Chapter 2

Getting Up and Running with the OpenMP Cluster Programming Model

Emilio Francesquini (UFABC)
e.francesquini@ufabc.edu.br
http://lattes.cnpq.br/8949216028517727

Hervé Yviquel (UNICAMP)
hyviquel@unicamp.br
http://lattes.cnpq.br/3339703725728623

Marcio Pereira (UNICAMP)
mpereira@ic.unicamp.br
http://lattes.cnpq.br/2671525924181612

Sandro Rigo (UNICAMP)
srigo@unicamp.br
http://lattes.cnpq.br/8308517667746974

Guido Araújo (UNICAMP)
guido@unicamp.br
http://lattes.cnpq.br/8683914780987242

Abstract

In this text we present a short introduction to the new OpenMP Cluster (OMPC) distributed programming model. The OMPC runtime allows the programmer to annotate their code using OpenMP target offloading directives and run the application in a distributed environment seamlessly using a task-based programming model. OMPC is responsible for scheduling tasks to available nodes, transferring input/output data between nodes, and triggering remote execution all the while handling fault tolerance. The runtime leverages the LLVM infrastructure and is implemented using the well-known MPI library.

2.1. Introduction

The increasing utilization of supercomputers and data centers, particularly in the domain of scientific research, that employ heterogeneous architectures (with accelerators such as FPGAs and GPUs) has led to an escalation in the complexity of the program parallelization process [2, 4]. For instance, real-world high-performance applications frequently comprise a fusion of diverse technologies, such as MPI (Message Passing Interface), multi-threading (either pure, *e.g.* POSIX, or facilitated by libraries such as OpenMP), accelerator-specific programming languages (*e.g.*, CUDA, Verilog), and, occasionally, the inclusion of checkpointing libraries to provide a degree of Fault Tolerance (FT).

In response to this complexity, one of the most widely adopted approaches is the use of directives in the application code. These directives serve as guiding instructions both to the compiler and to the runtime environment, enabling them to fully leverage the available computational resources. Among these solutions, OpenMP [5] stands as a prominent example. Recent versions of the OpenMP standard allow programmers to annotate their code to enable task parallelism as well as to perform the computation on accelerators (in a process called *computation offloading*). Within OpenMP, the OpenMP Target Library library assumes a specialized role, streamlining the parallelization of applications to harness various types of accelerators, including GPUs and FPGAs. These accelerators are referred to as *devices*.

While computation offloading has been widely employed for executing computations on devices within the same computing node, the distribution of these tasks across several nodes of a cluster has commonly been a manual and intricate process. OpenMP Cluster project (OMPC) capitalizes on OpenMP's offloading capabilities and introduces the concept of a "remote device". This concept allows for the offloading of computations to remote nodes. OMPC effectively conceals the underlying communication, which is MPI-based, behind OpenMP task dependencies. Importantly, OMPC adheres to the OpenMP standard, affording application developers the opportunity to harness intra- and inter-node parallelism using a unified set of tools.

In this text[1] we present OMPC [2], an OpenMP task parallelism-based execution model for clusters. OMPC allows for the offloading of complex scientific tasks across HPC cluster nodes in a transparent and balanced way. OMPC has some interesting features such (a) an underlying MPI communication layer for inter-node communication; (b) an event handler that transparently offloads tasks and data to cluster nodes; (c) a cluster-wide HEFT-based task scheduler [7] that balances the workload distribution across

[1]This text results from the compilation of the authors' prior papers and documentation. At the beginning of each section, we provide references enabling readers to access more comprehensive information if they desire further details.

[2]OMPC is available at `https://ompcluster.gitlab.io/`

the cluster; (d) a transparent fault-tolerant mechanism. All these features are provided transparently to the programmer that only interfaces with the OpenMP programming model. OMPC has been successfully used for the development of applications ranging from benchmarks (*e.g.*, TaskBench) to real world applications (*e.g.*, seismic applications for oil/gas reservoir detection, high energy plasma simulations,...). More information and details about the inner workings of OMPC can be seen on the papers [8, 3].

2.2. OpenMP Cluster (OMPC)[3]

OMPC builds upon the OpenMP Target Library which was created to allow the offloading of computation to accelerators, also called devices. Devices can be, for example, GPUs or FPGAs in a single computing node. Using this library, OpenMP users can offload computation directly to the accelerators simply using the OpenMP Target directives. These target directives are very similar to an existing OpenMP feature called *task*, both of which we now explain in further details.

2.2.1. OpenMP's `task` and `target` directives

OpenMP 3.0 introduced the concept of tasks, which enable a higher level of parallelism by allowing code fragments annotated with the `task` directive to execute asynchronously. These annotated code segments are treated as individual tasks and are dispatched to the OpenMP runtime by the *control thread*. Dependencies between tasks are specified using the `depend` clause, which defines both the input variables that a task relies on and the output variables it modifies. This arrangement effectively creates a task graph in which dependencies between tasks are represented as arcs between the nodes (tasks). Once a task's dependencies are satisfied, it is added to a *ready queue* by the OpenMP runtime, from which a pool of *worker threads* can draw tasks for execution. The management of task dependencies, data handling, and thread creation and synchronization are among the core responsibilities of the OpenMP runtime.

