OmpCluster

The OmpCluster Team

Mar 01, 2023

CONTENTS:

CHAPTER

ONE

PROGRAMMING APPLICATIONS

This page presents how to program OpenMP *tasks* based applications for distributed systems using the OmpCluster runtime.

For more details about the OpenMP directives, you can consult the specification of OpenMP.

OmpCluster relies on the application programming interface defined by OpenMP. It uses directives to program remote processes running on computer clusters with distributed memory architectures.

Between the Device Constructs that we use in this project, we have:

1.1 Execute code on a target device

```
omp target [clause[[,] clause],...] structured-block
omp declare target  [function-definitions-or-declarations]
```

1.2 Manage the device data environment

This construction allows you to transfer data between the host (the head process) and the devices (the worker nodes), where the target regions will be executed.

```
map ([map-type:] list) map-type := alloc | tofrom | to | from | release | delete
```

If map-type is **to** or **tofrom**, this new item is initialized with the value of the original list item in list in the host data environment.

```
#pragma omp target            \
  map(to:...)                 \
  map(tofrom:...)
{
  ...
}
```

If map-type is **from** or **alloc**, the initial value of the list item in the device data environment is undefined.

```
#pragma omp target             \
  map(from:...)                \
  map(alloc:...)
{
  ...
}
```

1.3 Asynchronous target task

`nowait` clause eliminates the implicit barrier so the parent task can make progress even if the target task is not yet completed. By default, an implicit barrier exists at the end of the target construct, which ensures the parent task cannot continue until the target task is completed.

```
#pragma omp target nowait
{
  ...
}
```

1.4 Task dependencies

`depend(dependence-type:list)` establishes scheduling dependencies between the target task and sibling tasks that share list items. The dependence-type can be **in**, **out**, or **inout**.

If dependence-type is **in** or **inout**, a scheduling dependence for the target task on the sibling task is created. Where the task that we are creating depends that the data inserted in the clause **in** or **inout** is ready.

```
#pragma omp target nowait          \
  depend(in:...)                   \
  depend(inout:...)
{
  ...
}
```

If dependence-type is *out* or *inout*, a scheduling dependence for the target task on the sibling task is created. Where the task we are creating will generate the outputs described in the **out** and **inout** clause.

```
#pragma omp target nowait          \
  depend(out:...)                  \
  depend(inout:...)
{
  ...
}
```

1.5 Data environment

`firstprivate(list)` declares the data variables in *list* to be private to the target task and shared by every thread team that runs the region. A new item is created for each list item that is referenced by the target task. Each new data variable is initialized with the value of the original variable at the time the target construct is encountered.

```
#pragma omp target nowait          \
  firstprivate(list)
{
  ...
}
```

1.6 Asynchronous target data task

Target data tasks are basically tasks dedicated to communication between the head process and the worker processes.

Those type of tasks allows the programmer to describe data to be send to the worker processes with a larger life scope. Indeed, the data will stay alive in the worker processes between the `enter` and the `exit` directive. By using those, the OMPC runtime will be able to optimize the communication within the worker processes.

Here is an example:

It is important to point out that every target tasks which is going to use a data sent by a target data task must use that first position of the array as a dependency. This differs from the OpenMP standard but is compulsory for the OMPC runtime to be able to keep track of the data used by the target tasks and manage the communication between worker processes correctly.

If the dependencies are not set correctly (pointing to the first position of the value) the execution of the program will most probably fail on a segfault.

CHAPTER

TWO

BASIC USAGE

2.1 Compile and run programs

Compiling OpenMP code for OmpCluster requires to use a specific OpenMP target `x86_64-pc-linux-gnu` which indicates to the compiler to compile the OpenMP target code region for a device. For example, the *mat-mul* example can be compiled using the following command:

```
clang -fopenmp -fopenmp-targets=x86_64-pc-linux-gnu mat-mul.cpp -o mat-mul
```

Then, you can run the newly created program but, contrary to classical OpenMP programs, programs need to be executed using `mpirun` or `mpiexec` tools to use the OmpCluster distributed task runtime just as any other MPI program:

```
mpirun -np 3 ./mat-mul
```

In this case, the runtime will automatically create 3 MPI processes (one head and two workers): the head process will offload OpenMP target regions to be executed on the workers following the currently implemented scheduling strategy.

The runtime also supports the offloading to remote MPI processes (located on other computers or containers), those can be configured using the `-host` or `-hostfile` flags of `mpirun` (note that flags may differ between MPI implementations). However, just as any MPI programs, the user needs to copy the binary on all computers/containers before executing it (using `pdcp` command or NFS directory).

As you might have noticed, it is somehow hard to follow what the OmpCluster runtime is doing when executing the application. You can enable information messages from the runtime by setting the following command when running the program:

```
LIBOMPTARGET_INFO=-1 mpirun -np 3 ./mat-mul
```

2.2 Container

To easily experiment with the OmpCluster tools, we provide a set of docker images containing a pre-compiled Clang/LLVM with all necessary OpenMP and MPI libraries to compile and run any OmpCluster programs.

All our images are based on Ubuntu 20.04. However, different configurations are available with different versions of CUDA, and MPICH or OpenMPI. Choose the docker image tag according to your favorite configuration or `latest` to use the default configuration.

You can execute your applications with the OmpCluster runtime on any computer using docker and the following command:

```
docker run -v /path/to/my_program/:/root/my_program -it ompcluster/runtime:latest /
↪bin/bash
cd /root/my_program/
```

This flags `-v` is used to share a folder between the operating system of the host and the container. You can get more information on how to use Docker in the official Get Started guide).

You can also use Singularity, using the following commands for example:

```
singularity pull docker://ompcluster/runtime:latest
singularity shell ./runtime_latest.sif
cd /path/to/my_program/
```

See here for more information about Singularity. Note that some cluster environments may have the new Singularity version that is called Apptainer.

2.2.1 Cluster job manager execution

The OmpCluster runtime can be used with a cluster job manager, like Slurm. After compiling your code using a container, you can launch the job as any MPI program. For example, using Slurm:

```
srun -N 3 --mpi=pmi2 singularity exec ./runtime_latest.sif ./my_program
```

Every job manager suport many configurations. For example, you can refere to the Slurm documentation.

2.2.2 Existing images and configurations

The container images that we provide follows this naming convention: `ompcluster/<image_name>:<tag>`.

Several images are available on our docker-hub repository, here is a tentative to list them:

- `hpcbase`: the base image for all other containers. It contains the MPI implementation, CUDA, Mellanox drivers, etc.
- `runtime`: this image contains the pre-built Clang and OmpCluster runtime based on the stable releases.
- `runtime-dev`: this image contains the pre-built Clang and OmpCluster runtime based on the Git repository. This version should be considered as unstable and should not be used in production.
- Application-specific images (`awave-dev`, `beso-dev`, `plasma-dev`, etc): Those images are based on the runtime image but contains additional libraries and tools required to develop some applications.

CHAPTER

THREE

EXAMPLES

3.1 First example

To better understand this model, we make an example with block-based matrix multiplication. Given two matrices, the program will multiply them and produce a new matrix.

