UNIL | Université de Lausanne

Universität Zürich[UZH]

Basics on code optimization & OpenMP I

Simon Scheidegger
simon.scheidegger@gmail.com
July 12th, 2018
Open Source Macroeconomics Laboratory – BFI/UChicago

Including adapted teaching material from books, lectures and presentations by
B. Barney, B. Cumming, G. Hager, R. Rabenseifner, O. Schenk, G. Wellein

This lecture: Basics on code optimization & OpenMP session I

1. Basic optimization of serial code

2. Shared memory parallelism – OpenMP (background)

3. Exploring the some first features of OpenMP

Some literature & other resources

Full standard/API specification:
- http://openmp.org

Tutorials:
- https://computing.llnl.gov/tutorials/openMP/

Books:
- “Introduction to High Performance Computing for Scientists and Engineers”
  Georg Hager, Gerhard Wellein

Basic optimization of serial code

In the age of multi-1000-processor parallel computers, writing code that runs efficiently on a single CPU has grown slightly old-fashioned.

Nevertheless there can be no doubt that single-processor optimizations are of premier importance.

→ If a **speed-up of two** can be achieved by some **simple code changes**, the user will be satisfied with much fewer CPUs in the parallel case.

→ This frees resources for other users and projects, and puts hardware that was often acquired for considerable amounts of money to better use.

→ If an existing parallel code is to be optimized for speed, it must be the first goal to **make the single processor** run as fast as possible.

Profiling*

- Gathering information about a program's behaviour, specifically its use of resources, is called **profiling**.

- The most important "resource" in terms of high performance computing is **runtime**.

→ Hence, a common profiling strategy is to find out how much time is spent in the different functions, and maybe even lines, of a code in order to **identify hot spots**, i.e., the parts of the program that require the dominant fraction of runtime.

- These hot spots are subsequently analysed for possible optimization opportunities.

→ Software for profiling (e.g. GNU gprof,...)

Definition of Profiling

The performance needs of software vary, but it's probably not surprising that many applications have **very stringent speed requirements**.

In general, for all but the simplest applications, the better the performance, the more useful and popular the application will be. For this reason, performance considerations are (or should be) in the forefront of many application developers' minds.

Unfortunately, much of the effort that is expended attempting to make applications faster is wasted, because **developers will often micro-optimize their software** without fully exploring how the program operates at a macro scale. For instance, you might spend a large amount of time making a particular function run twice as fast, which is all well and good, but if that function is called very rarely (when a file is opened, say) then reducing the execution time from 200ms to 100ms isn't going to make much difference to the overall execution time of the software.

A more fruitful use of your time would be spent optimizing those parts of the software that are called more frequently.

It is therefore vital that you **have accurate information on exactly where the time is being spent within your applications** -- and for real input data -- if you hope to have a chance of optimizing it effectively. This activity is called code profiling.

Profiling a Program → Where does It spend Its Time?

GNU compiler – gprof

https://sourceware.org/binutils/docs/gprof/

In general, code should be written with the following three goals, in order of importance:

1. **Make the software work correctly**. This must always be the focus of development. In general, there is no point writing software that is very fast if it does not do what it is supposed to! Obviously, correctness is something of a grey area; a video player that works on 99 percent of your files or plays video with the occasional visual glitch is still of some use, **but in general, correctness is more important than speed**.

2. **Make the software maintainable**. This is really a sub-point of the first goal. In general, if software is not written to be maintainable, then even if it works to begin with, sooner or later you (or someone else) will end up breaking it trying to fix bugs or add new features.

3. **Make the software fast**. Here is where profiling comes in. Once the software is working correctly, then start profiling to help it run more quickly.

→ gprof can profile C, C++, and Fortran applications.

Profiling enabled while compilation

In this first step, we need to make sure that the profiling is enabled when the compilation of the code is done. This is made possible by adding the '**-pg**' option in the compilation step.

-pg : Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

→ compile code: > g++ **-gp** myfile.cpp -o example_gprof.exec

→ run code: **>./example_gprof.exec**

→ **gmon.out** was generated, contains all profiling information

→ run gprof >gprof "NameOfYourExecutable", i.e. **>gprof example_gprof.exec** (probably pipe it into some file by **>gprof example_gprof.exec | tee out.txt**

Example gprof

https://people.sc.fsu.edu/~jburkardt/f_src/gprof/gprof.html

1. Go to the folder

> cd OSM2018/day2/code_day2/gprof_example

2. Compile the code by typing

> make

3. run the code in debug

>./example_gprof.exec (FORTRAN)

>./example_gprof_cpp.exec (CPP)

>gprof example_gprof.exec | tee profile.txt

4. Look at the output

>less profile.txt

First chunk of output

This time consists only of
time spent in this function, and
not in anything that function calls!!

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 83.99      0.47     0.47   501499     0.00     0.00  daxpy_
 10.72      0.53     0.06  2000000     0.00     0.00  d_random_
  3.57      0.55     0.02        1    20.01   498.49  dgefa_
  1.79      0.56     0.01      999     0.01     0.01  idamax_
  0.00      0.56     0.00      993     0.00     0.00  d_swap_
  0.00      0.56     0.00        2     0.00    30.02  d_matgen_
  0.00      0.56     0.00        2     0.00     0.00  timestamp_
  0.00      0.56     0.00        1     0.00   560.41  MAIN__
  0.00      0.56     0.00        1     0.00     1.87  dgesl_

