Programming BlueField DPU using ODOS

OpenMP DPU Offloading Support

Muhammad Usman, Sergio Iserte, Antonio J. Peña

Contents

1 Administrivia

1.1 Learning Outcomes

By the end of this lab you will be able to;

1. Understand capabilities of OpenMP for programming BlueField DPUs.
2. Program BlueField DPUs for domain-specific applications.

1.2 Resources

- LLVM with OpenMP offloading support for BlueField
- Access to cluster with BlueField DPU devices

2 Introduction

2.1 What is BlueField DPU?

The BlueField Data Processing Unit (DPU) by NVIDIA is a versatile hardware platform designed for data center infrastructure. Combining high-performance networking and security acceleration, BlueField DPUs are frequently deployed as Smart Network Interface Cards (SmartNICs), known for their capacity to offload and expedite various data center workloads, particularly networking and security tasks, thereby lightening the load on the host CPU. Their presence in data centers contributes to improved operational efficiency and performance, enhancing overall data processing capabilities and security.

2.2 What is OpenMP Target Offloading?

OpenMP target offloading is a powerful feature that extends the capabilities of the OpenMP parallel programming model to include offloading computations to accelerators like GPUs or other coprocessors. With OpenMP target offloading, developers can leverage parallelism on both the CPU and accelerator devices, thereby enhancing the performance and efficiency of compute-intensive applications. This feature enables the seamless migration of compute-intensive tasks to accelerators while maintaining a unified codebase, making it a valuable tool for applications requiring high computational power and speed.

2.3 How does OpenMP Target Offloading for BlueField Work?

This DOCA-based ODOS solution is composed of three main modules:

- Cross-compilation: LLVM feature to generate a binary that can run in the DPU device.
- OpenMP BlueField plugin: Runtime for interacting with BlueField.
- OpenMP DOCA service: Runs on BlueField DPUs to accept and complete requests received from the host, through the OpenMP plugin.

```
+------------------------------+             +------------------------------+
|                              |             |                              |
|   +--------------------+     |             |                              |
|   |                    |     +------------>|   OpenMP BlueField Plugin    |
|   |                    |     |<------------+                              |
|   | Target Image       |     |   OpenMP    |                              |
|   +--------------------+     |   Runtime   |                              |
|  Host Image                  |             |                              |
+------------------------------+             +-------------+-^--------------+
                                                           | | DOCA
                                                           | | Comm
                                                           | | Channel
                                             +-------------v-+--------------+
                                             |                              |
                                             |                              |
                                             |                              |
                                             |     OpenMP DOCA Service      |
                                             |                              |
                                             |                              |
                                             |                              |
                                             +------------------------------+
```

Figure 1: Methodology of OpenMP Target Offloading for BlueField DPU devices

A fat binary with embedded cross-compile code for BlueField (ARM architecture) is provided to OpenMP plugin by OpenMP runtime. The runtime then uses DOCA Comm Channel module to communicate with the DOCA OpenMP Service running on DPU. The service receives requests and run the tasks as required by the runtime through plugin. These tasks include loading image, allocation and copying of data and running target codes.

3 Building ODOS-enabled LLVM

Precompiled binaries will be shared in cluster. The following instructions are included herewith as a reference for compilation later.

3.1 Setup LLVM Build

Setup path for build directory and installation directory

```
$ export PREFIX=$HOME/opt/llvm_bf/
$ export BUILDDIR=$HOME/tmp/build_llvm_bf/
```

Setup the build system using `cmake` command

```
$ cmake -S llvm-project/llvm                                  \
        -B llvm_objdir                                        \
        -DCMAKE_INSTALL_PREFIX="$PREFIX"                      \
        -DCMAKE_BUILD_TYPE=Release                            \
        -DCMAKE_C_COMPILER="`which gcc`"                      \
        -DCMAKE_CXX_COMPILER="`which g++`"                    \
        -DCMAKE_EXE_LINKER_FLAGS="$LDFLAGS"                   \
        -DLLVM_BUILD_UTILS=OFF                                \
        -DLLVM_ENABLE_PROJECTS="clang"                        \
        -DGCC_INSTALL_PREFIX="/usr"                           \
        -DCLANG_ENABLE_ARCMT=OFF                              \
        -DCLANG_ENABLE_STATIC_ANALYZER=OFF
```

