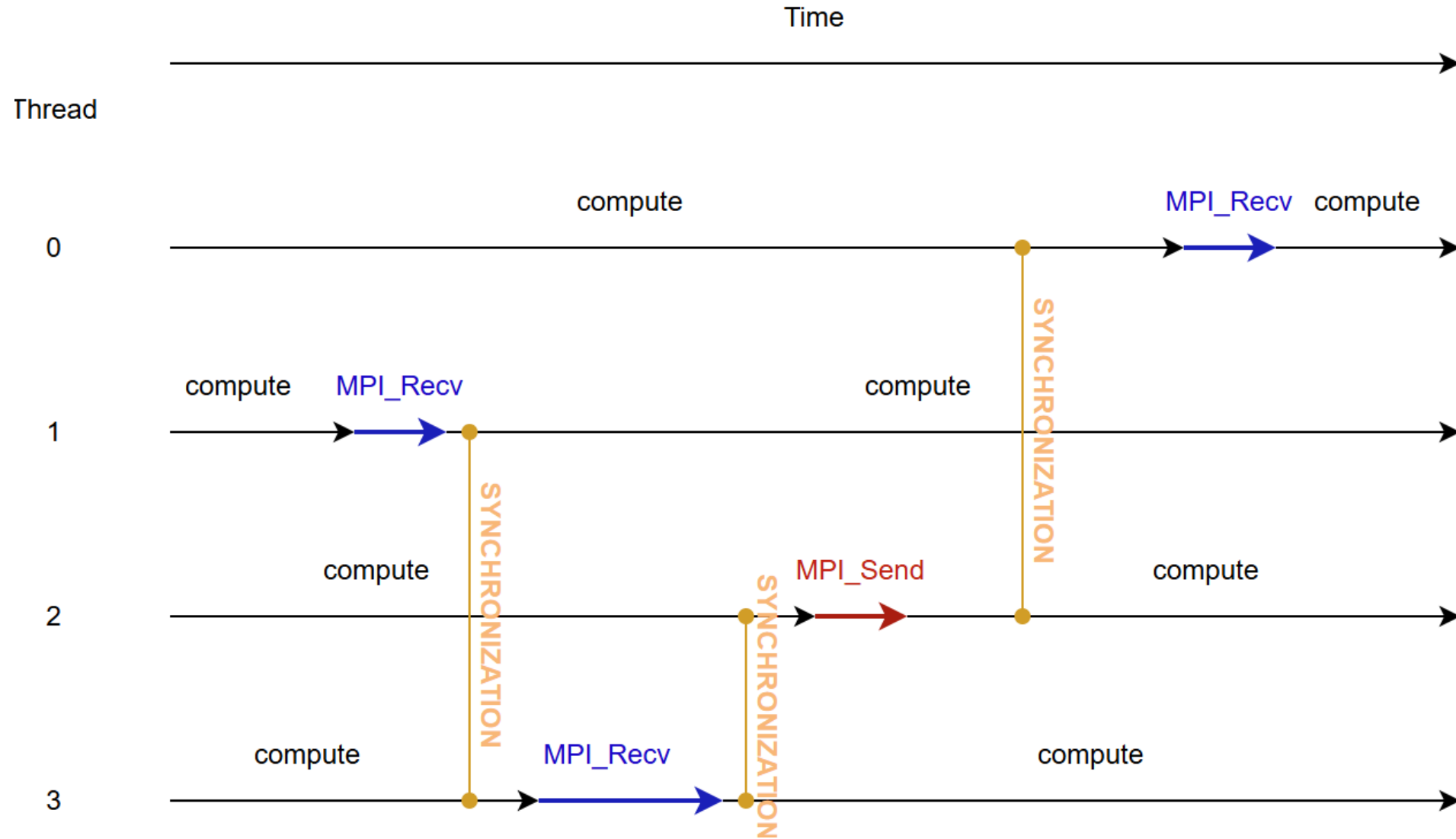
Threading Levels in MPI - Single



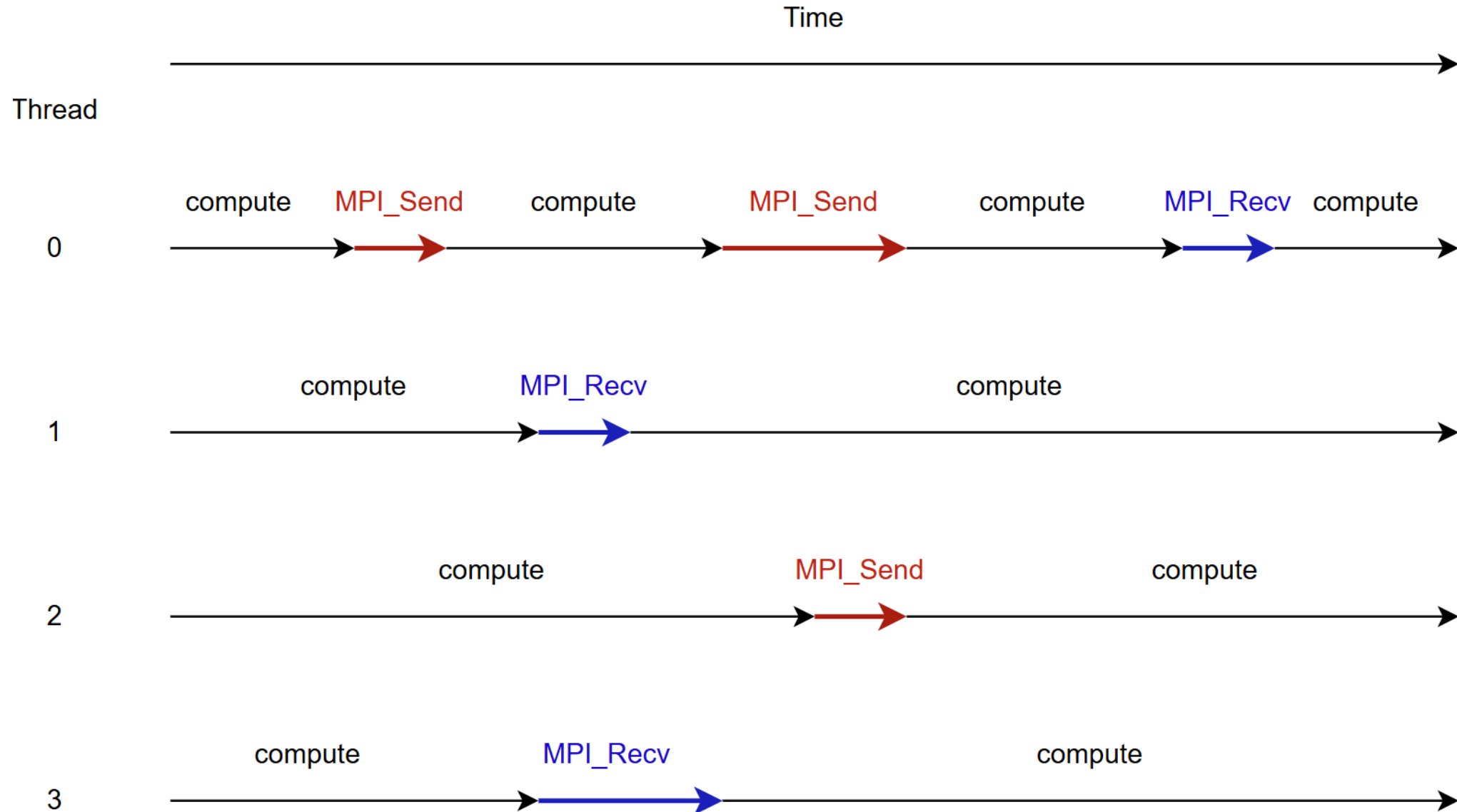
Threading Levels in MPI - Funneled



Threading Levels in MPI - Serialized



Threading Levels in MPI - Multiple



Static Initializers

You can use the following instead of, e.g., `pthread_mutex_init`:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Thread Pinning

What if I want to force a thread to run on a specific core?

```
// Do not include anything before the following three lines
```

```
#define _GNU_SOURCE
```

```
#include <pthread.h>
```

```
#include <sched.h>
```

```
void* thread_func(void* thread_args){
```

```
    ...
```

```
    cpu_set_t cpuset;
```

```
    pthread_t thread = pthread_self();
```

```
    /* Set affinity mask to include core 3 */
```

```
    CPU_ZERO(&cpuset);
```

```
    CPU_SET(3, &cpuset);
```

```
    s = pthread_setaffinity_np(thread, sizeof(cpuset), &cpuset);
```

```
    ...
```

```
}
```

Timing Code

```
#include <sys/time.h>

#define GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}

int main(){
    ...
    double start, finish, elapsed;
    GET_TIME(start);
    ...
    // Code to be timed
    ...
    GET_TIME(finish);
    elapsed = finish - start;
    printf("The code to be timed took %e seconds\n", elapsed);
    ...
}
```

Questions?

Concluding Remarks (1)

- A thread in shared-memory programming is analogous to a process in distributed memory programming.
- However, a thread is often lighter-weight than a full-fledged process.
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.

Concluding Remarks (2)

- When indeterminacy results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a **race condition**.

Concluding Remarks (3)

- A **critical section** is a block of code that updates a shared resource that can only be updated by one thread at a time.
- So the execution of code in a critical section should, effectively, be executed as serial code.