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
»          else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
»          function is profiled, else blank.
```

Second chunk of output

The second table is different.
It displays the time a function takes, and everything it calls.

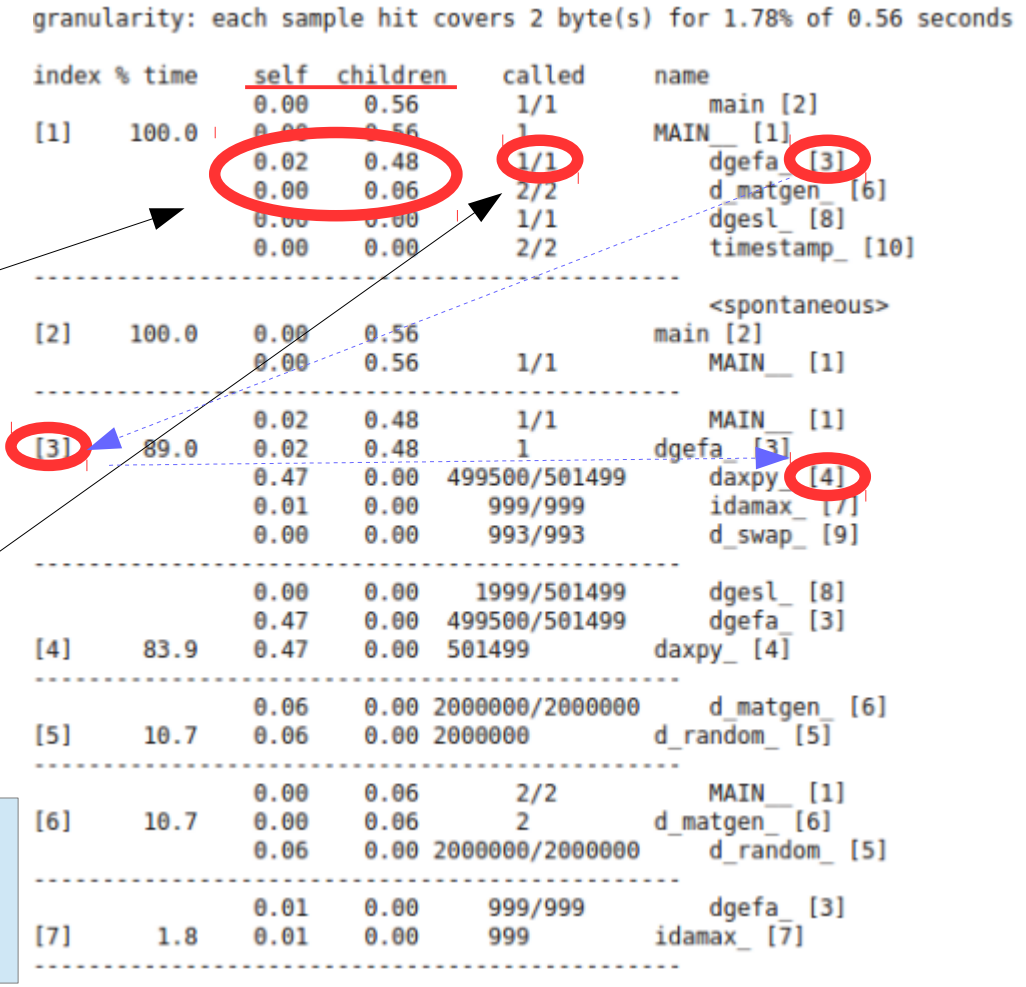
→ main takes all the time.

→ this table contains information on **how much time a function spends in all of his children**.

→ e.g. dgefa_ was called 1x from main.

Note: some compiler switch on optimizations automatically.

→ **never run with -O3 for profiling!**
→ use **-O2** flag in that case! (this should not re-arrange your code too much.



```
granularity: each sample hit covers 2 byte(s) for 1.78% of 0.56 seconds

index % time    self  children    called     name
                0.00    0.56       1/1          main [2]
[1]    100.0    0.00    0.56       1          MAIN__ [1]
                0.02    0.48       1/1          dgefa_ [3]
                0.00    0.06       2/2          d_matgen_ [6]
                0.00    0.00       1/1          dgesl_ [8]
                0.00    0.00       2/2          timestamp_ [10]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    0.56                    main [2]
                0.00    0.56       1/1          MAIN__ [1]
-----------------------------------------------
                0.02    0.48       1/1          MAIN__ [1]
[3]     89.0    0.02    0.48       1          dgefa_ [3]
                0.47    0.00  499500/501499      daxpy_ [4]
                0.01    0.00     999/999         idamax_ [7]
                0.00    0.00     993/993         d_swap_ [9]
-----------------------------------------------
                0.00    0.00    1999/501499      dgesl_ [8]
                0.47    0.00  499500/501499      dgefa_ [3]
[4]     83.9    0.47    0.00  501499          daxpy_ [4]
-----------------------------------------------
                0.06    0.00 2000000/2000000     d_matgen_ [6]
[5]     10.7    0.06    0.00 2000000         d_random_ [5]
-----------------------------------------------
                0.00    0.06       2/2          MAIN__ [1]
[6]     10.7    0.00    0.06       2          d_matgen_ [6]
                0.06    0.00 2000000/2000000     d_random_ [5]
-----------------------------------------------
                0.01    0.00     999/999         dgefa_ [3]
[7]      1.8    0.01    0.00     999          idamax_ [7]
-----------------------------------------------
```

Questions?

1. Advice – http://lmgtfy.com/

http://lmgtfy.com/?q=gprof

Common sense optimizations

- Very simple code changes can often lead to a significant performance boost.
- Some of those hints may **seem trivial**, but experience shows that many scientific codes can be improved by the simplest of measures.

→ **e.g. do less work**

```
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4   if(complex_func(A(i)) < THRESHOLD) then
5     FLAG = .true.
6   endif
7 enddo
```

If complex_func() has no side effects, the only information that gets communicated to the outside of the loop is the value of **FLAG**. In this case, depending on the probability for the conditional to be true, much computational effort can be saved by leaving the loop as soon as FLAG changes state.

EXIT THE LOOP →

```
1 logical :: FLAG
2 FLAG = .false.
3 do i=1,N
4   if(complex_func(A(i)) < THRESHOLD) then
5     FLAG = .true.
6     exit
7   endif
8 enddo
```

Common sense optimizations II

Avoid branching!

In this multiplication of a matrix with a vector, the upper and lower triangular parts get different signs and the diagonal is ignored. The **if** statement serves to decide about which factor to use.

→ Fortunately, the loop nest can be transformed so that all if statements vanish:

```
1  do j=1,N
2    do i=1,N
3      if(i.ge.j) then
4        sign=1.d0
5      else if(i.lt.j) then
6        sign=-1.d0
7      else
8        sign=0.d0
9      endif
10     C(j) = C(j) + sign * A(i,j) * B(i)
11   enddo
12 enddo
```

