High Performance
Computing &
Big Data Services

hpc.uni.lu

hpc@uni.lu

@ULHPC

LUXEMBOURG
LET'S MAKE IT HAPPEN

Uni.lu HPC School 2020

PS5: Scalable Science: Parallel computations with OpenMP/MPI

Uni.lu High Performance Computing (HPC) Team

Dr. E. Krishnasamy

University of Luxembourg (UL), Luxembourg

http://hpc.uni.lu

Latest versions available on Github:

UL HPC tutorials: https://github.com/ULHPC/tutorials

UL HPC School: http://hpc.uni.lu/hpc-school/

PS5 tutorial sources: ulhpc-tutorials.rtfd.io/en/latest/parallel/basics/

Main objectives of this session

Part 1

- *OpenMP-Shared Memory Programming*
  - ↪ Introduction to OpenMP progamming
  - ↪ How to use directives and clauses in OpenMP
  - ↪ Performance analysis and tuning

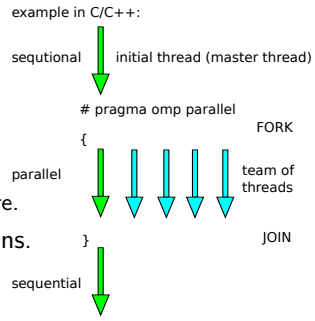
Part 2

- *MPI-Distributed Memory Programming*
  - ↪ Introduction to MPI programming
  - ↪ An overview of MPI global operations
  - ↪ Performance analysis and tuning

Summary

Parallel region

- **Fork-Join** model of parallel execution.
- **FORK: master** thread creates a **team** of parallel threads (master thread is also a part of team of the threads).
  - ↪ in C: **#pragma omp parallel** will create a threads
  - ↪ default thread number is defined by the given architecture.
- **JOIN:** team threads complete statements in parallel regions.
  - ↪ then they synchronize & terminate, leaving only the master thread to continue.



Parallel region example

```
int main()
{
  // printing out from master thread
  cout << "Hello world from master thread"<<endl;

  // creating the parallel region (team of threads)
  #pragma omp parallel num_threads(5)
  {
    cout << "Hello world from thread id"
    << omp_get_thread_num() << "from the team size of"
    << omp_get_num_threads()
    << endl;
  } // parallel region is closed
   cout << "end of the parallel region from master thread" << endl;
}
```

Checking the number of threads

- **Run-Time Library Routines** provides useful functions to query the threads.
- `omp_get_num_threads()`
  - ↪ returns number of threads within the team in the parallel region.
- `omp_get_thread_num()`
  - ↪ returns unique thread id number from the team of treads.
- `omp_in_parallel()`
  - ↪ returns **TRUE** if placed within the parallel region; otherwise returns **FALSE**.

Setting and controlling the number of threads

- **Environmental variable**: `export OMP_NUM_THREADS=<number of threads>`
- **Clause**: `num_threads(<number of threads>)`
  - ↪ for example, `#omp parallel num_threads(4)`
- `omp_set_num_threads(<number of threads>)`
  - ↪ to define number of threads in the parallel region, called in the serial region
- `int=omp_get_max_threads()`
  - ↪ returns the maximum number of threads
- `int=omp_get_num_procs()`
  - ↪ returns the number of processors that are available to the program

Summary

Shared and Private variables

- **private:** Each thread will have own copy to the private variable. And the private variable is only accessible within the parallel region not outside of the parallel region. By default, the loop iteration counters are considered as private.
  - ↪ this private variable does not have the originally initialized value.
- **shared:** All the threads have access to the shared variable, by default in the parallel region, all the variables are considered as a shared variable expect the loop iteration counter variables.
- Shared variables should be handled carefully; otherwise it causes **race conditions** in the program.
- Avoid writing a full list of variables in the code.

```
#pragma omp parallel default(shared) private(var_list)
#pragma omp parallel default(private) shared(var_list)
#pragma omp parallel private(var_list) shared(var_list)
```

Example: Shared and Private variables

```
#include<iostream>
#include<omp.h>
using namespace std;
int main()
{
  string i = "x", j = "y";
  int k = 3;
#pragma omp parallel private(i,k) shared(j) num_threads(2)
  {
    i += "a"; j += "b"; k += 7;
    cout << "i becomes " << i << "j becomes "
         << j << " and k becomes " << k << endl;
  }
    cout << "unchanged k value is " << k
         << "unchanged i value is " << i
         << "changed j is " << j << endl;
  return 0;
}
```

Firstprivate and Lastprivate

- **firstprivate:** is similar to private clause but each thread will have an initialized copy of the variables passed as fristprivate.
- **lastprivate:** is also similar to private clause but each thread will have an uninitialized copy of the variables passed as lastprivate.
  - ↪ at the end of the parallel loop or sections, the final variable value will be the last thread accessed value in section or in parallel loop.

```
int var = 5;
#pragma omp parallel for lastprivate(var) num_threads(4)
for(int i = 0; i < 4; i++)
{
var = i;
cout << "var value in parallel loop " << var << endl;
}
cout << "var value after parallel loop " << var << endl;
```

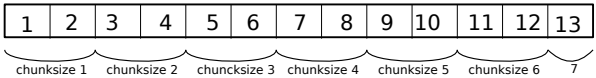
For loop parallelism

- **For**: for directive will execute for loop in a parallel loop.
- It should be placed within the parallel region `#pragma omp parallel`
  - ↪ or just as `#pragma omp parallel for`.

