CERN

Behind the scenes perspective: Into the abyss of profiling for performance

Part II

Servesh Muralidharan & David Smith

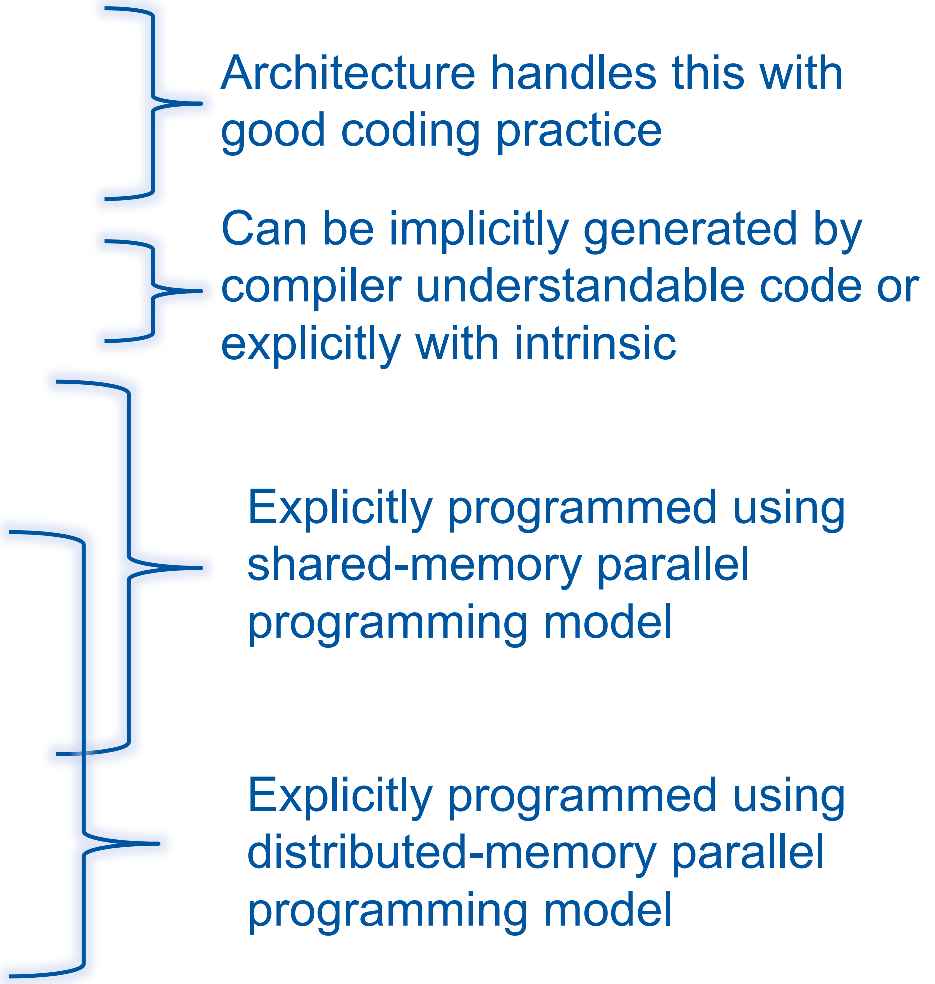
IT-DI WLCG-UP

CERN

29 Aug'18

Sources of Parallelism in Modern Architectures

1. Instruction level parallelism (ILP)
2. Pipelining
3. Vector Operations
4. Hardware Threads
5. Multicore
6. Multi Socket
7. Cluster
8. Grid



Architecture handles this with good coding practice

Can be implicitly generated by compiler understandable code or explicitly with intrinsic

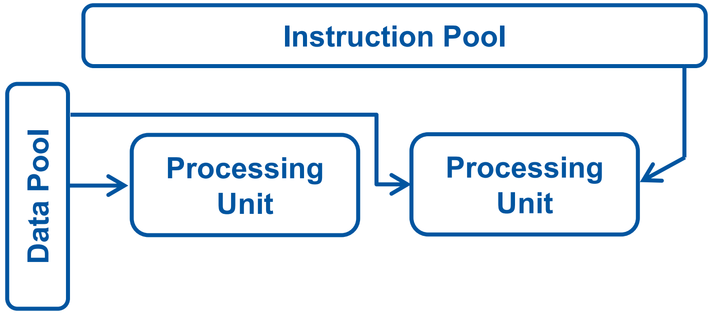
Explicitly programmed using shared-memory parallel programming model