Concluding Remarks (4)

- **Busy-waiting** can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.
- It can be very wasteful of CPU cycles.
- It can also be unreliable if compiler optimization is turned on.

Concluding Remarks (5)

- A **mutex** can be used to avoid conflicting access to critical sections as well.
- Think of it as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.

Concluding Remarks (6)

- A **semaphore** is the third way to avoid conflicting access to critical sections.
- It is an unsigned int together with two operations: `sem_wait` and `sem_post`.
- Semaphores are more powerful than mutexes since they can be initialized to any nonnegative value.

Concluding Remarks (7)

- A **barrier** is a point in a program at which the threads block until all of the threads have reached it.
- A **read-write lock** is used when it's safe for multiple threads to simultaneously read a data structure, but if a thread needs to modify or write to the data structure, then only that thread can access the data structure during the modification.

Concluding Remarks (8)

- Some C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.
- This type of function is not **thread-safe**.

Questions?

Multicore Architecture

Caching

Caching

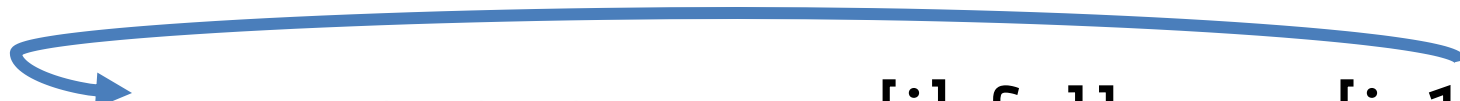
- A collection of memory locations that can be accessed in less time than some other memory locations.
- A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory (it is usually physically closer).
- Also uses more performing (but also more expensive) technology (e.g., SRAM instead of DRAM). So it is going to be faster but smaller

What to cache

- We assume locality (of both for instructions and data). I.e., accessing one location is followed by an access of a nearby location
- **Spatial locality** – accessing a nearby location.
- **Temporal locality** – accessing it in the near future.

