

High Performance Computing

ADVANCED SCIENTIFIC COMPUTING

Prof. Dr. – Ing. Morris Riedel

Adjunct Associated Professor

School of Engineering and Natural Sciences, University of Iceland, Reykjavik, Iceland

Research Group Leader, Juelich Supercomputing Centre, Forschungszentrum Juelich, Germany

LECTURE 6

[in @Morris Riedel](#)

[@MorrisRiedel](#)

[@MorrisRiedel](#)

Parallel Programming with OpenMP

October 10, 2019

Room V02-258



UNIVERSITY OF ICELAND
SCHOOL OF ENGINEERING AND NATURAL SCIENCES
FACULTY OF INDUSTRIAL ENGINEERING,
MECHANICAL ENGINEERING AND COMPUTER SCIENCE



JÜLICH
Forschungszentrum

JÜLICH
SUPERCOMPUTING
CENTRE



HELMHOLTZ
RESEARCH FOR GRAND CHALLENGES



HELMHOLTZ
ARTIFICIAL INTELLIGENCE
COOPERATION UNIT

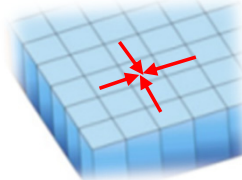
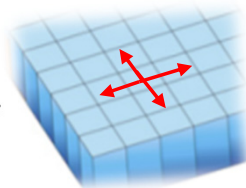
Review of Practical Lecture 5.1 – MPI Communicators & Data Structures

- Example: Nearest Neighbour & Cartesian
- High-Level I/O Hierarchical Data Format (HDF)

```
...
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods,
                reorder, &cartcomm);
MPI_Comm_rank(cartcomm, &rank);
...
MPI_Cart_coords(cartcomm, rank, 2, coords);
...
MPI_Cart_shift(cartcomm, 0, 1,
               &nbrs[UP], &nbrs[DOWN] );
MPI_Cart_shift(cartcomm, 1, 1,
               &nbrs[LEFT], &nbrs[RIGHT] );

printf("rank= %d coords= %d %d" having
       neighbours(u,d,l,r)=%d %d %d %d \n",
       rank, coords[0], coords[1],
       nbrs[UP], nbrs[DOWN], nbrs[LEFT], nbrs[RIGHT]);

// do some work with MPI communication operations...
...
```



```
[morris@jotunn hpdbscan]$ pwd
/home/morris/2019-HPC-Course/hpdbscan
[morris@jotunn hpdbscan]$ ls -al
total 1342196
drwxrwxr-x  2 morris morris    4096 okt  2 13:16 .
drwxrwxr-x 12 morris morris    4096 okt  2 20:21 ..
-rwxr-xr-x  1 morris morris 1302382632 okt  2 13:16 bremen.h5
-rwxr-xr-x  1 morris morris  72002416 okt  2 20:40 bremenSmall.h5
-rw-rw-r--  1 morris morris      0 okt  2 13:13 HPDBSCAN-199934.err
-rw-rw-r--  1 morris morris    490 okt  2 13:14 HPDBSCAN-199934.out
-rw-rw-r--  1 morris morris      0 okt  2 13:14 HPDBSCAN-199935.err
-rw-rw-r--  1 morris morris    492 okt  2 13:17 HPDBSCAN-199935.out
-rwxr-xr-x  1 morris morris    535 okt  2 13:14 submit-clustering-bremen.sh
```

```
#!/bin/bash
#SBATCH --job-name=HPDBSCAN
#SBATCH -o HPDBSCAN-%j.out
#SBATCH -e HPDBSCAN-%j.err
#SBATCH -n 4

# load modules
module load gnu/5.3.0
module load hdf5/1.8.17
module load openmpi/1.10.2
module load HPDBSCAN/mpi

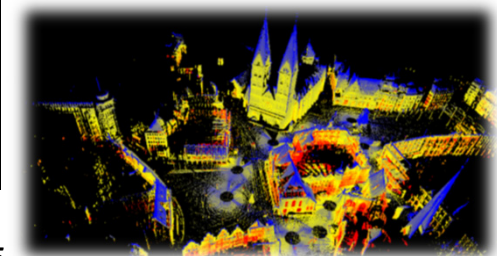
# executable
HPDBSCAN=dbscan

# your own copy of bremen small
BREMEENSMALLDATA=/home/morris/2019-HPC-Course/hpdbscan/bremenSmall.h5

# your own copy of bremen big
BREMEENBIGDATA=/home/morris/2019-HPC-Course/hpdbscan/bremen.h5

mpirun $HPDBSCAN -m 300 -e 500 $BREMEENSMALLDATA
```

```
[morris@jotunn hpdbscan]$ h5dump -n bremenSmall.h5
HDF5 "bremenSmall.h5" {
FILE CONTENTS {
  group /
  dataset /COLORS
  dataset /Clusters
  dataset /DBSCAN
}
```



[1] M. Goetz and M. Riedel et al,
Proceedings IEEE Supercomputing Conference, 2015

Outline of the Course

1. High Performance Computing
2. Parallel Programming with MPI
3. Parallelization Fundamentals
4. Advanced MPI Techniques
5. Parallel Algorithms & Data Structures
6. Parallel Programming with OpenMP
7. Graphical Processing Units (GPUs)
8. Parallel & Scalable Machine & Deep Learning
9. Debugging & Profiling & Performance Toolsets
10. Hybrid Programming & Patterns

11. Scientific Visualization & Scalable Infrastructures
12. Terrestrial Systems & Climate
13. Systems Biology & Bioinformatics
14. Molecular Systems & Libraries
15. Computational Fluid Dynamics & Finite Elements
16. Epilogue

+ additional practical lectures & Webinars for our hands-on assignments in context

- Practical Topics
- Theoretical / Conceptual Topics

Outline

■ Shared-Memory Programming Concepts

- OpenMP with Parallel & Serial Regions
- Fork/Join & Master and Worker Threads
- OpenMP Standard & Portability
- Hybrid Programming Motivation & PyCOMPSs/COMPSs
- OmpSs & OpenMP Data-Flow & Task-Based Evolutions

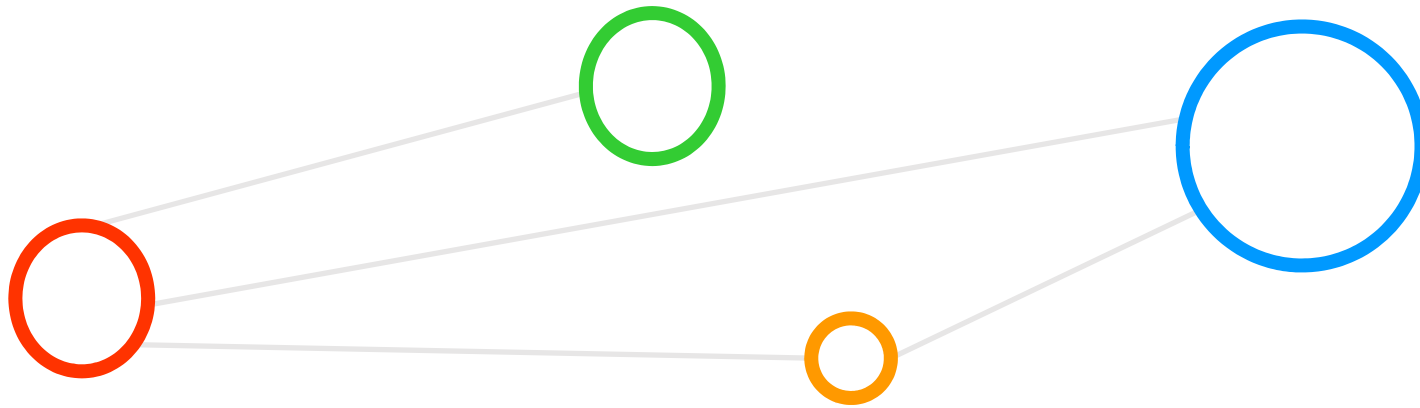
■ OpenMP Parallel Programming Basics

- Basic building blocks
- Local/shared variables & Loops
- Synchronization & Critical Regions
- Selected Comparisons with MPI & Evolutions
- HPDBSCAN Clustering OpenMP & Jacobi Application Example

- Promises from previous lecture(s):
- *Lecture 1:* Lecture 6 will give in-depth details on the shared-memory programming model with OpenMP and using its compiler directives
- *Lecture 3:* Lecture 6 will offer more elaborate shared memory parallel programming examples in context of different HPC application domains



Shared-Memory Programming Concepts

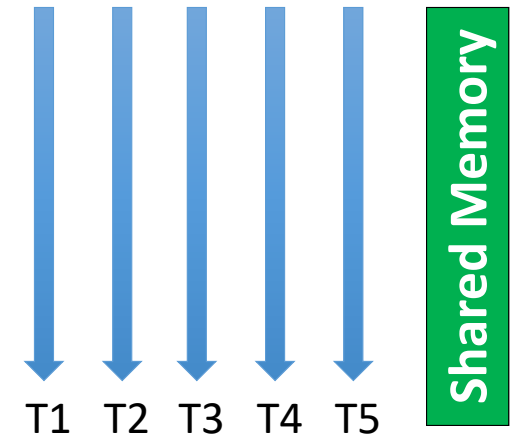


Shared-Memory Computers – Revisited (cf. Lecture 1)

- A shared-memory parallel computer is a system in which a number of CPUs work on a common, shared physical address space

[2] Introduction to High Performance Computing for Scientists and Engineers

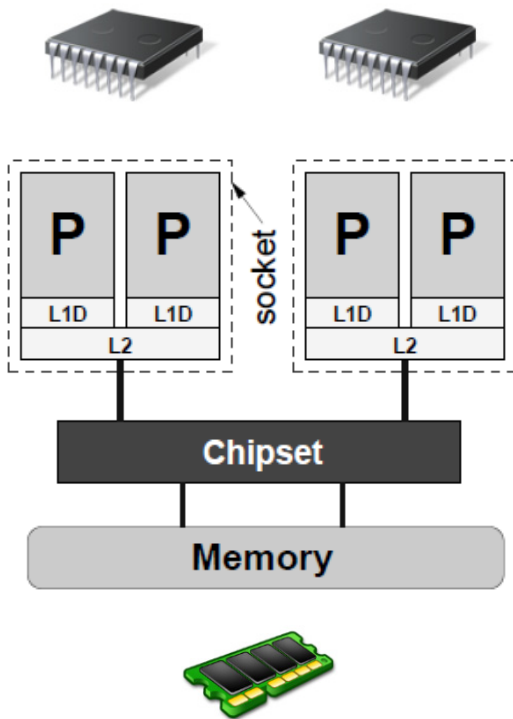
- Two varieties of shared-memory systems:
 1. Unified Memory Access (UMA)
 2. Cache-coherent Nonuniform Memory Access (ccNUMA)
- The Problem of ‘Cache Coherence’ (in UMA/ccNUMA)
 - Different CPUs use Cache to ‘modify same cache values’
 - Consistency between cached data & data in memory must be guaranteed
 - ‘Cache coherence protocols’ ensure a consistent view of memory



Shared-Memory with UMA – Revisited (cf. Lecture 1)

- UMA systems use 'flat memory model': Latencies and bandwidth are the same for all processors and all memory locations.
- Also called Symmetric Multiprocessing (SMP)

[2] Introduction to High Performance Computing for Scientists and Engineers



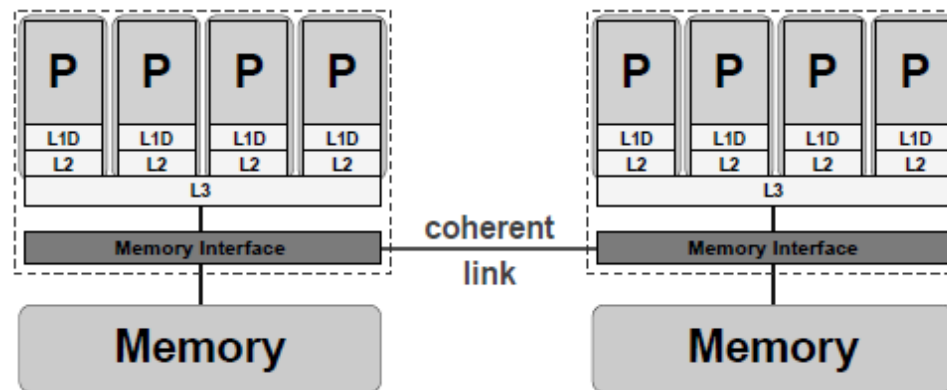
Selected Features

- **Socket** is a physical package (with multiple cores), typically a replacable component
- Two dual core chips (2 core/socket)
- P = Processor core
- L1D = Level 1 Cache – Data (fastest)
- L2 = Level 2 Cache (fast)
- Memory = main memory (slow)
- **Chipset = enforces cache coherence and mediates connections to memory**

Shared-Memory with ccNUMA – Revisited (cf. Lecture 1)

- ccNUMA systems share logically memory that is physically distributed (similar like distributed-memory systems)
- Network logic makes the aggregated memory appear as one single address space