If there are two input matrices A and B of size N and the output matrix C. The multiplication of matrices A and B can be performed using the following code:

```
void MatMul(int &A, int &B, int &C) {
  for (int i = 0; i < N; ++i){
    for (int j = 0; j < N ; ++j) {
      C[i][j] = 0;
      for (int k = 0; k < N ; ++k) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Fig. 1: matrix-multiplication-good

It is possible to use a block partitioned matrix product that involves only algebra on submatrices of the factors. The partitioning of the factors is not arbitrary, however, and requires conformable partitions between two matrices *A* and *B* such that all submatrix products that will be used are defined.

The following figure shows the block-based matrix multiplication.

In this case, the code would be:

```
void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
  // Go through all the blocks of the matrix.
  for (int i = 0; i < N / BS; ++i)
  {
    for (int j = 0; j < N / BS; ++j)
    {
      float *BlockC = C.GetBlock(i, j);
      for (int k = 0; k < N / BS; ++k) {
        float *BlockA = A.GetBlock(i, k);
        float *BlockB = B.GetBlock(k,j);
        // Go through the block.
        for(int ii = 0; ii < BS; ii++)
          for(int jj = 0; jj < BS; jj++) {
```

(continues on next page)

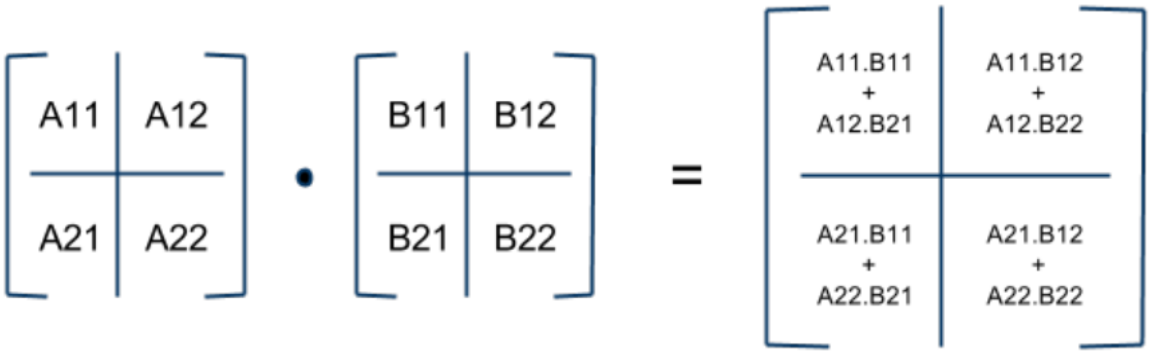


Fig. 2: block_multiplication

(continued from previous page)

```
            for(int kk = 0; kk < BS; ++kk)
              BlockC[ii + jj * BS] += BlockA[ii + kk * BS] * BlockB[kk + jj * BS];
          }
      }
    }
  }
}
```

In our example, the BlockMatrix class is only used as a utility wrapper to split the whole matrix into blocks so as to benefit from data locality (each block is contained by a different array).

We can parallelize the code by performing the multiplication of each pair of blocks in a different node using the following code:

```
void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
  #pragma omp parallel
  #pragma omp single
  for (int i = 0; i < N / BS; ++i)
    for (int j = 0; j < N / BS; ++j) {
      float *BlockC = C.GetBlock(i, j);
      for (int k = 0; k < N / BS; ++k) {
        float *BlockA = A.GetBlock(i, k);
        float *BlockB = B.GetBlock(k,j);
        #pragma omp target depend(in: *BlockA, *BlockB) \
                           depend(inout: *BlockC) \
                           map(to: BlockA[:BS*BS], BlockB[:BS*BS]) \
                           map(tofrom: BlockC[:BS*BS]) nowait
        for(int ii = 0; ii < BS; ii++)
          for(int jj = 0; jj < BS; jj++) {
            for(int kk = 0; kk < BS; ++kk)
              BlockC[ii + jj * BS] += BlockA[ii + kk * BS] * BlockB[kk + jj * BS];
          }
      }
    }
}
```

As we carry out the multiplication of each block in the node, we have to send the block of matrix A, the block of matrix B as input (`map(to: BlockA[:BS*BS], BlockB[:BS*BS])`), and the block of matrix C as output

and input (`map(tofrom: BlockC[:BS*BS])`). The multiplication process depends on input blocks A and B (`depend(in: BlockA[0], BlockB[0])`) and block C as output (`depend(inout: BlockC[0])`).

It is also possible to optimize the code even more by using a second level of parallelism within each node using the traditionnal `parallel for` directive as shown below:

```
void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
  #pragma omp parallel
  #pragma omp single
  for (int i = 0; i < N / BS; ++i)
    for (int j = 0; j < N / BS; ++j) {
      float *BlockC = C.GetBlock(i, j);
      for (int k = 0; k < N / BS; ++k) {
        float *BlockA = A.GetBlock(i, k);
        float *BlockB = B.GetBlock(k,j);
        #pragma omp target depend(in: *BlockA, *BlockB) \
                           depend(inout: *BlockC) \
                           map(to: BlockA[:BS*BS], BlockB[:BS*BS]) \
                           map(tofrom: BlockC[:BS*BS]) nowait
        #pragma omp parallel for
        for(int ii = 0; ii < BS; ii++)
          for(int jj = 0; jj < BS; jj++) {
            for(int kk = 0; kk < BS; ++kk)
              BlockC[ii + jj * BS] += BlockA[ii + kk * BS] * BlockB[kk + jj * BS];
          }
      }
    }
}
```

The problem with this implementation is that the 3 blocks must be transmitted between the head and worker processes for each target task without any possibility for the runtime to improve the communication between the nodes.

To fix it, the input blocks can be sent previously using target data tasks (`target enter data map(...) depend(...) nowait`) and the resulting blocks retrieved back using opposite target data tasks (`target exit data map(from: ...) depend(...) nowait`) at the end. In this case, the OMPC scheduler and data manager will be able to optimize the mapping of the target tasks on the worker processes and the communication between them. Here would be the resulting code:

```
void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
  #pragma omp parallel
  #pragma omp single
  {
    // Maps all matrices´ blocks asynchronously (as tasks).
    for (int i = 0; i < N / BS; ++i) {
      for (int j = 0; j < N / BS; ++j) {
        float *BlockA = A.GetBlock(i, j);
        #pragma omp target enter data map(to: BlockA[:BS*BS]) \
                                      depend(out: *BlockA) nowait
        float *BlockB = B.GetBlock(i, j);
        #pragma omp target enter data map(to: BlockB[:BS*BS]) \
                                      depend(out: *BlockB) nowait
        float *BlockC = C.GetBlock(i, j);
        #pragma omp target enter data map(to: BlockC[:BS*BS]) \
                                      depend(out: *BlockC) nowait
      }
    }

    for (int i = 0; i < N / BS; ++i)
```

(continues on next page)

(continued from previous page)

```
      for (int j = 0; j < N / BS; ++j) {
        float *BlockC = C.GetBlock(i, j);
        for (int k = 0; k < N / BS; ++k) {
          float *BlockA = A.GetBlock(i, k);
          float *BlockB = B.GetBlock(k,j);
          // Submits the multiplication for the ijk-block
          // Data is mapped implicitly and automatically moved by the runtime
          #pragma omp target depend(in: *BlockA, *BlockB) \
                             depend(inout: *BlockC)
          #pragma omp parallel for
          for(int ii = 0; ii < BS; ii++)
            for(int jj = 0; jj < BS; jj++) {
              for(int kk = 0; kk < BS; ++kk)
                BlockC[ii + jj * BS] += BlockA[ii + kk * BS] * BlockB[kk + jj * BS];
            }
        }
      }

      // Removes all matrices´ blocks and acquires the final result asynchronously.
      for (int i = 0; i < N / BS; ++i) {
        for (int j = 0; j < N / BS; ++j) {
          float *BlockA = A.GetBlock(i, j);
          #pragma omp target exit data map(release: BlockA[:BS*BS]) \
                                       depend(inout: *BlockA) nowait
          float *BlockB = B.GetBlock(i, j);
          #pragma omp target exit data map(release: BlockB[:BS*BS]) \
                                       depend(inout: *BlockB) nowait
          float *BlockC = C.GetBlock(i, j);
          #pragma omp target exit data map(from: BlockC[:BS*BS]) \
                                       depend(inout: *BlockC) nowait
        }
      }

  }
}
```

It is important to point out that every target tasks must use the first position of the block as a dependency. As said earlier, this is compulsory for the runtime to be able to keep track of the data used manage the communication between worker processes correctly.

3.2 More examples

More examples are available here: OMPC examples.

Currently, the following examples are available:

- hello-world
- fibonacci
- reduce-sum
- matmul
- cholesky

CHAPTER

FOUR

PROFILING

This document provides a cookbook on how to collect, process and analyze OMPC traces.

4.1 Collecting a trace

The OmpCluster runtime has built-in support for collecting execution traces in the JSON format. To enable it, simply export the following environment variable:

```
export OMPCLUSTER_PROFILE="/path/to/file_prefix"
```

A task graph in a DOT format also can be generated by exporting the environment variable:

```
export OMPCLUSTER_TASK_GRAPH_DUMP_PATH="/path/to/graph_file_prefix"
```

and then run your application normally. At the end of the execution, the runtime will create a couple of timeline files named `<file_prefix>_<process_name>.json` and two graph files named `<graph_file_prefix>_graph_<graph_number>.dot`. You should see one JSON file for each MPI process. Analyzing the traces separately is cumbersome but the OMPCBench tool can help you with that.

4.1.1 Merge timelines

Clone and install the OMPCBench tool in your machine and then follow the steps in the README to install it in a virtualenv. After installing, you can merge the timeline of all the processes into one using the following command:

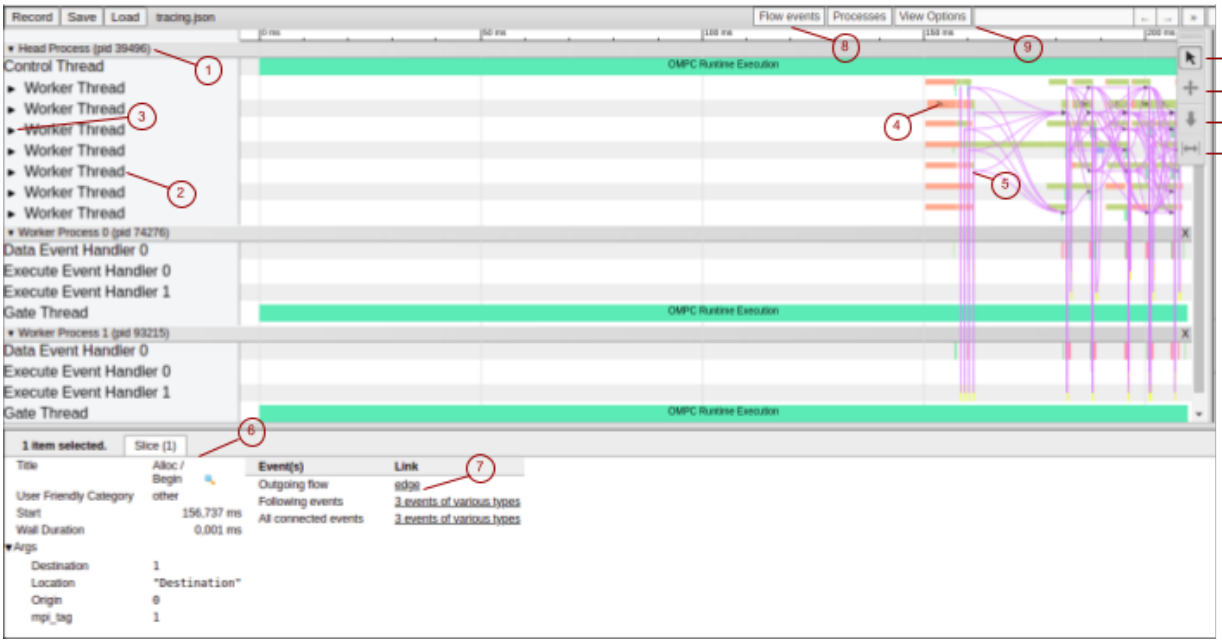
```
# Run the following command inside the virtualenv:
ompcbench merge --no-sync # (Optional) Synchronize timelines disabled. The clocks may␣
↪differ between processes, so by default the timelines are synchronized.
                --developer # (Optional) Generate a timeline for runtime developers␣
↪(with more information and no filters applied).
                --ompc-prefix /path/to/file_prefix # Specify the common prefix or␣
↪directory of the timelines to merge.
                --ompt-prefix /path/to/file_prefix # (Optional) Specify the common␣
↪prefix or directory of the OmpTracing timelines to merge.
                --output tracing.json # (Optional) Merged timeline name. If not␣
↪passed, default name is tracing.json
```

The file `tracing.json` will be created and you can proceed to the next stage of inspection. If the task graph file is in the traces folder, the timeline will contain task dependencies and identifiers.

> **Tip**: Run `ompcbench merge --help` for explanations and more options.

4.2 Inspecting the trace

In order to view your trace in a proper timeline, you can navigate to the URL `chrome://tracing` in the Chrome Browser, click "Load" in the top-left corner and then open your merged timeline or just drag and drop the file into the window. You should now be seeing your application trace, including the runtime operations. An example timeline is presented below.



Timeline OMPC

The timeline points indicated by numbers represent as follows:

1. Process separation: all threads below belongs to the referenced process.
2. Thread separation: all events on the right belongs to the referenced thread.
3. Arrow to hide events: used to decrease the height of the timeline, as events from that thread are compressed vertically. It is useful when users need to analyze events that are vertically distant on the timeline. The arrow next to the process name has a similar function, but completely hides the threads and events of that process.
4. Timeline events: the label indicates what it represents on OMPC. All the colors are chosen by Chrome Tracing except for the events named "Task XX", where events of the same color have the same source location and XX is the task id.
5. Arrows that indicate relations between different events.
6. Event information: when an event is selected, by clicking on it, this panel shows some event information. The first lines are information provided by the Chrome Tracing tool (as event start, duration, and arrows) and the args section is specific information about this event provided by OMPC.
7. Provides information about any arrow from or to this event. If click on it, it will show the two events linked.
8. If click on it, is possible to obtain a more clear view of the timeline by hiding the events arrows.
9. Used to search for events by label or any of its arguments.
10. Chrome Tracing tool to select events. This feature must be enabled to exhibit event info by clicking on it.
11. Chrome Tracing tool to move across the timeline. It is useful when the timeline is zoomed in to a specific point.

12. Chrome Tracing tool to zoom the timeline. It is useful to analyze events more precisely and see events that have a fine duration (like communication events). It is possible to zoom a specific event by pressing 'f' on the keyboard.
13. Chrome Tracing tool to measure the duration between two events on the timeline. It is useful when events are in a different process.

4.2.1 OMPC Events

The OMPC Events of the user version are listed below:

- OMPC Runtime Execution: total duration of application.
- Target Enter (Nowait): represents a target enter data map (nowait) region or an entrance in a target data map region (nowait).
- Target Exit (Nowait)": represents a target exit data map (nowait) region or an exit in a target data map region (nowait).
- Target (Nowait): target (nowait) region, represents an execution task in the head node.
- Execute / Target Execution or Task XX: total duration of task execution on worker nodes. If the graph file was provided, the name will be Task XX, where XX represents the task id, otherwise, the name will be "Execute / Target Execution".
- Execute: total duration of task execution on head node.
- Alloc: when OMPC allocates data in the worker nodes. In the worker nodes this event is divided into a pair of Alloc / Begin and Alloc / End.
- Submit: In the head node, represents a data submission to worker nodes. In the worker nodes, represent received data (from the head node through Submit event or from another worker through the Forward event) except in the developer version of the timeline (Forward events always have an associated Submit event representing a data submission). In the worker nodes, this event is divided into a pair of Submit / Begin and Submit / End.
- Delete: the data allocated on the worker nodes are freed. In the worker nodes, this event is divided into a pair of Delete / Begin and Delete / End.
- Retrieve: represents data received in the head node from a worker node. In the worker nodes, this event is divided into a pair of Retrieve / Begin and Retrieve / End.
- Forward: represents the head node sending a message to a worker to send data to another worker. The worker node receives the data by the Submit event. In the worker nodes, this event is divided into a pair of Forward / Begin and Forward / End.
- Variable names: if the user compiles the application with debug symbols, setting CMake flag to `-DCMAKE_BUILD_TYPE=RelWithDebInfo`, events that have variable names associated (e.g. Submit) will be nested to a variable name event.

4.2.2 OMPC Args

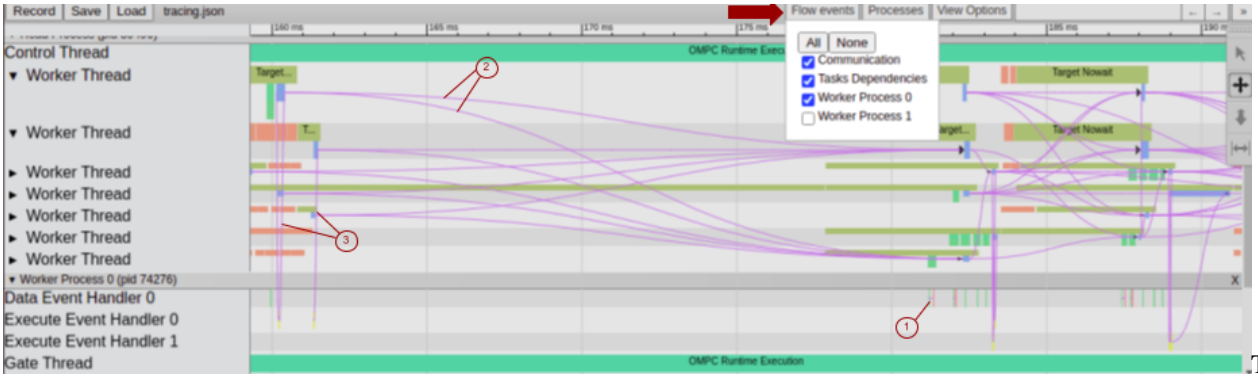
Some of the arguments that can appear in OMPC events are listed below:

- Origin: number of the process where the event was created.
- Destination: number of the process where the event was executed.
- Location: some events have a pair (origin and destination), the location indicates which of the pairs the event is on.
- mpi_tag: event id that is the same for the origin and destination pair.

- task_id: the identifier of the task that corresponds to the task graph.
- source_location: the file, line, and collum that the event was executed. For the origin and destination args, the number represents the ids of head and worker nodes. 0 represents the head node, 1 represents the worker node 0, and so on.

4.2.3 OMPC Dependencies

The OMPC timeline has dependencies (arrows) that indicate relations between different events. These dependencies can be disabled to clear the timeline view, and the names presented in the image below represents:



Timeline Dependencies

1. Communication: dependencies between event communication pairs (Begin and End) in the worker nodes.
2. Tasks: data dependencies between tasks in the head nodes.
3. Worker Process X: dependencies between Execute event in the Head node and Task events in the worker node X, where X is the id of the worker node.

4.2.4 OMPC Threads

- Head Node:
  - Control Thread: responsible for scheduling the task graph. Worker Thread: responsible for data communication and task creation.
- Worker Node:
  - Data Event Handler: handles data communication events (receiving and submitting data).
  - Execute Event Handler: handle the execution of tasks.

4.3 Filter usage

The Filter command is similar to the merge command, but with fewer options. You can pass an OMPC prefix and it will merge and automatically filter and synchronize the files. The advantage of using the filter is when the user already has a merged timeline in a developer version and needs to simplify it.