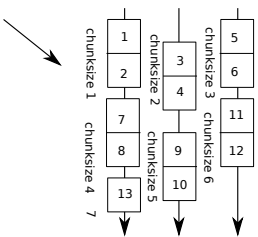
```
#pragma omp for
for (int i = 0; i < n; i++){
...(for loop body)
}
```

- By default loop counter index becomes private.
- **Loop-scheduling**: has the following clauses:
  - ↪ static
  - ↪ dynamic
  - ↪ guided
  - ↪ auto
  - ↪ runtime

Static (scheduling)



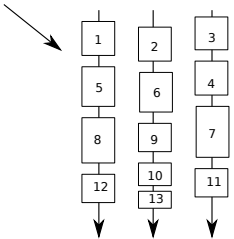
- The number of iterations is divided by chunksize.
- If the chunksize is not provided, the number of iterations will be divided by the size of a team of threads.
  - ↪ for example, n=64, num_threads=4; each thread will execute the 16 iterations in parallel.
- This is useful when the computational cost is similar to each iteration.



Dynamic (scheduling)

1	2	3	4	5	6	7	8	9	10	11	12	13

- The number of iterations is divided by chunksize.
- If the chunksize is not provided, it will consider the default value as 1.
- It will request for chunk of data until there are no more chunk of data available.
- There is no pattern for how different thread access the different chunk of data.
- This is useful when the computational cost is different in the iteration. This will allow quickly to place the chunk of data in the queue.



Guided, Auto and Runtime (scheduling)

- Similar to dynamic scheduling, that is a number of iteration divided chunksize.
- But the chunk of the data size is decreasing, which is proportional to the number of unsigned iterations divided by the number of threads.
- If the chunksize is not provided, it will consider the default value as 1.
- This is useful when there is poor load balancing at the end of the iteration.
- **Auto:** Compiler chooses the optimized chunksize for a number of iterations.
- **Runtime:** Environmental variable `OMP_SCHEDULE` is used to define parallel for loop scheduling.

Collapse

- Useful for nested loops.
- A total number of iterations will be partitioned into the available number of threads.
- Example: if the outer loop index is equal to the threads then each thread will execute 10 iterations in the innermost loop.

```
#pragma omp parallel for collapse(2) num_threads(10)
 for(int i=0; i<10; i++)
   for(int j=0; j<100; j++)
   {
    calculation(i,j);
   }
```

Reduction

- Useful for increment or summation of array into a shared nemerical variable.
- `reduction(operators: variable)`
  - ↪ arithmetic reductions: `+,*,-,max,min`
  - ↪ logical operator reductions in C: `& && | || ^`

```
int sum=0;
#pragma omp parallel for reduction(+:sum)
for(int i=0; i<n; i++)
sum += array[i];
```

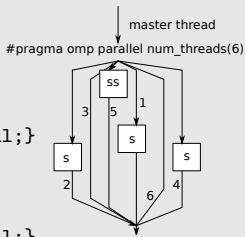
Section, Sections and Single

- Different task can be executed in parallel.
- `sections` can be used when more task is required.
- `single` is an alternative just for a single task.

```
//#pragma omp parallel sections num_threads(6)
#pragma omp parallel num_threads(6)
  {
//#pragma omp section
#pragma omp single
{
  cout <<"Thread id in single " << omp_get_thread_num()
  << " from team of threads "<< omp_get_num_threads() <<endl;
}
cout <<"Parallel region thread id"
<< omp_get_thread_num() << " from team of threads "
<< omp_get_num_threads() <<endl; }
```

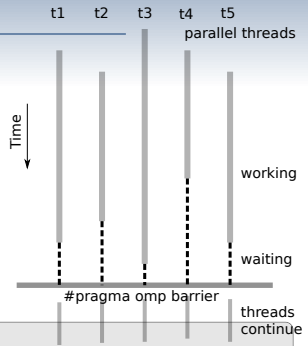
Section, Sections and Single

```
#pragma omp parallel num_threads(6)
 {
#pragma omp sections
   {
      cout <<"Thread id " << omp_get_thread_num()
      << " from team of threads "<< omp_get_num_threads() <<endl;
#pragma omp section
        {
        cout <<"Thread id " << omp_get_thread_num()
        << " from team of threads "<< omp_get_num_threads() <<endl;}
#pragma omp section
        {
        cout <<"Thread id " << omp_get_thread_num()
        << " from team of threads "<< omp_get_num_threads() <<endl;}
#pragma omp section
        {
        cout <<"Thread id " << omp_get_thread_num()
        << " from team of threads "<< omp_get_num_threads() <<endl;}}}
```