Note: the above command requires `ninja` to build LLVM

3.2 Compile LLVM Build

Compile using `ninja` build system

```
$ ninja -C $BUILDDIR install
```

3.3 Setup OpenMP Build

Setup path for build directory and installation directory

```
$ export PREFIX=$HOME/opt/llvm_bf/
$ export BUILDDIR=$HOME/tmp/build_llvm_openmp_bf/
```

Setup the build system using `cmake` command:

```
$ cmake -S llvm-project/openmp                                \
        -B openmp_objdir                                       \
        -DLLVM_ROOT="$PREFIX"                                  \
        -DCMAKE_INSTALL_PREFIX="$PREFIX"                       \
        -DCMAKE_BUILD_TYPE=Release                             \
        -DCMAKE_C_COMPILER="$PREFIX/bin/clang"                 \
        -DCMAKE_CXX_COMPILER="$PREFIX/bin/clang++"             \
        -DCMAKE_EXE_LINKER_FLAGS="$LDFLAGS"                    \
        -DCMAKE_EXPORT_COMPILE_COMMANDS=ON                     \
        -DOPENMP_ENABLE_LIBOMPTARGET_PROFILING=OFF             \
        -DLIBOMP_HAVE_OMPT_SUPPORT=OFF                         \
        -DLIBOMP_INSTALL_ALIASES=OFF                           \
        -DLIBOMPTARGET_ENABLE_DEBUG=OFF                        \
        -DLLVM_BUILD_TOOLS=ON
```

3.4 Compile OpenMP Build

Compile using `ninja` build system

```
$ ninja -C $BUILDDIR install
```

3.5 Check BlueField LLVM/OpenMP Target Info

We can check if BlueField is enabled in our installation of LLVM/OpenMP by using the following command:

```
$ llvm-omp-device-info
...
Device (4):
    BlueField DPU device
        iface     :  ib0
        ib dev    :  mlx5_2
        doca id   :  2
        pci addr  :  42:00:0
        comm channel:
            max msg size            :   4080
            max send    queue size  :   8192
            max receive queue size  :   8192
...
```

4 Building and Running ODOS-enabled Applications

4.1 Loading modules

4.1.1 Modules on Host Side

```
$ module load ODOS
```

4.1.2 Modules on DPU Side

```
$ module load ODOS
```

4.2 Compiling

```
$ clang -fopenmp -fopenmp-targets=aarch64-unknown-linux \
        app.c -o app
```

4.3 Running

4.3.1 DOCA OpenMP Service in DPU

On the DPU side, run the DOCA OpenMP service:

```
$ doca-omp-service
```

4.3.2 Application in Host

On host side, run the compiled code

```
$ ./app
```

Note: make sure that DOCA OpenMP service is running on the DPU.

5 Labs

5.1 Setup

5.1.1 Shells

Open up two terminals to access both host and DPU. Access HPC Advisory Council Clusters login node on both.

```
$ ssh username@gw.hpcadvisorycouncil.com
```

5.1.2 Node Allocation

```
$ salloc -N 2 -p thor --time 03:00:00 -w thor0XX,thorbf3aXX
```

5.1.3 Accessing Nodes

Access host node

```
$ ssh thor0XX
```

and access DPU node

```
$ ssh thorbf3a0XX
```

5.1.4 Loading Modules

On both sides, load the ODOS module and cmake

```
$ module load cmake ODOS
```

5.1.5 Compile

A script exists to compile all the labs.

```
$ sh compile.sh
```

5.1.6 Run

Run the DOCA OpenMP Service on DPU side

```
$ doca-omp-service
```

Run the application on host side

```
$ ./task_a/build/hello
```

Name of the binary generated can be different based on the task.
Moreover, task c and d require more changes and will be discussed in the relevant section.