Explicitly programmed using distributed-memory parallel programming model

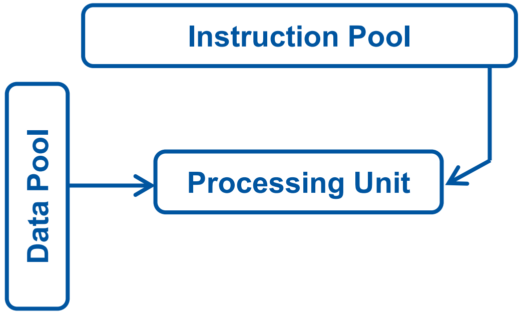
Flynn’s taxonomy: <u>Can be</u> a programmer’s guideline

- Proposed in 1966
- Instruction streams
  - single (SI) or multiple (MI)
- Data streams
  - single (SD) or multiple (MD)
- Types
  - SISD
  - SIMD
  - MISD
  - MIMD

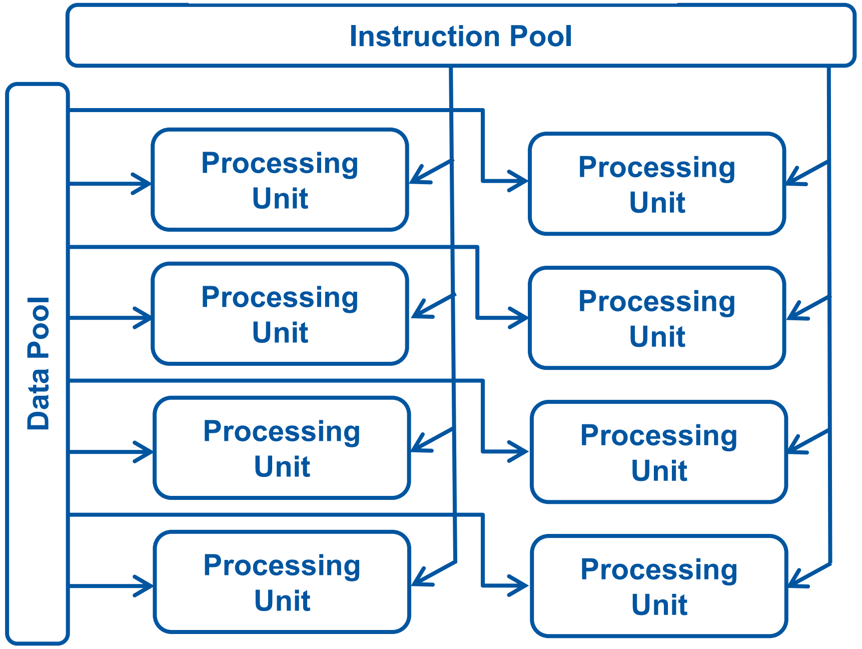
MISD


Instruction Pool
Data Pool
Processing Unit
Processing Unit

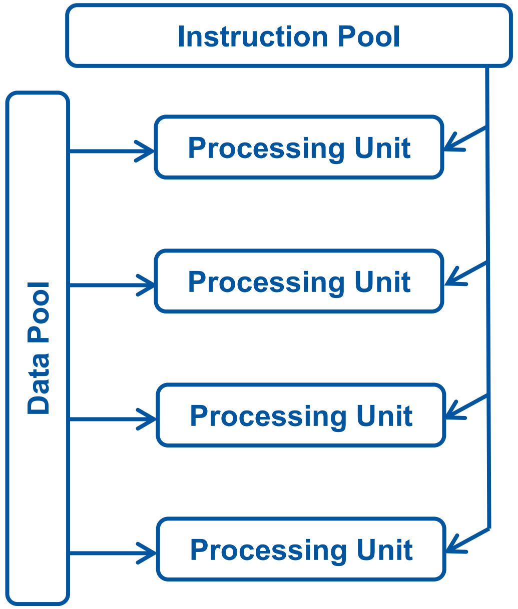
SISD


Instruction Pool
Data Pool
Processing Unit

MIMD


Instruction Pool
Data Pool
Processing Unit
Processing Unit
Processing Unit
Processing Unit
Processing Unit
Processing Unit
Processing Unit
Processing Unit