Originally, the OpenMP `task` directive was designed with multicore architectures in mind. However, as acceleration devices like GPUs and FPGAs gained prominence, OpenMP evolved to support these devices through the "OpenMP accelerator model" (introduced in OpenMP 4.X). This expansion introduced the `target` directive, which shares similarities with the `task` directive but is intended for offloading computations to acceleration devices instead of CPU cores. It utilizes clauses like `depend` to define dependencies, and it employs the `map` clause to specify the data transfer direction between the host and the accelerator (e.g., `to`, `from`, and `tofrom`). Additionally, the `nowait` clause was introduced to indicate that the `target` directive is non-blocking, allowing it to execute asynchronously on the accelerator.

[3]This section was based on the paper [8], where you can find more details about OMPC inner workings.

```
1 #pragma omp target enter data map(to: A[:N]) nowait depend(out: *A)
2 #pragma omp target nowait depend(inout: *A)
3 foo(A)
4 #pragma omp target nowait depend(inout: *A)
5 bar(A)
6 #pragma omp target exit data map(release: A[:N]) nowait depend(out:
  ↪ *A)
```

Listing 1: OpenMP target tasks

In this scheme, `target` directives are treated as tasks, allowing us to represent the program execution as a task graph. OMPC expands the concept of devices to include nodes of a cluster, enabling the distribution of computations, such as the one involving `foo`, across any node in the cluster. This distribution occurs seamlessly while at the same time maintaining the code identical to the one used for a single node.

As an example, let's examine the code snippet Listing 1. Within this code, we encounter two tasks, namely, `foo` and `bar`, both marked with the `target` directive from OpenMP. This directive, in accordance with OpenMP's specifications [1], enables these tasks to be offloaded for execution on accelerators, such as GPUs.

In this context, what occurs is that the code and associated data for tasks `foo` and `bar` are dispatched to an accelerator for processing. However, it's essential to note that the host machine, which is responsible for executing the code found in Listing 1, also plays a role in managing the data movement required to satisfy the dependencies between these tasks. This means that the host orchestrates the necessary data transfers to ensure that `foo` and `bar` can execute efficiently on the accelerator while meeting their dependencies.

Let's delve into the semantics of this code to gain a better understanding. Starting with line 1, we encounter a critical operation: the vector `A` is transferred from the host memory to the accelerator memory. This process is referred to as *offloading*. Moving on to lines 2-3, we see that these lines play a crucial role in executing the `foo` task on the accelerator. The code of `foo` is dispatched to the accelerator, where it is executed. The result of this execution is stored directly on `A`. Subsequently, `A` is brought back to the host memory. To highlight the importance of this data movement, the `depend(inout: *A)` clause is employed, indicating that `A` is both read and written by the `foo` task. Lines 4-5 mirror a similar computation, this time involving the `bar` task. In line 4, the `target` directive assigns `bar` to an accelerator while specifying that it will read `A`, which was previously written by `foo` and resides in the host memory. Like before, the runtime orchestrates the process: it reads `A` from host memory, transfers it to the accelerator assigned to `bar`, executes the `bar` task, stores the result in accelerator memory, and

finally returns `A` to host memory. Line 6 marks the end of the execution. Notably, all the `target` directives in lines 2-4 include a `nowait` clause, signifying that both `foo` and `bar` are executed asynchronously. Consequently, it is the responsibility of OpenMP runtime to ensure that all `depend` clauses specified in lines 2-4 are satisfied, in accordance with the programmer's directives.

2.2.2. OMPC/OpenMP integration

The use of OpenMP directives makes for a streamlined approach to application parallelization on large computing clusters. OMPC makes use of OpenMP Target Library , which consists of two distinct layers: the "Plugin" layer and the "Agnostic" layer.

The "Plugin" layer comprises various plugin implementations, with each plugin specializing in offloading computations to specific accelerators. For instance, a CUDA plugin provides the code to allow offloading to GPUs. On the other hand, the "Agnostic" layer encompasses the generic aspects of OpenMP Target Library , including program and task execution management, as well as data handling.

Figure 2.1 illustrates the operational framework of OpenMP Target Library. Starting from the user program, device and host code are generated and encapsulated within a fat binary when compiling for a chosen device. This is done using the Clang compiler from LLVM, which contains the OpenMP Target Library implementation. This executable then utilizes the generated dynamic libraries to offload computations to the device specified during compilation. The components introduced by OMPC to OpenMP Target Library are highlighted in green in Figure 2.1.

The OMPC plugin encompasses a few critical elements: MPI, Event System, and Fault Tolerance components, which collectively form the core of OMPC. These components enable the system to distribute instructions and data between nodes and allow the offloading of computations across multiple nodes within a cluster. Notably, all communication is accomplished using MPI.

In the agnostic layer, OMPC introduces the scheduling of tasks through the High-Performance Earliest Finish Time (HEFT) scheduler [7]. Through an evaluation of the task graph, OMPC can optimally determine the node for executing each task. Furthermore, OMPC incorporates a Data Manager component, which plays a pivotal role in orchestrating data movements across nodes. This component bears the responsibility for optimizing inter-node communications, effectively preventing unnecessary data transfers and thereby improving the overall communication efficiency within the cluster.