[2] Introduction to High Performance Computing for Scientists and Engineers



Selected Features

- Eight cores (4 cores/socket); L3 = Level 3 Cache
- Memory interface = establishes a coherent link to enable one 'logical' single address space of 'physically distributed memory'

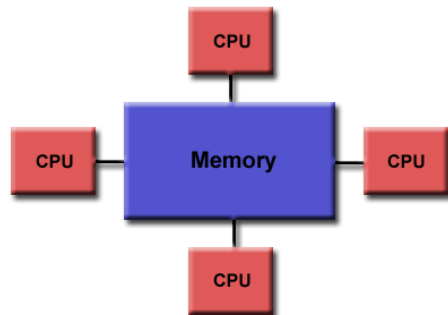
Programming with Shared Memory using OpenMP – Revisited (cf. Lecture 1)

- Shared-memory programming enables immediate access to all data from all processors without explicit communication
- OpenMP is dominant shared-memory programming standard today
- OpenMP is a set of compiler directives to ‘mark parallel regions’

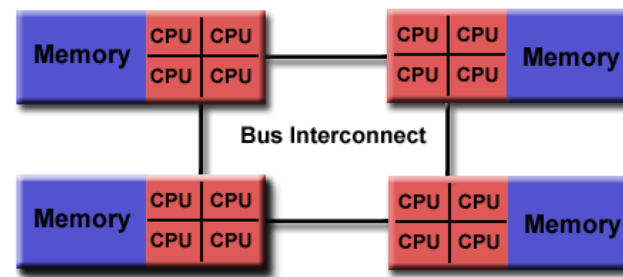
[3] OpenMP API Specification

■ Features

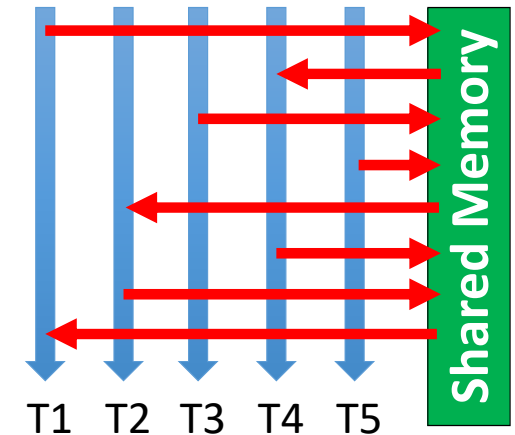
- Bindings are defined for C, C++, and Fortran languages
- Threads TX are ‘**lightweight processes**’ that mutually access data



(uniform memory access)



(non-uniform memory access)



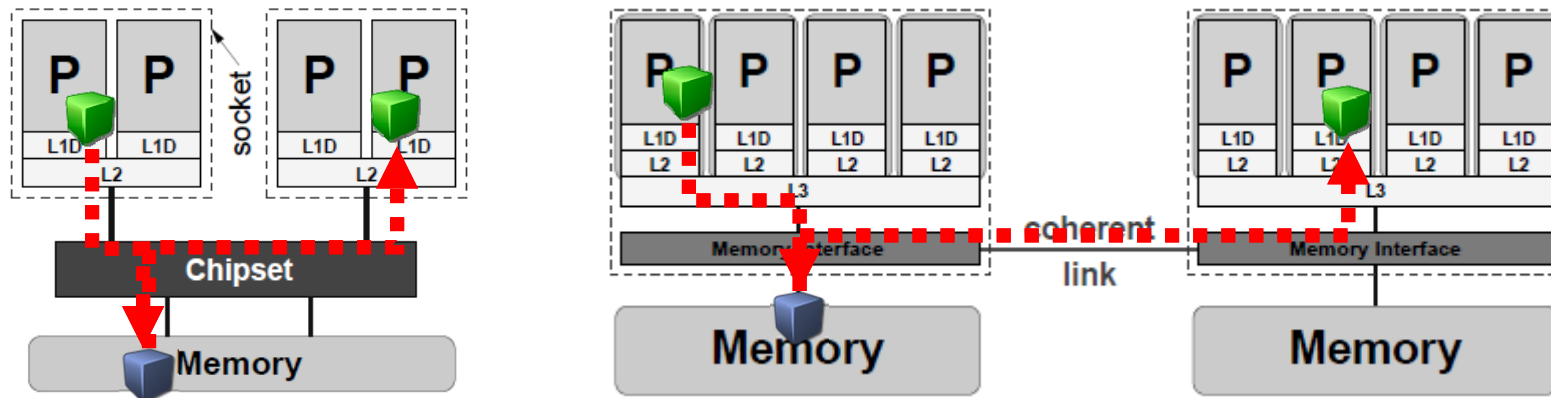
[7] LLNL OpenMP Tutorial

What means a 'Shared Address Space'?

- Shared-memory programming enables immediate access to all data from all processors without explicit communication
- OpenMP is dominant shared-memory programming standard today
- OpenMP is a set of compiler directives to 'mark parallel regions'

[3] OpenMP API Specification

(programming model: work on shared address space – 'local access to memory')



What is OpenMP?

- OpenMP is a library for specifying ‘parallel regions in serial code’
 - Defined by major computer hardware/software vendors → portability!
 - Enable scalability with parallelization constructs w/o fixed thread numbers
 - Offers a suitable data environment for easier parallel processing of data
 - Uses specific environment variables for clever decoupling of code/problem
 - Included in standard C compiler distributions (e.g. gcc)
- Threads are the central entity in OpenMP
 - Threads enable ‘work-sharing’ and share address space (where data resides)
 - Threads can be synchronized if needed
 - Lightweight process that share common address space with other threads
 - Initiating (aka ‘spawning’) n threads is less costly than n processes (e.g. variable space)

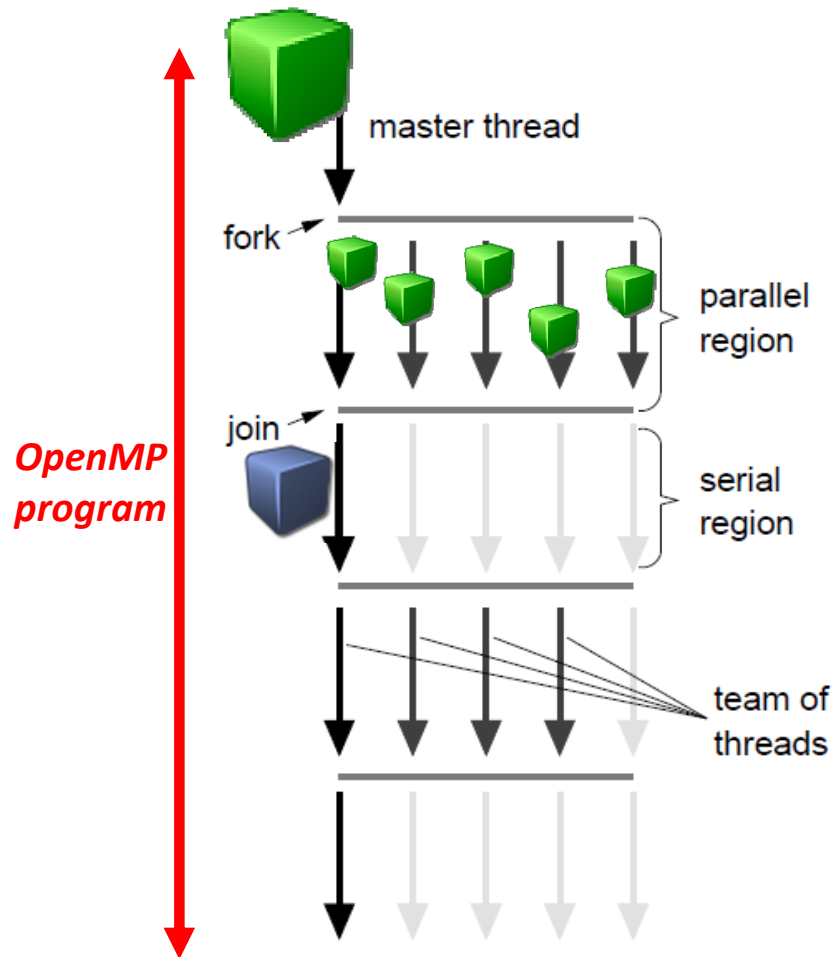


[3] OpenMP API Specification



- Recall ‘computing nodes’ are independent computing processors (that may also have N cores each) and that are all part of one big parallel computer
- Threads are lightweight processes that work with data in memory

Important Terminology

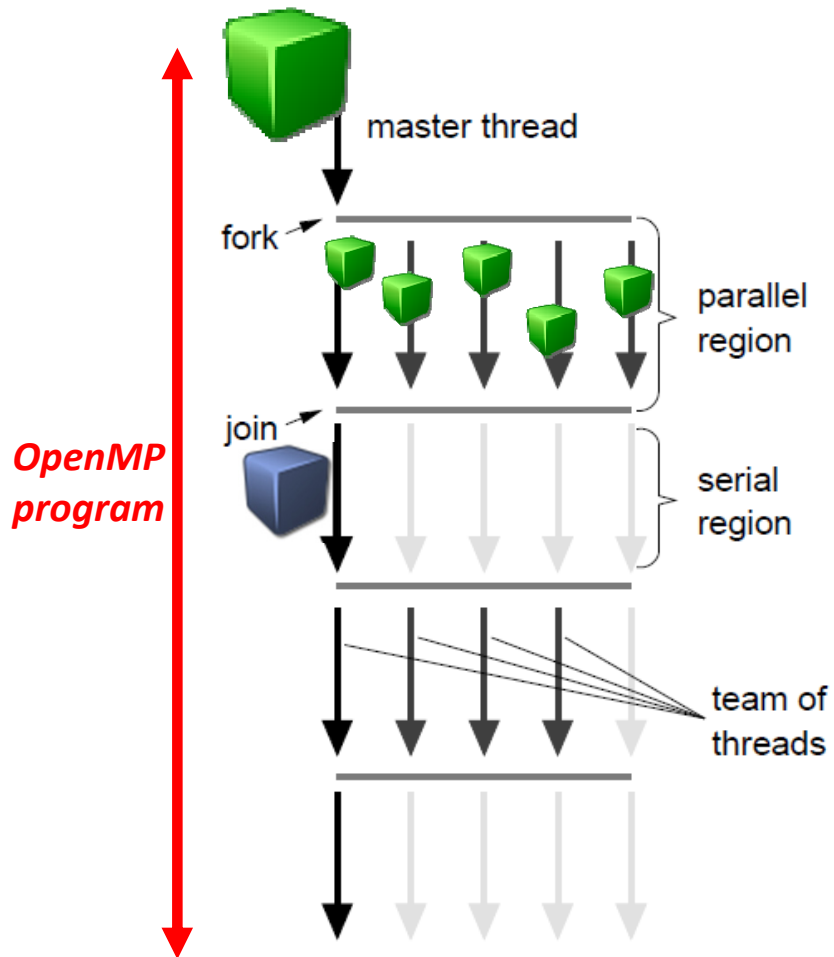


- **Thread:** An execution entity with a stack and associated static memory, called thread private memory
- **OpenMP Thread:** A thread that is managed by the OpenMP runtime system
- **Team:** A set of one or more threads participating in the execution of a parallel region
- **Task:** A specific instance of executable code and its data environment that the OpenMP implementation can schedule for execution by threads
- **Base Language:** A programming language that serves as the foundation of the OpenMP specification
- **Base Program:** A program written in the base language
- **OpenMP Program:** A program that consists of a base program that is annotated with OpenMP directives or that calls OpenMP API runtime library routines.
- **Directive:** In C/C++, a `#pragma` that specifies OpenMP program behavior



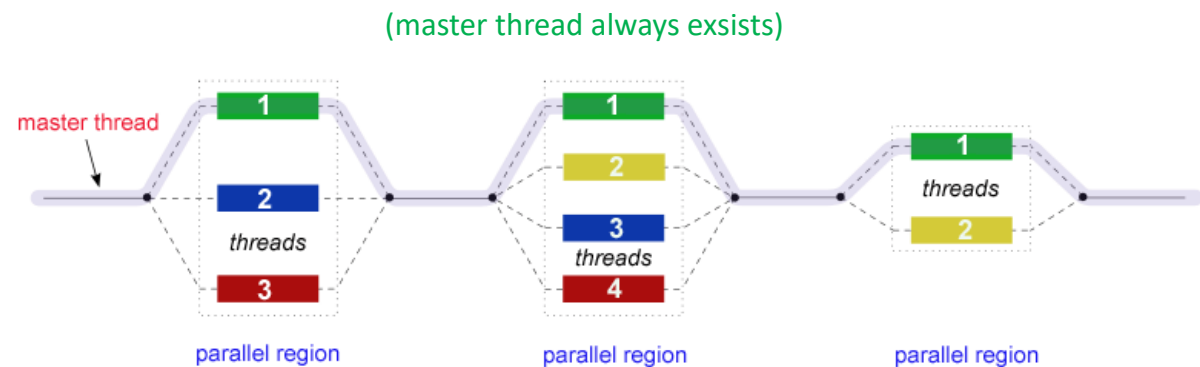
[3] OpenMP API Specification

Understanding Parallel & Serial Regions



- `fork()` initiated by master thread (exists always) creates team of threads
- Team of threads concurrently work on shared-memory data actively in parallel regions
- `join()` initiates the 'shutdown' of the parallel region and terminates team of threads
- Team of threads maybe also put to sleep until next parallel region begins
- Number of threads can be different in each parallel region