SIMD


Instruction Pool
Data Pool
Processing Unit
Processing Unit
Processing Unit
Processing Unit

Using SIMD

- Choose between code manageability and portability and speed:
- Use different levels of abstraction
  - Assembly
  - Intrinsic
  - Wrapper functions or classes in C or C++ (using intrinsics)
  - Custom languages like Cilk/Cilk++
  - Autovectorization

SIMD: SSE, AVX & FMA

- The use of SIMD instructions in vectorized code can give good performance gains

(S)SSE 1,2,3	SSE 4.1, 4.2	AVX	AVX2 + FMA	AVX-512
From 1999 Width 128b	From 2007 Width 128b	From 2011 Width 256b	From 2013 Width 256b	From 2016 Width 512b

- 256b -> 8 floats or 4 doubles (possibility of 4 or 8 speedup)

Fused Multiply Add

- These are instructions which can do calculations of the form:

  `A <- A*C + B or`

  `A <- B*C + A`

- Can bring gain because one instructions replaces a multiply and add instruction (reduces throughput cycles and latency)
- Is also a SIMD instruction

FMA and floating point

- FMA is specified to round only once
  - Therefore FMA is a change which can affect the result of a calculation compared to using separate multiply and add operations
- Enable with:
- GCC:
  - Will use FMA if it is compiling for an architecture that has it
    - May be explicitly set with –mfma or –mno-fma

Autovectorization

- Depends on compiler and version
- Advantages
  - Can get great speedups (x2 or more) with little change of the source code
  - Compiler can generate a report to help
  - Source code remains architecture independent
- Disadvantages
  - Can be delicate. A small change of the source can stop the compiler autovectorizing, causing a large change in performance
  - Lots of compiler options that affect the autovectorization
  - Usually you could have bigger gains using intrinsics or assembly
  - Some compiler options which help autovectorization can change the way FP operations are done: you have to be aware when that may be important

Autovectorization

- Two main places where automatic vectorization can be done
  - In loops
    - compiler tries to do several iterations of the loop at once using SIMD
    - May unroll a number of the loops so to fill pipeline and improve ILP
    - May peel loops to allow aligned access to data
  - Combining similar independent instructions into vector instructions
    - Known as SLP vectorizer

Autovectorization difficulties

```
for(i=0;i<*p;++i) {
  A[i] = B[i] * C[i];
  sum += A[i];
}
```

(example from slide by Georg Zitzlsberger, Intel)

Possible to vectorize? Concerns may be:

- Is the loop range invariant during the loop
- Is A[] aliased with the other arrays or with sum, is sum aliased with B[] or C[]
- Is the + operator associative?
- Is the vectorized version expected to be faster?

Using autovectorization

- GCC:
  - Switch on using –ftree-vectorize
    - Off by default
  - Information on autovecotization analysis
    - using –ftree-vectorizer-verbose=X (X=0-7, 7 is most information)
    - Or examine generated code with gdb
  - May be necessary to use –ffast-math
    - Often with reductions (e.g. summations)
    - This will cause the compiler to relax certain some constrains, for example allow it to assume associative properties
- ICC:
  - On by default, modify with –x and -ax
    - Would switch off by using –no-vec
  - ICC defaults to ignoring parentheses to specify floating point associativity and generally can make more aggressive optimizations on floating point calculations
    - Controlled by -fprotect-parens and –fp-model
  - Information on autovectorization analysis
    - -qopt-report=2 –qopt-report-phase=vec
    - -opt-report-help and –opt-report-phase={hpo,ipo}
    - Or examine with gdb…

FP caution

- E.g. summing a number of values: One technique to reduce rounding error over long sums is Kahan summation

```
double sum = 0.0, C = 0.0, Y, T;
size_t i;
/* Kahan summation of values in A[i] */
for (i=0; i<length; i++) {
   Y = a[i] - C;
   T = sum + Y;
   C = (T-sum) - Y;
   sum = T;
}
```

- If the compiler reasons C=0 the correction is lost. Perhaps pairwise summation could be used instead.