2.2.3. The OMPC Programming Model

This section outlines the key distinctions between the proposed OpenMP Cluster (OMPC) model and the OpenMP Accelerator Model. OMPC was purposefully designed to seam-

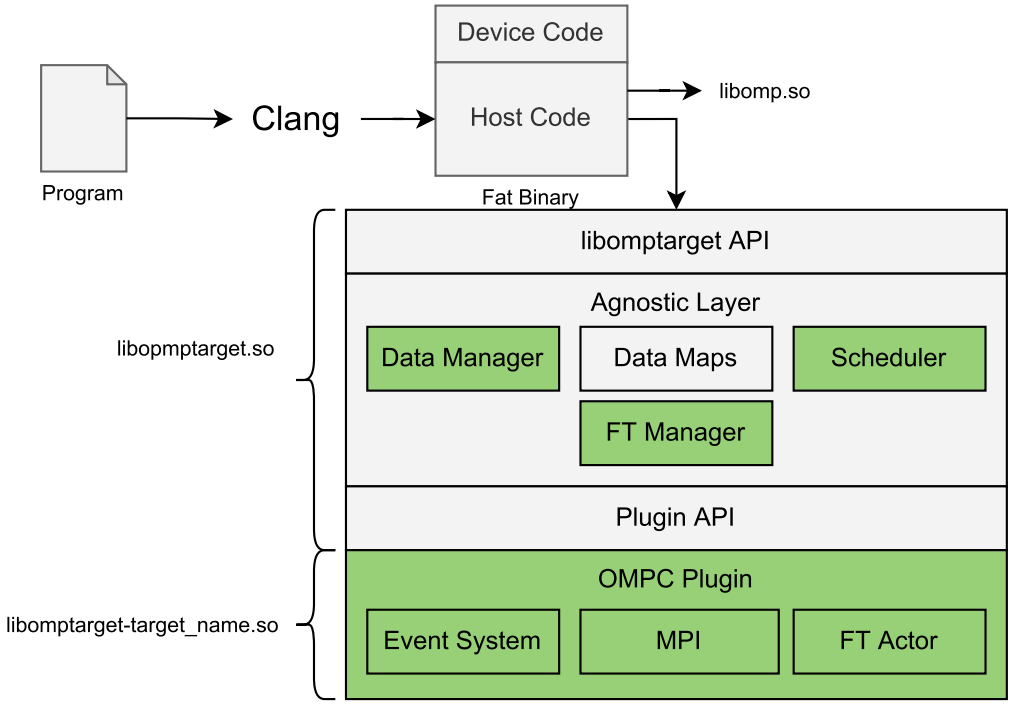

Figure 2.1. Model for OpenMP Target Library . In green, we display the OMPC additions to the library [8].

lessly extend OpenMP semantics to cluster environments. To achieve this, a straightforward abstraction was employed, expanding the concept of "cores" in OpenMP to "nodes" in OMPC. To illustrate this concept further, consider the following use cases:

1. In OpenMP, a `target` directive assigns a task to an accelerator on the same machine, while in OMPC, the task is assigned to a cluster node.

2. While in OpenMP, the `depend` clause handles data movement between host and accelerator memories (both on the same machine), in OMPC, the `depend` clause utilizes underlying MPI calls to make the data transfers between cluster nodes.

Overall, by recognizing that a core in OpenMP corresponds to a node in OMPC, it becomes clear that the foundational semantics of the OpenMP specification still apply in the OMPC context. For instance, the OpenMP code provided in Listing 1 remains unchanged when executed on a cluster using the OMPC runtime. In this scenario, tasks `foo` and `bar` are assigned to cluster nodes, and vector `A` is seamlessly transferred between nodes using OMPC's efficient implementation of the depend clause, which leverages MPI calls to efficiently move `A` from `foo` to `bar`.