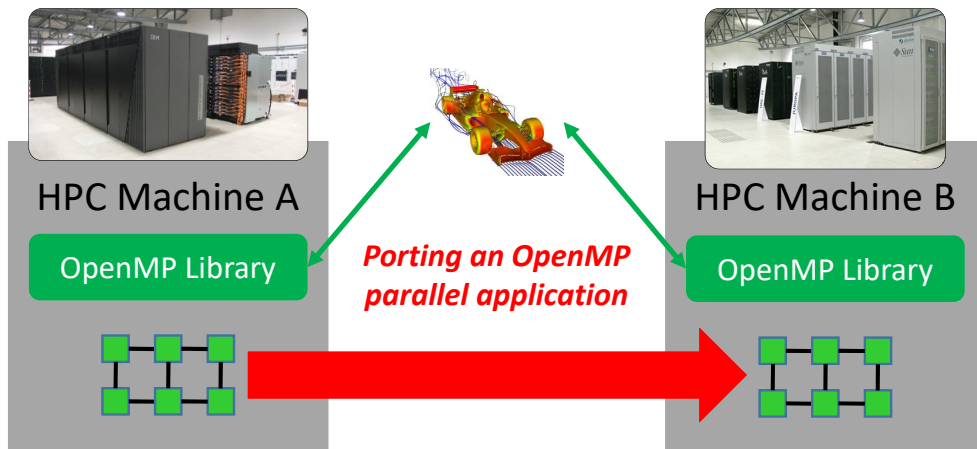
Modified from [2] Introduction to High Performance Computing for Scientists and Engineers



[7] LLNL OpenMP Tutorial

OpenMP Standard enables Portability

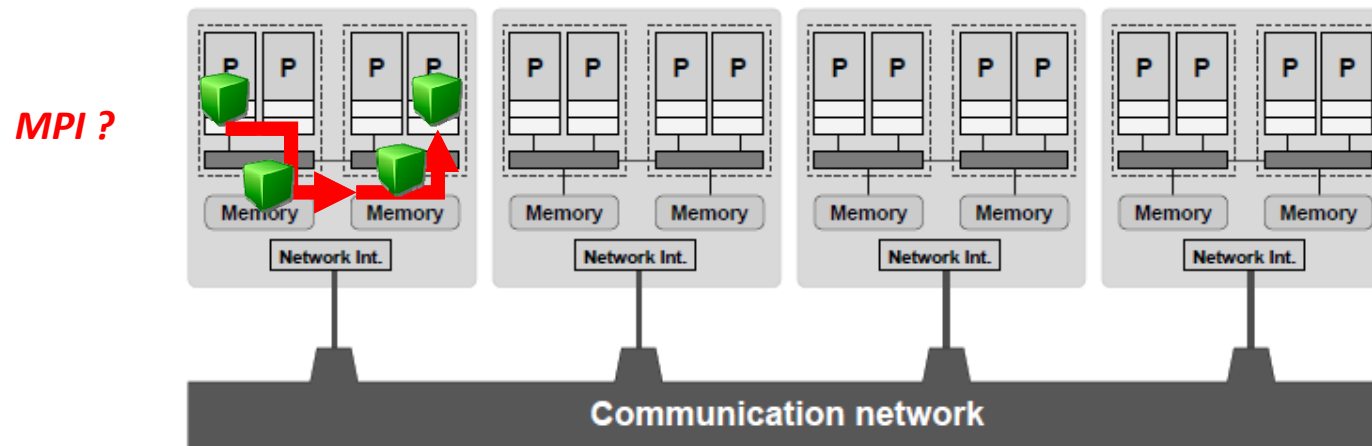
- Key **reasons** for requiring a standard programming library
 - **Technical advancement** in supercomputers is extremely fast
 - Parallel computing experts switch organizations and **face another system**
- Applications using proprietary libraries where **not portable**
 - Create whole **applications from scratch** or **time-consuming code updates**
- OpenMP is **parallel programming model for UMA and ccNUMA**



- OpenMP is an open standard that significantly supports the portability of parallel shared-memory applications
- But different vendors might implement it differently

Programming Hybrid Systems – Motivation

- Inefficient ‘on-node communications’ when using MPI instead of OpenMP
 - MPI uses ‘buffering techniques’ to transfer data (cf. Lecture 2 & 4)
 - Transfers may require ‘multiple memory copies’ to get data from A to B
 - Comparable to a ‘memory copy’ between different MPI processes
- Take advantage of shared memory techniques where feasible
 - OpenMP threads can read memory on the same node

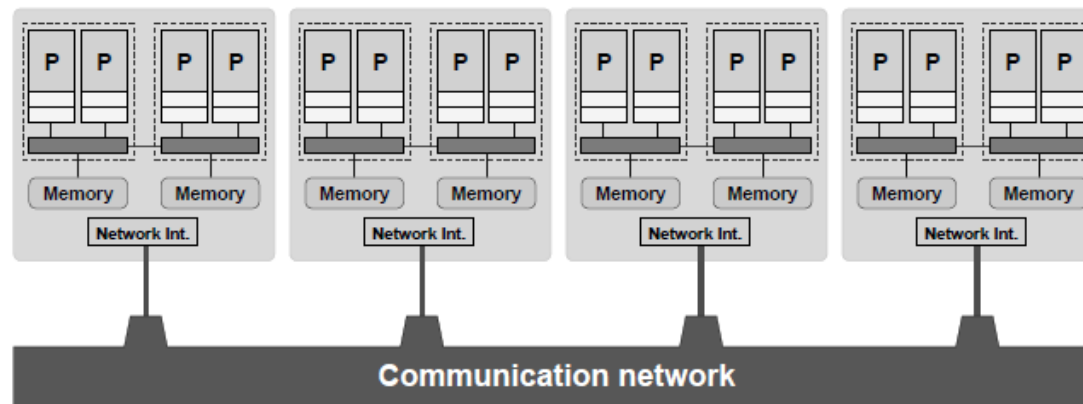


Modified from [2] Introduction to High Performance Computing for Scientists and Engineers

Hierarchical Hybrid Computers – Revisited (cf. Lecture 1)

- A hierarchical hybrid parallel computer is neither a purely shared-memory nor a purely distributed-memory type system but a mixture of both
- Large-scale ‘hybrid’ parallel computers have shared-memory building blocks interconnected with a fast network today

[2] Introduction to High Performance Computing for Scientists and Engineers



■ Features

- Shared-memory nodes (here ccNUMA) with local NIs
- NI mediates connections to other remote ‘SMP nodes’

➤ Lecture 10 will provide insights into hybrid programming models and introduces selected patterns used in parallel programming

Programming Hybrid Systems & Patterns – Revisited (cf. Lecture 1)

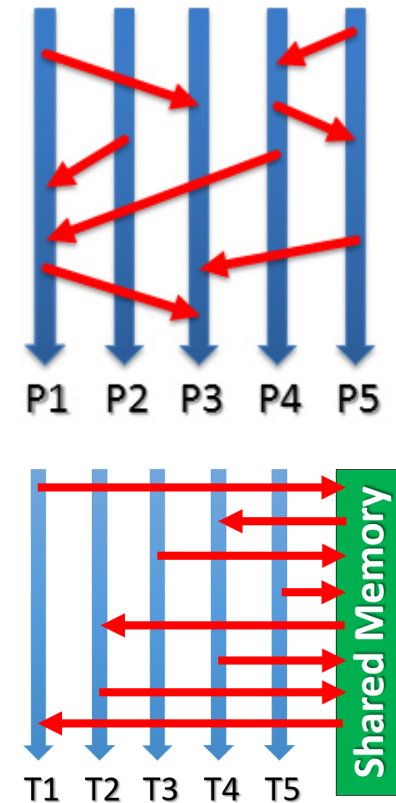
- Hybrid systems programming uses MPI as explicit internode communication and OpenMP for parallelization within the node
- Parallel Programming is often supported by using 'patterns' such as stencil methods in order to apply functions to the domain decomposition

■ Experience from HPC Practice

- Most parallel applications still take no notice of the hardware structure
- Use of **pure MPI for parallelization remains the dominant programming**
- Historical reason: old supercomputers all distributed-memory type
- **Use of accelerators is significantly increasing in practice today**

■ Challenges with the 'mapping problem'

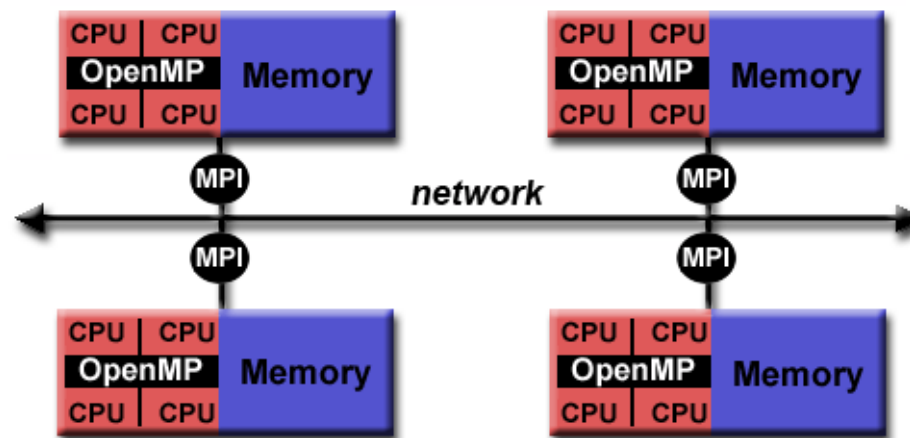
- Performance of hybrid (as well as pure MPI codes) depends crucially on factors not directly connected to the programming model
- It largely depends on the **association of threads and processes to cores**
- **Patterns (e.g., stencil methods) support the parallel programming**



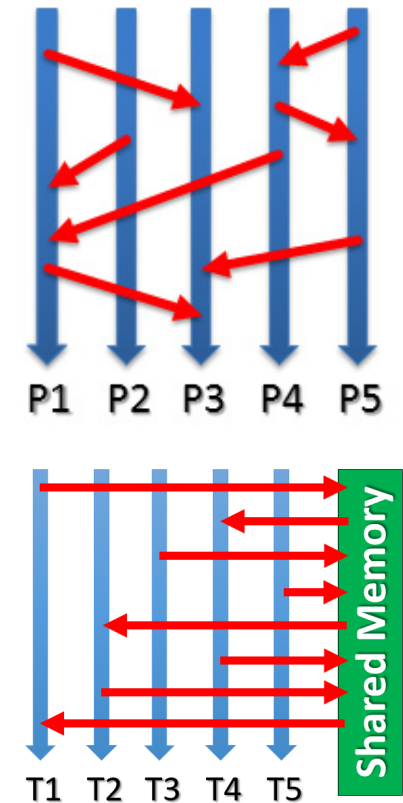
➤ Lecture 10 will provide insights into hybrid programming models and introduces selected patterns used in parallel programming

Programming Hybrid Systems with MPI & OpenMP

- Hybrid systems programming uses MPI as explicit internode communication and OpenMP for parallelization within the node
- Parallel Programming is often supported by using 'patterns' such as stencil methods in order to apply functions to the domain decomposition



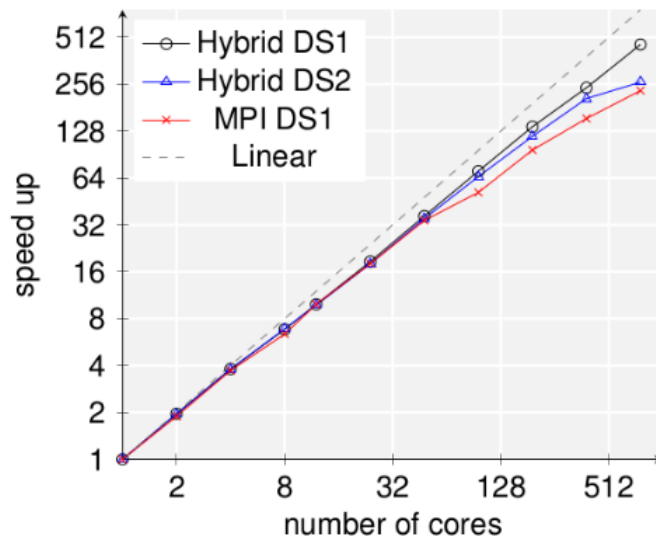
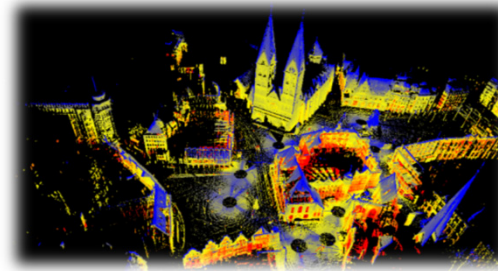
[7] LLNL OpenMP Tutorial