Locality: example

float $z[1000]$; $z[i]$ follows $z[i-1]$ in memory
(spatial locality)



$\cdot \quad \cdot \quad \cdot$
 $sum = 0.0$; (temporal locality) $Accesses\ to\ z[i]\ follows\ access\ to\ z[i-1]$



for ($i = 0$; $i < 1000$; $i++$)
 $sum += z[i]$;

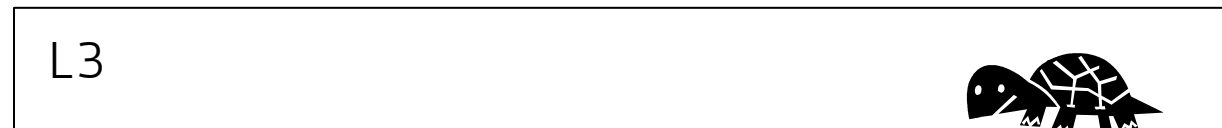
Cache lines

- Data is transferred from memory to cache in **blocks/lines**
- i.e., when $z[0]$ is transferred from memory to cache, also $z[1]$, $z[2]$, ..., $z[15]$ might be transferred
- Doing one transfer of 16 memory locations, is better than doing 16 transfers of one memory location each
- When accessing $z[0]$ you need to wait for the transfer, but then you will find the other 15 elements in cache already

Questions?

Cache levels

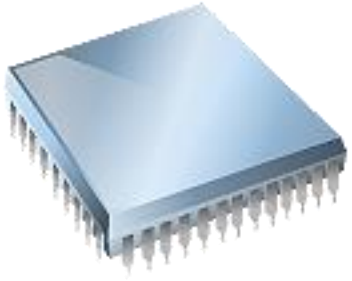
smallest & fastest



largest & slowest

- Data stored in L1 might or might not be stored in L2/L3 as well (it depends on the type of cache)
- The CPU first checks if the data is in L1, if not, checks in L2, etc...

Cache hit



fetch x

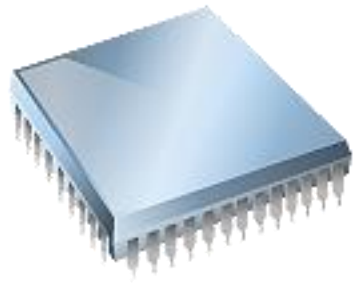
| | | |
|----|---|-----|
| L1 | x | sum |
|----|---|-----|

| | | | |
|----|---|---|-------|
| L2 | y | z | total |
|----|---|---|-------|

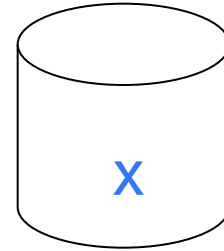
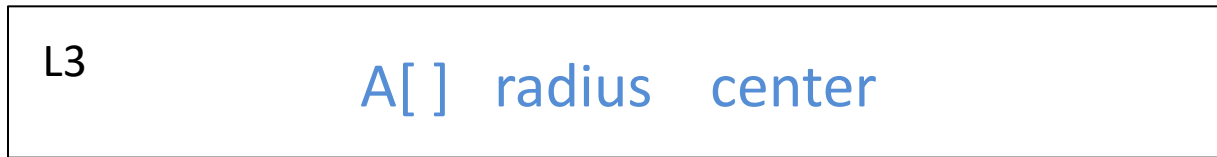
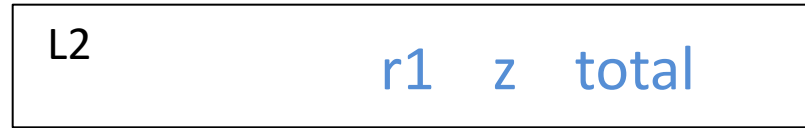
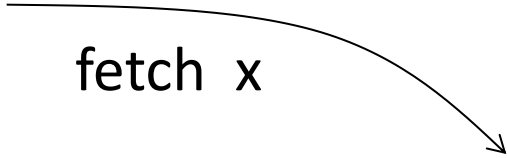
| | | | | |
|----|------|--------|----|--------|
| L3 | A[] | radius | r1 | center |
|----|------|--------|----|--------|

L1 hit!

Cache miss



fetch x



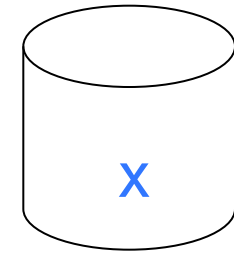
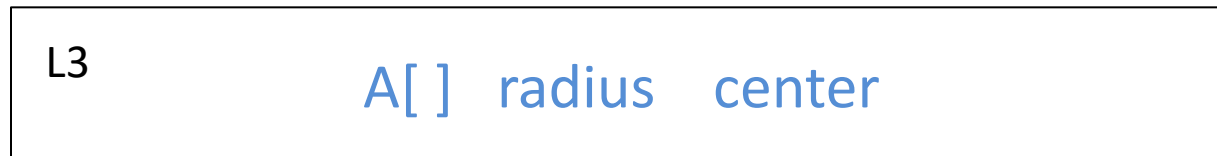
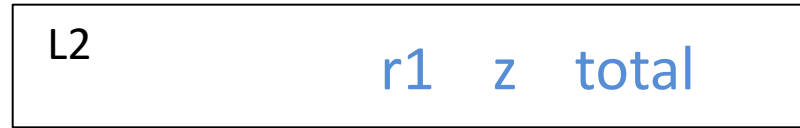
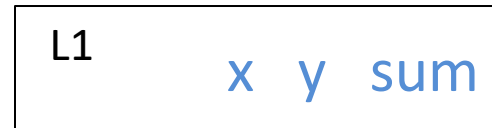
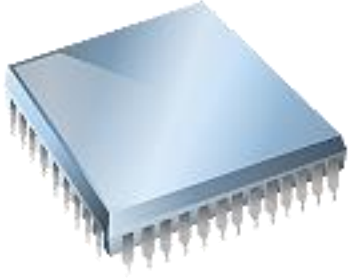
main
memory

L1 miss! L2 miss! L3 miss! Main memory hit!

Why do we care?