Barrier

- `#pragma omp barrier` will synchronize all the threads(waiting for other threads to finish their work) before it continues.
- Synchronization is useful in some parts in the code depends on the problem.




```
#pragma omp parallel
 {
  int tid = omp_get_thread_num();
  x[tid] = my_task(tid);
#pragma omp barrier
  y[tid] = x[tid] + tid*2;
 }
```

Critical, Atomic and Nowait

- **critical** will allow only one thread at once (one-by-one) within the critical section.
  - ↪ multiple threads can not access the critical section in parallel.
  - ↪ this will avoid the race condition.

```
int global_sum = 0;
#pragma omp parallel num_threads(5)
 {
#pragma omp for
for(int i = 0; i < n; i++){int local_sum = compute(i);
#pragma omp critical
if(global_sum < local_sum){global_sum = local_sum; }}}
```

- **atomic** is similar to `critical` but has a performance advantage.
  - ↪ just a single statement after followed by atomic declaration.
  - ↪ safely update the shared numeric variable and supports the memory location update.
- **nowait** will disable the implicit barrier from the worksharing constructs(workshare, single and sections).

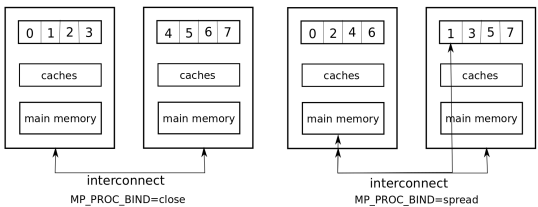
Example for atomic and nowait

```
int sum = 0;
#pragma omp parallel num_threads(5)
 {
  for(int i = 0; i < 10; i++){
    #pragma omp atomic
     sum++;}
 }
```

```
#pragma omp parallel num_threads(5)
 {
    cout << " beginning of parallel region " <<endl;
#pragma omp for nowait //nowait
    for (int i = 0; i < 10; ++i) {
      cout << " no wait " <<endl;}
    cout << " end of parallel region " <<endl;
 }
```

Environmetal varibles

- For better memory access.
- Allows you to control over the processes or threads.
- GNU:`OMP_PLACES=<sockets, cores, threads>` and `OMP_PROC_BIND=<close, spread, master>`.
- Intel: `KMP_AFFINITY=<compact, disabled, explicit, none, scatter>`.



Task

- Task region is executed by any thread in the team.
- Tasks can be explicitly synchronized by the `barrier` or `taskwait`.
- Alternative to `sections` and only supportive after **OpenMP 3.0+**.

```
#pragma omp parallel num_threads(10)
 {
#pragma omp single
   {
#pragma omp task
    cout << "Task 1 executed by thread id " << omp_get_thread_num() <<endl;
#pragma omp task
    printf("Task 2 executed by thread id " << omp_get_thread_num() <<endl;
#pragma omp taskwait //wait for other task to finish their job
   }
 }
```

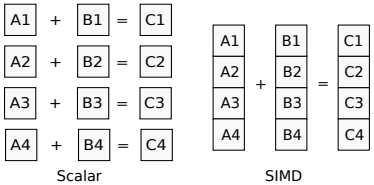
Task dependencies

- Here `task 1` and `task 2` will be executed in sequential.
- But `task 2` and `task 3` can be executed in parallel.

```
#pragma omp parallel num_threads(5)
{
#pragma omp single
{
#pragma omp task depend(out:a)
{
a = 2;cout << "task 1" <<endl;}
#pragma omp task depend(in:a) depend(in:b) // or  #pragma omp task depend(in:a,b)
{
b = 3; a = 1;cout <<"task 2" <<endl;}
#pragma omp task depend(out:c)
{
c = 4;cout <<"task 3" <<endl;}
}
}
```

Single Instruction Multiple Data

- According to Flynn's taxonomy computer architecture can be classified into four.
  - ↪ one of them is Single Instruction Multiple Data (SIMD).
  - ↪ same instruction is executed on the multiple data.
- **OpenMP 4.x** directives support SIMD code generation.



Example: simd and collapse

- To vectorize the loop

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

- For nested loop

```
#pragma omp simd collapse(2)
for (int i=0; i<10; ++i){
for (int j=0; j<10; ++j){
    c[i][j] = a[i][j] / b[i][j];
    }
}
```

Example: reduction and declare

- reduction

```
int total=0;
#pragma omp simd reduction(+:total)
   for (int i=0; i<size; ++i){
      total += a[i] + b[i];}
```

- To create simd version of function to be used in simd.

```
#pragma omp declare simd
double square (double x){
   return x * x;}
#pragma omp simd
   for (int i=0; i<n; ++i){
   c[i] = square(a[i];}
```