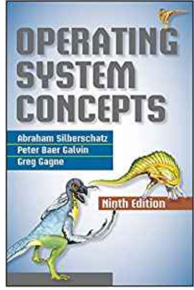
SoA vs AoS

- It is better to load the contents of a vector register from a contiguous piece of memory rather than gather the values
- Can arrange that the data layout fits this access pattern. e.g. use of Structure of Arrays instead of Array of Structures. e.g.

```
struct point {
        double x,y,z;
} location[1000];
```

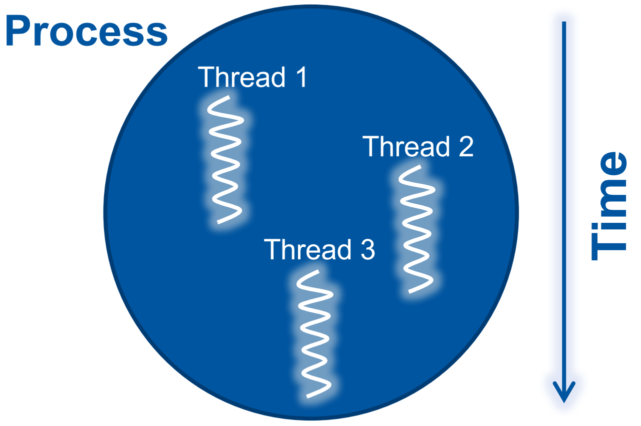
```
struct points {
        double x[1000];
        double y[1000];
        double z[1000];
} locations;
```

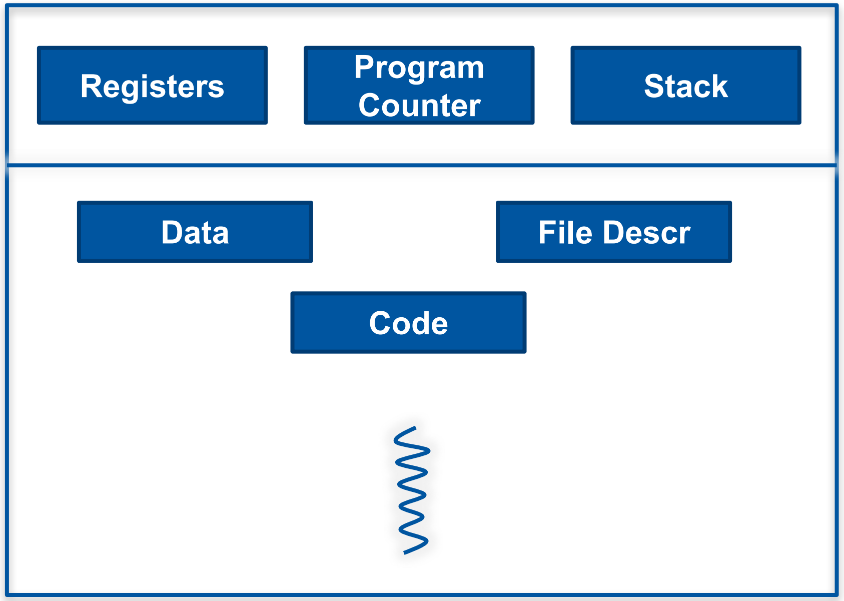
Software processes & threads



- Process (OS process):
  - Encapsulated entity of a program running in its own private address space
  - It consists of a private copy of program code and data along with file descriptors and permissions
  - It has a dedicated heap and stack space from which data is accessed and modified.
- Thread
  - Lightweight execution context that runs under a process
  - They share address space, program code and operating system resource of their parent process
  - Can be created and destroyed with low overhead in comparison to that of a process
  - Consists of a small amount of thread local storage space