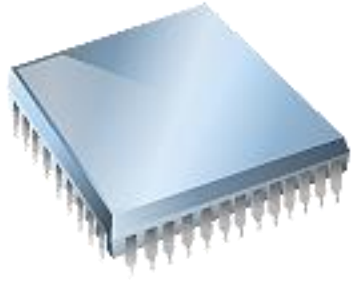
- To write efficient/performant parallel code:
 - Its sequential parts must be efficient/performant
 - Try to think about how your application accesses the data.
Random accesses are much worst than linear accesses
 - The coordination between these sequential parts must be done efficiently

Consistency

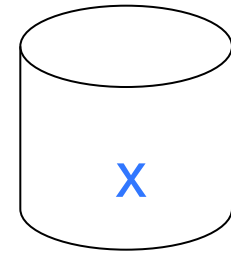
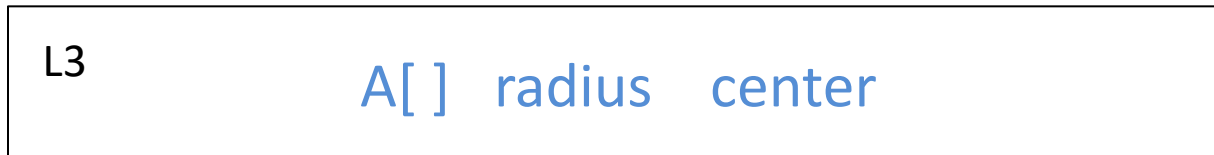


main
memory

Consistency



update x to x'



main
memory

Consistency

- When a CPU writes data to cache, the value in cache may be inconsistent with the value in main memory.
- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.
- **Write-back** caches mark data in the cache as **dirty**. When the cache line is replaced by a new cache line from memory, the **dirty** line is written to memory.

Exercises on caches

- Github, lec12, cache_01_*.c
- Use "perf" to measure number of cache misses. What's happening? How to fix cache_01_slow.c?
- You can do that at a finer grain from the code itself, by using a library called PAPI (<https://icl.utk.edu/papi/>)
- Similar effects can happen when swapping from memory to disk

Questions?

Caching on Multicores

Cache coherence

- Programmers have no control over caches and when they get updated.

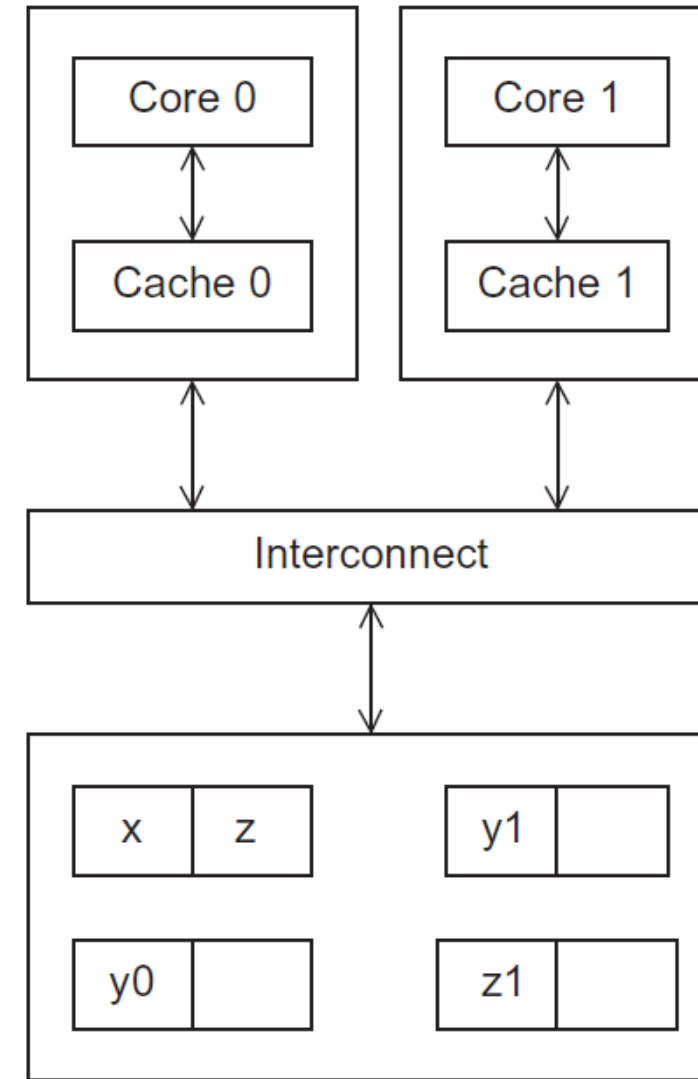


Figure 2.17

A shared memory system with two cores and two caches

Cache coherence

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

| Time | Core 0 | Core 1 |
|------|---------|-----------|
| 0 | y0 = x; | y1 = 3*x; |

Core 0 cache:

x = 2

y0 = 2

Core 1 cache:

x = 2

y1 = 6

Cache coherence

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

| Time | Core 0 | Core 1 |
|------|---------|------------------------------|
| 0 | y0 = x; | y1 = 3*x; |
| 1 | x = 7; | Statement(s) not involving x |

Core 0 cache:

x = 7

y0 = 2

Core 1 cache:

x = 2

y1 = 6

Cache coherence

y0 privately owned by Core 0

y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

| Time | Core 0 | Core 1 |
|------|------------------------------|------------------------------|
| 0 | y0 = x; | y1 = 3*x; |
| 1 | x = 7; | Statement(s) not involving x |
| 2 | Statement(s) not involving x | z1 = 4*x; |

Core 0 cache:

x = 7

y0 = 2

Core 1 cache:

x = 2

y1 = 6

z1 = 4*2 or 4*7?

N.B.: occurs for both WT and WB policies

Snooping Cache Coherence

- The cores share a bus .
- Any signal transmitted on the bus can be “seen” by all cores connected to the bus.
- When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.
- If core 1 is “snooping” the bus, it will see that x has been updated and it can mark its copy of x as invalid.
- Not much used anymore: broadcast is expensive, nowadays we have multicores with 64/128 cores

Directory Based Cache Coherence

- Uses a data structure called a **directory** that stores the status of each cache line.
- E.g., a bitmap/list saying which cores has a copy of that line
- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are **invalidated**.