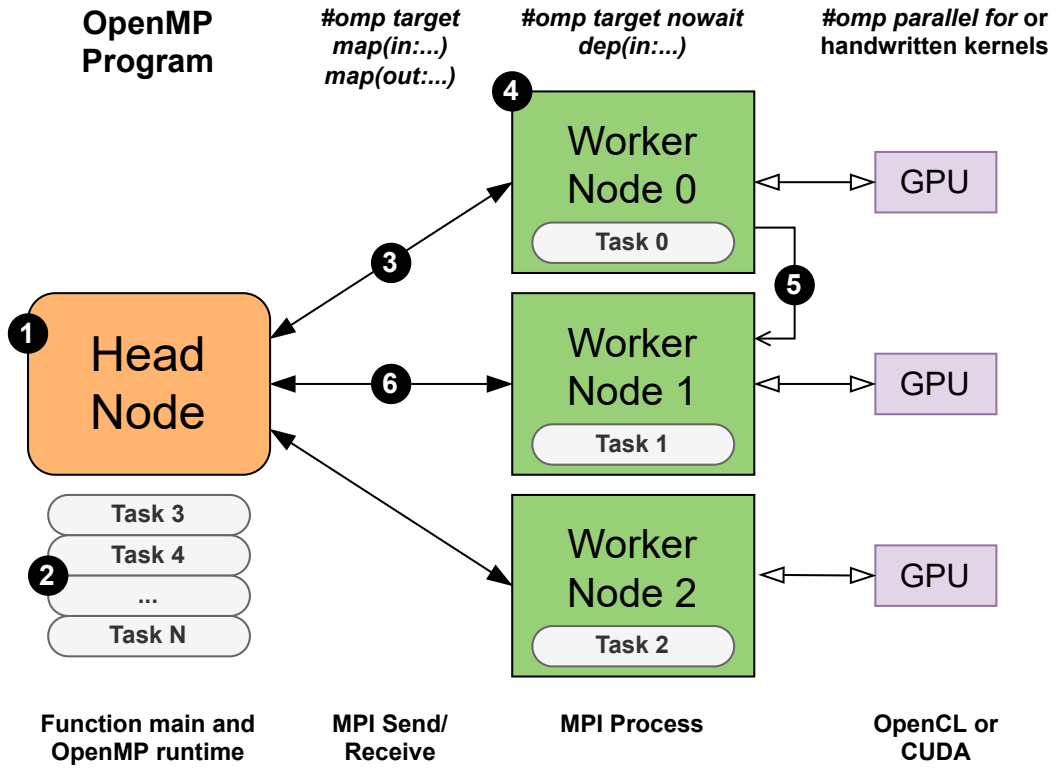

Figure 2.2. Execution Model of the Heterogeneous Cluster Device using OpenMP

Execution model As depicted in Figure 2.2, a standard cluster setup comprises a *head node* responsible for executing the OMPC runtime and a collection of *worker nodes*. This workflow encapsulates the operation of code execution within the cluster, balancing tasks across nodes, and managing data exchanges, all orchestrated by the OMPC runtime. The execution workflow for the annotated code within the cluster unfolds as follows:

1. The user initiates their program from the head node.

2. When the OpenMP kernel is encountered, the OpenMP runtime generates tasks automatically (without executing them) and places them into a dedicated pool, where they await execution. For clarity, let us assume that tasks `foo` and `bar` from the code in Listing 1 correspond to "Task 0" and "Task 1" in Figure 2.2.

3. The OMPC runtime distributes tasks, for example, `foo`, along with their input data (*e.g.*, vector `A`), to be executed on specific worker nodes (*e.g.*, `foo` on "Node 0").

This distribution leverages calls to the underlying MPI subsystem and adheres to a scheduling strategy, such as HEFT, to ensure a balanced computational workload.

4. Worker nodes process the received data.

5. The OMPC runtime forwards the results (*i.e.*, the output of `foo`) to tasks dependent on them (e.g., `bar` at *Node 1*). This transfer is accomplished using MPI calls. Subsequently, the task is removed from the dependency graph, and dependencies are updated. On the head node side, worker node tracking is managed through the dependency graph, following as described by the OpenMP specification.

6. To conclude the computation, OMPC retrieves vector `A` and places it back in the head node.

A crucial aspect to emphasize is the versatility of the code offloaded to the nodes. The code within `foo` or `bar` represents regular code that can potentially harness a second level of parallelism. For instance, if `foo` were to contain a loop annotated with a `parallel for` directive, it could still benefit from OpenMP parallelism within the node. Moreover, `foo` or `bar` could also be written in languages like OpenCL or CUDA, following the practices typically employed in distributed clusters. As previously mentioned, OMPC was intentionally designed to seamlessly integrate with the standard OpenMP runtime.

Furthermore, OMPC was developed with fault tolerance in mind. To facilitate this, each node in OMPC, including both the head node and worker nodes, features a heartbeat mechanism arranged in a ring topology. This arrangement enables nodes to monitor the status of their neighbors. Consequently, if a node experiences a failure, the system promptly detects it and initiates the process of restarting the affected tasks. The implementation of fault tolerance on OMPC is underway and will be released in a future version.

2.3. Using OMPC

In this section, we show how to setup the environment up and get up and running with OMPC[4].

2.3.1. Compilation and execution

To compile OpenMP code for OmpCluster, you need to specify an OpenMP target, denoted as `x86_64-pc-linux-gnu` to instruct the compiler to compile the OpenMP

[4]This section and the next are revised versions of the public documentation of OMPC, written by the authors and the OMPC Team (available at: `https://ompcluster.readthedocs.io`).

target code region for a particular device. For instance, you can compile the `mat-mul` example using the following command:

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

Listing 2: Compiling an application with OMPC.

Then, the newly generated program can be executed. However, unlike conventional OpenMP programs, OmpCluster programs require to be executed using MPI. To that end, tools such as `mpirun` and `mpiexec` should be employed. This is necessary so that the OMPC's distributed runtime system is able to use the configured (for MPI) infrastructure. The command line should be like:

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

Listing 3: Executing an OMPC application with MPI.

In the example provided in Listing 3, the runtime will automatically generate three MPI processes: one *head process* and two *worker processes*. The head process is responsible for offloading OpenMP target regions, which are then executed on the worker processes, following the currently implemented scheduling strategy.

The runtime also supports offloading to remote MPI processes (located on different computers or containers). These remote configurations can be established using the `-host` or `-hostfile` available on the `mpirun` command (please note that specific flag names may vary among different MPI implementations). However, similar to any MPI program, it is essential for the user to ensure that the binary executable is copied to all relevant computers or containers before execution. This can be accomplished using commands such as `pdcp` or by employing an NFS (Network File System) directory for seamless access across the networked nodes.

Execution logs are available using the flag `LIBOMPTARGET_INFO`, set as an environment variable. The execution in this case will look like this:

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

Listing 4: Executing an OMPC application with MPI.

2.3.2. Containerized images

You do not have to go through the process of downloading and compiling the most recent OMPC version from the official website at `https://ompcluster.gitlab.io/`. Instead, we strongly encourage you to opt for pre-compiled Docker images. These images come equipped with Clang/LLVM, along with all the essential OpenMP and MPI libraries required to run OMPC programs seamlessly. You can conveniently access these pre-compiled images at `https://hub.docker.com/r/ompcluster/`. This approach simplifies your setup process and ensures that you have all the necessary tools readily available.