Summary

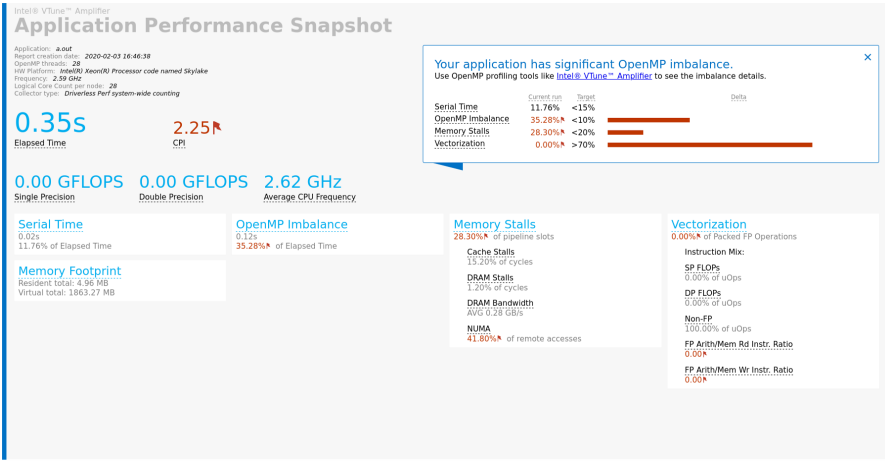
Intel Application Performance Snapshot (APS)

- ASP can be used to see the CPU utilisation, memory access efficiency and vectorisation.
  - ↪ `$ aps -help` will list out profiling metrics options in APS.

```
$ module purge                              # remove the previous modules if there are any
$ module load swenv/default-env/latest      # load the software environment
$ module load tools/VTune/2019_update4      # load the Intel APS
$ module load toolchain/intel/2019a         # load the intel compiler
# Set the max. number of threads from -c option passed to slurm (Ex: 28 on iris)
$ export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASKS:-1}
$ icc -qopenmp example.c                    # compile the code

# execute and profile the code
$ aps --collection-mode=all -r report ./a.out
$ aps-report -g report                      # create the .html file
$ firefox report_<postfix>.html             # openmp the .html file in the firefox
$ aps-report report_output                  # command line output (alternative option)
```

Intel APS GUI



Intel Inspector

- Intel Inspector detects and locates the memory, deadlocks, and data races in the code
- For example, memory access and memory leaks can be found.

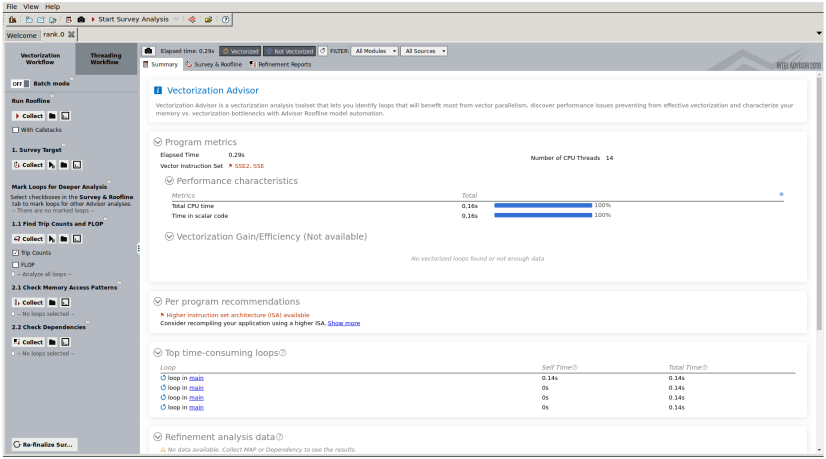
```
$ module purge                                # remove the previous modules if there are any
$ module load swenv/default-env/latest        # load the software environment
$ module load toolchain/intel/2019a           # load the intel compiler
$ module load tools/Inspector/2019_update4 # load the intel inspector
# Set the max. number of threads from -c option passed to slurm (Ex: 28 on iris)
$ export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASKS:-1}
$ icc -qopenmp example.c                      # compile the code
# execute and profile the code
$ inspxe-cl -collect mi1 -result-dir mi1 -- ./a.out
$ cat inspxe-cl.txt                           # open the file to see if there is any memory leak
=== Start: [2020/12/12 01:19:59] ===
0 new problem(s) found
=== End: [2020/12/12 01:20:25] ===
```

Intel Advisor

- Intel Advisor: set of collection tools for the metrics and traces that can be used for further tunning in the code.
  - ↪ `survey`: analyse and explore an idea about where to add efficient vectorisation.

```
$ module purge                              # remove the previous modules if there are any
$ module load swenv/default-env/latest      # load the software environment
$ module load toolchain/intel/2019a         # load the intel compiler
$ module load perf/Advisor/2019_update4 # load the intel advisor
# Set the max. number of threads from -c option passed to slurm (Ex: 28 on iris)
$ export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASKS:-1}
$ icc -qopenmp example.c                  # compile the code
# collect the survey metrics
$ advixe-cl -collect survey -project-dir result -- ./a.out
# collect the report
$ advixe-cl -report survey -project-dir result
# open the gui for report visualization
$ advixe-gui
```