```
1  do j=1,N
2    do i=j+1,N
3      C(j) = C(j) + A(i,j) * B(i)
4    enddo
5  enddo
6  do j=1,N
7    do i=1,j-1
8      C(j) = C(j) - A(i,j) * B(i)
9    enddo
10 enddo
```

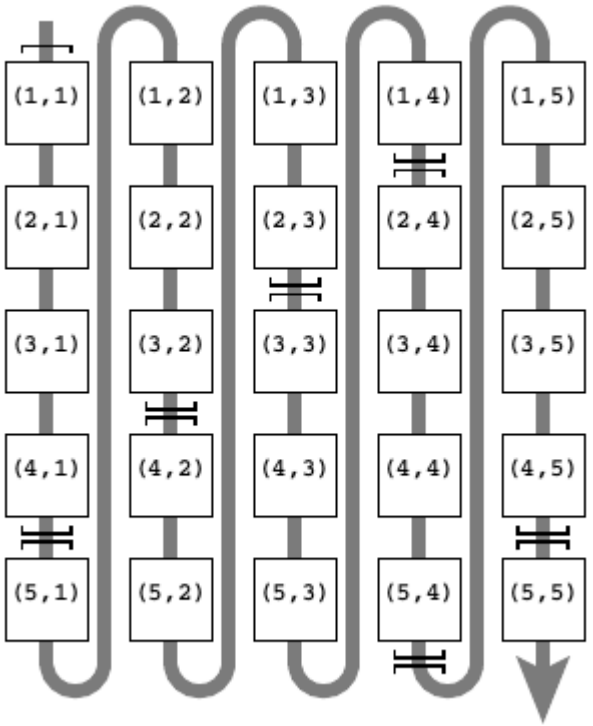
Data access (example)

Stride-N access

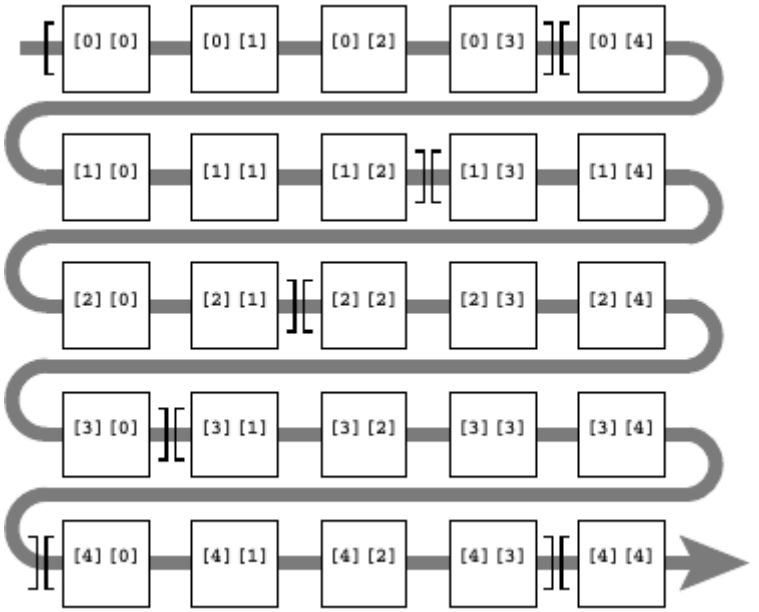
```
1  do i=1,N
2    do j=1,N
3      A(i,j) = i*j
4    enddo
5  enddo
```

Stride-1 access

```
for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    a[i][j] = i*j;
  }
}
```



Fortran: Column major



C: Row major

Small exercise on loop ordering

check directory:

> cd OSM2018/day2/code_day2/loop/loop.f90

compile:

>./compile_loop.sh

run:

>./test_loop

To be done:

a) Inspect code

b) run code (play with array size & re-compile)

Using SIMD instruction sets → Vectorization

Vectorization performs multiple operations in parallel on a core with a single instruction (SIMD – single instruction multiple data).

Data is loaded into vector registers that are described by their width in bits:

-256 bit registers: 8 x float, or 4x double
-512 bit registers: 16x float, or 8x double

Vector units perform arithmetic operations on vector registers simultaneously.

Vectorization is key to maximising computational performance.

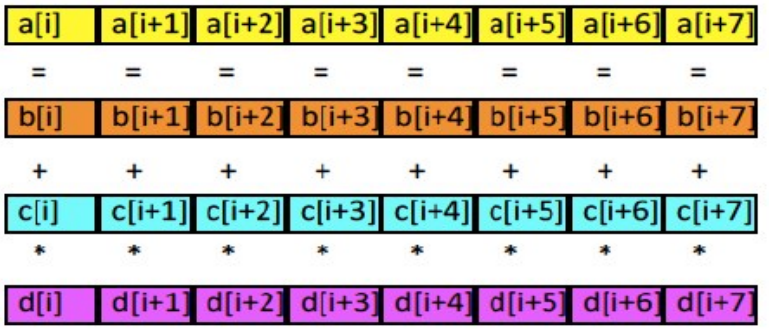
Vectorization illustrated

sequential

```
a [i] = b[i] + c[i] * d[i];
```

8× vectorization

```
a [i:8] = b[i:8] + c[i:8] * d[i:8];
```



In an optimal situation all this is carried out by the compiler automatically. Compiler directives can be used to give hints as to where vectorization is safe and/or beneficial.