Processes & threads




Registers

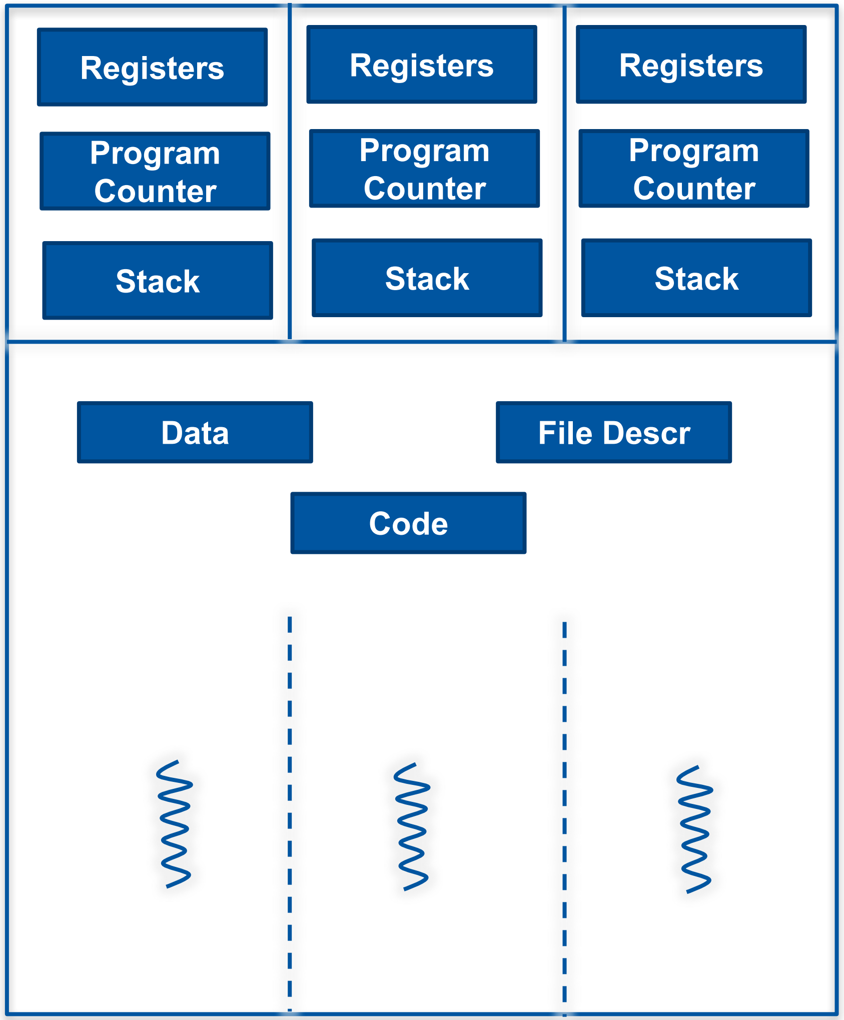
Program Counter

Stack

Data

File Descr

Code


Registers

Program Counter

Stack

Registers

Program Counter

Stack

Registers

Program Counter

Stack

Data

File Descr

Code

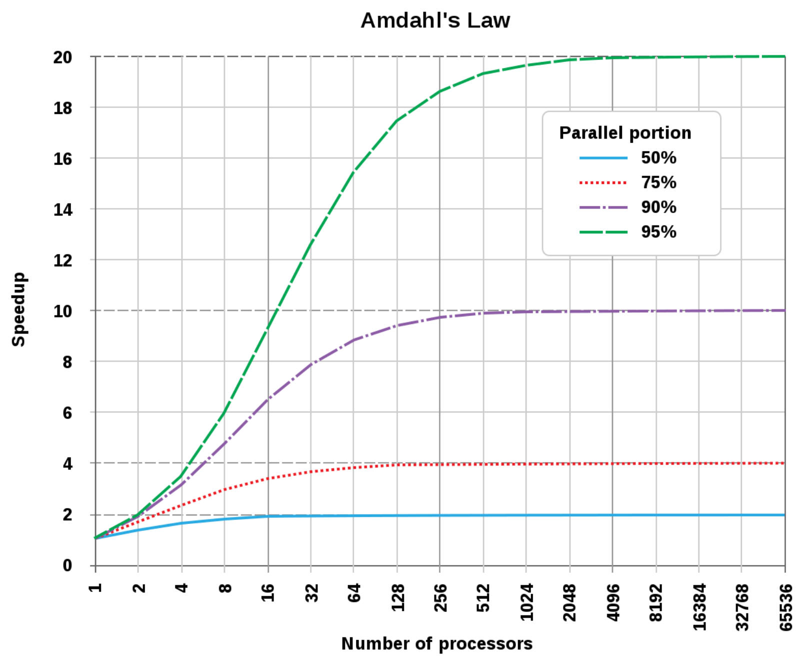
Parallel computing

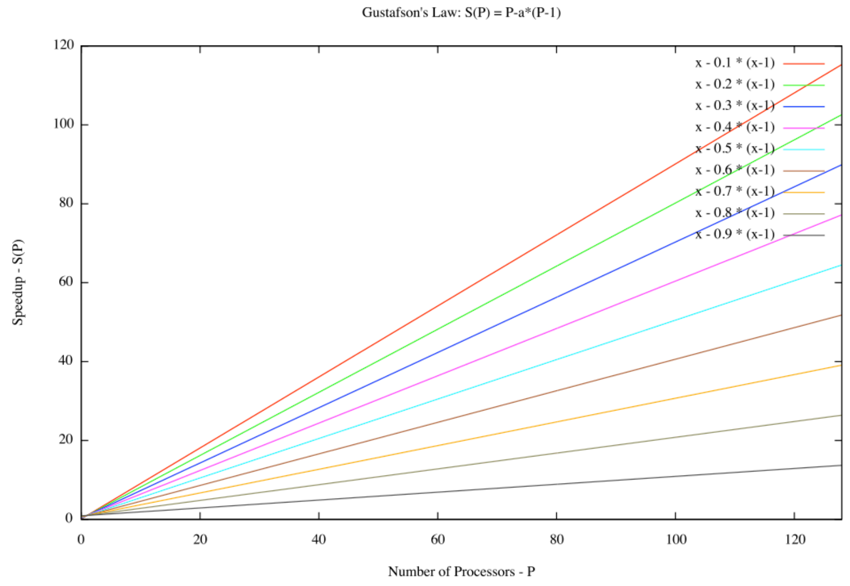
- Performing certain computations simultaneously using multiple resources
- **Amdahl's law & Gustafson's law**
  - Speedup only comes from the parallelizable part of the code
  - i.e. Serial part of the code will impact or limit the achievable performance

Amdahl's law $$S_\text{latency}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

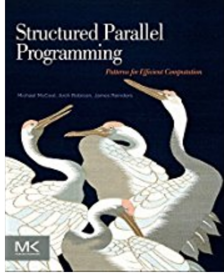
Gustafson's law $$S_\text{latency}(s) = 1 - p + sp,$$

Where,
S_latency is the theoretical overall speedup
s is the speedup in the parallel part
p is the percentage of the execution time of serial part