All images are built based on Ubuntu 20.04. However, we offer various configurations with different CUDA versions, as well as the choice of MPICH or OpenMPI. You can select the appropriate Docker image tag that matches your preferred configuration, or simply use *latest* to use the default setup.

The container images adhere to the following naming convention:

```
ompcluster/<image_name>:<tag>
```

In our Docker Hub repository, you can find several available images. We list below a few of the most used:

hpcbase This serves as the base image for all other containers. It includes the MPI implementation, CUDA, Mellanox drivers, etc.

runtime This image contains pre-built Clang and the stable OMPC runtime based on stable releases.

runtime-dev This image also contains pre-built Clang and the OmpCluster runtime but is sourced directly from the Git repository. Please note that this version is considered unstable and should not be used in production.

Application-specific These images (`awave-dev`, `beso-dev`, `plasma-dev`, etc) are built on the runtime image and incorporate additional libraries and tools necessary for the development of specific applications.

Once you've selected the most suitable image for your needs, you can run the application within the container using the following method:

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

Listing 5: Executing an OMPC application within a Docker container.

The flag `-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 (`https://docs.docker.com/get-started/`).

Running on Singularity is also supported as seen below:

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

Listing 6: Executing an OMPC application within a Singularity container.

For additional information, please consult the Singularity Documentation at `https://sylabs.io/guides/3.2/user-guide/`. It is worth noting that certain cluster environments may adopt the newer Singularity version known as Apptainer, which you can learn more about at `http://apptainer.org/docs/user/latest/`.

2.3.3. Slurm

You can seamlessly integrate the OmpCluster runtime with a cluster job manager, such as Slurm. Once you've compiled your code within a container, launching the job becomes as straightforward as running any MPI program. Here's an example of how to do it using Slurm:

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

Listing 7: Executing an OMPC application within a Singularity container.

Please refer to Slurm Documentation (`https://slurm.schedmd.com/quickstart.html`) for more information.

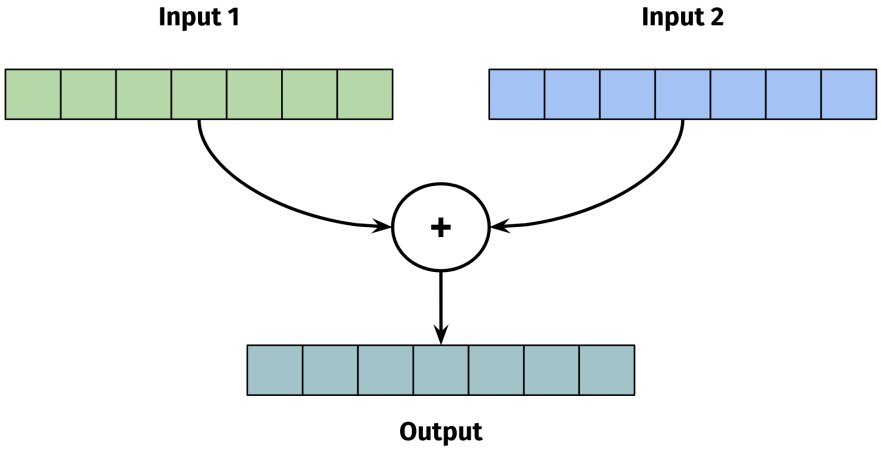

Figure 2.3. Vector sum.

2.4. Usage examples

In this section we present three applications using OMPC. The first is a trivial vector addition, the second is a vector reduction and the third is a matrix multiplication. We start with the sequential trivial implementation and improve on this implementation until we have an optimized OMPC version. For additional examples of code using OMPC see the examples page located at: `https://gitlab.com/ompcluster/ompc-examples`.

2.4.1. Example 1: Vector Addition

In this first example we want to sum two vectors, element wise. In other words given two vectors `A` and `B` of size `N`, we want to calculate a vector `C`, also of size `N` such that `C[i] = A[i] + B[i]`, for $0 \leq i \leq n$. Figure 2.3 illustrates this process.

A trivial implementation of vector addition in C++ can be seen in the code snippet below:

```
1 void vadd(int *A, int *B, int *C, size_t N) {
2   for (size_t i = 0; i < N; i++){
3     C[i] = A[i] + B[i];
4   }
5 }
```

Listing 8: Simple vector addition in C++.

Since we want to parallelize this code, we need some way of dividing the pro-

cessing in parts to be executed by each available computing node. To do that, we split the input vectors in chunks or blocks, and then, using OMPC, distribute these blocks. Listing 9 shows the code used to do that.

```
1  void vadd(int *A, int *B, int *C, size_t N) {
2    for (size_t i = 0; i < N; i++) {
3      C[i] = A[i] + B[i];
4    }
5  }
6
7  void blocked_vadd(int *in1, int *in2, int *out, int N, int BS) {
8    for(int i = 0; i < N / BS; i++) {
9      int *A  = &in1[BS * i], *B  = &in2[BS * i], *C  = &out[BS * i];
10     #pragma omp target nowait           \
11             map(to: A[:BS], B[:BS])     \
12             map(from: C[:BS])           \
13             depend(in: A[0], B[0])      \
14             depend(out: C[0])
15     vadd(A, B, C, BS);
16   }
17   #pragma omp taskwait
18 }
```

Listing 9: Blocked vector addition in C++.