```
1  ! vectorized part
2  rest = mod(N,4)
3  do i=1,N-rest,4
4    load R1 = [x(i),x(i+1),x(i+2),x(i+3)]
5    load R2 = [y(i),y(i+1),y(i+2),y(i+3)]
6    ! "packed" addition (4 SP flops)
7    R3 = ADD(R1,R2)
8    store [r(i),r(i+1),r(i+2),r(i+3)] = R3
9  enddo
10 ! remainder loop
11 do i=N-rest+1,N
12   r(i) = x(i) + y(i)
13 enddo
```

Advanced Vector Extensions (AVX)

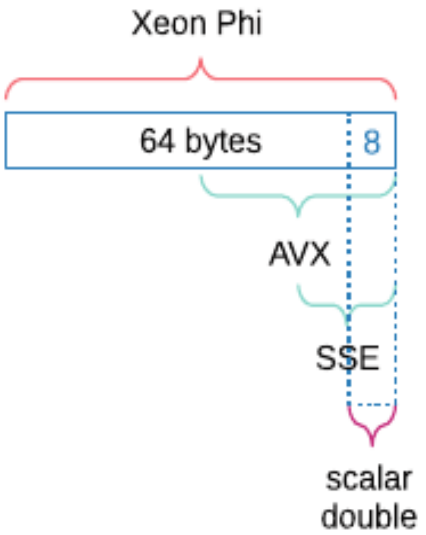


Fig. 7. Vector registers on modern CPUs: a scalar program can utilize only 1/4 of computational parallelism on AVX-enabled CPUs, e.g. the SandyBridge.

How to use vectorization

- use vector intrinsics (**see example**)
  - explicit hardware-specific instructions.
  - high performance.
  - non-portable and hard to maintain.
  - → see, e.g. https://software.intel.com/sites/landingpage/IntrinsicsGuide/

-automatic compiler vectorization
  - compiler will vectorize where it is possible.
  - compilers can do a poor job.

-use libraries that are already vectorized
  - let somebody else do the work for you.

Does my code vectorize?

→ Not clear a priori.

→ Compilers can generate reports that summarise which loops vectorized.

→ You can ask for different levels of detail e.g. only loops that failed to vectorize e.g. whether to explain why a loop didn't vectorize.

→ The flags vary from compiler to compiler, e.g.:

- **Intel** : `-vec-report=n`, or `-opt-report=n`
- **GCC**: `-ftree-vectorizer-verbose=n`
- **Cray**: `-h list=a`

You can also use the `disassemble` command in **gdb***, if you like reading assembly.

*we will address the gnu debugger (gdb) later in this lecture suite

Small example on vectorization

check directory:

> cd OSM2018/day2/code_day2/vectorization/avx-intrinsics.cpp

compile:

>./compile_avx.sh

run:

>./avx-example

To be done:

a) Inspect code

Compilers

- Most high-performance codes benefit, to varying degrees, from employing **compiler-based optimizations**, e.g. standard optimization options (**-O0, -O1,. . .**).

- Every modern compiler has command line switches that allow a (more or less) fine-grained tuning of the available optimization options.

- Sometimes it is even worthwhile **trying a different compiler** just to check whether there is more performance potential. One should be aware that the compiler has the extremely complex job of mapping source code written in a high-level language to machine code, thereby utilizing the processor's internal resources as well as possible.

- However, there is no guarantee that this is actually the case and the programmer should at least be aware of the basic strategies for automatic optimization and potential stumbling blocks that prevent the latter from being applied. **It must be understood that compilers can be surprisingly smart and stupid at the same time**.

- A common statement in discussions about compiler capabilities is **"The compiler should be able to figure that out." This is often a false assumption.**

Break with mountains

Shared memory parallelism: OpenMP

(**Open M**ulti **P**rocessing)

- **OpenMP** website is a good source of information:

→ openmp.org

→ You can find there:

- tutorials and examples for all levels.
- the standard.
- quick references guide.

Reminder: Free Lunch is over

- For a long time high-performance computing had a "free-lunch".

- The density of transistors in chips increased, decreasing the size of integrated circuits.
  - → same number of transistors with less power.
  - → More transistors to add functionality.

- The clock speeds steadily rose, increasing the number of operations per second (from MHz to GHz).

- The free lunch has been over for a few years now.
  - → We are reaching the limitations of transistor density.
  - → Increasing clock frequency requires too much power.

We used to focus on floating point operations per second! now we also think about floating point operations per Watt!

Multicore

- The number of transistors is still increasing:
  - → Sandy Bridge 28 nm, Haswell 22 nm, Broadwell 14 nm.
  - → Projected hard limit of **5-7 nm**.

- This has lead to three trends:
  - → more cores the
    Sandy Bridge processor on Cray XC30 has 8 cores
    Xeon Phi KNL will has 72 cores (e.g. Nersc's Cori)

  - → reducing clock speed

  - → simplify/specialized cores

An extreme example of this are GPUs, which have in the order of 100/1000s of cores specialized for tasks common in graphics.

Shared memory systems

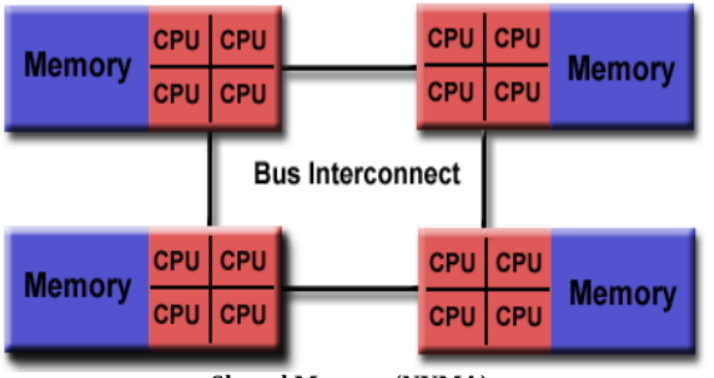
- **Process can access same GLOBAL memory**

- **Uniform Memory Access (UMA) model**
  - Access time to memory is uniform.
  - Local cache, all other peripherals are shared.

Shared Memory (UMA)

- **Non-Uniform Memory Access (NUMA) model**
  - Memory is physically distributed among processors.
  - Global virtual address spaces accessible from all processors.