Parallel Programming



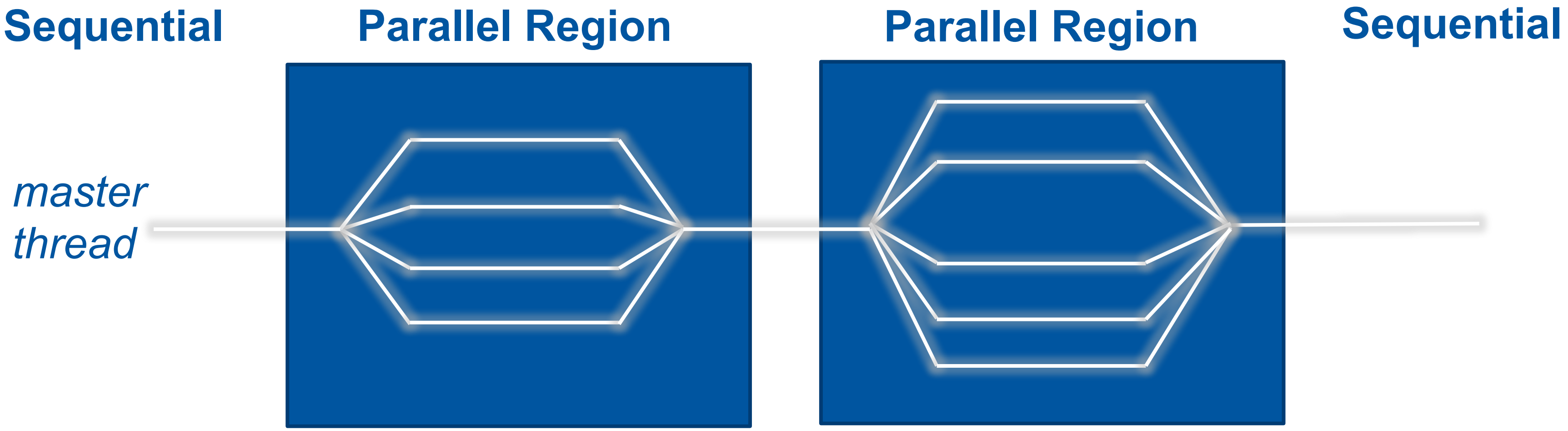
- Is a large topic. Many tools and techniques, a few:
  - pthreads is a standard API for managing threads
    - Fundamental API for threading in Linux
  - Cilk Plus
    - Language support by compiler extensions: appears as C/C++ with extensions
  - TBB (threading building blocks from Intel)
    - C++ large use of templates
    - commercial binary distribution with support or open source
  - C++11 threads
  - CUDA and OpenCL
    - GPU or CPU/GPU unified programming models
  - OpenMP
  - http://concurrency.web.cern.ch/GaudiHive
    - A framework from the HEP community

OpenMP

- A specification:
  - See https://www.openmp.org
  - Are compiler directives, routines and variables that can be used to specify high-level **parallelism** in C, C++ and Fortran
- GCC
  - 4.4 – OpenMP 3.0
  - 4.9 – OpenMP 4.0
  - 6.1 – OpenMP 4.5
- Clang
  - 3.7 – OpenMP 3.1
- Intel
  - 12 – OpenMP 3.1
  - 16 – OpenMP 4.0
  - 17 – OpenMP 4.5

OpenMP

- Code looks similar to a serial version
  - #pragma are used to indicate handling of parallel parts
  - Usually uses a fork-join model



OpenMP

- May need to compile with –fopenmp (check your compiler)
- Most OpenMP features are used through pragmas

```
#pragma omp construct [clause [clause] … ]
```

- You can change the number of threads via environment or an API or specify it in the pragma

```
export OMP_NUM_THREADS=16
```

Parallel regions

- ❑ Threads (up to the number configured) are created, if needed, when the pragma is crossed
- ❑ Threads execute the parallel region, the sequential part continues once all the threads have come to the end of the region
- ❑ Data is shared, but stack variables declared in the parallel region are private

```
#pragma omp parallel
{
     function_called_in_parallel();
}
function_sequential();
```

Parallel for-loops

- ❑ Loop iterations become threads
- ❑ Data is shared between threads (i.e. iterations), except loop index
- ❑ Threads wait at the end of the for loop
- ❑ The pragma is specified directly before the loop

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i < N; i++) {
    function(i);
  }
}
```

- ❑ The two pragmas above are equivalent to

```
#pragma omp parallel for
```

Sharing control

- Consider

```
double x, y;
#pragma omp parallel for
for(i=0;i<N;i++) {
  x = a[i]*4;
  y = b[i] * b[i];
  b[i] = x/y;
}
```

- This will probably not give the intended result: x and y are shared between the threads of the parallel for loop

The private clause

- Used to give each thread a private copy of a variable which was already declared outside
- The variable is uninitialized