```
# Run the following command inside the virtualenv:
ompcbench filter --ompc-prefix /path/to/file_prefix # Specify the common prefix or␣
↪directory of the timelines to merge.
                 --output tracing.json # (Optional) Merged timeline name. If not␣
↪passed, default name is tracing.json
```

(continues on next page)

(continued from previous page)

4.4 OmpTracing usage

OmpTracing tool extract information about the OpenMP runtime like the duration of loops, parallel regions, and tasks. The OMPC timeline collects only information about tasks created using the target compile directive, while OmpTracing provides information about the ones created using the task compile directive. OmpTracing can be used just by executing the following command, and merged with OMPC timeline by using ompt-prefix merge option:

```
export OMP_TOOL_LIBRARIES=/path/to/libomptracing.so
```

For more information about how to select events to be monitored, consult this documentation.

CHAPTER

FIVE

DEBUGGING

Debugging distributed application is always a complex task. Here is a set of advices and tips to help you in this journey.

5.1 Single Process Execution

Debugging OmpCluster programs might be a bit tricky since they run on multiple MPI processes, so it is usually a good idea to start by fixing the execution of the application on a single node without using mpirun.

5.2 Runtime Information

You can enable the debug message of the libomptarget runtime by setting the environment variable `LIBOMPTARGET_INFO=-1`. Just re-execute the application and you should see many messages in the execution log including some more error messages.

It is recommended to build your application with debugging information enabled, this will enable filenames and variable declarations in the information messages. Use the `-g` flag in Clang/GCC or configure CMake with `-DCMAKE_BUILD_TYPE=Debug` or `-DCMAKE_BUILD_TYPE=RelWithDebInfo`.

More information on how to use this environment variable is available [here][libomptarget-info].

5.2.1 Advanced Runtime Information

`LIBOMPTARGET_DEBUG` and `OMPCLUSTER_DEBUG` can also be used to enable additional logs. This feature is only available if libomptarget was built with `-DOMPTARGET_DEBUG`. The debugging output provided is intended for use by libomptarget and OMPC developers. More user-friendly output is presented when using `LIBOMPTARGET_INFO`.

5.3 The GNU Debugger (GDB)

GDB is one of the most popular terminal debuggers out there, even though it is very serial-oriented. In order to make the debugging endeavor smoother, be sure to enable debug information when compiler your program. Use the `-g` flag in Clang/GCC or configure CMake with `-DCMAKE_BUILD_TYPE=Debug` or `-DCMAKE_BUILD_TYPE=RelWithDebInfo`. When troubleshooting MPI applications, we usually launch one instance of GDB per MPI process. The following sections discuss some tools to aid you.

Here is a non-exhaustive table of useful GDB commands:

Command	Description
`run`	Run your program until it exits.
`continue`, `c`	Run your program until it hits a breakpoint.
`break file:line`	Set a breakpoint in a file followed by a line.
`backtrace`, `bt`	Show the stack trace.
`step`, `s`	Execute until next statment, stepping into function calls.
`next`, `n`	Execute until next statement, stepping over function calls.
`finish`, `fin`	Execute until current function returns.
`print expression`	Evaluates `expression` and print the result.
`info break`	List all breakpoints.
`info threads`	List all threads.
`info locals`	List all local variables.

The official GDB documentation has a lot more commands and better descriptions, be sure to check it out.

5.4 The LLVM Debugger (LLDB)

The LLVM debugger is a more modern alternative to GDB, and as the name suggests, it is part of the LLVM compiler infrastructure. You must also compile your program with debug information enabled and then launch it with `lldb -- <program> <args>`. The commands are different from GDB, you can check a tutorial here and the equivalent commands from GDB here.

LLDB commands follow a well defined structure:

```
<noun> <verb> [-options [option-value]] [argument [argument...]]
```

Here is a non-exhaustive list:

LLDB Command	GDB Equivalent
`run`, `process launch`	`run`
`step`, `thread step-in`	`step`
`next`, `thread step-over`	`step`
`finish`, `thread step-out`	`finish`
`breakpoint set --file file --line line`	`break file:line`
`breakpoint list`	`info break`
`frame variable`	`info locals`

5.5 Debugging with TMPI

> **Note**: This is the recommended way of debugging. Both TMPI and Tmux are installed in our containers such that you can debug your application everywhere with no setup required other than the container image itself.

TMPI (repo) is a bash script that launches multiple MPI processes in a Tmux window and attaches one pane for each process. Combined with GDB, it is possible to debug distributed applications more or less easily. By default TMPI enables pane synchronization which means that the keys you type in one pane are also sent to the others.

TMPI usage is:

```
tmpi <nprocs> <commmand>
```

Where <nprocs> is the number of processes to be launched and <command> is the command you with to run. A more concrete example would be:

```
tmpi 4 gdb --args <program> <args>
```

> **Tip**: If your command is really long or you need to set variables before execution you can write a bash script and then make TMPI invoke your script instead.

Tmux has a lot of interesting features, it is highly recommended that you take some time to learn how to use this tool properly. In the meantime, check this cheatsheet out for quick reference.

> **Tip**: Here is is @leiteg's Tmux configuration, there are a bunch of shortcuts to make the experience of Tmux feel smoother. Feel free to use it as you like. If you are unsure what the options do ask him on Slack or simply `man tmux`. :wink:

5.5.1 Closing panes after execution

TPMI sets the Tmux option `remain-on-exit on`, which keeps the panes after the command finishes. To close the tmux window, you can use `Ctrl + B, &, y`. If you are exclusively using TMPI with GDB, you do not need `remain-on-exit on`, since gdb does not exit when the executable finishes. To unset this option, comment line 128 from TMPI: `#tmux set-window-option -t ${window} remain-on-exit on &> /dev/null`.

5.5.2 Unreproducible bugs

> **Note**: This is not thoroughly tested, it may not work properly.

When working with parallel/distributed code, it is very common to run into a bug which does not always occur and consequently is really hard to reproduce and debug. A trick using TMPI and GDB can be used to run a command multiple times and quit GDB automatically if the program succeeds or leave it open otherwise.

How to do it: First you will need to change your TMPI script and comment line 128 which has the following contents: `#tmux set-window-option -t ${window} remain-on-exit on &> /dev/null`. This will configure your Tmux window to close when all the processes (in this case, GDB) exit. Next, we need to tell GDB itself to quit when everything runs fine. Use the following script that runs the same program repeatedly in order to catch one faulty execution:

```
for i in {1..100}; do
    tmpi 2 gdb -quiet -ex='!sleep 1' -ex='set confirm on' -ex=run -ex=quit --args ./
↪program args
done
```

How it works: GDB flags work out the magic:

Flag	What it does
`-quiet`	Supress GDB startup message.
`-ex='! sleep 2'`	Introduce a small delay in GDB. This is needed because sometimes GDB exists before the TMPI script finishes executing and then the latter complains "window not found".
`-ex='set confirm on'`	Confirm before doing dangerous operations. More specifically, confirms before quitting a program in progress.
`-ex=run`	Start running program immediately.
`-ex=quit`	Quit GDB right after running. If the program exits successfully, no confirmation is neeeded and GDB quits. If the program received a signal (error), then it hangs waiting for confirmation, just say "no" and you have your debug session.

5.5.3 Missing RTTI information

Sometimes GDB may not correctly parse the RTTI information embedded into clang debug binaries. In such cases, one can use LLDB to correctly debug an MPI program compiled by clang.

```
tmpi 4 lldb -- <program> <args>
```

Mind you that the commands accepted by LLDB may not directly match the ones supported by GDB. For more information, see the *LLDB section*.

5.6 Common Errors

5.6.1 Fatal error

You might get the following error which is quite common but not not very helpful:

```
Libomptarget fatal error 1: failure of target construct while offloading is mandatory
```

This error basically means the offloading of the computation failed.

In this case, it is usually helpful to enable the debug message of the libomptarget runtime using `LIBOMPTARGET_INFO=-1`. Then, re-run the application and you should see many messages in the execution log including some more interesting errors.

5.6.2 Undefined symbol

```
Target library loading error: /tmp/tmpfile_zSK0IW: undefined symbol: xxx"
```

This means `xxx` is used in the target region and should be declared as such, using the `declare target` pragmas.

5.6.3 Segfault error

In case you get a segfault, you can try to debug the program using TMPI and gdb. Using `printf` might also be useful.

CHAPTER

SIX

ADVANCED USAGE

6.1 Tuning

This documents gathers the strategies users can take advantage of when tuning and optimizing OMPC applications.

6.1.1 Threads

Currently, the number of threads from the head process must match the number of execute event handlers of all workers to get the best performance from the runtime. Our suggestion is to set the environment variable `LIBOMP_NUM_HIDDEN_HELPER_THREADS` to `OMPCLUSTER_NUM_EXEC_EVENT_HANDLERS` * num workers. This is a known limitation of our runtime and we are working on fixing it upstream (see here) and in one of our next releases.

6.1.2 Scheduler

In order to map `target` tasks to devices (*i.e.* worker nodes), the OmpCluster runtime uses the HEFT scheduling algorithm by default.

The Round-Robin scheduling algorithm is also available. This algorithm is fast to execute but generally produces bad schedules. Still, the users can experiment with it by setting the following environment variable:

```
export OMPCLUSTER_SCHEDULER="roundrobin"
```

Setting this variable will use round-robin instead of heft the next time your application is executed.

6.1.3 Blocking Scheduler

By default, the OMPC scheduler allows multiple target tasks to be mapped to a worker at a time. This is useful to use all cores of each worker, especially if it does not have parallel computations within the tasks.

However, it might not be the ideal behavior for all applications, especially when target tasks already perform parallel computations (e.g. a `parallel for` within a `target nowait`). In that case, the blocking behavior can be enabled for any scheduling strategy (`round_robin`, `heft`, etc) by setting the following environment variable:

```
export OMPCLUSTER_BLOCKING_SCHEDULER=1
```

When set, the scheduler behaves as a blocking scheduler: it only allow a single target task to be mapped to a worker ate a time. This is useful to avoid competition in using the hardware resources of the workers.

Tuning HEFT

HEFT is a heuristic-based list-scheduling algorithm that makes decisions based on:

- **Computation cost**: how many time units is required for executing a single task;
- **Communication cost**: how many time units for transferring data between two tasks.

Unfortunately, the runtime does not know have this information ahead of time and expects the user to provide them via environment variables. Take a look at the example below. The first variable, `OMPCLUSTER_HEFT_COMP_COST` indicates how long does it take to execute a task. Secondly, the variable `OMPCLUSTER_HEFT_COMP_COEF` specifies a coefficient in relation to the computation cost. For example, a coefficient of 2 indicates that communication costs twich as much as computation. The actual units here (miliseconds, seconds, minutes) are not as important as the relationship between computation and communication cost.

```
# Computation cost in time units (could be miliseconds, seconds, ...)
# E.g. computation of a task takes 30 time units
export OMPCLUSTER_HEFT_COMP_COST="30"

# Communication cost as a coefficient of computation cost
# E.g. communication of a dependency takes 2x as much time as a computation.
export OMPCLUSTER_HEFT_COMP_COEF="2.0"
```

> **Tip**: Adjusting the costs may required some experimenting and iteration from the user-side in order to find a good balance. Dumping the task graph (see next section) can also help here.
>
> **Note**: The OmpCluster runtime does not yet support tasks and dependencies with different costs. If your application fits this category we suggest you set an average value to cover both cases.

Dumping the Task Graph

If you wish to inspect the final scheduled graph you can use the `OMPCLUSTER_TASK_GRAPH_DUMP_PATH` to specify a path where to dump using the GraphViz dot language.

```
# Specify a path where the task graph should be dumped (required)
export OMPCLUSTER_TASK_GRAPH_DUMP_PATH="<path>/<file_prefix>"

# Show the edge weights in the graph (optional)
export OMPCLUSTER_HEFT_DUMP_EDGE_LABEL=1
```

Dumping internal HEFT data

> **Note**: This option is useful for developers who are looking to troubleshoot the execution of the algorithm itself. Reguler users can safely ignore this option.

The following environment variable will instruct the HEFT scheduler to dump its internal state before the application exists. This is useful if you are looking to inspect the EST, EFT, AST, AFT tables.