  - Access time to local and remote data is different.

→ **OpenMP**, but other solutions available (e.g. Intel's TBB).



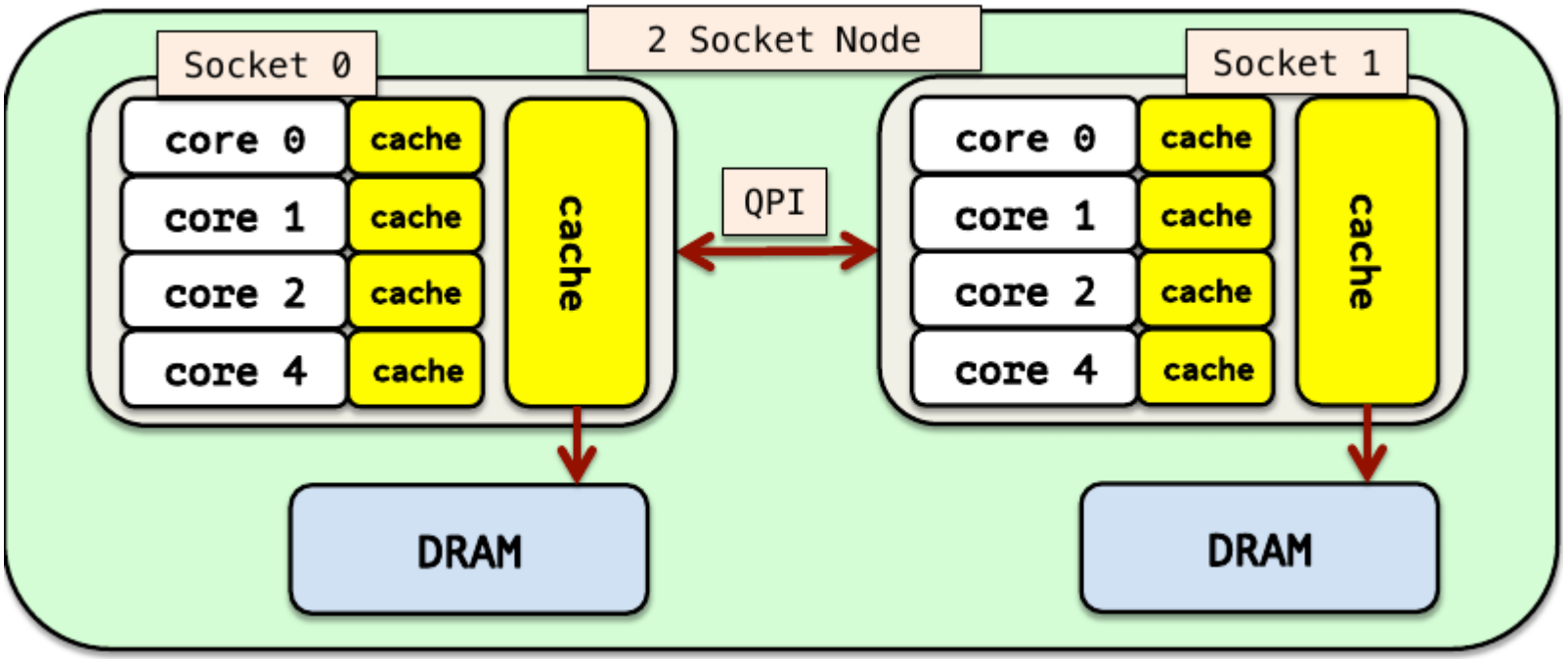
Shared Memory (NUMA)

N-

Multicore Nodes

A CPU-based node can have multiple sockets, each having multiple cores.

Cores on a socket share cache and uniform DRAM access.

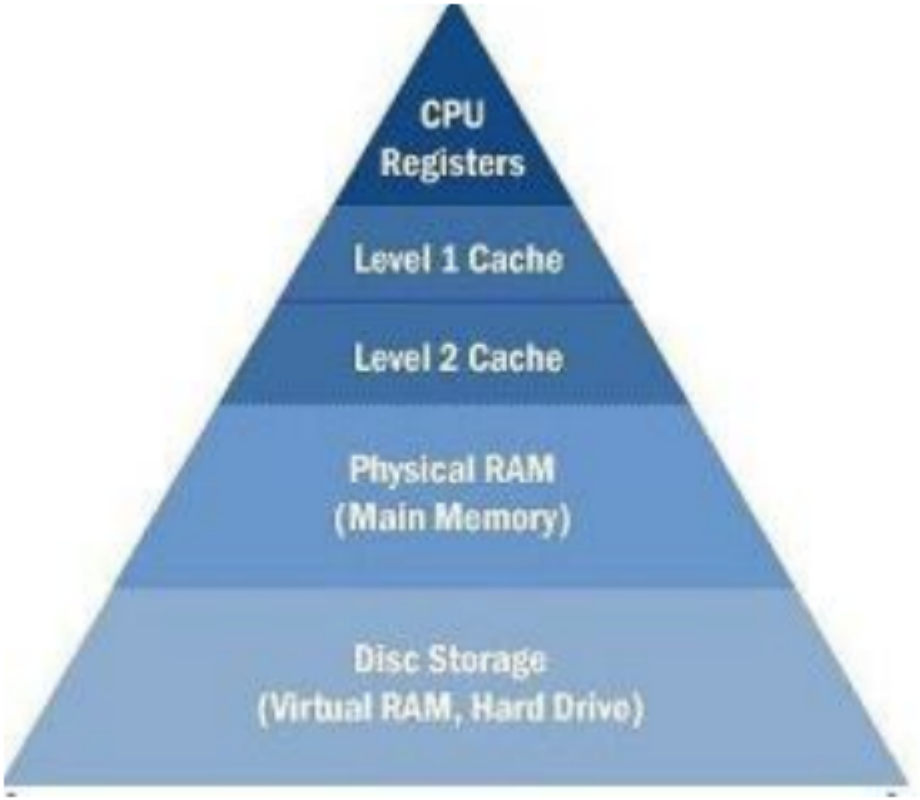


Memory Hierarchy

L1 caches
→ Instruction cache
→ Data cache

L2 cache
→ Joint instruction/data cache
→ Dedicated to individual core processor

L3 cache
→ Not all systems
→ Shared among multiple cores



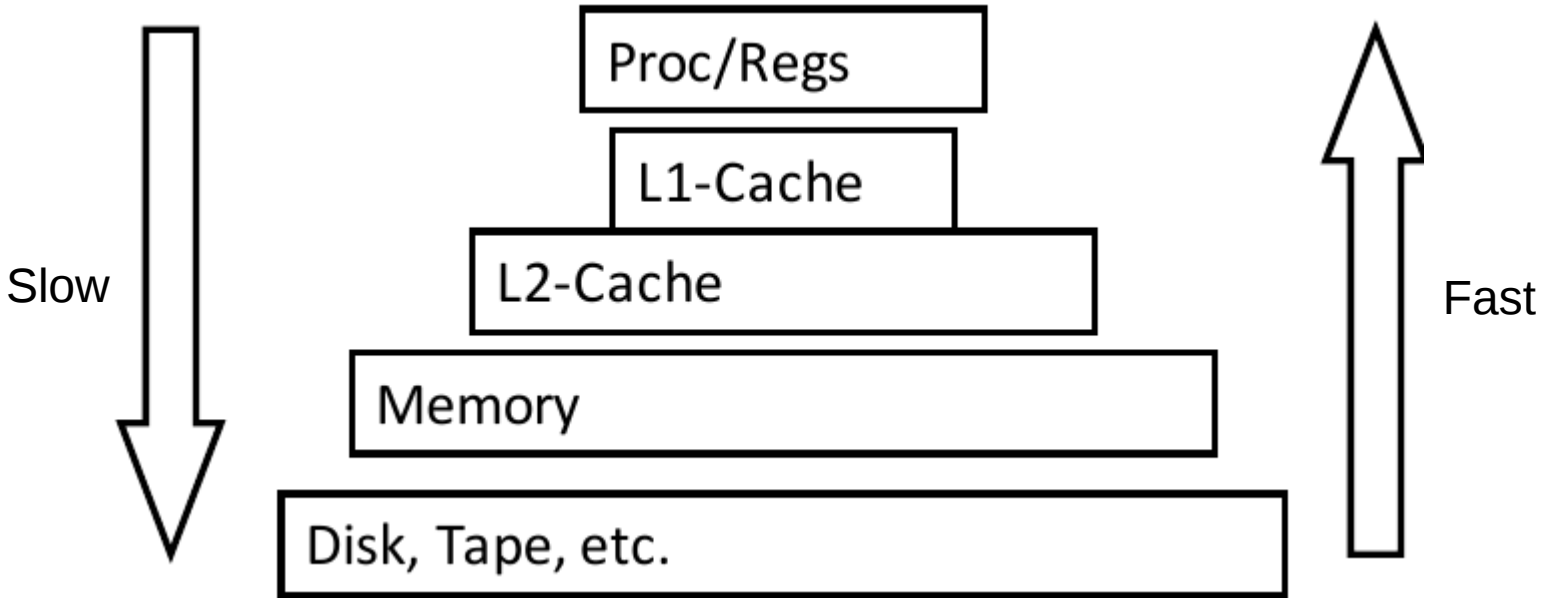
Memory interface
→ Address translation and management (sometimes)

I/O interface

Memory speed & Cache

- Small, fast storage used to improve average access time to slow memory.

- Exploits **spatial** and **temporal** locality.



Data access speed

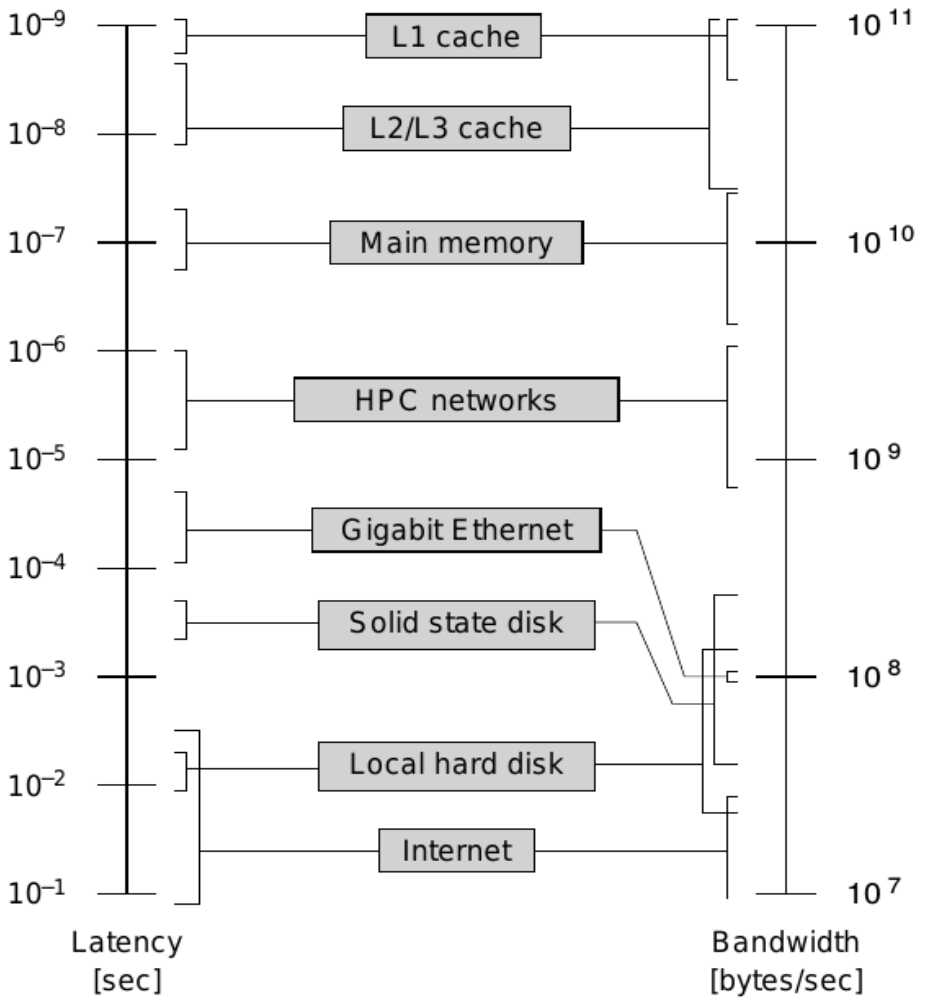


Figure 3.1: Typical latency and bandwidth numbers for data transfer to and from different devices in computer systems. Registers have been omitted because their "bandwidth" usually matches the computational capabilities of the compute core, and their latency is part of the pipelined execution.

Fig. from Hager & Wellein

Spatial and temporal locality

Temporal Locality:
→ a property that if a program accesses a memory location, there is a much higher than random probability that the same location would be accessed again.

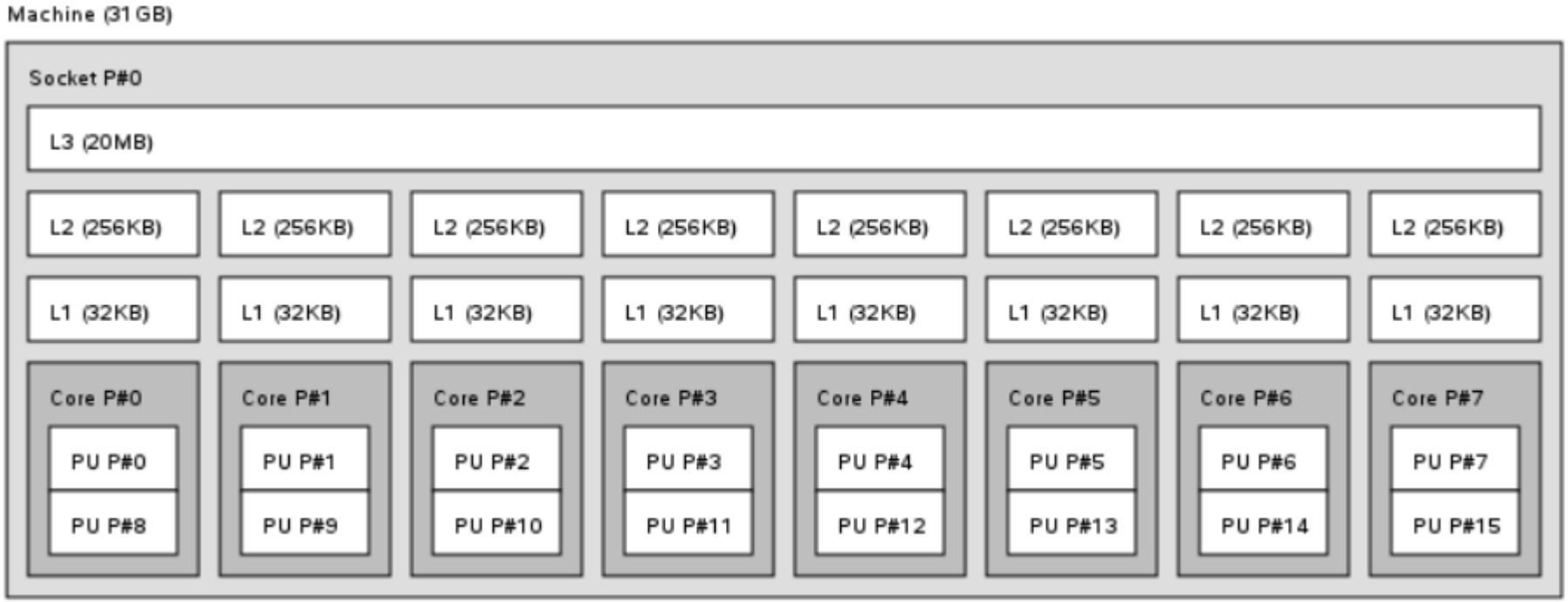
Spatial Locality:
→ a property that if a program accesses a memory location, there is a much higher than random probability that the nearby locations would be accessed soon.

Spatial locality is usually easier to achieve than temporal locality.

A couple of key factors affect the relationship between locality and scheduling :

→ Size of dataset being processed by each processor
→ How much reuse is present in the code processing a chunk of iterations.

Topology – CRAY XC30 nodes



Topology II – CRAY XC40 nodes

Shared Memory – Pro's & Con's

Pro's

- Global address space provides a **userfriendly programming perspective** to memory.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

Con's

- **Lack of scalability** between memory and CPUs. Adding more CPUs geometrically increases traffic on the shared memory-CPU path...
- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

What is OpenMP?

Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors (e.g. GNU, Intel, Cray, PGI,...).

OpenMP provides a portable, scalable model for developers of shared memory parallel applications.

Supports C/C++ and Fortran on a wide variety of architectures.

→ API may be used to explicitly direct multi-threaded, shared memory parallelism.

The API comprised of three main components:

1) Compiler Directives
2) Runtime Library Routines
3) Environment variables

Goals of OpenMP

- Standardization

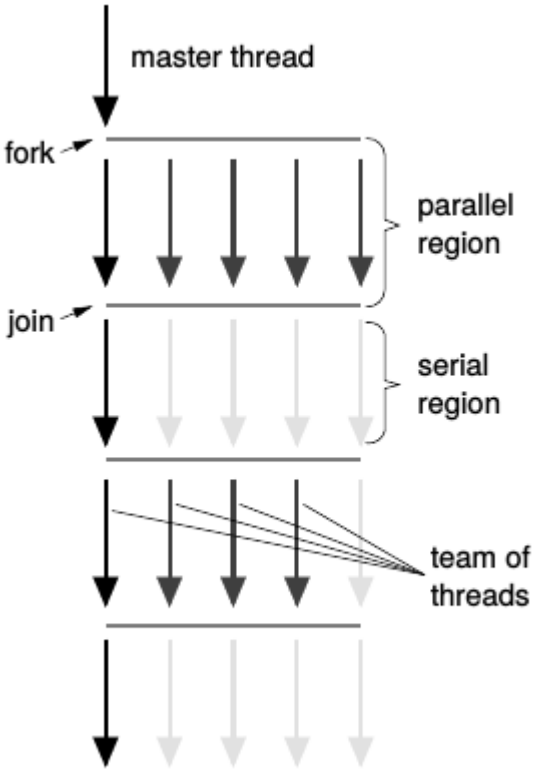
- Ease of use:
  - → Concise and simple set of directives.
  - → **Possible to get good speed-up with a handful of directives** (now at release 4.0).
  - → You can incrementally add it to the code without major changes.

- Should I use OpenMP?
  - → **Path least resistance to parallelize your code...**
  - → For many-core architectures like Xeon Phi, lightweight threading is required since MPI does not scale there...

Fork and Join Model

- OpenMP uses **fork** and **join** model for threading.

- The application starts with a master thread:
  - **FORK**: a team of parallel worker threads is started at the beginning of each parallel block.
  - The block is executed in parallel by each thread.
  - **JOIN**: the worker threads are synchronized at the end of the parallel block and join with the master thread.



- Threads are numbered **0:*N*-1** (*N* is the total number of threads).

- The **master** thread is always numbered 0.

OpenMP compiler directives

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise usually by **specifying the appropriate compiler flag**.

OpenMP compiler directives are used for various purposes:

- Spawning a parallel region.
- Dividing blocks of code among threads.
- Distributing loop iterations between threads.
- Serializing sections of code.
- Synchronization of work among threads.

- Compiler directives have the following syntax:

  sentinel directive-name [clause, ...]

  For example:

Fortran	**!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)**
C/C++	**#pragma omp parallel default(shared) private(beta,pi)**

Compiling OpenMP

- Most compilers require a **flag** to enable OpenMP compilation
  → without any flag, the **#pragma** or **!$** directives are ignored by the compiler and a serial application is created.

- Compilers that don't understand OpenMP will simply ignore the directives (no portability problems).