Intel Advisor: advixe-gui

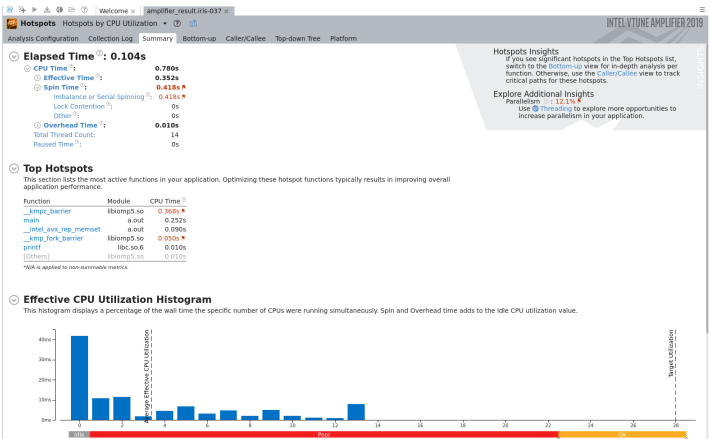


Intel VTune Amplifier

- Identifying the time consuming part in the code.
- And also the identify the cache misses and latency.

```
$ module purge                                  # remove the previous modules if there are any
$ module load swenv/default-env/latest          # load the software environment
$ module load toolchain/intel/2019a             # load the intel compiler
$ module load tools/VTune/2019_update4          # load the VTune amplifier
$ module load vis/GTK+/3.24.8-GCCcore-8.2.0# load the GUI dependency for the \
                                                # VTune amplifier
# Set the max. number of threads from -c option passed to slurm (Ex: 28 on iris)
$ export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASKS:-1}
$ icc -qopenmp example.c                        # compile the code
# execute the code and collect the hotspots
$ amplxe-cl -collect hotspots -r amplifier_result ./a.out
$ amplxe-gui                                    # open the GUI of VTune amplifier
```

Intel VTune Amplifier: amplxe-gui

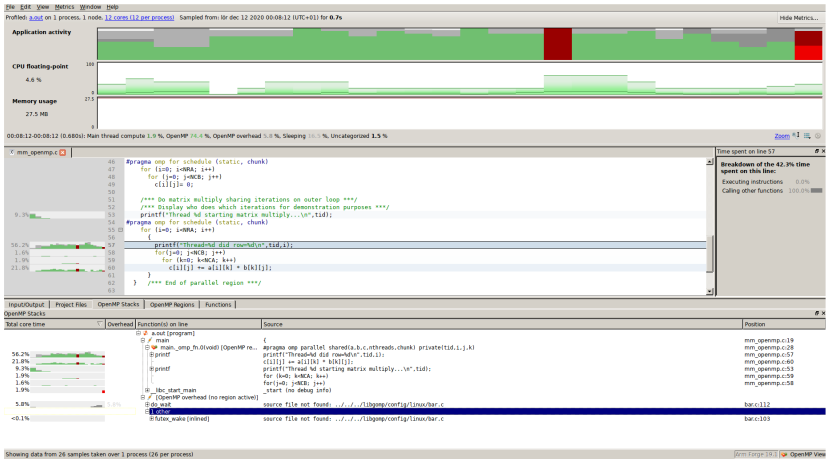


ARM Forge

- Arm forge will hlep to debugging, profiling, and analyzing.
- It also supports the MPI, UPC, CUDA, and OpenMP programming models for a different architecture with different variety of compilers.
- Both DDT and MAP are supporting the GUI interface.

```
$ module purge                          # remove the previous modules if there are any
$ module load swenv/default-env/latest  # load the software environment
$ module load tools/ArmForge/19.1       # load the ArmForge tool (profiling and debugging)
$ module load toolchain/foss/2019a      # gcc compiler
# Set the max. number of threads from -c option passed to slurm (Ex: 28 on iris)
$ export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASKS:-1}
$ gcc example.c -g -fopenmp             # compilation with debugging tool
$ map --profile --no-mpi ./a.out        # execute and profile the code
$ map xyz.map                           # open the profiled result in GUI
```

ARM Forge: map GUI



Practical Session on OpenMP

Your Turn!

Hands-on Pre-requisites

▶ url ◀ | github | src

- Access to ULHPC facility `ssh`
- Clone/Pull `ULHPC/tutorials` repository `~/git/github.com/ULHPC/tutorials`
- Prepare dedicated directory `~/tutorials/OpenMP-MPI` for this session

```
(access)$> mkdir -p ~/tutorials/OpenMP-MPI
(access)$> cd ~/tutorials/OpenMP-MPI
# create a symbolic link to the reference material
(access)$> ln -s ~/git/github.com/ULHPC/tutorials/parallel/basics ref.d
```