Scientific Application Example: Data Mining & Clustering

■ Hybrid data mining algorithm example

- Parallel Density-based Spatial Clustering for Applications with Noise (DBSCAN)
- Using MPI and OpenMP to scale better
- Standalone OpenMP is also possible to use



```
#!/bin/bash
#SBATCH --job-name=HPDBSCAN
#SBATCH -o HPDBSCAN-%j.out
#SBATCH -e HPDBSCAN-%j.err
#SBATCH --nodes=2
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=4
#SBATCH --time=00:20:00
#SBATCH --cpus-per-task=4
#SBATCH --reservation=ml-hpc-1

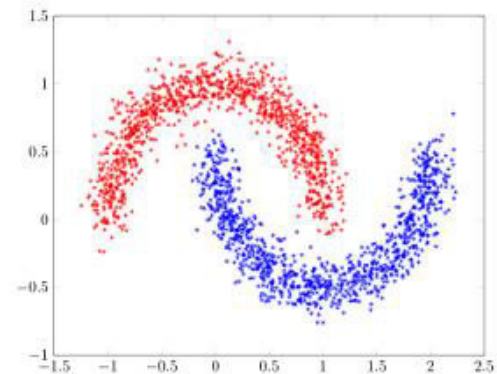
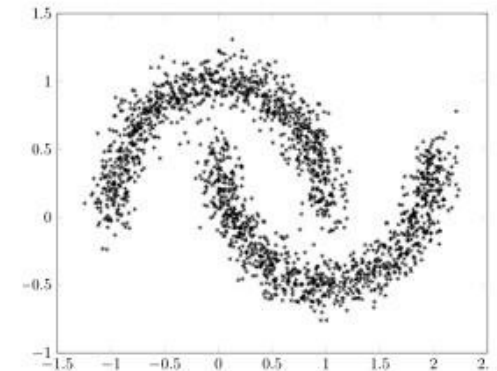
export OMP_NUM_THREADS=4

# location executable
HPDBSCAN=/homea/hpclab/train001/tools/hpdbscan/dbscan

# your own copy of bremen small
BREMENSMALLDATA=/homea/hpclab/train001/bremenSmall.h5

# your own copy of bremen big
BREMENBIGDATA=/homea/hpclab/train001/bremen.h5

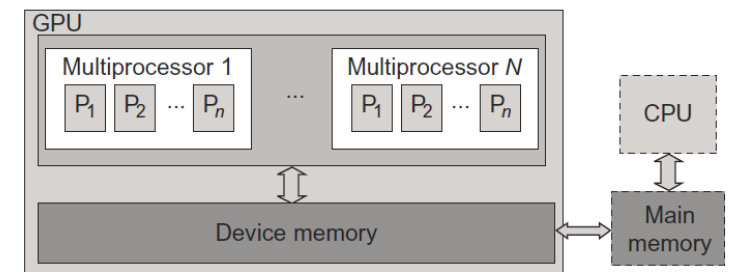
srun $HPDBSCAN -m 100 -e 300 -t 12 $BREMENSMALLDATA
```



[1] M. Goetz and M. Riedel et al,
Proceedings IEEE Supercomputing Conference, 2015

Many-core GPGPUs – Revisited (cf. Lecture 1)

- Use of very many simple cores
 - High throughput computing-oriented architecture
 - Use massive parallelism by executing a lot of concurrent threads slowly
 - Handle an ever increasing amount of multiple instruction threads
 - CPUs instead typically execute a single long thread as fast as possible
- Many-core GPUs are used in large clusters and within massively parallel supercomputers today
 - Named General-Purpose Computing on GPUs (GPGPU)
 - Different programming models emerge



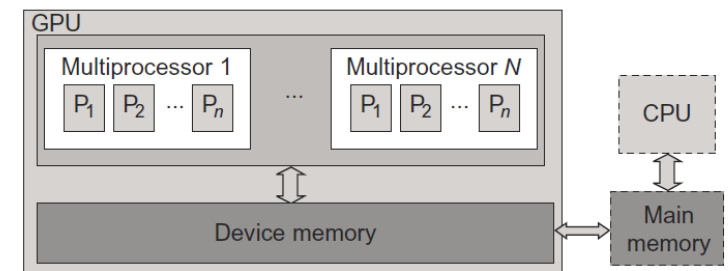
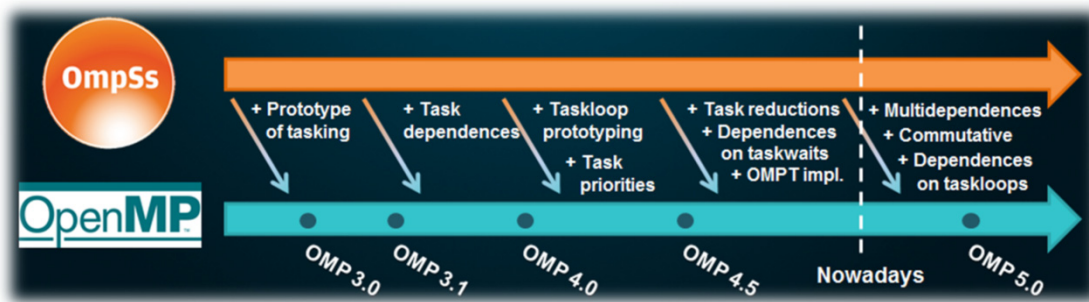
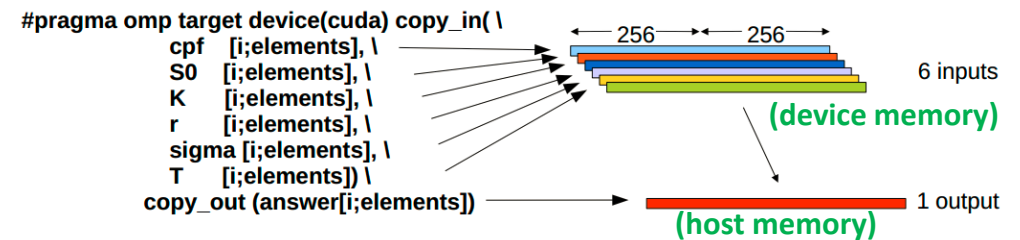
[9] Distributed & Cloud Computing Book

- Graphics Processing Unit (GPU) is great for data parallelism and task parallelism
- Compared to multi-core CPUs, GPUs consist of a many-core architecture with hundreds to even thousands of very simple cores executing threads rather slowly

DEEP-EST EU Project – OmpSs & OpenMP Evolutions

- OmpSs is an innovative programming model **influencing OpenMP**
 - Based on tasks and (data) dependencies – tasks as elementary unit of work
 - Extend OpenMP model: **better data-flow & heterogeneity** (e.g. GPGPUs)
 - New Version: **OmpSs-2**

[11] OmpSs Web Page



[9] Distributed & Cloud Computing Book

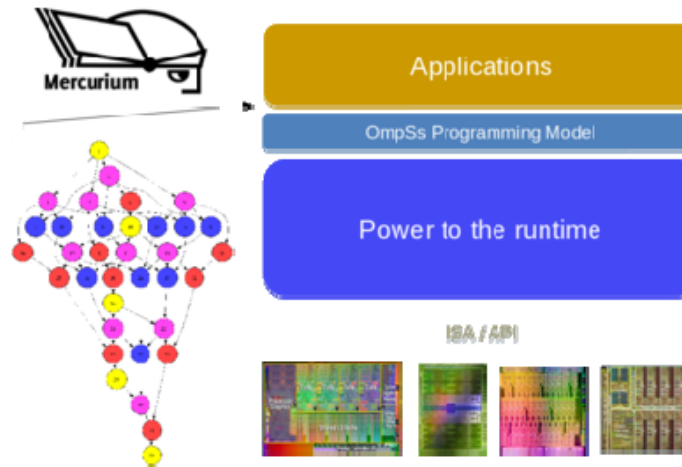


[10] DEEP Projects Web Page

OmpSs Programming Model – Adding Task & Data Dependencies

OmpSs Programming Model

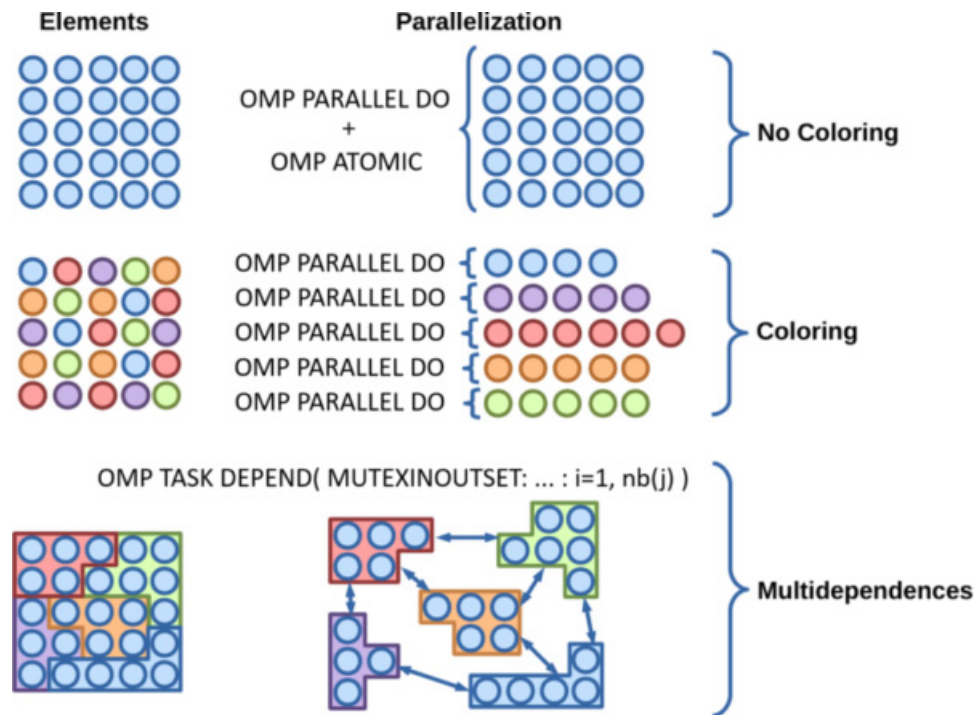
```
void Cholesky( float *A[NT][NT] ) {  
  int i, j, k;  
  for (k=0; k<NT; k++) {  
    #pragma omp task inout (A[k][k])  
    spotrf (A[k][k]) ;  
    for (i=k+1; i<NT; i++) {  
      #pragma omp task in (A[k][k]) inout (A[k][i])  
      strsm (A[k][k], A[k][i]);  
    }  
    for (i=k+1; i<NT; i++) {  
      for (j=k+1; j<i; j++) {  
        #pragma omp task in (A[k][i], A[k][j]) inout (A[j][i])  
        sgemm( A[k][i], A[k][j], A[j][i]);  
      }  
      #pragma omp task in (A[k][i]) inout (A[i][i])  
      ssyrk (A[k][i], A[i][i]);  
    }  
  }  
}
```



- OmpSs main goal is to act as a forefront and nursery of ideas for a data-flow task-based programming model so these ideas can ultimately be incorporated in the OpenMP industrial standard

[12] OmpSs BSC Programming Models

Enabling Parallelization Approaches with Task Multi-Dependencies



[13] MontBlanc OmpSs Multi-Task Dependencies

COMPSs & PyCOMPSs

■ COMPSs (COMP Superscalar)

- Coarse-grained programming model oriented to distributed environments
- Powerful runtime that leverages low-level APIs (e.g., Amazon EC2 clouds)
- Manages data dependencies (objects and files)

- **COMP Superscalar (COMPSs) is a framework which aims to ease the development and execution of applications for distributed infrastructures, such as Clusters, Grids and Clouds.**
- **PyCOMPSs is the Python binding of COMPSs**
- **PyCOMPSs follows OpenMP & OmpSs approach: from a sequential Python code, it is able to run in parallel and distributed**