Exercises on caches

- Github, lec12, cache_fs_*.c

False Sharing

- Data is fetched for memory to cache in lines. Each line can contain several variables
- E.g., if a cache is 64 Bytes long, it can contain 16 4-byte integers (which were consecutive in memory)
- When data is invalidated, the entire line is invalidate
- Even if two threads access two different variables, if those are on the same cache line, this would still cause an invalidation

False Sharing - Solution

- Try to force variables which are accessed by different threads to be on different cache lines
- Padding (but be aware, the compiler might change the order of the fields in a struct). To avoid that, do something like (for GCC, assuming 64 bytes cache lines):

```
struct alignTo64ByteCacheLine {  
    int _onCacheLine1 __attribute__((aligned(64)))  
    int _onCacheLine2 __attribute__((aligned(64)))  
}
```

- How to get the cache line size?
 - From the code: `sysconf (_SC_LEVEL1_DCACHE_LINESIZE)`
 - From the shell: `getconf LEVEL1_DCACHE_LINESIZE`
- Do all the updates on a variable local to the thread (e.g., allocated on the stack, and then write it on heap only at the end)

Exercise

```
perf stat -e cache-misses ./cache_fs_slow  
perf stat -e cache-misses ./cache_fs_fast
```

Alternatively, if supported:

```
perf c2c record ./cache_fs_slow  
perf c2c report
```

Questions?

Matrix-Vector Multiplication

Pthreads matrix-vector multiplication

```
void *Pth_mat_vect(void* rank) {  
    long my_rank = (long) rank;  
    int i, j;  
    int local_m = m/thread_count;  
    int my_first_row = my_rank*local_m;  
    int my_last_row = (my_rank+1)*local_m - 1;  
  
    for (i = my_first_row; i <= my_last_row; i++) {  
        y[i] = 0.0;  
        for (j = 0; j < n; j++)  
            y[i] += A[i][j]*x[j];  
    }  
  
    return NULL;  
} /* Pth_mat_vect */
```

Run-times and efficiency

Matrix size denoted as # Rows x # Cols

| Threads | Matrix Dimension | | | | | |
|---------|------------------|-------|-------------|-------|---------------|-------|
| | 8,000,000 × 8 | | 8000 × 8000 | | 8 × 8,000,000 | |
| | Time | Eff. | Time | Eff. | Time | Eff. |
| 1 | 0.393 | 1.000 | 0.345 | 1.000 | 0.441 | 1.000 |

In both cases we perform the same number of operations (64,000,000).
Why 8 x 8M takes more than 8K x 8K?

| | | | |
|-------------|-------------|----------|---------------|
| a_{00} | a_{01} | \cdots | $a_{0,n-1}$ |
| a_{10} | a_{11} | \cdots | $a_{1,n-1}$ |
| \vdots | \vdots | | \vdots |
| a_{i0} | a_{i1} | \cdots | $a_{i,n-1}$ |
| \vdots | \vdots | | \vdots |
| $a_{m-1,0}$ | $a_{m-1,1}$ | \cdots | $a_{m-1,n-1}$ |

| |
|-----------|
| x_0 |
| x_1 |
| \vdots |
| x_{n-1} |

| |
|---|
| y_0 |
| y_1 |
| \vdots |
| $y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$ |
| \vdots |
| y_{m-1} |

The input vector x has 8,000,000 elements.
Probably I can't keep it between the computation of $y[i]$ and $y[i+1]$

Run-times and efficiency

Matrix size denoted as # Rows x # Cols

| Threads | Matrix Dimension | | | | | |
|---------|------------------|-------|-------------|-------|---------------|-------|
| | 8,000,000 × 8 | | 8000 × 8000 | | 8 × 8,000,000 | |
| | Time | Eff. | Time | Eff. | Time | Eff. |
| 1 | 0.393 | 1.000 | 0.345 | 1.000 | 0.441 | 1.000 |

In both cases we perform the same number of operations (64,000,000).
Why 8M x 8 takes more than 8K x 8K?

The output vector y has 8,000,000 elements rather than 8,000 (more cache misses).
Once you load $y[i]$, you only use it for 8 operations

Run-times and efficiency

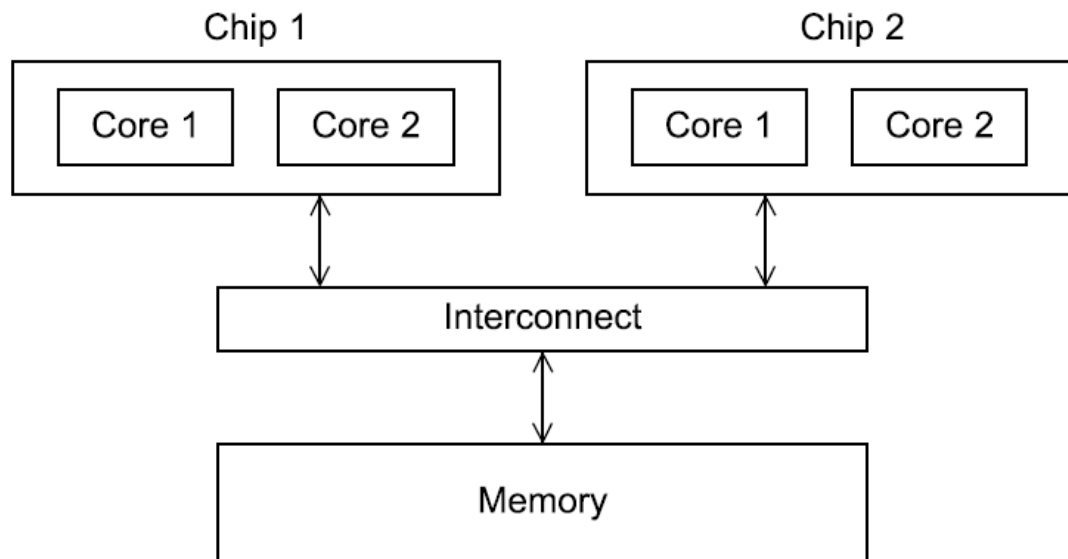
Matrix size denoted as # Rows x # Cols

| Threads | Matrix Dimension | | | | | |
|---------|------------------|-------|-------------|-------|---------------|-------|
| | 8,000,000 × 8 | | 8000 × 8000 | | 8 × 8,000,000 | |
| | Time | Eff. | Time | Eff. | Time | Eff. |
| 1 | 0.393 | 1.000 | 0.345 | 1.000 | 0.441 | 1.000 |
| 2 | 0.217 | 0.906 | 0.188 | 0.918 | 0.300 | 0.735 |
| 4 | 0.139 | 0.707 | 0.115 | 0.750 | 0.388 | 0.290 |