The `vadd` function remains the same. However, we now also have the function `blocked_vadd`. This function breaks the vectors in blocks of size `BS`, given as input, and distributes these blocks throught each available computing node. `N / BS` tasks (and blocks) are created by the `for` on Line 8 (this function assumes `N` is a multiple of `BS`). Then on Line 9, local variables `A`, `B` and `C` are created. These are nothing more than new pointers/references to the original vectors (`in1`, `in2`, `out`), taking into account the task number and the block size. These blocks are then distributed to the working nodes according to the pragma on Lines 10-14. On Line 11 we map `BS` elements from `A` and `B` to the work node (using `to:`), and on Line 12 we copy the result back from the work nodes to the head node (using `from:`). Note the pragma `omp taskwait` on Line 17. This pragma is needed because since the tasks are executing asynchronously, we have to explicitly wait for them to finish before returning from the function.

It is also possible to use an accelerator on the remote node, and let OMPC distribute the work. For instance, one could use FPGAs to perform the same vector addition we just implemented. The support for FPGAs in OMPC is still in an early stage of development but, it already works and will be released as full implemantation in a future version of OMPC.

To do so, we use the OpenMPs `variant` pragma to annotate a function that contains an alternative implementation of the kernel. This first step is show below:

```
1 // FPGA prototype
2 void vadd_hw(int *in1, int *in2, int *out, unsigned int num);
3
4 // CPU prototype
5 #pragma omp declare variant( vadd_hw ) match( device={arch(alveo)} )
6 void vadd(int *in1, int *in2, int *out, unsigned int num);
```

Listing 10: OpenMP `variant` pragma.

The pragma on Line 5 states that if the device being used for offloading matches `alveo`[5], then the variant implementation (`vadd_hw`) should be used instead of the regular implementation of `vadd`. The code for execution on the FPGA is straightforward:

```
1 // HLS CODE
2 void vadd_hw(int *in1, int *in2, int *out, unsigned int num) {
3 #pragma HLS INTERFACE m_axi port=a bundle=gmem0
4 #pragma HLS INTERFACE m_axi port=b bundle=gmem1
5 #pragma HLS INTERFACE m_axi port=c bundle=gmem0
6 for (int i = 0; i < num; i++)
7   out[i] = in1[i] + in2[i];
8 }
```

Listing 11: Vector addition, FPGA version.

To make this work, one needs to write the kernel code in C, then add HLS directives and, finally, compile it into a xclbin file. To compile the OMPC code, we change the target to `alveo`), but the execution command line remains exactly the same:

```
1 $ clang++ -fopenmp -fopenmp-targets=alveo -fno-openmp-new-driver
   ↪  vadd.cpp -o vadd
2 $ mpirun -np $(N_PROCESSES) ./main
```

Listing 12: Executing an application using OpenMP `variant` pragma.

[5] `alveo` is the commercial name for a series of adatable accelerator cards (FPGAs) from Xilinx.

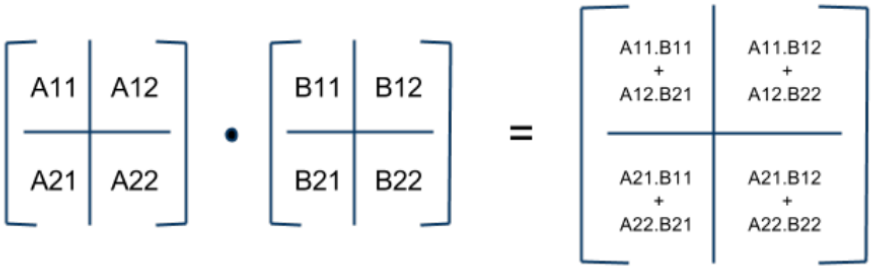

Figure 2.4. Block matrix multiplication

2.4.2. Example 3: Matrix Multiplication

To gain a clearer insight into this model, let's illustrate it with an example involving block-based matrix multiplication. In this scenario, we have two matrices, and our program's objective is to multiply them, resulting in a new matrix as the output.

Given two input matrices, `A` and `B`, each of size `N`, we would like to calculate `C` such that `C = A x B`. A first, naive, implementation in C++ would look like this:

```
1 void MatMul(int &A, int &B, int &C) {
2   for (int i = 0; i < N; ++i)
3     for (int j = 0; j < N ; ++j) {
4       C[i][j] = 0;
5       for (int k = 0; k < N ; ++k)
6         C[i][j] += A[i][k] * B[k][j];
7     }
8 }
```

Listing 13: Simple matrix multiplication in C++.

This first implementation, however, has a poor behavior regarding memory locality. Blocked (or block) matrix multiplication, is a technique used to multiply large matrices by dividing them into smaller blocks or submatrices. Instead of performing the entire matrix multiplication at once, in this technique one breaks the task into smaller, more manageable subproblems.

It's important to note that the partitioning of these factors isn't arbitrary; instead, it necessitates conformable partitions between two matrices, `A` and `B`, ensuring that all submatrix products that will be used are well-defined. Figure 2.4 illustrates this process.

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

Listing 14: Blocked matrix multiplication in C++.

In our example, the `BlockMatrix` class serves as a utility wrapper designed to partition the entire matrix into blocks, thereby capitalizing on data locality. Each block is encapsulated within a separate array. To achieve parallelization, we can compute the multiplication of each pair of blocks on distinct nodes, employing 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];
          }
      }
    }
}
```

Listing 15: Blocked parallel matrix multiplication in C++ using OMPC.

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 further optimize the code by using a second level of parallelism within each node using the `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];
          }
      }
    }
}
```

Listing 16: Blocked parallel matrix multiplication in C++ using OMPC.