[14] PyCOMPSs

```
from pycompss.api.task import task
from pycompss.api.parameter import FILE_INOUT

@task(filePath = FILE_INOUT)
def increment(filePath):
    # Read value
    fis = open(filePath, 'r')
    value = fis.read()
    fis.close()

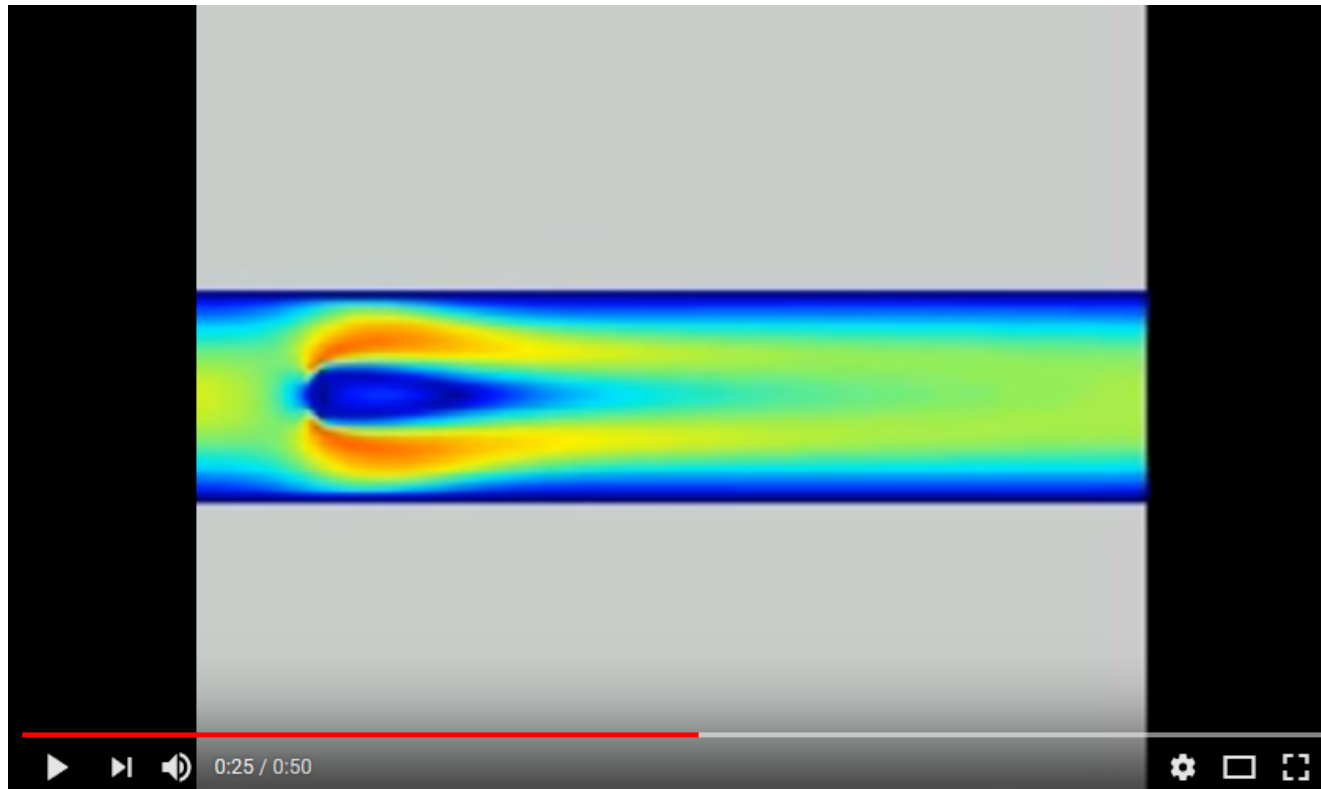
    # Write value
    fos = open(filePath, 'w')
    fos.write(str(int(value) + 1))
    fos.close()

def main_program():
    from pycompss.api.api import compss_open

    # Check and get parameters
    if len(sys.argv) != 2:
        exit(-1)
    initialValue = sys.argv[1]

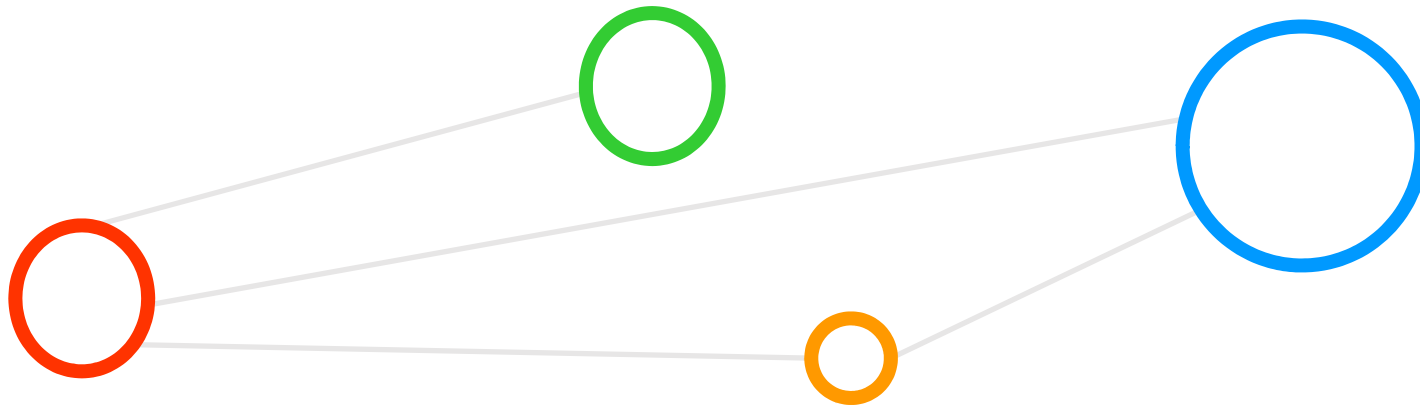
    fileName="counter"
```


[Video] Scientific Application Example using OpenMP



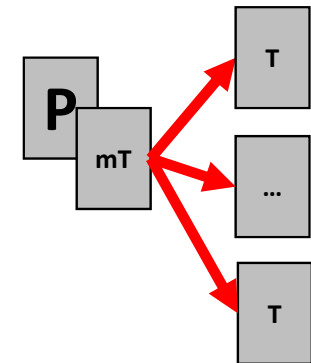
[4] Lattice Boltzmann – Flow past an obstacle, YouTube Video

OpenMP Parallel Programming Basics



Start 'Thinking' Parallel

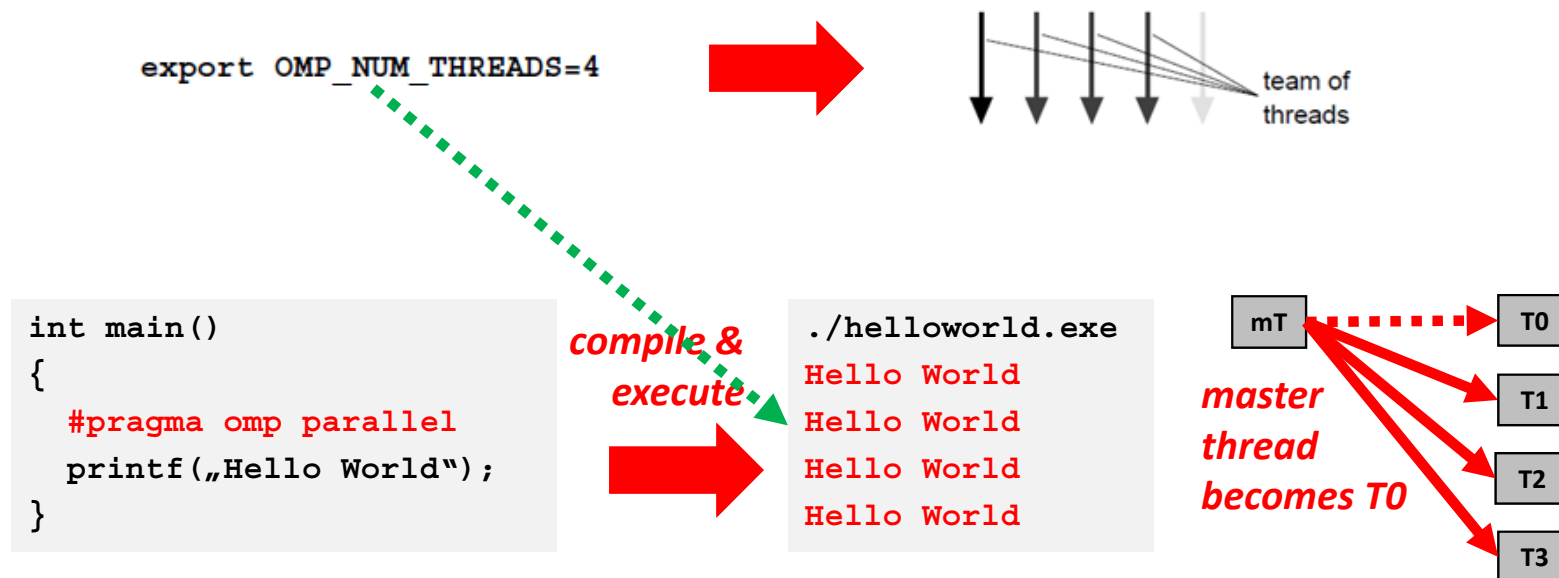
- Parallel OpenMP program
 - Knows about the **existence of a certain number of threads** that all work together as part of a bigger picture
- OpenMP programs
 - Written in a **sequential** programming language and some parts are **executed in parallel**
 - Run on a processor that 'spawns' numerous threads
- **Parallelization**
 - **Dedicated n parallel regions** is key to the design in OpenMP ($n = 1, 2, 3, \dots$)
 - E.g. **loops/additions are good candidates for parallelization**
 - (if individual loop iterations are independent from each other)
- Start with the **basic building blocks** using OpenMP
 - Defining code that enables '**parallel computing**', step-by-step is possible



Number of Threads & Scalability

- The real number of threads normally **not known at compile time**
 - (There are methods for doing it in the program → do not use them!)
 - Number is **set in scripts and/or environment variable** before executing
 - Parallel programming is done **without knowing number of threads**

- OpenMP programs should be always written in a way that it does not assume a specific number of threads that in turn enables a scalable program
- Compiler directives are used such as `#pragma omp parallel`



OpenMP Basic Building Blocks: Hello World Example

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

```
#include <stdio.h>
```

```
int main(argc,argv)
```

```
int argc; char *argv[]; {
```

```
    int nthreads, tid;
```

```
    #pragma omp parallel private(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 in parallel region = %d\n", nthreads);
```

```
        }
```

```
    }
```

```
}
```

- The OpenMP library contains OpenMP API definitions

- Shared variable nthreads; local variable tid

- The Sentinel is a special string that starts an OpenMP compiler directive: '#pragma omp'

- private defines local variables for each thread
- Each thread works independently and thus needs space to 'store' private local results

- omp_get_thread_num() function provides unique Thread ID (0...n-1)

- Same code executed n times with n threads, but tid is unique and thus different for each thread

- Similar like MPI ranks, here the Thread ID can be used to perform different executions per threads

- Only the master (tid=0) provides output of how many threads are existing in the parallel region

- omp_get_num_threads() function obtains number of active threads in the current parallel region

➤ Practical examples and assignments in this course focus on parallel programming with MPI, but OpenMP is important in practice too

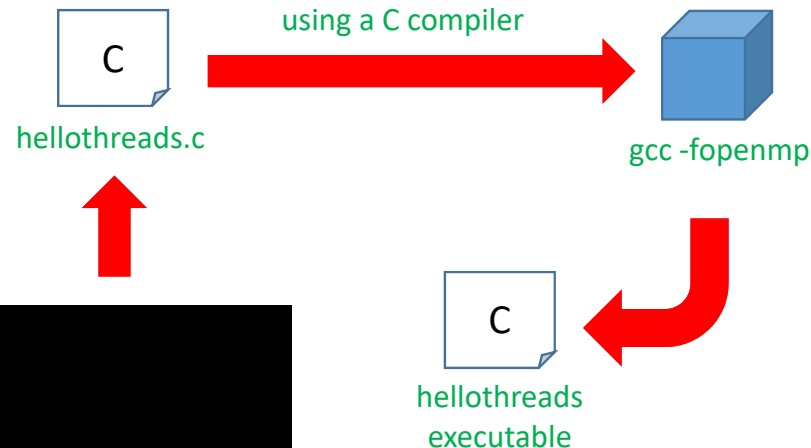
Edit C Program with OpenMP Directives & Functions & Compilation

- Using basic gcc compiler

- 'module load gnu openmpi'