```
double x, y;
#pragma omp parallel for private(x,y)
for(i=0;i<N;i++) {
  x = a[i]*4;
  y = b[i] * b[i];
  b[i] = x/y;
}
```

Variations on sharing control

- As well as *private*:
  - *firstprivate*: initializes each private copy to the value from the master thread
  - *lastprivate*: copies the value from the thread, which executed the last iteration of the loop, to the master thread
  - *shared*: is the default, but for documentation or if the default is changed you can uses this clause
  - Plus others, e.g. those which concern threadprivate variables

Reductions

- Reductions will implicitly produce a local copy of the reduction variable in each thread
- Each thread updates its copy
- At the end of the construction the reduction operation merges sub-results into a single value and puts it in the reduction variable
- Operations: `+ * - ^ & | && || min max`

```
#pragma omp for reduction(op:var)
```

Reduction example

```
double dotprod=0;
#pragma omp parallel for reduction(+:dotprod)
for(i=0;i<N;i++) {
  dotprod += a[i] * b[i];
}
```

Exercise 4 – Use SIMD in the matrix multiplication

- Starting with the blocked version of the matrix multiplication see if autovectorization has an effect
  - The Makefile already has the matrixmul-simd target.
  - Autovectorization may work or might need a small change
    - Compare the execution time to a similar multiplication not using autovectorization
    - See if you have SIMD instructions (you may use Intel SDE, see next slide)
    - Check the compiler report if it didn’t work

Intel SDE

- Is the software development emulator
  - In this case we can use it to count and classify different types of instructions

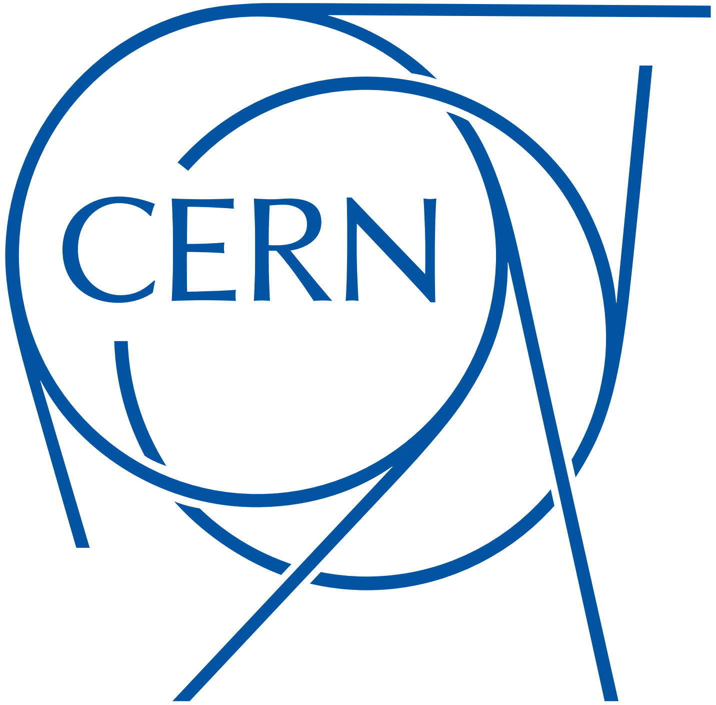
```
export PATH=/home/gss2018/exercises/sde-external-8.16.0-2018-01-30-lin:$PATH
sde -iform 1 -omix test.out -top_blocks 5000 -- ./my_executable
```

- Look in test.out for lines ending in _1, _2 or _4 representing scalar or packed operations, e.g.

```
cat test.out | egrep '^\*.*_[124]'
```

Exercise 5 - OpenMP

- ❑ Use OpenMP to make the matrix multiplication from the previous exercise use multiple threads
  - Set OMP_NUM_THREADS=6, 12, 24, 36 and then 48
  - Run a multiplication and use top to look at the running process. Note that %CPU should be >100%.
  - Compare the runtime each time and
  - Use perf to measure the instructions and cycles; vary the number of threads and note how each changes


CERN