However, this implementation is inneficient. The issue arises from the need to transmit all three blocks between the head and worker processes for each target task, with no opportunity for the runtime to optimize the inter-node communication.

To fix that problem, the input blocks can be sent in advance using *target enter data tasks* (`target enter data map(...) depend(...) nowait`) and the resulting blocks retrieved back using the inverse *target exit data tasks* (`target exit data map(from: ...) depend(...) nowait`) at the end. In this scenario, the OMPC scheduler and data manager gain the capability to optimize both the allocation of target tasks across worker processes and the communication among them. The resulting code would look as follows:

```
1  void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
2  #pragma omp parallel
3  #pragma omp single
4  {
5    // Maps all matrices' blocks asynchronously (as tasks).
6    for (int i = 0; i < N / BS; ++i) {
7      for (int j = 0; j < N / BS; ++j) {
8        float *BlockA = A.GetBlock(i, j);
9        #pragma omp target enter data map(to: BlockA[:BS*BS]) \
10                                       depend(out: *BlockA) nowait
11       float *BlockB = B.GetBlock(i, j);
12       #pragma omp target enter data map(to: BlockB[:BS*BS]) \
13                                       depend(out: *BlockB) nowait
14       float *BlockC = C.GetBlock(i, j);
15       #pragma omp target enter data map(to: BlockC[:BS*BS]) \
16                                       depend(out: *BlockC) nowait
17     }
18   }
19
20   for (int i = 0; i < N / BS; ++i)
21     for (int j = 0; j < N / BS; ++j) {
22       float *BlockC = C.GetBlock(i, j);
23       for (int k = 0; k < N / BS; ++k) {
24         float *BlockA = A.GetBlock(i, k);
25         float *BlockB = B.GetBlock(k,j);
26         // Submits the multiplication for the ijk-block
27         // Data is mapped implicitly and automatically moved by the
             ↪ runtime
28         #pragma omp target depend(in: *BlockA, *BlockB) \
29                                 depend(inout: *BlockC)
30         #pragma omp parallel for
31         for(int ii = 0; ii < BS; ii++)
32           for(int jj = 0; jj < BS; jj++) {
33             for(int kk = 0; kk < BS; ++kk)
34               BlockC[ii + jj * BS] += BlockA[ii + kk * BS] *
                   ↪ BlockB[kk + jj * BS];
35           }
36       }
37     }
38
39     // Removes all matrices' blocks and acquires the final result
         ↪ asynchronously.
40     for (int i = 0; i < N / BS; ++i) {
41       for (int j = 0; j < N / BS; ++j) {
42         float *BlockA = A.GetBlock(i, j);
43         #pragma omp target exit data map(release: BlockA[:BS*BS]) \
44                                      depend(inout: *BlockA) nowait
45         float *BlockB = B.GetBlock(i, j);
46         #pragma omp target exit data map(release: BlockB[:BS*BS]) \
47                                      depend(inout: *BlockB) nowait
48         float *BlockC = C.GetBlock(i, j);
49         #pragma omp target exit data map(from: BlockC[:BS*BS]) \
50                                      depend(inout: *BlockC) nowait
51       }
52     }
53
54 }
55 }
```

Listing 17: Optimized blocked parallel matrix multiplication in C++ using OMPC.

It's crucial to emphasize that each target task must use the first position of the block as a dependency. As mentioned earlier, this is a mandatory requirement for the runtime to accurately monitor the data usage and effectively manage communication among worker processes.

2.5. Profiling

Sometimes we need a deeper understanding and information about the execution of our application to optimize it and remove performance bottlenecks. In this section we provide a cookbook on how to collect, process and analyze OMPC traces.

2.5.1. Collecting a trace

The OmpCluster runtime includes native support for gathering execution traces in the JSON format. The activation is done through an environment variable:

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

Listing 18: Execution traces are enabled via an environment variable which contais the prefix for the trace file name.

OMPC can also generate a task graph in DOT format. As it is the case for the trace file, the task graph generation can also can be enabled via an environment variable:

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

Listing 19: Task graph generation can be enabled via an environment variable which contais the prefix for the DOT file names.

Once you've enabled tracing and/or task graph dump, you can proceed with running the application as usual. Upon completion of the execution, the runtime will generate timeline files with the naming convention `<file_prefix>_<process_name>.json` and two graph files named `<graph_file_prefix>_graph_<graph_number>.dot`. Each MPI process will have its corresponding JSON file. Analyzing these traces individually can be a bit challenging; hence, we offer the *OMPCBench* tool (available at `https://gitlab.com/ompcluster/ompcbench`) to simplify the analysis process.

2.5.2. Merging timelines

After a successful application execution, multiple trace files may be generated. Merging these files into a single consolidated trace can be beneficial.