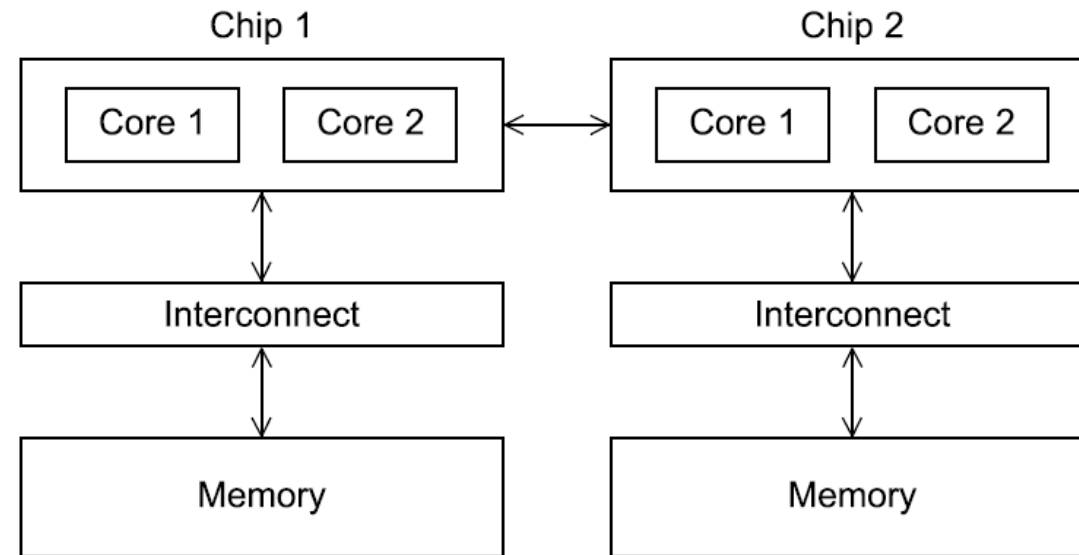
- Why the efficiency of the multithread version is much worst for 8 x 8M?
- The output vector y only has 8 rows. Let's say it is a vector of floats, with float on 8 bytes.
- The entire vector y would be 64 bytes, which would fit in a cache line.
Thus, even if each thread access a different y[i], we have false sharing issues since they are located on the same cache line
- If the number of rows is larger, this would still happen, but only in the "border" element, which would be a small fraction of the number of rows. E.g., in the 8,000x8,000 case, with 4 threads, each thread would compute 2,000 elements of y. Even if they falsely share 8 elements, this is just 8 out of 2,000, so it is not going to make much of a difference

Questions?

Memory Organization

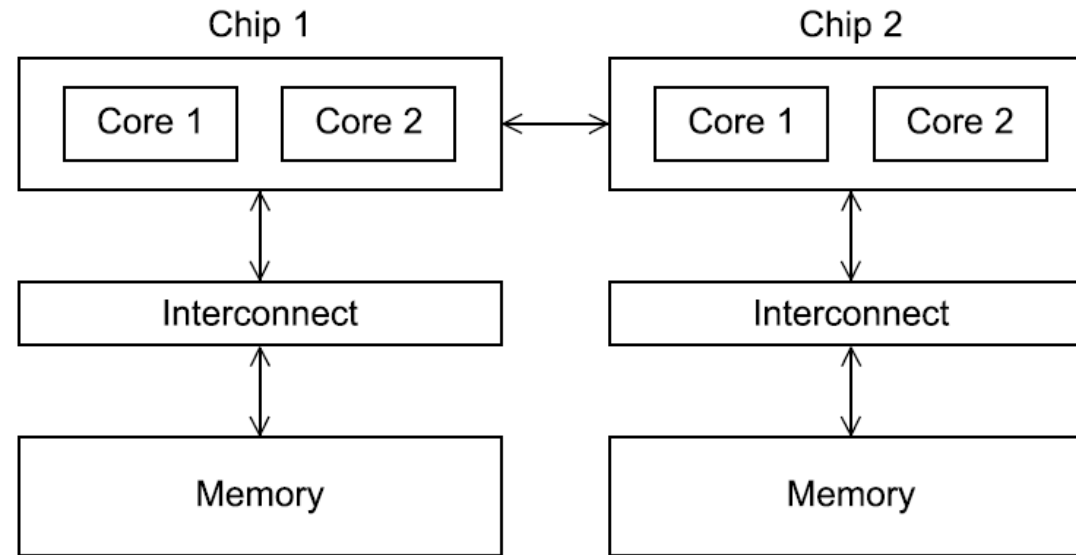


UMA
(Uniform Memory Access)



NUMA
(Non-Uniform Memory Access)

NUMA Systems



- The cost of accessing the memory changes depending on where the data is allocated
- Accessing a “local” memory is cheaper than accessing a “remote” memory
- Is it possible to specify where the data must be allocated (see `numa.h` library)
- It is possible to have some control from the command line (see `numactl`)

It is even more complicated...

Some cores might be «closer» to the NIC, so it might have sense that those cores are those doing MPI_Send/MPI_Recv/etc...

Questions?