```
Cray    : on by default for -O1 and greater, disable with -h noomp
Intel   : off by default, enable with -openmp
GNU     : off by default, enable with -fopenmp
PGI     : off by default, enable with -mp
```

Runtime library

- OpenMP API includes a growing number of **runtime library routines**.

These are used for a variety of purposes:

- Setting and querying the number of threads.
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size.
- Setting and querying the dynamic threads feature.
- Querying if in a parallel region, and at what level.
- Setting and querying nested parallelism.
- Setting, initializing and terminating locks and nested locks.
- Querying wall clock time and resolution.

- For C/C++, all of the runtime library routines are actual subroutines.
- For Fortran, some are actually functions, and some are subroutines.

Runtime library II

- OpenMP has runtime library routines for controlling your application, including

Function:	omp_get_num_threads()
C/ C++	int omp_get_num_threads(void);
Fortran	integer function omp_get_num_threads()
Description: Returns the total number of threads currently in the group executing the parallel block from where it is called.	

Function:	omp_get_thread_num()
C/ C++	int omp_get_thread_num(void);
Fortran	integer function omp_get_thread_num()
Description: For the master thread, this function returns zero. For the child nodes the call returns an integer between 1 and omp_get_num_threads()-1 inclusive.	

- There are many others, however these are probably the most commonly used.

Runtime library III

The runtime library requires that the OpenMP **header/module** is included:

```c
#include <omp.h>
...
int threads = omp_get_max_threads();
int outside = omp_get_num_threads();
int inside;
#pragma omp parallel
{
  inside = omp_get_num_threads();
}
printf("%d in, %d out, %d max \n",
       inside, outside, threads);
```

```
use omp_lib
...
integer :: threads, inside, outside
threads = omp_get_max_threads()
outside = omp_get_num_threads()
!$omp parallel
inside = omp_get_num_threads()
!$omp end parallel
print *, inside, ' in ', outside, ' out ',
         threads, ' max'
```

```
> OMP_NUM_THREADS=8 ./a.out
8 in, 1 out, 8 max
```

Running OpenMP applications

- The default number of threads is set with an environment variable **OMP_NUM_THREADS**

csh/tcsh	`setenv OMP_NUM_THREADS 8`
sh/bash	`export OMP_NUM_THREADS=8`

- Compiling and running:

```
g++ 1a.hello_world.cpp -fopenmp -o 1a.hello_world.exec
export OMP_NUM_THREADS=8
./1a.hello_world.exec
```

compile openmp code

set 8 threads

run

“Hello world” in OpenMP (CPP)

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

int main ()
{
int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */

}
```

“Hello world” in OpenMP (CPP)

```
#include <omp.h>

main ()
{
   int nthreads, tid;
   #pragma omp parallel private(nthreads, tid)
   {
      tid = omp_get_thread_num();
      printf("Hello World from thread = %d\n", tid);
      if (tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
   }
}
```

Non shared copies of data for each thread

OpenMP directive to indicate START segment to be parallelized

Code segment that will be executed in parallel

OpenMP directive to indicate END segment to be parallelized

Data scoping

- Any variables that existed before a parallel region still exist inside, and are **by default shared between all threads**.

- True work sharing, however, makes sense only if **each thread can have its own, private variables**.

- OpenMP supports this concept by defining a separate stack for every thread.