```
export OMPCLUSTER_HEFT_LOG="/path/to/heft.log"
```

6.2 Environment variables

This section describes the environment variables than can be used to tune the runtime.

6.2.1 OpenMP Target Runtime

Non-exhaustive list of the settings for the LLVM OpenMP Target runtime library. The full list is available on the upstream website. Please notice some settings might differ since our version of LLVM is not synchronized with the last version of LLVM.

LIBOMPTARGET_INFO

The variable controls whether or not the runtime provides additional information during the execution. The output provided is intended for use by application developers. It is recommended to build your application with debugging information enabled, this will enable filenames and variable declarations in the information messages. More information on how to use this environment variable is available here.

LIBOMPTARGET_DEBUG

The variable controls whether or not debugging information will be displayed. This feature is only available if libomptarget was built with `-DOMPTARGET_DEBUG`. The debugging output provided is intended for use by libomptarget developers.

LIBOMPTARGET_MEMORY_MANAGER_THRESHOLD

`LIBOMPTARGET_MEMORY_MANAGER_THRESHOLD` sets the threshold size for which the libomptarget memory manager will handle the allocation. Any allocations larger than this threshold will not use the memory manager and be freed after the device kernel exits. Contrary to the other libomptarget plugins, the OmpCluster runtime is using a Bump allocator with a default threshold value of 8MB. If `LIBOMPTARGET_MEMORY_MANAGER_THRESHOLD` is set to 0 the memory manager will be completely disabled.

LIBOMP_NUM_HIDDEN_HELPER_THREADS

The variable configures the number of threads used by the OpenMP runtime to offload target tasks. Those threads are called hidden helper threads. By default, the number of hidden helper threads is 8.

Note: Currently, the number of hidden helper threads should match the total number of execute event handlers to get the best performance.

6.2.2 OMPC Runtime

General settings of the OmpCluster runtime

OMPCLUSTER_PROFILE

The variable defines the path and file suffix used to dump the trace files generated by the profiler. The output is a set of JSON files with the following names `path/filename_suffix + "_" + process_name + ".json"`.

There is *no default value*, if not set, the profiling is not performed and the execution traces are simply not saved to any file.

OMPCLUSTER_PROFILE_LEVEL

It controls what profiling events that will be added to the generated trace. Use `-1` to print them all. Default value is 1 and should be sufficient for most users.

OMPCLUSTER_DEBUG

Sets the level of OmpCluster debug messages. This feature is only available if libomptarget was built with `-DOMPTARGET_DEBUG`. Default value is 0.

OMPCLUSTER_MPI_FRAGMENT_SIZE

Maximum buffer size sent in a single MPI message. If a buffer is larger than this threshold, it is *automatically* splitted in separated messages. The default value is 100000000 bytes (100 MiB).

OMPCLUSTER_NUM_EXEC_EVENT_HANDLERS

It controls the number of threads responsible for executing target regions spawned per process. The default value is 1.

OMPCLUSTER_NUM_DATA_EVENT_HANDLERS

It controls the number of threads responsible for data-related events spawned per process. The default value is 1.

OMPCLUSTER_EVENT_POLLING_RATE

Polling rate used by event handlers in microseconds. Small values reduce waiting time between checks, but increases CPU usage. Default value is 1 us.

OMPCLUSTER_BCAST_STRATEGY

Sets how OmpCluster transfers data that must be broadcast (i.e. sent to all nodes).

Mode	Description
disabled	Data on synchronous target data map regions will be sent to each device when needed by the memory management system.
p2p	Data on synchronous target data map regions will be sequentially sent to every device through peer-to-peer communication.
mpibcast	Data on synchronous target data map regions will be sent to every device using MPI_Bcast.
dynamicbcast	Data on synchronous target data map regions will be sent to every device using the Dynamic Broadcast algorithm.

Default value is `disabled`.

OMPCLUSTER_ENABLE_PACKING

If set to `1` enables communication events to be packed, while a value of `0` does not. When enabled, the runtime packs communication metadata and buffers that have less than `OMPCLUSTER_PACKING_THRESHOLD` bytes. Dafault value is `0`.

OMPCLUSTER_PACKING_THRESHOLD

The value passed to `OMPCLUSTER_PACKING_THRESHOLD` defines the maximum size of the buffers that will be packed in a single MPI message. If `OMPCLUSTER_PACKING_THRESHOLD=0` and `OMPCLUSTER_ENABLE_PACKING=1`, then only the communication metadata is packed. If `OMPCLUSTER_ENABLE_PACKING` is set to `0`, `OMPCLUSTER_PACKING_THRESHOLD` is not used. Default value is `0`.

6.2.3 OMPC Scheduler

Settings of the OmpCluster runtime specific to the task scheduler

OMPCLUSTER_SCHEDULER

Selects the strategy used by the scheduler to assign tasks to specific MPI processes. Currently, there are three available options, described below.

Scheduler	Description
roundrobin	Dynamic round-robin: schedules task when executing them by continuously going over the list of processes
graph_roundrobin	Static round-robin: schedules task when creating them by continuously going over the list of processes
heft	Heterogeneous Earliest Finish Time (HEFT): more advanced heuristic that takes into account the communication time. See wikipedia for more details.

Default value is `heft`.

OMPCLUSTER_BLOCKING_SCHEDULER

This variable is used to configure the blocking behavior of the OMPC scheduler. A value of 0 enables multiple target regions (*target tasks*) to be executed at the same time in parallel by each MPI process, and a value of 1 does not. Default value is 0.

Please notice that the dynamic round-robin strategy will not assign a task to a busy process (already executing a task) if the blocking behavior is enabled.

OMPCLUSTER_TASK_GRAPH_DUMP_PATH

Path used to dump the task graph generated by the scheduler, indicating to which MPI rank each task was assigned. The output is a dot file.

There is *no default value*, if not set, the task graph is simply not saved to any file.

6.2.4 HEFT parameters

Parameters used by the HEFT scheduler. Only used if `OMPCLUSTER_SCHEDULER=heft`. Those parameters are especially useful since the runtime is not yet able to predict the communication and computation time of the tasks.

6.2.5 OMPCLUSTER_HEFT_COMM_COEF

Coefficient of the communication cost. Having a higher coefficient means that communication will have more weight in relation to computation. Default value is 1.

6.2.6 OMPCLUSTER_HEFT_COMP_COST

Default computation cost of tasks. Default value is 100.

6.2.7 OMPCLUSTER_HEFT_COMM_COST

Default communication cost of tasks. Default value is 1.

6.2.8 Fault Tolerance

Settings of the OmpCluster runtime specific to the fault tolerance.

OMPCLUSTER_FT_DISABLE

OMPCLUSTER_CP_USEVELOC

OMPCLUSTER_CP_EXECCFG

OMPCLUSTER_CP_TESTCFG

OMPCLUSTER_HB_TIMESTEP

OMPCLUSTER_HB_TIMEOUT

OMPCLUSTER_HB_PERIOD

OMPCLUSTER_CP_MTBF

OMPCLUSTER_CP_WSPEED

CHAPTER

SEVEN

OMPC PLASMA

This library is an extension of the PLASMA library for distributed memory systems.

7.1 Building

To use OMPC PLASMA, we provide a docker image `ompcluster/plasma-dev:lastest` containing a pre-compiled Clang/LLVM with all the OpenMP and MPI libraries needed to compile and run OMPC PLASMA.

You can execute OMPC PLASMA on any computer using docker or Singularity.

To install OMPC PLASMA we use the following commands:

```
git clone https://gitlab.com/ompcluster/plasma.git
cd plasma/
mkdir build
cd build
export CC=clang
export CXX=clang++
export OpenBLAS_ROOT=/usr/local/include/openblas/
cmake ..
make -j$(nproc)
```

7.2 Usage

OMPC PLASMA should be run using parameters. To observe the parameters, execute the following command:

```
./plasmatest --help
```

In general, OMPC PLASMA should be executed with the following parameters:

```
./plasmatest routine --dim=$dim --nrhs=$dim --nb=$nb --test=$test
```

These parameters represent:

- `routine`: This parameter represents the application of linear algebra. Currently OMPC PLASMA supports four applications: `spotrf`, `sgemm`, `ssyrk` and `strsm`.
- `$dim`: The matrix size.
- `$nb`: The block size. This number should be divisor of `$dim`.
- `$test`(yln): Determine whether or not the results should be verified.

There are other parameters, which depend on each routine that the user wants to execute.

7.3 Example

Here is a example how to run the OMPC PLASMA in a cluster using SLURM:

```
#!/bin/bash
#SBATCH --job-name=plasma-job
#SBATCH --output=plasma-output.txt
#SBATCH --nodes 3

module purge
module load mpich/4.0.2-ucx

##### OMPC settings
export OMPCLUSTER_NUM_EXEC_EVENT_HANDLERS=4
expsort LIBOMP_NUM_HIDDEN_HELPER_THREADS=8
export OMPCLUSTER_HEFT_COMM_COEF=0.00000000008
export OMPCLUSTER_HEFT_COMP_COST=20000000000