Hands-on: Parallel OpenMP jobs

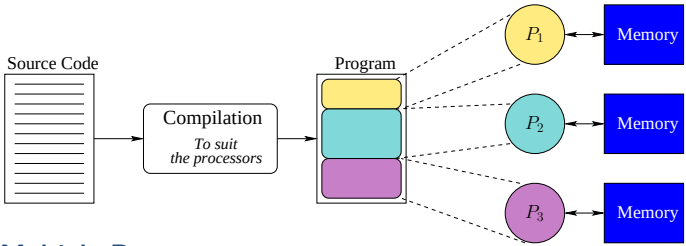
Hands-on: Work with OpenMP examples and Profiling tools ▶ url ◀ | github | src

- **Reserve** an **interactive** job to launch 4 OpenMP threads
- Check and **compile** `src/hello_openmp.c`
  - ↪ against both toolchains `bin/[intel_]hello_openmp`
- **execute** the generated binaries
  - ↪ set `$OMP_NUM_THREADS`
- prepare a **launcher script** `runs/launcher.OpenMP.sh`
- repeat on a more serious program `src/matrix_mult_openmp.c`
  - ↪ `bin/[intel_]matrix_mult_openmp`

- **Note**: More exercise can be referred **here**

Summary

SPMD Programming model

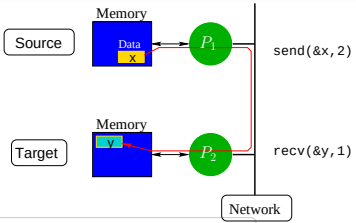


- SPMD: **Simple Program, Multiple Data**
  - ↪ same programs for each processors
  - ↪ executed at independent points
  - ↪ processes identified by a rank
  - ↪ each process knows the piece of code he works on
  - ↪ common in master-worker computations

```
if (my_rank == 0) { /* master */
   //... load input and dispatch ...
} else { /* workers */
   //... wait for data and compute ...
}
```

MPI (Message Passing Interface)

- **Message Passing Model**:
- each "processor" runs a process
- processes communicate by exchanging messages
  - ↪ *analogy*: **mail**



Message Passing Interface (MPI) Standard

- **Goal**:
  - ↪ **portable**, **efficient** & **flexible** standard for message passing
  - ↪ industry standard
- Reference website `https://www.mpi-forum.org/`
- **Latest version: 3.1** (June 2015) – specifications

Important environment management routines

- `MPI_Init( int *argc, char ***argv )`
- `MPI_Comm_size( MPI_Comm comm, int *size )`
- `MPI_Comm_rank( MPI_Comm comm, int *rank )`
- `MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
- `MPI_Finalize( void )`

MPI status

- `MPI_Status` represents the recieving status from the `MPI_Recv`
  - ↪ status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR
- Example: process 0 sends the `int buffer` with `tag`, this will be recieved by process 1.

```
int buffer_sent = 12345;
int tag = 67890;
MPI process 0 sends value 12345 with tag 67890.
MPI process 1 received value 12345 from rank 0, with tag 67890 and error code 1.
```

Data types in MPI

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Summary

MPI communications

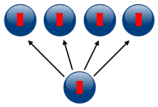
- Two types of communications
  - ↪ **blocking:** the process will not return to the return call until the sending data have been copied. That is process sending will wait until the process receiving the information.
  - ↪ **non-blocking:** these functions return quickly even before the message have been copied.
- MPI communication mode
  - ↪ **standard mode:** it can be either synchronous or buffered.
  - ↪ **synchronous mode:** send returns only receive has been started.
  - ↪ **buffered mode:** send returns as soon as the message has been put into the buffer or received.

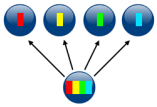
Point-to-point communication

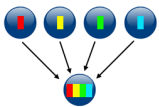
- `MPI_Send` and `MPI_Recv` are called Point-to-Point (P2P) communication from MPI routines.
  - ↪ involves sending the message from one specific process to and receives the message from another specific process.
- P2P for blocking:
  - ↪ standard mode: `MPI_Send` and `MPI_Recv`
  - ↪ synchronous mode: `MPI_Ssend` and `MPI_Recv`
  - ↪ buffered mode: `MPI_Bsend` and `MPI_Recv`
- P2P for non-blocking:
  - ↪ standard mode: `MPI_ISend` and `MPI_Irecv`
  - ↪ synchronous mode: `MPI_Issend` and `MPI_Irecv`
  - ↪ buffered mode: `MPI_Ibsend` and `MPI_Irecv`
- Non-blocking calls require `MPI_wait` to synchronous routine.

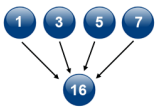
MPI collective routines

- Broadcast: single processor sends (same information) scalar or vector value to all processor
- Reduce: many processes return computation result to the root process.
- Allreduce: MPI process can do the reduce (reduction) computation, and this computed value will be available to another MPI process too.
- Scatter: it is similar to `MPI_Bcast`, but `MPI_Scatter` send the chunk of data (array) to many MPI process.
- Gather: it is the inverse of `MPI_Scatter`, that is gathering all the information from the process to root process.