Misc

Profiling

- Let's say I have a piece of code with many functions
- How to know where most of the time is spent and, thus, where it would make most sense to optimize? (Remember: *"Premature optimization is the root of all evil"*)
- We could explicitly time each function (boring, time consuming)
- We can use a profiler that does that for us (see https://hpc-wiki.info/hpc/Gprof_Tutorial)

Debugging

- Mostly through gdb (you can find many online resources)
- valgrind is a quite helpful tool as well

Chapter 5

Shared Memory Programming with OpenMP

Roadmap

- Writing programs that use OpenMP.
- Using OpenMP to parallelize many serial for loops with only small changes to the source code.
- Task parallelism.
- Explicit thread synchronization.
- Standard problems in shared-memory programming.

OpenMP

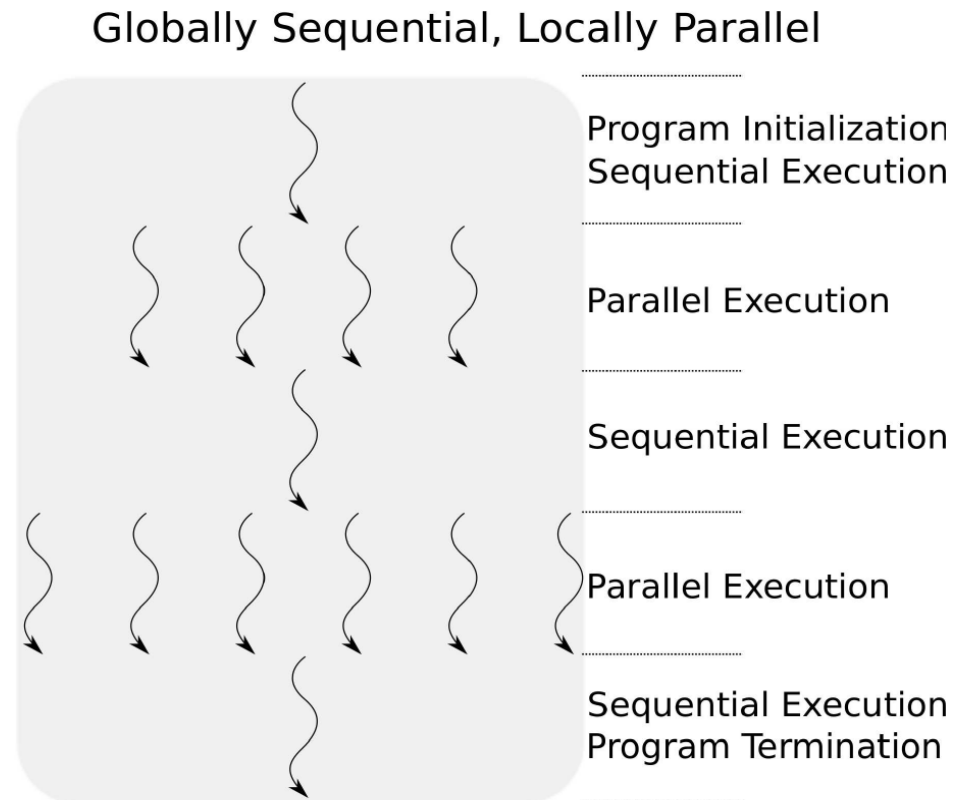
- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for shared-memory systems.
- System is viewed as a collection of cores or CPU's, all of which have access to main memory.

OpenMP

- OpenMP aims to decompose a sequential program into components that can be executed in parallel.
- OpenMP allows an “incremental” conversion of sequential programs into parallel ones, with the assistance of the compiler.
- OpenMP relies on compiler directives for decorating portions of the code that the compiler will attempt to parallelize.

OpenMP

- OpenMP programs are Globally Sequential, Locally Parallel.
- Programs follow the fork-join paradigm:



Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C specification.
- Compilers that don't support the pragmas ignore them.

`#pragma`

OpenMP pragmas

- `# pragma omp parallel`
 - Most basic parallel directive.
 - The number of threads that run the following structured block of code is determined by the run-time system.

```

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */

```

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

```
./omp_hello 4
```

compiling



running with 4 threads



```
Hello from thread 0 of 4  
Hello from thread 1 of 4  
Hello from thread 2 of 4  
Hello from thread 3 of 4
```

possible
outcomes



```
Hello from thread 1 of 4  
Hello from thread 2 of 4  
Hello from thread 0 of 4  
Hello from thread 3 of 4
```

```
Hello from thread 3 of 4  
Hello from thread 1 of 4  
Hello from thread 2 of 4  
Hello from thread 0 of 4
```

Thread Team Size Control

- **Universally:** via the `OMP_NUM_THREADS` environmental variable:
\$ echo \${OMP_NUM_THREADS} # to query the value
\$ export OMP_NUM_THREADS=4 # to set it in BASH
- **Program level :** via the `omp_set_num_threads` function, outside an OpenMP construct.
- **Pragma level :** via the `num_threads` clause.
- **Precedence:**
 - Universally/env. Variable
 - Program level
 - Pragma level