5.1.7 Clean

```
$ sh compile.sh
```

5.2 Automated Setup

There are 2 scripts provided for automated execution of the tasks. Following is the listing of both scripts:

5.2.1 Slurm Script for Service

How to Run

1. Set the available node using `vim` command.
2. Run the command:

```
$ sbatch --export=NONE ./run_service.sh

```

Scipt Snippet

```
#!/bin/bash -l

#SBATCH -p thor
#SBATCH -w thorbf3a030
#SBATCH -o service-%j.out

# sbatch ./run_service.sh

# the following line is for task c
if [ -n "$LIB_" ]; then
        echo "lib path: $LIB_"
        export LD_LIBRARY_PATH=$LIB_:$LD_LIBRARY_PATH
fi
MODULEPATH=/global/home/groups/rdmaworkshop/doca-omp-service/
    modulefiles/
```

```
module load doca $MODULEPATH/doca-omp-service.aarch64

doca-omp-service
```

5.2.2 Slurm Script for Task

How to Run

1. Set the available node using `vim` command.

2. Run the command and provide the task binary in the argument:

   ```
   $ sbatch --export=NONE ./run_task.sh ./task_a/hello
   ```

   pre-compile task binaries are provided in the repository

Script Snippet

```
#!/bin/bash -l

#SBATCH -p thor
#SBATCH -w thor030
#SBATCH -o task-%j.out

# sbatch ./run_task.sh ./task_a/hello

# the following line is for task c
if [ -n "$LIB_" ]; then
        echo "lib path: $LIB_"
        echo "the path includes following libs:"
        ls $LIB_/*.so
        export LD_LIBRARY_PATH=$LIB_:$LD_LIBRARY_PATH
fi

module load doca

COMMAND_=$@

echo "cmd : $COMMAND_"

$COMMAND_
```

5.3 Task A: Hello World

The task offloads code block to BlueField DPU. The example also demostrates that DPU is capable of running linux systemcalls.

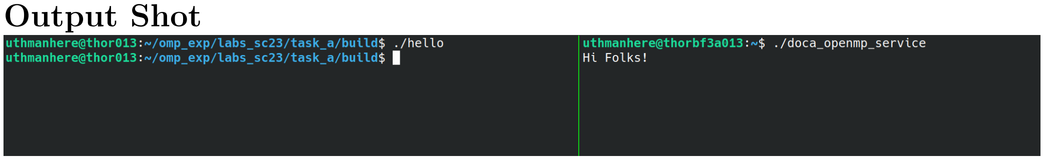
Code Snippet

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

int main()
{
#pragma omp target
        puts("Hi Folks!");

        return 0;
}
```

Compilation Script

```
$ $LLVM/bin/clang                                    \
        -fopenmp -fopenmp-targets=aarch64-unknown-linux \
        hello.c -o hello
```

Output Shot



Running using Script

1. Set the available node in `run_service.sh` using `vim` command.

2. Run the command:

```
$ sbatch --export=NONE ./run_service.sh

```

3. Set the available node in `run_task.sh` using `vim` command.

4. Run the command and provide the task binary in the argument:

```
$ sbatch --export=NONE ./run_task.sh ./task_a/hello

```

5. Observe the output in `task.out` file (representing host node) and `service.out` file (representing dpu node).

5.4 Task B: OpenMP with Target Code

The task demonstrates that offloaded code can also use OpenMP features and hece make use of much more features of underlying hardware inside DPU.

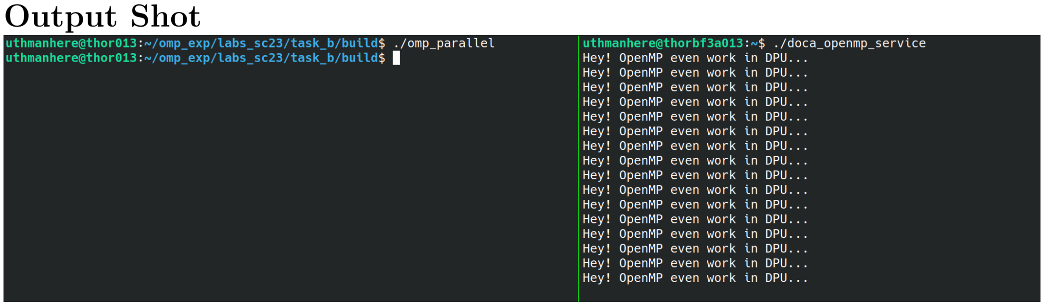
Code Snippet

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

int main()
{
#pragma omp target
#pragma omp parallel
        puts("Hey! OpenMP even work in DPU...");

        return 0;
}
```

Compilation Script

```
$ $LLVM/bin/clang                                          \
        -fopenmp -fopenmp-targets=aarch64-unknown-linux \
        omp_parallel.c -o omp_parallel
```

Output Shot



Running using Script

1. Set the available node in `run_service.sh` using `vim` command.
2. Run the command:

```
$ sbatch  --export=NONE ./run_service.sh
```