##### OpenMP settings
export OMP_NUM_THREADS=4
export OPENBLAS_NUM_THREADS=1

srun --mpi=pmi2 -n 3 singularity exec plasma-dev_latest.sif plasma/build/plasmatest␣
↪spotrf --dim=1024 --nrhs=1024 --nb=256 --test=y
```

OMPC configurations depend on how the user executes the program in the cluster. In the example, OMPC PLASMA runs on 2 worker nodes, each node will work with 4 threads.

CHAPTER

EIGHT

OMPTRACING

This tool can be used to trace the execution of an OpenMP (or OmpCluster) application.

8.1 Usage

To enable OmpTracing you just need to set the `OMP_TOOL_LIBRARIES` environment variable to the path of the OmpTracing library.

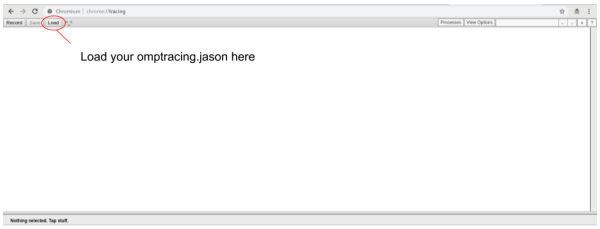
```
export OMP_TOOL_LIBRARIES=/path/to/libomptracing.so
./your-own-omp-program
```

If you use one of the container images provided by OmpCluster, OmpTracing library is already provided in `/opt/omptracing/lib/libomptracing.so`, otherwise you can compile it from its repository.

After the execution of the program ends, OmpTracing should have produced two files: a JSON file containing the tracing of the execution and a DOT file describing the task graph produced by the OpenMP runtime.

8.2 Tracing

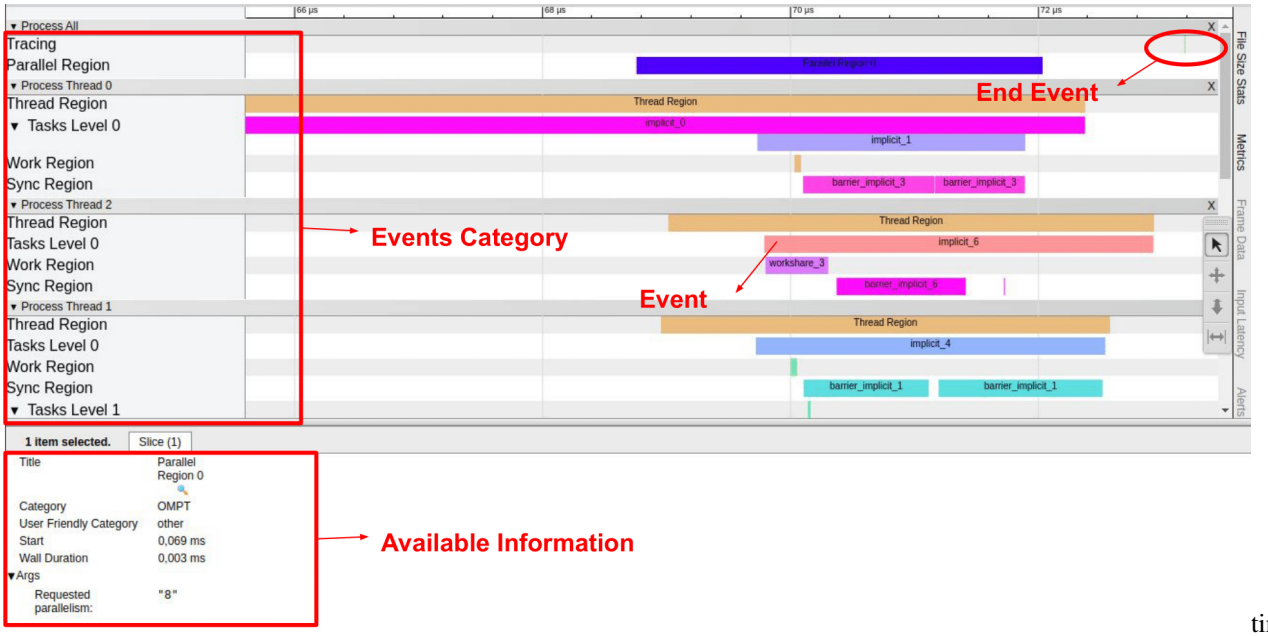
You can use Chrome browser as an graphical interface to see the tracing of the program execution. Just open Chrome (or Chromium), enter `chrome://tracing` in the address field. A new interface should appear where you can load the omptracing.json file just as presented in the image below.



tracing

A timeline example is presented in the image below. An OmpTracing timeline displays events, categories (that separate events), and extra information about a selected event. The registered events are the horizontal colored bars (e.g. the implicit_6 event indicated), the left panel separates the events by their categories such as parallel region, thread region, and others. Each event has its category or subcategory id that composes the event name, for example, the implicit_6 event indicated has Tasks Level 0 category (left panel) and implicit subcategory with 6 id (event name). The number in the task level category on the left panel represents the level of the task parent tree. When an event is selected, all

the available information (args) is presented in the bottom section. The selection of a specific event can be performed by clicking on the event's bar on the timeline. The categories are divided by threads (Process Thread in the left panel) and the general categories are aligned in Process All. The tracing category registers the beginning and the end of the application. In the figure, the begin event is hidden because of the image zoom, and the end event is circled.



timeline

The categories are explained below.

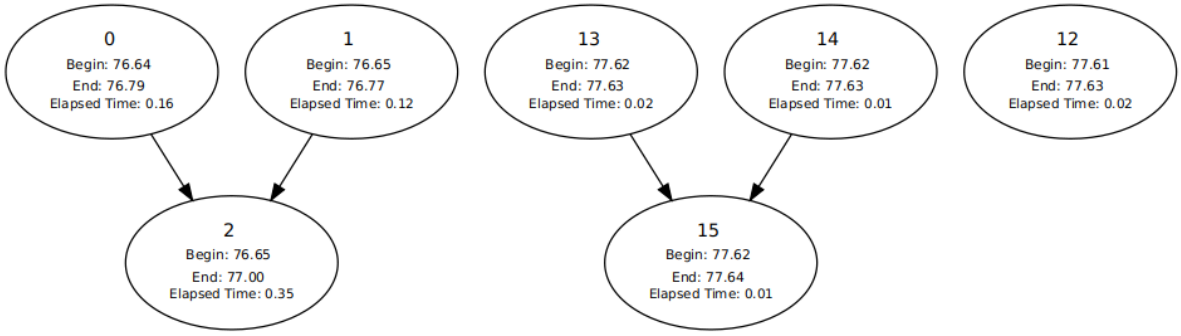
- **Parallel Region:** it marks the begin and the end of parallel regions. Each region specifies an ID and the number of threads requested.
- **Thread Region:** it marks the begin and the end of thread regions. The thread type is specified. See OMPT documentation to get more informations.
- **Tasks:** it marks the begin and the end of tasks. The tasks are divided in levels. A parent task is a task that created another one, and tasks of level zero have no parent tasks. The level information is described in the label section, as well as the parent task id and the task type. See OMPT documentation to get more informations.
- **Work Region:** it marks the begin and the end of work regions like loop, taskloop, sections, workshare, single regions and distribute regions and gave information about it. See OMPT documentation to get more informations.
- **Implicit Task Region:** it marks the begin and the end of implicit task regions and gave your number of threads/teams. See OMPT documentation to get more informations.
- **Master Region:** it marks the begin and the end of master regions. See OMPT documentation to get more informations.
- **Sync Region:** it marks the begin and the end of sync regions like barrier implicit/explicit, taskwait, taskgroup and reduction. See OMPT documentation to get more informations.

8.3 Task Graph

OmpTracing also produce a dot file representing the task-dependency graph of the OpenMP program. You can produce a PDF file to visualize it using the following command:

```
dot -Tpdf graph.gv > graph.pdf
```

A graph example is presented in the image below. The graph shows the tasks labeled with OmpTracing identifier numbers and the dependencies are represented by arrows. Each task contains your specific begin time, end time, and elapsed time.

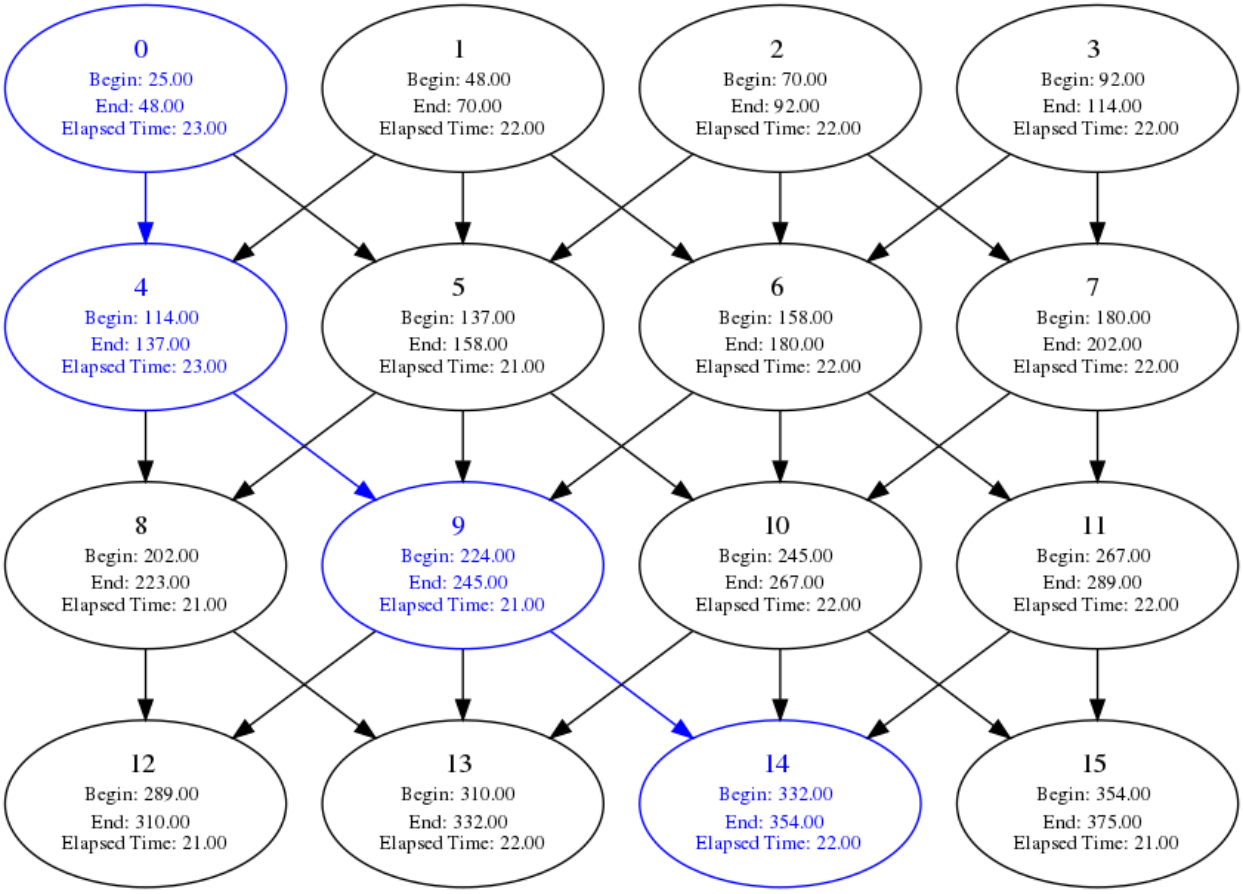


timeline

8.4 Configuration

OmpTracing permits to configure which information will be present in the timeline by passing a JSON configuration file like the model below. An updated example is available here in the OmpTracing repository. The monitored events array select which events will be tracing, the max tasks field represents the maximum number of tasks supported, the critical path field indicates if the critical path will be highlighted in the graph generated (the blue path in the graph example below), and the graph time field indicates if the time information will be presented in the graph. Just add or remove some events to the monitored events array to add or remove the tracing of this information.

```
{
  "monitored events": [
    "task",
    "task create",
    "thread",
    "parallel",
    "work",
    "implicit task",
    "master",
    "sync region",
    "taskwait"
  ],
  "max tasks":2048,
  "graph time":"yes",
  "critical path":"yes"
}
```



tracing

OmpTracing registers only those events defined in the config selector file or uses a default selection. There are two ways to pass a config selector file to OmpTracing:

- By assigning the file path to `OMPTRACING_CONFIG_PATH` environment variable;
- By placing the file with the name "config.json" in the same folder as the application.

If OmpTracing can not read the file passed, it will use the default config selection.

8.5 Tagging

OmpTracing allows adding tagging on the parallel code by adding omptracing header (omptracing.h) and find omptracing package on CMake. The functions `omptracingTagBegin` and `omptracingTagEnd` permit you to choose the name category (tag name) and the event name while the functions `omptracingTagEventBegin` and `omptracingTagEventEnd` permit you to choose just the event name and use a default OmpTracing tag name. See the example code below.

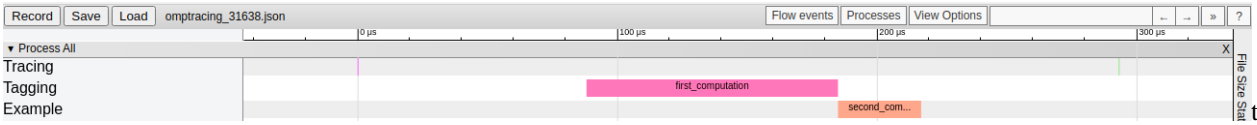
```
#include "omptracing.h"

...

// Tag Name: Default
// Event Name: first_computation
omptracingTagEventBegin("first_computation");
...
```

(continues on next page)

(continued from previous page)

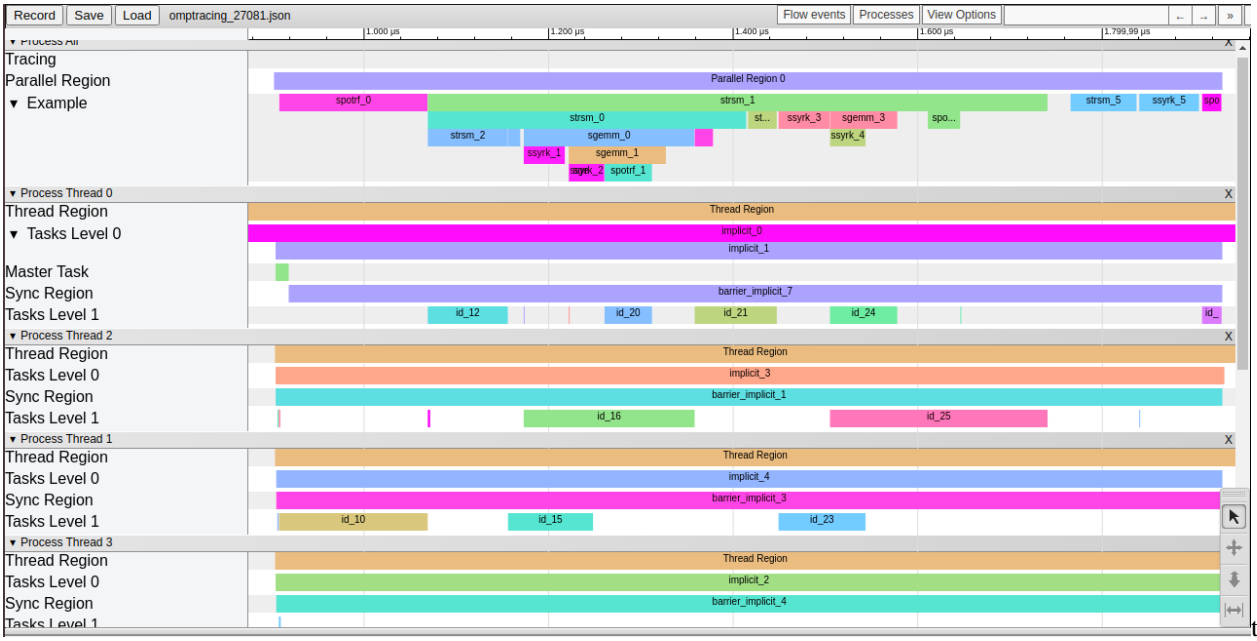
```
  omptracingTagEventEnd("first_computation");

  // Tag Name: Example
  // Event Name: second_computation
  omptracingTagBegin("Example", "second_computation");
  ...
  omptracingTagEnd("Example", "second_computation");
```

The timeline generated is present in the image below.



tracing

A more practical example is present in the next image. It shows a spotrf core tagging of Plasma application. The marked tags are in the "Example" category and it shows spotrf, strsm, ssyrk, and sgemm Plasma computation.



tracing

8.5.1 Linking to OmpTracing library

Contrary to the timeline and the taskgraph which are activated through OMPT environment variable, tagging requires to link the OmpTracing library in a more tradiditionnal manner. This can be achieved using for example the following CMake code:

```
find_package(omptracing CONFIG REQUIRED)
include_directories(${omptracing_INCLUDE_DIRS})

...

target_link_libraries(mytarget omptracing)
```

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search