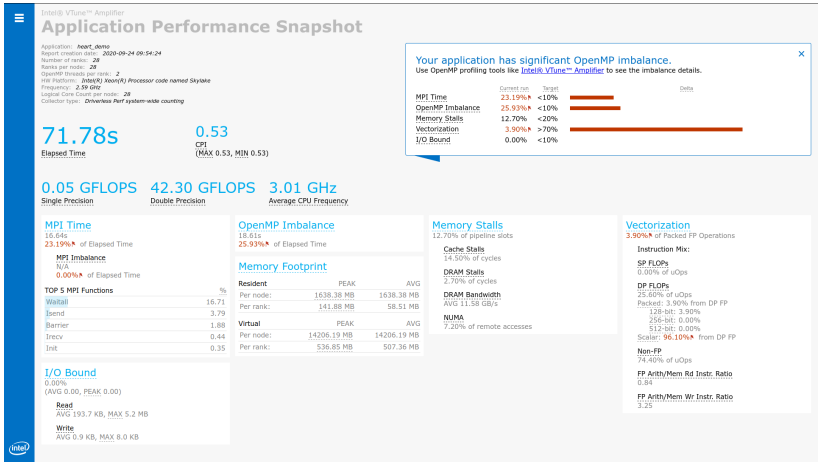
Summary

Intel Application Performance Snapshot (APS)

- ASP works large MPI applications workloads & can help to analyse scalability issues.
- APS MPI analysis: details on analysis charts. Ex: data transfer per rank and node-to-node-data transfer.
- Filtering capabilities is quite useful. Ex:volume of data and lines.

```
$ module purge                              # remove the previous modules if there are any
$ module load swenv/default-env/latest      # load the software environment
$ module load tools/VTune/2019_update4      # load the Intel APS
$ module load toolchain/intel/2019a         # load the intel compiler
$ mpiicc example.c                          # compile the code
# execute and profile the code
$ srun -n 32 aps --collection-mode=mpi aps-report -g report
$ aps-report -g report                      # create the .html file
$ firefox report_<postfix>.html             # openmp the .html file in the firefox
$ aps-report report                         # command line output (alternative option)
```

Intel APS GUI



Intel Inspector

- Intel Inspector detects and locates the memory, deadlocks, and data races in the code
  - ↪ For example, memory access and memory leaks can be found.

```
$ module purge                                  # remove the previous modules if there are any
$ module load swenv/default-env/latest          # load the software environment
$ module load toolchain/intel/2019a             # load the intel compiler
$ module load tools/Inspector/2019_update4 # load the intel inspector
$ mpiicc example.c                              # compile the code

# in the batch job
srun -n 14 inspxe-cl -collect=ti2 -r result ./a.out
$ cat inspxe-cl.txt                             # open the file to see if there is any memory leak
=== Start: [2020/12/12 01:19:59] ===
0 new problem(s) found
=== End: [2020/12/12 01:20:25] ===
```

Intel Advisor

- Intel Advisor: set of collection tools for the metrics and traces that can be used for fine tunning
  - ↪ `survey`: analyse & explore an idea about where to add efficient vectorisation.

```
$ module purge                              # remove the previous modules if there are any
$ module load swenv/default-env/latest      # load the software environment
$ module load toolchain/intel/2019a         # load the intel compiler
$ module load perf/Advisor/2019_update4     # load the intel advisor
$ mpiicc example.c                          # compile the code

# collect the survey metrics (in the batch job)
srun -n 14 advixe-cl --collect survey --project-dir result -- ./a.out
# open the gui for report visualization
$ advixe-gui result
```

Intel VTune Amplifier

- Identifying the time consuming part in the code.
- And also the identify the cache misses and latency.

```
$ module purge                                  # remove the previous modules if there are any
$ module load swenv/default-env/latest          # load the software environment
$ module load toolchain/intel/2019a
$ module load tools/VTune/2019_update4          # load the VTune amplifier
$ module load vis/GTK+/3.24.8-GCCcore-8.2.0# load the GUI dependency for the \
                                                # VTune amplifier
$ mpiicc example.c
srun -n 14 amplxe-cl -collect uarch-exploration -r result -- ./a.out
$ amplxe-gui result
```

Intel Trace Analyzer and Collector (ITAC)