3. Set the available node in `run_task.sh` using `vim` command.

4. Run the command and provide the task binary in the argument:

```
$ sbatch --export=NONE ./run_task.sh ./task_b/omp_parallel
```

5. Observe the output in `task.out` file (representing host node) and `service.out` file (representing dpu node).

5.5 Task C: Shared Libraries

The task demonstrates the use of shared libraries with BlueField DPU. It consists of multiple files as:

- Library header file - `log.h`
- Library source code file - `log.c`
- Application source code - `shared.c`

We generate 3 files from these files to be able to use them with LLVM/OpenMP Target Offloading infrastructure:

- Shared object - `liblog_x86.so`
- Cross-compile shared object - `liblog_aarch64.so`
- Application binary - `shared`

Setup Library Path Setup the path of the library that has been compiled on both host and dpu sides

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/task_c/build/
```

Source Code Library header file `log.h`

```
void printify();
```

Source Code Library source code file `log.c`

```
#include "log.h"

#include <stdio.h>

void printify()
{
    puts("Hello from the other side (shared object actually)");
}
```

Compilation script Script for native and cross-compiled shared objects

```
$ $LLVM/bin/clang log.c -shared          \
        -o liblog_x86.so
$ $LLVM/bin/clang log.c -shared          \
        -target aarch64-unknown-linux \
        -o liblog_aarch64.so
```

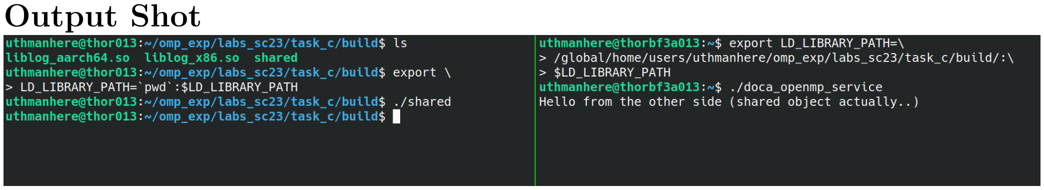
Source Code Application

```
#include <omp.h>
#include "log.h"
#include <stdio.h>

int main()
{
#pragma omp target
        printify();

        return 0;
}
```

Compilation script for application code

```
$ $LLVM/bin/clang -fopenmp \
        -fopenmp-targets=aarch64-unknown-linux   \
        shared.c -o shared                       \
        -L. -llog_x86                            \
        -Wl,--device-linker=-L.,--device-linker=-llog_aarch64
```

Output Shot



Running using Script

1. export the environment variable `LIB_` to refer to the directory containng shared objects. It is used to add the directory to `LD_LIBRARY_PATH` such that the binary can find these at runtme.

```
$ export LIB_=/path/to/shared/object/task_c/build/

```

2. Set the available node in `run_service.sh` using `vim` command.

3. Run the command:

```
$ sbatch --export=NONE ./run_service.sh
```

4. Set the available node in `run_task.sh` using `vim` command.

5. Run the command and provide the task binary in the argument:

```
$ sbatch --export=NONE ./run_task.sh ./task_c/build/shared
```

6. Observe the output in `task.out` file (representing host node) and `service.out` file (representing dpu node).

5.6 Task D: BlueField as a Network Accelerator

This task demonstrate capabilities of BlueField DPU as a network preproesor or network post processor. It employs a TCP/IP application with performs an asynchronous pingpong with increment on DPU, which CPU is free to run other compute-centric workloads.

Source Code Main snippet contains a target block for DPU offloading. The target block contains initialization of a TCP/IP network. Then it proceeds to run a ping pong application to run asychronous transfer and compute capabilities of the code.

```
int main(int argc, char *argv[])
{
        int fd;
        char mode;
        char *ip;

        if (argc < 2) {
                mode = 's';
        } else if (argc > 2 && argv[1][0] == 'c') {
                mode = 'c';
                ip   = argv[2]
        } else {
                err(-EINVAL,                \
                    "invalid argument." \
                    " Leave empty for server. c for client");
                return -EINVAL;
        }

#pragma omp target map(to:ip[0:strlen(ip)]) nowait
{
        fd = tcp_init(mode, ip);

        if (fd < 0) {
                err(-EAGAIN, "init failed.");
                goto err_out;
        }

        puts("connection established.");

        ping_pong(fd, mode);

        close(fd);
```

```
err_out:
}

#pragma omp taskwait

        return 0;
}
```

Source code Pingpong snippet first sends from client to server. The server increments the messages and sends it back the client. The client does the same i.e. increment the integer again and send it back to to server. It continues for pre-specified number of iterations.