To achieve this, you can begin by cloning and installing the OMPCBench tool on your machine. Follow the instructions outlined in the README file available at `https://gitlab.com/ompcluster/ompcbench/-/blob/main/README.md` to set up OMPCBench within a virtual environment. Once the installation is complete, you can merge the timelines of all the processes into a single trace by executing 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
```

Listing 20: Merging of traces using the ompcbench tool.

For further details and explanations regarding the options available with the ompcbench command, you can access the help documentation by running `ompcbench --help`, which will provide a comprehensive overview of the available options and their usage.

Upon execution, a file named `tracing.json` will be generated, allowing you to move on to the subsequent inspection stage. If the task graph file is located in the traces folder, the timeline will include task dependencies and identifiers.

2.5.3. Inspecting the trace file

The trace files are compatible with the Chrome Tracing tool of the Chrome Web Browser. To visualize your trace within a timeline, follow these steps:

1. Open the Chrome Browser and go to the URL `chrome://tracing`.

2. In the top-left corner, click “Load”.

3. Then, either open your merged timeline by selecting the appropriate file, or simply drag and drop the file into the browser window.

You should now have your application trace displayed, complete with runtime operations. An example timeline can be seen in Figure 2.5.

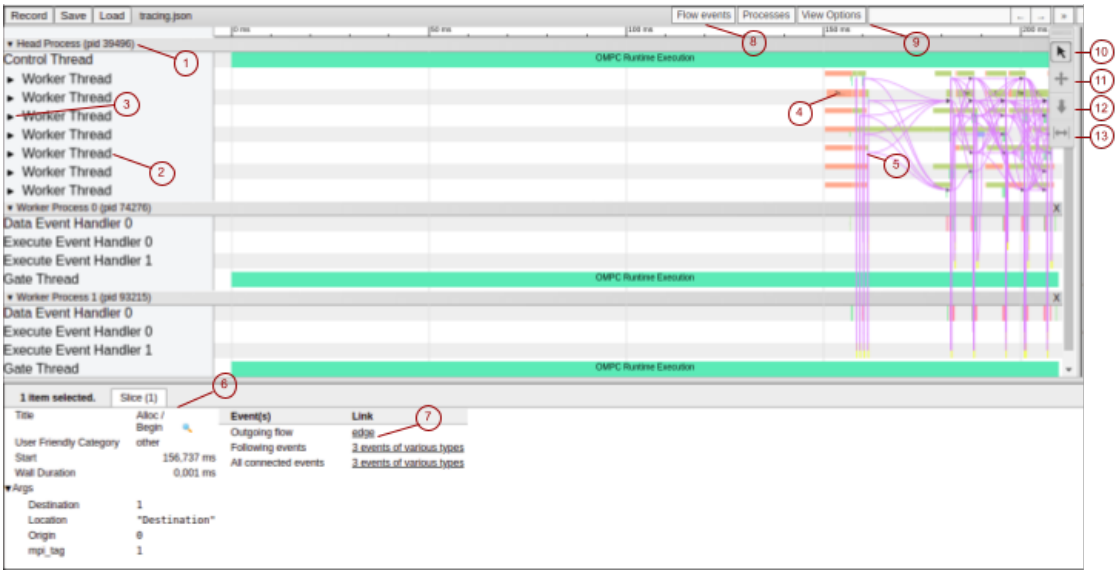

Figure 2.5. Visualization of a timeline using the `chrome://tracing` tool.

The numbers indicated in the timeline (Figure 2.5) are:

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 will show the two events linked.

8. If clicked on, shows 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 short duration (like communication events). It is possible to zoom into 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 different processes.

More information about OMPC Profiling can be seen on the paper by Pinho et al. [6] and on OMPC Documentation (`https://ompcluster.readthedocs.io/en/latest/profiling.html`).

2.6. Conclusion

Parallel and distributed computing are increasingly crucial in the era of Big Data, AI, and scientific computing. However, efficient parallel programming, especially in HPC environments, has historically been a challenging task reserved for specialists. This complexity often stems from the need to employ numerous tools and techniques simultaneously to achieve satisfactory results.

In this context, OMPC offers an alternative that simplifies the process of developing new HPC applications. OMPC is a distributed runtime based on tasks that leverages OpenMP's task programming model for parallelizing code. Unlike traditional OpenMP

tasks, OMPC distributes tasks across heterogeneous computers, allowing for the exploration of both shared-memory and distributed-memory parallelism while automatically handling all communications using MPI.

This text explores the core functionalities of OMPC and provides examples of its use. For additional information, please refer to the online documentation at `https://ompcluster.readthedocs.io/` or visit the project's website at `https://ompcluster.gitlab.io/`.

References

[1] OAR Board. Openmp application programming interface-version 5.2, 2021.

[2] Stephen P. Crago and John Paul Walters. Heterogeneous cloud computing: The way forward. *Computer*, 48(1):59–61, 2015.

[3] Pedro Henrique Di Francia Rosso and Emilio Francesquini. Ocftl: An mpi implementation-independent fault tolerance library for task-based applications. In Isidoro Gitler, Carlos Jaime Barrios Hernández, and Esteban Meneses, editors, *High Performance Computing*, pages 131–147, Cham, 2022. Springer International Publishing.

[4] Hans Werner Meuer, Erich Strohmaier, Jack Dongarra, and Horst D Simon. *The TOP500: History, Trends, and Future Directions in High Performance Computing*. Chapman & Hall/CRC, 1st edition, 2014.

[5] OpenMP. OpenMP Application Program Interface. Technical report, 2013.

[6] Vitoria Pinho, Hervé Yviquel, Marcio Machado Pereira, and Guido Araujo. Omptracing: Easy profiling of openmp programs. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 249–256, 2020.

[7] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[8] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Sousa, and Guido Araujo. The OpenMP cluster programming model. *51st International Conference on Parallel Processing Workshop Proceedings (ICPP Workshops 22)*, 2022.