- Note: there are many C compilers available, we here pick one for our particular HPC course that works with OpenMP
 - Note: If there are no errors, the file **hellothreads** is now a full C program executable that can be started by an OS

```
[morris@jotunn hellothreads]$ gcc -fopenmp -o hellothreads hellothreads.c
[morris@jotunn hellothreads]$ ls -al
total 16
drwxrwxr-x 2 morris morris  46 okt  9 20:08 .
drwxrwxr-x 3 morris morris  25 okt  9 20:04 ..
-rwxrwxr-x 1 morris morris 8844 okt  9 20:08 hellothreads
-rw-rw-r-- 1 morris morris  275 okt  9 20:05 hellothreads.c
```



```
#include <stdio.h>
#include <omp.h>
int main(void)
{
    printf("Hello from your main thread.\n");

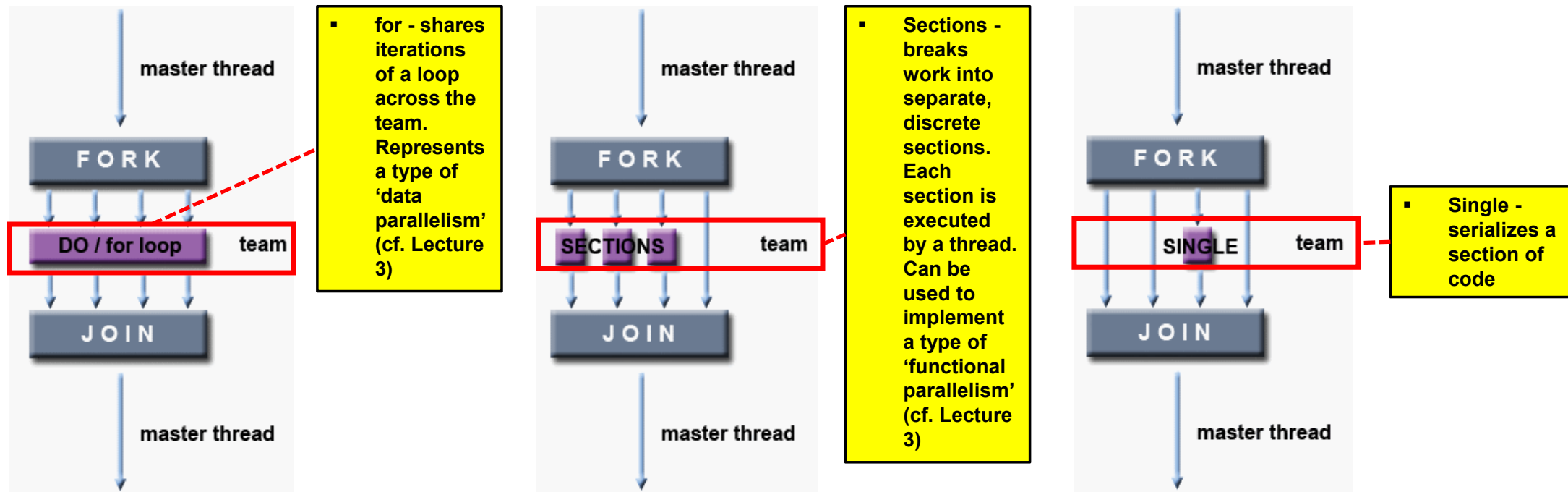
    #pragma omp parallel
    printf("Hello from thread %d of %d.\n", omp_get_thread_num(), omp_get_num_threads());

    printf("Hello again from your main thread.\n");
    return 0;
}
```



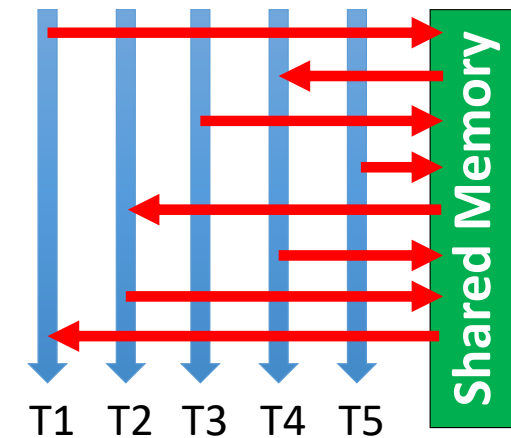
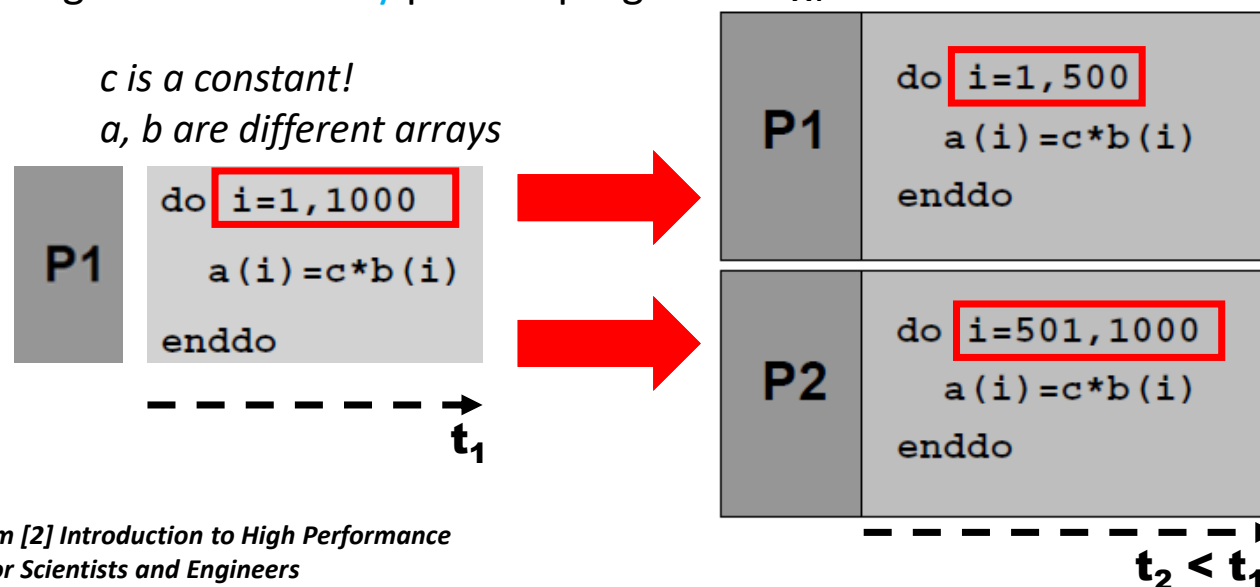
[1] Icelandic HPC Machines & Community

OpenMP Work Sharing Constructs – Overview



Data Parallelism: Medium-grained Loop Parallelization (cf. Lecture 3)

- Idea: Computations performed on individual array elements are **independent** of each other
 - Good for parallel execution by N processors (e.g., using **shared memory** parallel programming)



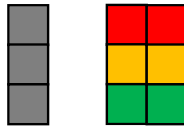
OpenMP Work Sharing Construct: Simple For Loop Example

```
#include <omp.h>

int main(int argv, char **argv)
{
    int n, i;
    double *x, *y;
    /* Get input size */
    n = atoi(argv[1]);
    x = (double *)malloc(n*sizeof(double));
    y = (double *)malloc(n*sizeof(double));
    #pragma omp parallel for private(i) shared (x,y)
    for (i=0; i<n; i++)
    {
        x[i] = x[i] + y[i];
    }

    /* x contains the result for all vector elements */
}
```

'simplified demo code'



$X = X + Y$

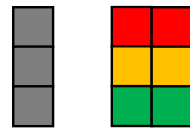
- The Sentinel is a special string that starts an OpenMP compiler directive
- Smart programming support by OpenMP: Loops are very often part of scientific applications
- Less burden for programmer: no manual definition of local variables (e.g. *i* automatically localized)
- Directive is optimized to enable a parallel loop (i.e. parallel for) starting a parallel region
- Private defines local variables for each thread
- Each thread works independently and thus needs space to 'store' local results – here *i* as index
- Shared defines global variables that exist only one time
- Each thread works independently but shared variables can be written and read from all threads

OpenMP Work Sharing Construct: Advanced For Loop Example

```
include <omp.h>

#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[]) {
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel region */
    return 0;
}
```



$$C = A + B$$

- Arrays a, b, c, and chunk will be shared by all threads
- Variable i is private to each thread: each thread will have its own unique copy of i

- Schedule: Describes how iterations of the loop are divided among the threads in the team; the default schedule is implementation dependent
- The iterations of the loop will be distributed dynamically in CHUNK sized pieces: when a thread finishes one chunk, it is dynamically assigned another
- Threads will not synchronize upon completing their individual pieces of work (nowait)

STATIC



DYNAMIC



[7] LLNL OpenMP Tutorial

OpenMP Work Sharing Construct: Sections Example

```
#include <omp.h>

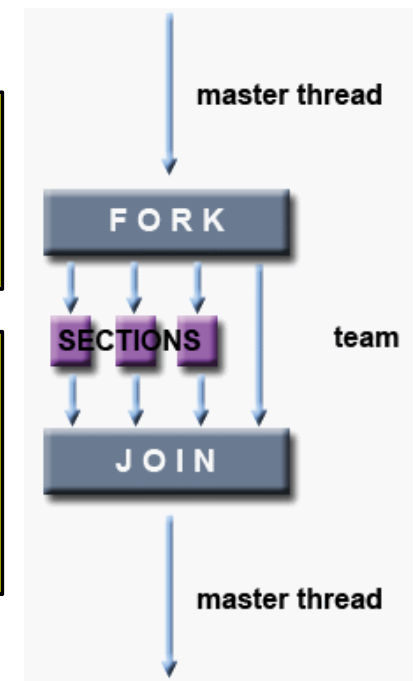
#define N 1000

main(int argc, char *argv[]) {
    int i;
    float a[N], b[N], c[N], d[N];
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
    }
    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];

            #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
        }
    }
}
```

- The sections directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team
- Independent section directives are nested within a sections directive

- Each section is executed once by a thread in the team.
- Different sections may be executed by different threads: It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such
- Different blocks of work in sections will be done by different threads



[7] LLNL OpenMP Tutorial

OpenMP Synchronization Construct: Critical Region Example

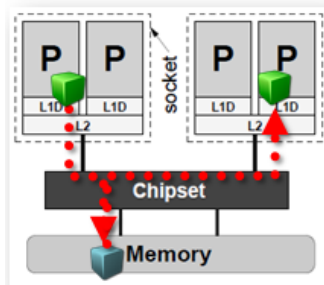
```
#include <omp.h>

main(int argc, char *argv[]) {
    int x;
    x = 0;

    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x + 1;

    } /* end of parallel region */
    return 0;
}
```

- If a thread is currently executing inside a critical region and another thread reaches that critical region and attempts to execute it, it will block until the first thread exits that critical region
- All threads in the team will attempt to execute in parallel, however, because of the critical construct surrounding the increment of x, only one thread will be able to read/increment/write x at any time
- Note the 'race conditions' of variable x otherwise: Race Condition in shared-memory: shared variable x will be set concurrently by the different threads – not with critical regions



[7] LLNL OpenMP Tutorial

OpenMP ThreadPrivate Directive – Persistence between Parallel Regions (1)

```
#include <omp.h>

int a, b, i, tid;
float x;
#pragma omp threadprivate(a, x)
main(int argc, char *argv[]) {
    omp_set_dynamic(0);
    printf("1st Parallel Region:\n");
    #pragma omp parallel private(b,tid)
    {
        tid = omp_get_thread_num();
        a = tid;
        b = tid;
        x = 1.1 * tid + 1.0;
        printf("Thread %d:  a,b,x= %d %d %f\n",tid,a,b,x);
    } /* end of parallel region */
    printf("Master thread doing serial work here\n");
    ...
}
```

- Threadprivate() directive specifies that variables are replicated, with each thread having its own copy
- Can be used to make global file scope variables (C/C++/Fortran local and persistent to a thread through the execution of multiple parallel regions
- Threadprivate() variables differ from private variables because they are able to persist between different parallel regions of a code

- Explicitly turn off dynamic threads


[7] LLNL OpenMP Tutorial

OpenMP ThreadPrivate Directive – Persistence between Parallel Regions (2)

```
#include <omp.h>

int a, b, i, tid;

float x;
#pragma omp threadprivate(a, x)
main(int argc, char *argv[]) {
    ...
    printf("Master thread doing serial work here\n");
    ...
    printf("2nd Parallel Region:\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
    } /* end of parallel region */
}
```



Output:

1st Parallel Region:

Thread 0: a,b,x= 0 0 1.000000

Thread 2: a,b,x= 2 2 3.200000

Thread 3: a,b,x= 3 3 4.300000

Thread 1: a,b,x= 1 1 2.100000

Master thread doing serial work here

2nd Parallel Region:

Thread 0: a,b,x= 0 0 1.000000

Thread 3: a,b,x= 3 0 4.300000

Thread 1: a,b,x= 1 0 2.100000

Thread 2: a,b,x= 2 0 3.200000

[7] LLNL OpenMP Tutorial

OpenMP Reduction Clause Example – Vector Dot Product Example

```
#include <omp.h>

main(int argc, char *argv[]) {
    int i, n, chunk;
    float a[100], b[100], result;
    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
    #pragma omp parallel for \
        default(shared) private(i) \
        schedule(static,chunk) \
        reduction(+:result)
    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
    printf("Final result= %f\n",result);
}
```

- Reduction clause performs a reduction operation on the variables that appear in its list
- A private copy for each list variable is created and initialized for each thread
- At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable
- Reduction operations are a smart alternative to manual critical regions definitions around operations of variables
- Reduction operation automatically localizes variable
- Several operations are common in scientific applications: +, *, -, &, |, ^, &&, ||, max, min
- REDUCTION() with operator + on variable s enables here ...
- Starting with a local copy of s for each thread
- During progress of parallel region each local copy of s will be accumulated separately by each thread
- At the end of the parallel region automatically synchronized and accumulated with resulting master thread variable