```
void ping_pong(int fd, char mode)
{
        int i, m;
        m = 0;

        for (i = 0; i < _ITERATIONS_; ++i) {
                if (mode == 'c') {
                        printf("to  server > %02d\n", m);
                        send(fd, &m, sizeof(m), 0);
                        recv(fd, &m, sizeof(m), 0);
                        ++m;
                } else {
                        recv(fd, &m, sizeof(m), 0);
                        ++m;
                        printf("to  client > %02d\n", m);
                        send(fd, &m, sizeof(m), 0);
                }
        }

}
```

Running

- Setup 4 nodes: 2 hosts and their 2 corresponding DPUs
- Arbitrarily choose one pair as server and the other pair as client
- Compile using the`compile.sh` script

- Run `doca-omp-service` on both DPUs
- On server-side host, run:

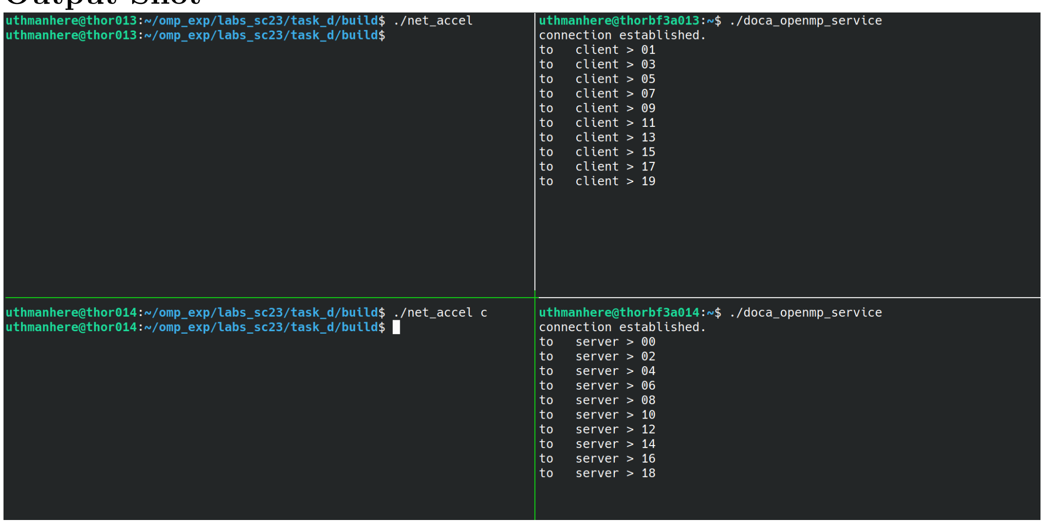
```
$ ./net_accel

```

- On client-side host, run:

```
$ ./net_accel c 192.168.3.2XX

```

where `XX` represent the last 2 digits in the name of server node.

Output Shot



Running using Script

1. Set the available node in `run_service.sh` using `vim` command for server service.
2. Run the command:

```
$ sbatch --export=NONE ./run_service.sh

```

3. Set the available node in `run_service.sh` using `vim` command for client service.

4. Run the command:

```
$ sbatch --export=NONE ./run_service.sh
```

5. Set the available node in `run_task.sh` using `vim` command for server task.

6. Run the command and provide the task binary in the argument:

```
$ sbatch --export=NONE ./run_task.sh ./task_d/net_accel
```

7. Set the available node in `run_task.sh` using `vim` command for client task.

8. Run the command and provide the task binary in the argument alongwith client mode and server IP address:

```
$ sbatch --export=NONE ./run_task.sh ./task_d/net_accel c
    192.168.2.2XX
```

where `XX` represent the last 2 digits in the name of server node (as `18` in `thorbf3a018`).

9. Observe the output in `task.out` file (representing host node) and `service.out` file (representing dpu node).

Complete Source Code