Thread Team Size Control

- The `omp_get_num_threads` call returns the active threads in a parallel region. If it is called in a sequential part it returns 1.
- The `omp_get_thread_num` returns the id of the calling thread (similar to the rank in MPI)



```

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();

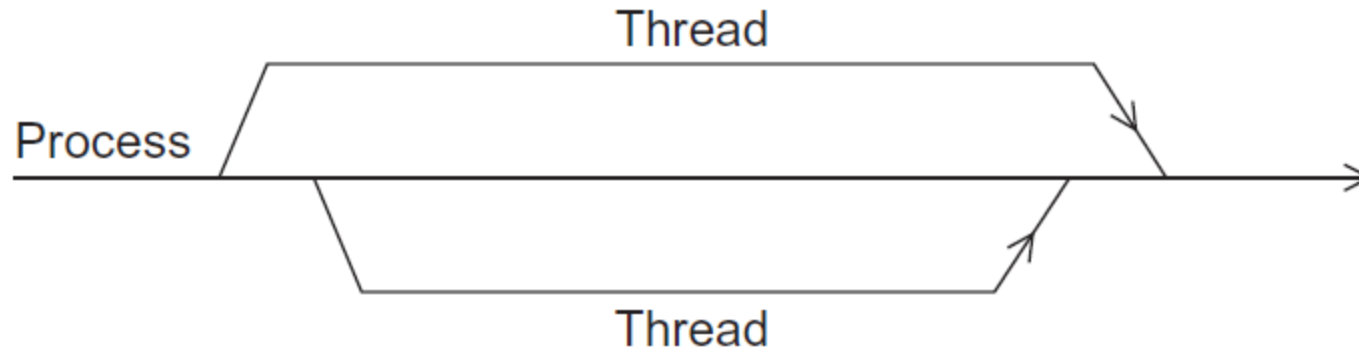
    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */

```

A process forking and joining two threads



clause

- Text that modifies a directive.
- The `num_threads` clause can be added to a `parallel` directive.
- It allows the programmer to specify the number of threads that should execute the following block.

```
# pragma omp parallel num_threads ( thread_count )
```


Of note...

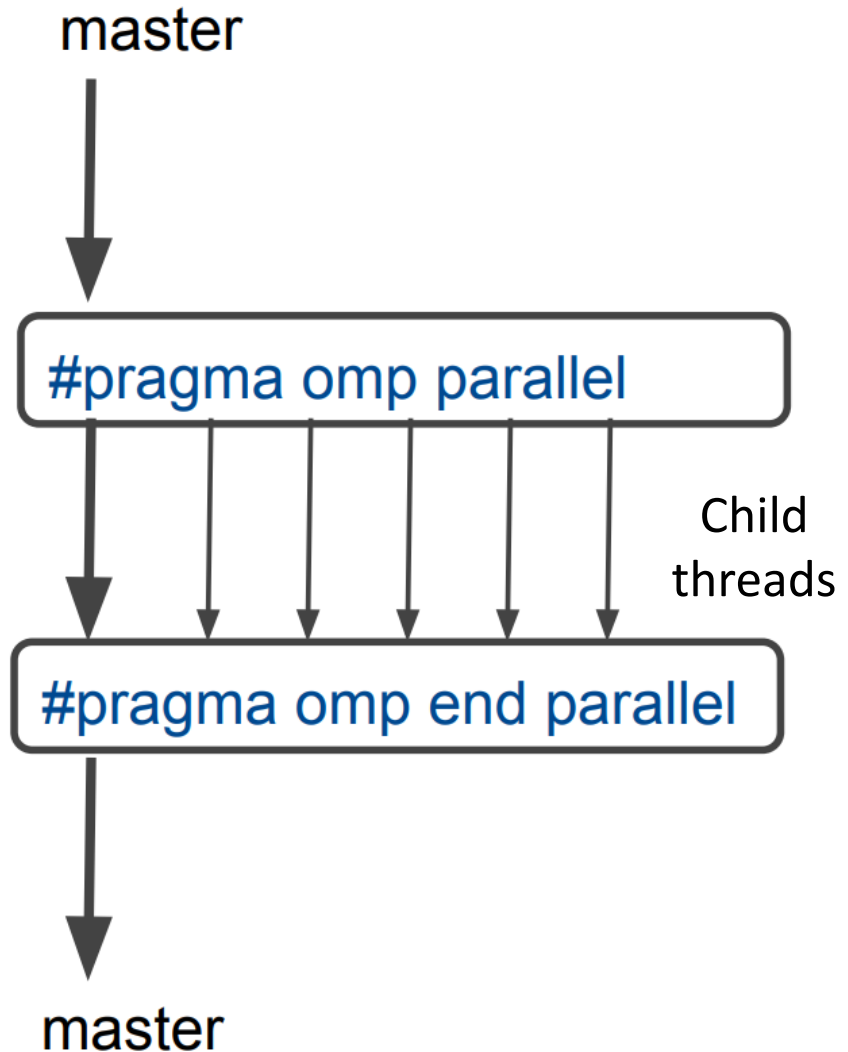
- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn't guarantee that this will actually start `thread_count` threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we're trying to start a lot of threads, we will almost always get the desired number of threads.
- After a block is completed, there is an implicit barrier.

Some terminology

- In OpenMP parlance the collection of threads executing the parallel block — the original thread and the new threads — is called a **team**
- **master:** the original thread of execution
- **parent:** thread that encountered a parallel directive and started a team of threads.
- In many cases, the parent is also the master thread.
- **child:** each thread started by the parent is considered a child thread.

Parallel Construct

```
#pragma omp parallel  
{  
...  
}
```



Parallel Construct

```
1  #include<stdio.h>
2  #include<omp.h>
3
4  int main()
5  {
6      printf("Only master thread here \n");
7
8      #pragma omp parallel
9      {
10         int tid = omp_get_thread_num();
11         printf("Hello I am thread number %d \n", tid);
12     }
13
14     printf("Only master thread here \n");
15 }
```

1

3

Only master thread here
Hello I am thread number 0
Hello I am thread number 1
Hello I am thread number 2
Hello I am thread number 3
Only master thread here

2

int tid

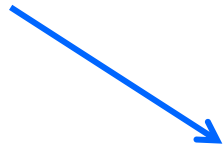
4

*Each thread has a
private copy of
variable*

- Specifying number of threads
 - `export OMP_NUM_THREADS=4`
- Execute:
 - `./basic.exe`

In case the compiler doesn't support OpenMP

```
# include <omp.h>
```



```
#ifdef _OPENMP  
# include <omp.h>  
#endif
```

In case the compiler doesn't support OpenMP

```
# ifdef _OPENMP
    int my_rank = omp_get_thread_num ( );
    int thread_count = omp_get_num_threads ( );
# else
    int my_rank = 0;
    int thread_count = 1;
# endif
```