STATIC

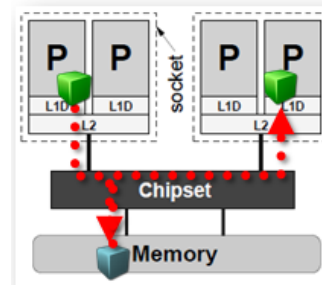
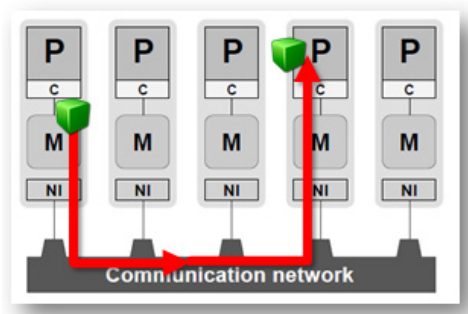


- Vector Dot Product Example: Result is a scalar!
- Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the team (schedule static)
- At the end of the parallel loop construct, all threads will add their values of "result" to update the master thread's global copy

[7] LLNL OpenMP Tutorial

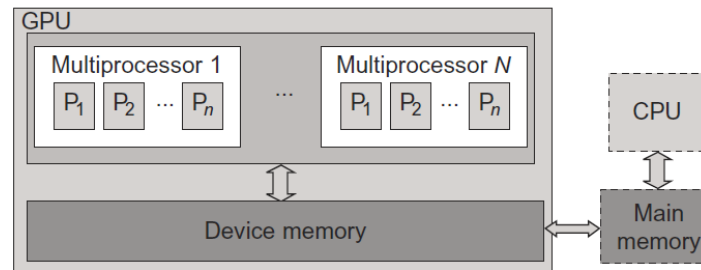
Selected Comparisons with MPI

- Some aspects are similar, because both enable **parallel** computing
 - Obtaining **unique IDs**: MPI ranks vs. OpenMP thread-num
 - **Master-worker approach** (if rank==0 vs. if tid ==0)
- No **explicit communication constructs** to enable inter-process communication in OpenMP → assuming shared-memory
 - **Data exchange**: Message exchanges between processes vs. shared variable
 - **Synchronization** functions nevertheless exist in both: e.g. barriers
 - **Clever automatisms for usual problems**: MPI reduce vs. OpenMP reduction



Recent Support of OpenMP for Programming GPUs with Directives

```
...
#pragma omp target map (tofrom:y), map(to:x)
#pragma omp teams num_teams(10) num_threads(10)
#pragma omp distribute
for (...) {
    ...
    #pragma omp parallel for
    for (...) {
    }
    ...
}
...
```

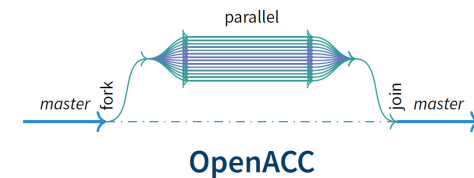
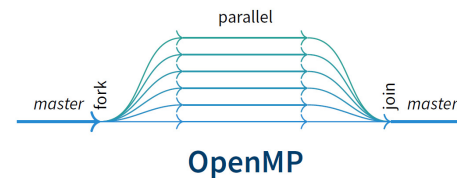


[9] Distributed & Cloud Computing Book

- OpenACC is similar to OpenMP, because it is modeled after OpenMP, but for accelerators

```
#pragma acc data copyout(y[0:N]) create(x[0:N])
{
    double sum = 0.0;
    #pragma acc parallel loop
    for (int i=0; i<N; i++) {
        x[i] = 1.0;
        y[i] = 2.0;
    }
    #pragma acc parallel loop
    for (int i=0; i<N; i++) {
        y[i] = i*x[i]+y[i];
    }
}
```

- OpenMP is the de-facto standard for multi-threaded programming on CPU
- OpenMP includes since version 4.0 (better since 4.5) also capabilities for programming GPUs



➤ Lecture 7 will offer more details on OpenMP relationships of programming GPUs and similarities to GPU programming using OpenACC

Monitoring, Debugging and Performance Analysis Tools for OpenMP

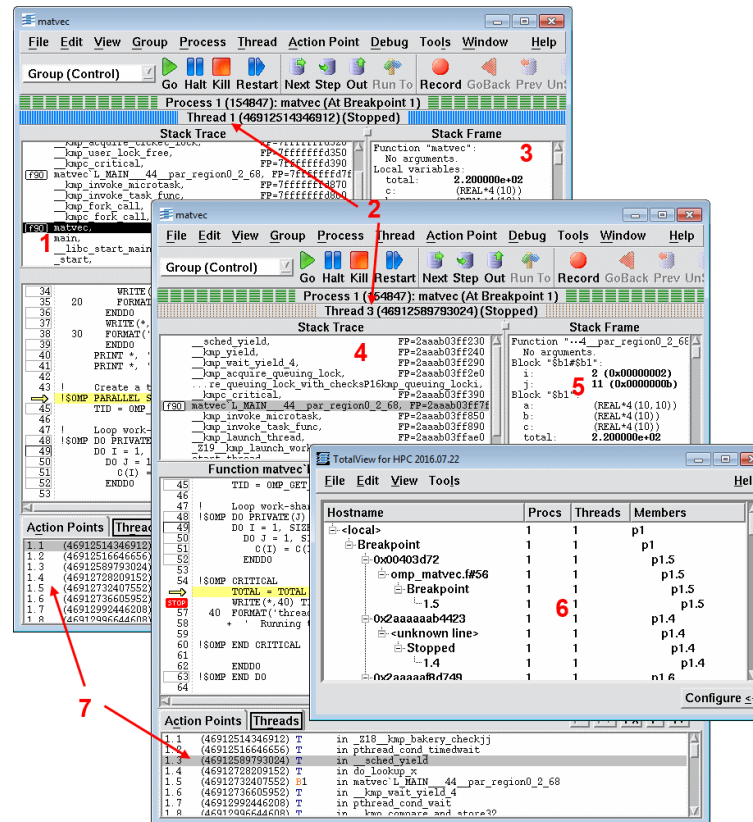
■ Different Tools exist

- E.g. **TotalView** Debugger
- E.g. Linux **top** command
- E.g. Linux **ps** command

```
% ps -lf
UID          PID  PPID  LWP  C  NLWP  STIME  TTY          TIME CMD
blaise      22529 28240 22529  0   5 11:31 pts/53    00:00:00 a.out
blaise      22529 28240 22530  99   5 11:31 pts/53    00:01:24 a.out
blaise      22529 28240 22531  99   5 11:31 pts/53    00:01:24 a.out
blaise      22529 28240 22532  99   5 11:31 pts/53    00:01:24 a.out
blaise      22529 28240 22533  99   5 11:31 pts/53    00:01:24 a.out
```

```
% ps -T
PID  SPID  TTY          TIME CMD
22529 22529 pts/53    00:00:00 a.out
22529 22530 pts/53    00:01:49 a.out
22529 22531 pts/53    00:01:49 a.out
22529 22532 pts/53    00:01:49 a.out
22529 22533 pts/53    00:01:49 a.out
```

```
% ps -lm
PID  LWP  TTY          TIME CMD
22529 - pts/53    00:18:56 a.out
- 22529 - 00:00:00 -
- 22530 - 00:04:44 -
- 22531 - 00:04:44 -
- 22532 - 00:04:44 -
- 22533 - 00:04:44 -
```



```
top - 14:13:21 up 2 days, 23:17, 20 users, load average: 3.34, 1.59, 0.73
Tasks: 471 total, 5 running, 465 sleeping, 1 stopped, 0 zombie
Cpu(s): 33.4%us, 1.7%sy, 0.0%ni, 56.6%id, 8.0%wa, 0.2%hi, 0.0%si, 0.0%st
Mem: 24479116k total, 19015304k used, 5463812k free, 117572k buffers
Swap: 4096564k total, 89432k used, 4007132k free, 16511060k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
18010	blaise	25	0	92292	1248	920	R	100.0	0.0	0:42.68	a.out
18012	blaise	25	0	92292	1248	920	R	100.0	0.0	0:42.62	a.out
18013	blaise	25	0	92292	1248	920	R	100.0	0.0	0:42.65	a.out
18014	blaise	25	0	92292	1248	920	R	99.7	0.0	0:42.61	a.out
617	root	15	0	0	0	0	D	1.3	0.0	0:15.36	pdfFlush
4344	root	15	0	0	0	0	S	0.7	0.0	1:37.12	kiblnsd_sd_02
4345	root	15	0	0	0	0	S	0.7	0.0	1:38.24	kiblnsd_sd_03
4352	root	15	0	0	0	0	S	0.7	0.0	1:37.56	kiblnsd_sd_10
5055	root	15	0	0	0	0	S	0.7	0.0	10:19.15	ptlrpcd

[7] LLNL OpenMP Tutorial

➤ Lecture 9 will provide a set of tools that can be used for monitoring, debugging, and performance analysis of MPI and OpenMP

Jacobi 2D Application Example – Shared Memory with OpenMP Possible

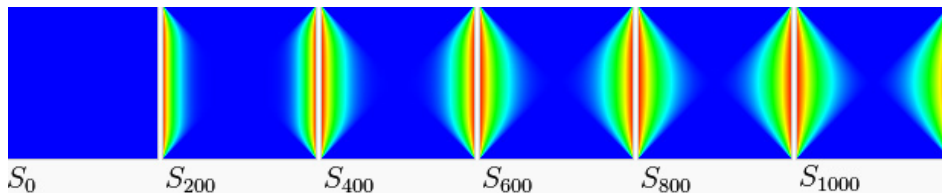
■ Solver

- Each diagonal element is solved and approximate value is plugged in
- The process is iterated until it **converges**

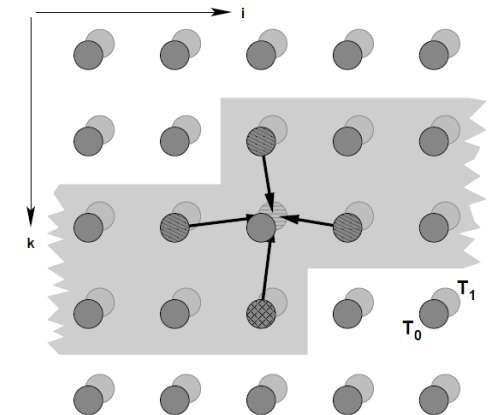
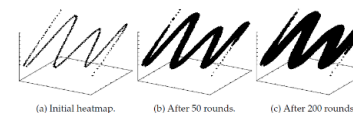
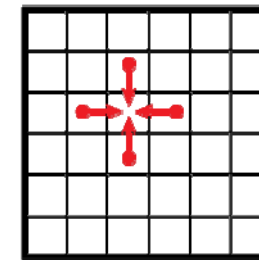
■ Update function 2D Jacobi iterative method example

- E.g. computes the arithmetic mean of a cell's four neighbours
- E.g. solving diffusion equations (**heat dissipation example**)

- The Jacobi iterative method is a stencil-based iterative method used in numerical linear algebra
- Algorithm for determining the solutions of diagonally dominant system of linear equations



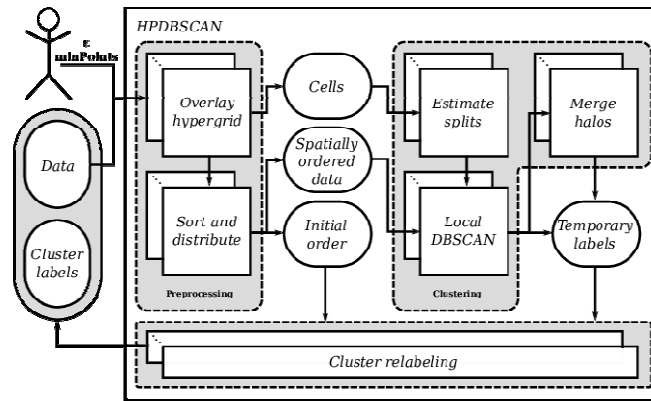
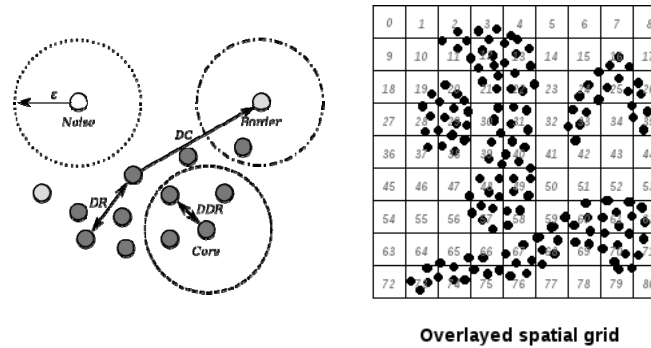
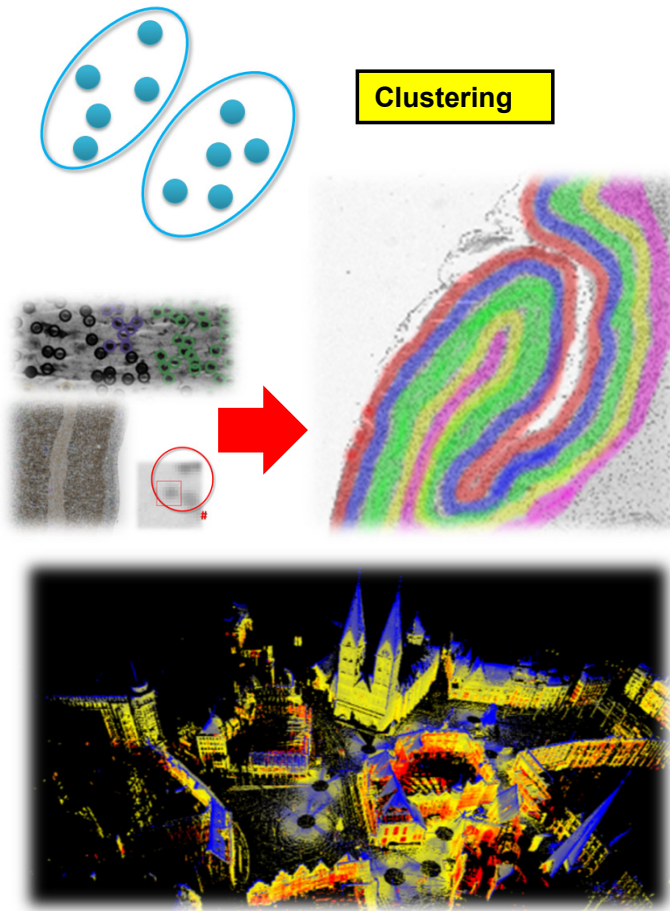
[5] Wikipedia on 'stencil code'



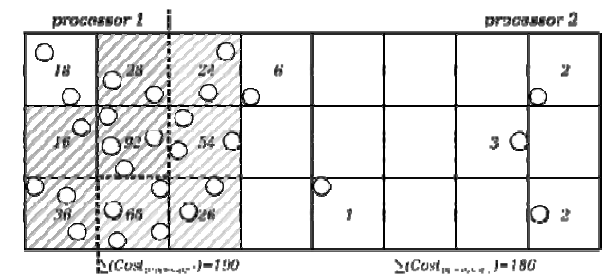
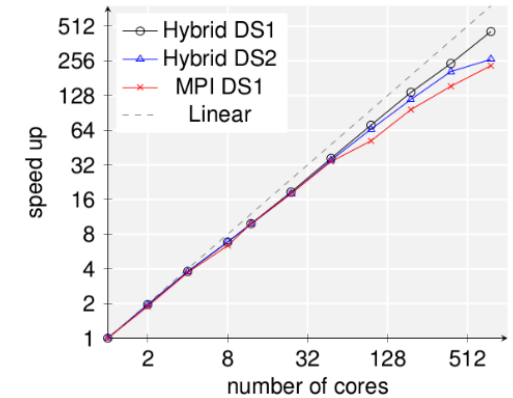
[2] Introduction to High Performance Computing for Scientists and Engineers

➤ Lecture 10 will provide more details about stencil-based iterative methods & used patterns in many different HPC application examples

'Big Data' Science Example – Parallel & Scalable Clustering Algorithm – Revisited



[1] M. Goetz and M. Riedel et al,
Proceedings IEEE Supercomputing Conference, 2015



HPDBSCAN Clustering OpenMP Application Example in Data Sciences

[8] M. Goetz HPDBSCAN Bitbucket Repository

// hpdbscan.h file

...

#include <hdf5.h>

#include <omp.h>

...

// local DBSCAN run

#pragma omp parallel for schedule(dynamic, 32) private(neighboring_points)
firstprivate(previous_cell) reduction(merge: rules)

for (size_t point = lower; point < upper; ++point) {

...

Clusters cluster(Dataset& dataset, int threads=omp_get_max_threads()) {

#ifdef WITH_OUTPUT

double execution_start = omp_get_wtime();

#endif

// set the number of threads

omp_set_num_threads(threads);

...

- Using the standard OpenMP omp.h header file
- Using #pragma omp parallel for loop compiler directive in combination with reduction operation

- How Many Threads within a parallel region?
- omp_get_max_threads() : generally reflects the number of threads as set by the OMP_NUM_THREADS environment variable
- omp_set_num_threads(threads) routine affects the number of threads to be used for subsequent parallel regions