1. A variable that exists before entry to a parallel construct can be privatized, i.e., made available as a **private instance for every thread**, by a **PRIVATE** clause to the OMP PARALLEL directive. The private variable's scope extends until the end of the parallel construct.

2. The index variable of a work-sharing loop is automatically made private.

3. Local variables in a subroutine called from a parallel region are private to each calling thread. This pertains also to copies of actual arguments generated by the call-by-value semantics, and to variables declared inside structured blocks in C/C++. However, local variables carrying the SAVE attribute in Fortran (or the static storage class in C/C++) will be shared. **Shared variables that are not modified in the parallel region do not have to be made private**.

Scope of Variables

OpenMP provides clauses that describe how variables should be shared between threads

– **shared**: all variables access the same copy of a variable.
  → this is the default behaviour.
  → WARNING: take care when writing to shared variables.

– **private**: each thread gets its own copy ofthe variable
  → private copy is uninitialized.
  → use *firstprivate* to initialize variable with value from master.

Example 1:”hello world from thread”

1. go to OSM2018/day2/code_day2/openmp:

> cd OSM_Lab/HPC_day2/code_day2/openmp

2. Have a look at the code

> vi 1.hello_world.f90 / vi 1a.hello_world.cpp

3. compile by typing:

> make

4. Experiment with different numbers of threads

> export OMP_NUM_THREADS=1 (play around with #threads)
> ./hello_world.exec (FORTRAN)
> ./1a.hello_world.exec (CPP)

Example 1:”hello world from thread”

5. experiment with slurm
   (the settings)

> cd day_2/code_day2/openmp
> vi submit_openmp_midway.sh

```
#!/bin/bash
# a sample job submission script to submit an OpenMP job to the sandyb
# partition on Midway1 please change the --partition option if you want to use
# another partition on Midway1

# set the job name to hello-openmp
#SBATCH --job-name=hello-openmp

# send output to hello-openmp.out
#SBATCH --output=hello-openmp.out

# this job requests node
#SBATCH --ntasks=1


# and request 8 cpus per task for OpenMP threads
#SBATCH --cpus-per-task=8

# this job will run in the sandyb partition on Midway1
#SBATCH --partition=sandyb


# set OMP_NUM_THREADS to the number of --cpus-per-task we asked for
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# Run the process with mpirun. Notice -n is not required. mpirun will
# automatically figure out how many processes to run from the slurm options
### openmp executable
./1.hello_world.exec
```

6. see **1a.hello_world.cpp** line 10: #pragma omp parallel private(nthreads, tid)

   → what happens if we remove **private(tid)** → try out!

Shared memory model

- OpenMP uses a shared memory model.

- All threads can read and write to the same memory locations simultaneously.

- By default variables are shared, so one copy is used by all threads.

- The result of computations where multiple threads attempt to read/write to a variable are undefined.

→ see OSM2018/day2/code_day2/openmp/2a.example_racing_cond.cpp

→ this is a very common parallel programming bug called **race condition**.

Example 2: Racing condition

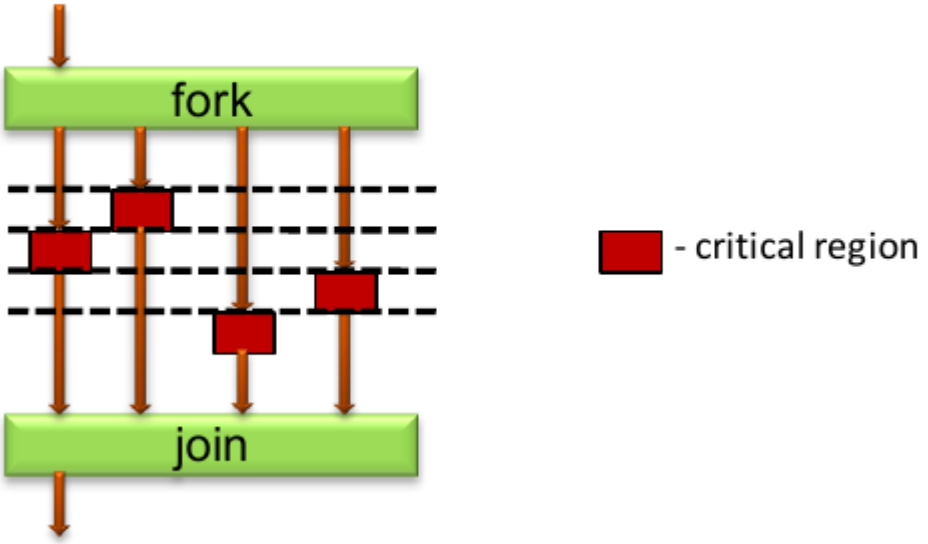
```cpp
#include <iostream>

#include <omp.h>

int main(void){
    int num_threads = omp_get_max_threads();
    std::cout << "sum with " << num_threads << " threads" << std::endl;

    int sum=0;

    #pragma omp parallel
    {
        sum += omp_get_thread_num()+1;
    }

    // use formula for sum of arithmetic sequence: sum(1:n) = (n+1)*n/2
    int expected = (num_threads+1)*num_threads/2;
    std::cout << "sum " << sum
              << (sum==expected ? " which matches the expected value "
                                : " which does not match the expected value ")
              << expected << std::endl;
    return 0;
}
```

Synchronization/critical regions

- Concurrent write access to a shared variable or, in more general terms, a shared resource, must be avoided by all means to circumvent race conditions.

- Critical regions solve this problem by making sure that at most one thread at a time executes some piece of code.

- If a thread is executing code inside a critical region, and another thread wants to enter, the latter must wait (block) until the former has left the region.



Example 2 fixed: Racing conditions

1. cd OSM2018/day2/code_day2/openmp/3a.racing_cond_fix.cpp

2. Have a look at the code

3. compile by typing:

> make

4. Experiment with different numbers of threads

the **CRITICAL** and **END CRITICAL** directives bracket the update to ***sum*** so that the result is always correct.

→ **WARNING: SYNCHRONIZATION (AND SERIAL CODE REGIONS) CAN QUICKLY LIMIT POTENTIAL SPEED-UP FROM PARALLELISM**

Example 3: Race conditions fixed

```
#include <iostream>

#include <omp.h>

int main(void){
    int num_threads = omp_get_max_threads();
    std::cout << "sum with " << num_threads << " threads" << std::endl;

    int sum=0;

    #pragma omp parallel
    {
        #pragma omp critical
        sum += omp_get_thread_num()+1;
    }

    // use formula for sum of arithmetic sequence: sum(1:n) = (n+1)*n/2
    int expected = (num_threads+1)*num_threads/2;
    std::cout << "sum " << sum
              << (sum==expected ? " which matches the expected value "
                                : " which does not match the expected value ")
              << expected << std::endl;
    return 0;
}
```

Another break with a nice mountain