```
#include <err.h>
#include <errno.h>

#include <omp.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>

#include <arpa/inet.h>

#define _PORT_ 3310
```

```c
#define _ITERATIONS_ (10)

int tcp_init(char MODE, char *server_ip)
{
	int fd, ret;
	int enable_reuse = 1;
	socklen_t socklen_;
	struct sockaddr_in sockaddr_;

	server_ip[strlen(server_ip)-1] = '\0';
	fd = socket(AF_INET, SOCK_STREAM, 0);
	if (fd == -1) {
		err(-EAGAIN, "failed to crate socket.");
		return -EAGAIN;
	}
	ret = setsockopt(fd,               \
		SOL_SOCKET, SO_REUSEADDR, \
		&enable_reuse, sizeof(enable_reuse));
	if (ret == -1) {
		err(-EAGAIN, "failed to set socket to use.");
		return -EAGAIN;
	}


	sockaddr_.sin_family = AF_INET;
	sockaddr_.sin_port = htons(_PORT_);

	if (MODE == 'c') {
		sockaddr_.sin_addr.s_addr = \
			inet_addr(_SERVER_IP_);
		ret = connect(fd,                          \
			(struct sockaddr *)&sockaddr_, \
			sizeof(sockaddr_));
		if (ret == -1) {
			err(-EAGAIN, \
			    "failed to connect socket.");
			return -EAGAIN;
		}
	} else {

		sockaddr_.sin_addr.s_addr = INADDR_ANY;

		ret = bind(fd,                              \
			(struct sockaddr *) &sockaddr_, \
			sizeof(sockaddr_));
```

```
                if (ret == -1) {
                        err(-EAGAIN, "failed to bind socket.");
                        return -EAGAIN;
                }

                ret = listen(fd, 1);
                if (ret == -1) {
                        err(-EAGAIN, \
                            "failed to listen socket.");
                        return -EAGAIN;
                }

                socklen_ = sizeof(sockaddr_);
                fd = accept(fd,                         \
                        (struct sockaddr *)&sockaddr_, \
                        &socklen_);
                if (fd == -1) {
                        err(-EAGAIN, \
                            "failed to listen socket.");
                        return -EAGAIN;
                }
        }

        return fd;
}

void ping_pong(int fd, char mode)
{
        int i, m;
        m = 0;

        for (i = 0; i < _ITERATIONS_; ++i) {
                if (mode == 'c') {
                        printf("to  server > %02d\n", m);
                        send(fd, &m, sizeof(m), 0);
                        recv(fd, &m, sizeof(m), 0);
                        ++m;
                } else {
                        recv(fd, &m, sizeof(m), 0);
                        ++m;
                        printf("to  client > %02d\n", m);
                        send(fd, &m, sizeof(m), 0);
                }
        }

```

```
}

int main(int argc, char *argv[])
{
        int fd;
        char mode;
        char *ip

        if (argc < 2) {
                mode = 's';
        } else if (argc > 2 && argv[1][0] == 'c') {
                mode = 'c';
                ip   = argv[2]
        } else {
                err(-EINVAL,             \
                    "invalid argument." \
                    " Leave empty for server. c for client");
                return -EINVAL;
        }

#pragma omp target map(to:ip[0:strlen(ip)]) nowait
{
        fd = tcp_init(mode, ip);

        if (fd < 0) {
                err(-EAGAIN, "init failed.");
                goto err_out;
        }

        puts("connection established.");

        ping_pong(fd, mode);

        close(fd);
err_out:
}

#pragma omp taskwait

        return 0;
}
```

6 MPI Compatibility

We are actively working on MPI compatibility. **Coming soon...**

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank; // Process rank
    int size; // Total number of processes

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) {
        printf("requires at least two processes.\n");
        MPI_Finalize();
        return 1;
    }

#pragma omp target
{
    int data = 0;

    if (rank == 0) {
        data = 42; // Data to be sent from rank 0
        MPI_Send(&data, 1, MPI_INT, \
                    1, 0, MPI_COMM_WORLD);
        printf("Process %d sent data: %d\n", rank, data);
    } else if (rank == 1) {
        MPI_Recv(&data, 1, MPI_INT, \
                    0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process %d received data: %d\n", rank, data);
    }
}
    MPI_Finalize();

    return 0;
}
```

7 Access ODOS

Get access to the ODOS framework in AccelCom website:
bsc.es/discover-bsc/organisation/scientific-structure/accelerators-and-communications-hpc/team-software
.

8 Contact Us

Don't hesitate contacting us at accelcom@bsc.es for further information.