```
#!/bin/bash
#SBATCH --job-name=HPDBSCAN
#SBATCH -o HPDBSCAN-%j.out
#SBATCH -e HPDBSCAN-%j.err
#SBATCH --nodes=2
#SBATCH --ntasks=4
#SBATCH --ntasks-per-node=4
#SBATCH --time=00:20:00
#SBATCH --cpus-per-task=4
#SBATCH --reservation=ml-hpc-1
```

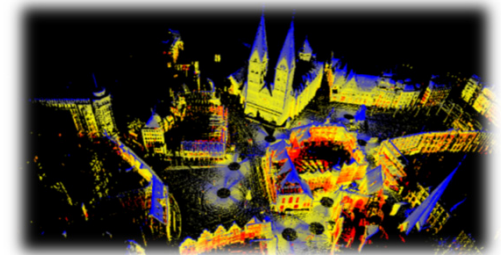
export OMP_NUM_THREADS=4

location executable
HPDBSCAN=/homea/hpclab/train001/tools/hpdbscan/dbscan

your own copy of bremen small
BREMENSMALLDATA=/homea/hpclab/train001/bremenSmall.h5

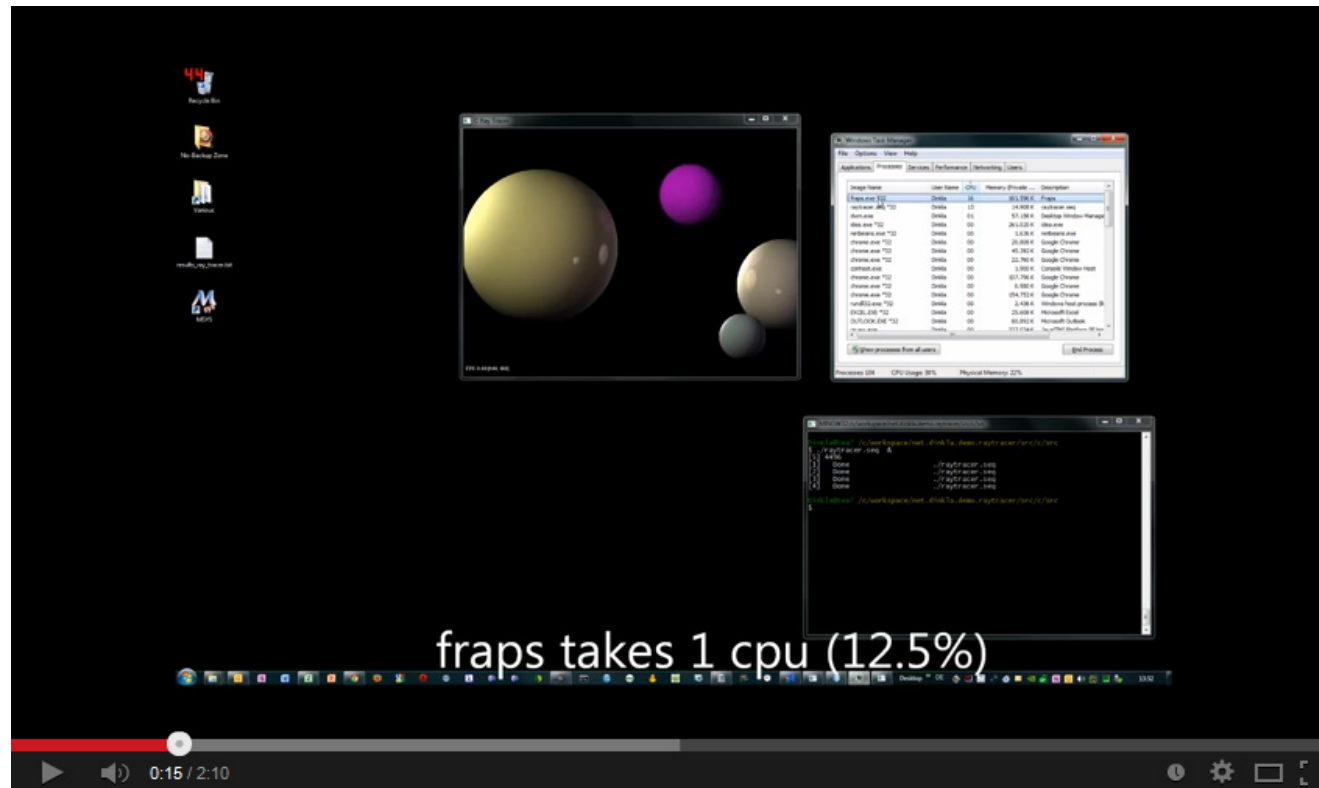
your own copy of bremen big
BREMENBIGDATA=/homea/hpclab/train001/bremen.h5

srun \$HPDBSCAN -m 100 -e 300 -t 12 \$BREMENSMALLDATA



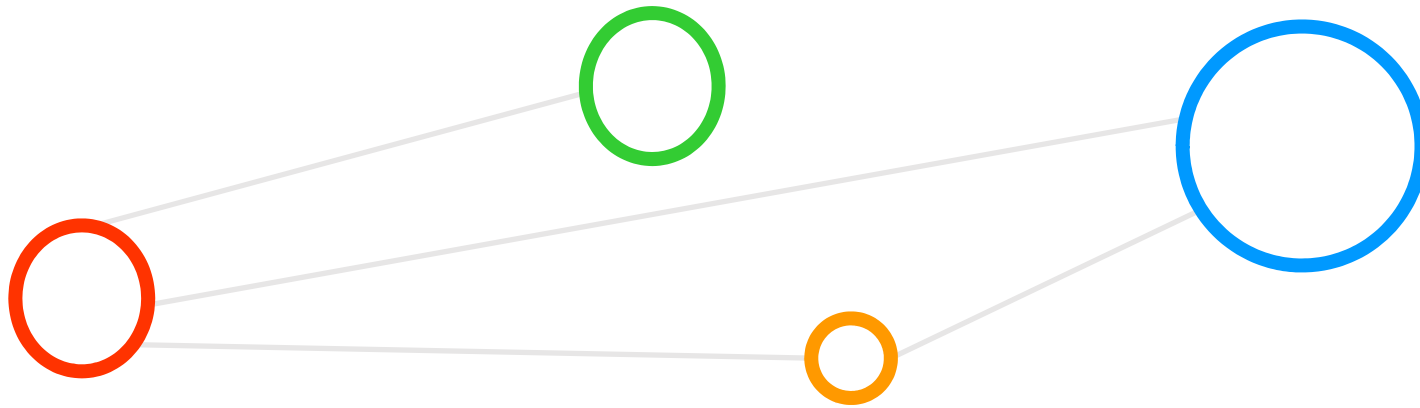
[1] M. Goetz and M. Riedel et al,
Proceedings IEEE Supercomputing Conference, 2015

[Video] Raytracing Application with OpenMP



[6] Speeding up a Ray tracer with OpenMP, YouTube Video

Lecture Bibliography



Lecture Bibliography (1)

- [1] M. Goetz, C. Bodenstein, M. Riedel, 'HPDBSCAN – Highly Parallel DBSCAN', in proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC2015), Machine Learning in HPC Environments (MLHPC) Workshop, 2015, Online: https://www.researchgate.net/publication/301463871_HPDBSCAN_highly_parallel_DBSCAN
- [2] Introduction to High Performance Computing for Scientists and Engineers, Georg Hager & Gerhard Wellein, Chapman & Hall/CRC Computational Science, ISBN 143981192X
- [3] The OpenMP API specification for parallel programming, Online: <http://openmp.org/wp/openmp-specifications/>
- [4] Lattice Boltzmann – Flow past an obstacle, Online: <https://www.youtube.com/watch?v=fspGcBpxguo>
- [5] Wikipedia on 'stencil code', Online: http://en.wikipedia.org/wiki/Stencil_code
- [6] Speeding up a Ray tracer with OpenMP, YouTube Video, Online: http://www.youtube.com/watch?v=S9Z5MeQS_LU
- [7] LLNL OpenMP Tutorial, Online: <https://computing.llnl.gov/tutorials/openMP/>
- [8] M. Goetz HPDBSCAN Bitbucket Repository, Online: <https://bitbucket.org/markus.goetz/hpdbscan/src/default/>
- [9] K. Hwang, G. C. Fox, J. J. Dongarra, 'Distributed and Cloud Computing', Book, Online: http://store.elsevier.com/product.jsp?locale=en_EU&isbn=9780128002049
- [10] DEEP Projects Web page, Online: <http://www.deep-projects.eu/>

Lecture Bibliography (2)

- [11] OmpSs, Online:
<https://pm.bsc.es/ompss-2>
- [12] OmpSs BSC Programming Models, Online:
<https://www.bsc.es/research-development/research-areas/programming-models/the-ompss-programming-model>
- [13] MontBlanc OmpSs Multi-Dependencies Extension Approved in OpenMP, Online:
<https://www.montblanc-project.eu/press-corner/news/ompss-multidependences-extension-approved-openmp-technical-report-6-tr6>
- [14] PyCOMPSs, Online:
https://hbp-hpc-platform.fz-juelich.de/?hbp_software=pycompss