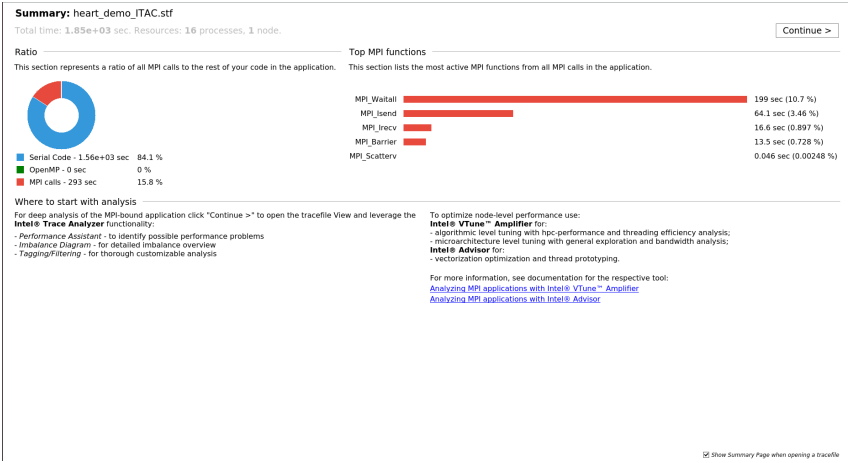
- ITAC provides the detailed analyses of the MPI application.
- Helps with improving the weak and strong scaling for smalland large applications.

```
$ module purge
$ module load toolchain/intel/2019a
$ module load tools/itac/2019.4.036
$ module load tools/VTune/2019_update4
$ module load vis/GTK+/3.24.8-GCCcore-8.2.0

$ mpiicc -trace example.c

srun -n 14 ./a.out
$ traceanalyzer a.out*.stf
```

ITAC


Summary: heart_demo_ITAC.stf
Total time: 1.85e+03 sec. Resources: 16 processes, 1 node.
Continue >
Ratio
This section represents a ratio of all MPI calls to the rest of your code in the application.
Serial Code - 1.56e+03 sec 84.1 %
OpenMP - 0 sec 0 %
MPI calls - 293 sec 15.8 %
Top MPI functions
This section lists the most active MPI functions from all MPI calls in the application.
MPI_Waitall 199 sec (10.7 %)
MPI_Isend 64.1 sec (3.46 %)
MPI_Irecv 16.6 sec (0.897 %)
MPI_Barrier 13.5 sec (0.728 %)
MPI_Scatterv 0.046 sec (0.00248 %)
Where to start with analysis
For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the Intel® Trace Analyzer functionality:
- Performance Assistant - to identify possible performance problems
- Imbalance Diagram - for detailed imbalance overview
- Tagging/Filtering - for thorough customizable analysis
To optimize node-level performance use:
Intel® VTune™ Amplifier for:
- algorithmic level tuning with hpc-performance and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;
Intel® Advisor for:
- vectorization optimization and thread prototyping.
For more information, see documentation for the respective tool:
Analyzing MPI applications with Intel® VTune™ Amplifier
Analyzing MPI applications with Intel® Advisor
Show Summary Page when opening a tracefile

ARM Forge

- Arm forge will hlep to debugging, profiling, and analyzing.
- It also supports the MPI, UPC, CUDA, and OpenMP programming models for a different architecture with different variety of compilers.
- Both DDT and MAP are supporting the GUI interface.

```
$ module purge                          # remove the previous modules if there are any
$ module load swenv/default-env/latest  # load the software environment
$ module load tools/ArmForge/19.1       # load the ArmForge tool (profiling and debugging)
$ module load toolchain/foss/2019a      # gcc compiler

$ mpicc example.c -g -fopenmp           # compilation with debugging tool

unset I_MPI_PMI_LIBRARY
map --profile mpirun -np 14 ./a.out
```

Hands-on: MPI jobs

Your Turn!

Hands-on: MPI

▶ url ◀ | github | src

- **Reserve** an **interactive** job to launch 6 MPI processes
  - ↪ across two nodes (2x3), for 30 minutes
- Check and **compile** `src/hello_mpi.c`
  - ↪ `bin/{openmpi,intel}_hello_mpi`
- **execute** the generated binaries
- prepare a **launcher script** `runs/launcher.MPI.sh`
- repeat on a more serious program `src/matrix_mult_mpi.c`
  - ↪ `bin/{openmpi,intel}_matrix_mult_mpi`

Hands-on: Source Code Examples & Profiling

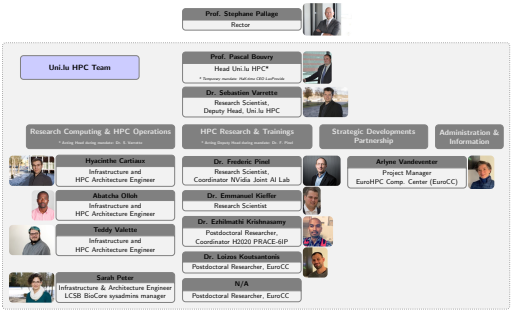
Your Turn!

Hands-on

- Work with examples too for understanding of basic MPI and communications
  ↪ see `parallel/mpi/examples/{src,scripts}/`
- Work with the profiling tools to understand the performance measuring and the tuning ideas.
  ↪ see `parallel/mpi/profiling/`

Questions?

High Performance Computing @ Uni.lu



1. Parallel region
2. Directives and clauses in OpenMP
3. Performance analysis and tuning
4. MPI: Message Passing Interface
5. MPI global communication
6. Performance analysis and tuning

University of Luxembourg, Belval Campus
Maison du Nombre, 4th floor
2, avenue de l'Université
L-4365 Esch-sur-Alzette
mail: hpc@uni.lu

https://hpc.uni.lu