Integrating TAMPI and ParaStationMPI NAM windows

Table of contents:

Quick Overview

Heat Benchmark

In this section, we exemplify the use of TAMPI and NAM windows through the Heat benchmark. We use an iterative Gauss-Seidel method to solve the Heat equation, which is a parabolic partial differential equation that describes the distribution of heat in a given region over time. This benchmark simulates the heat diffusion on a 2D matrix of floating-point elements during multiple timesteps. The 2D matrix is logically divided into 2D blocks and may have multiple rows and columns of blocks. The computation of an element at position `M[r][c]` in the timestep `t` depends on the value of the top and left elements (`M[r-1][c]` and `M[r][c-1]`) computed in the current timestep `t`, and the right and bottom elements (`M[r][c+1]` and `M[r+1][c]`) from the previous timestep `t-1`. We can extrapolate this logic in the context of blocks so that a block has a dependency on the computation of its adjacent blocks. Notice that the computation of blocks in a diagonal is fully concurrent because there is no dependency between them.

There are three different MPI versions, and all of them distribute the 2D matrix across ranks assigning consecutive rows of blocks to each MPI rank. Note that the matrix is distributed by blocks vertically but not horizontally. Therefore, an MPI rank has two neighboring ranks: one above and another below. The exceptions are the first and last ranks since they have a single neighbor. This distribution requires the neighboring ranks to exchange the external rows (halos) from their boundary blocks in order to compute their local blocks in each timestep.

This benchmark is publicly available in the ?https://pm.bsc.es/gitlab/DEEP-EST/apps/Heat repository. The first version is based on an MPI-only parallelization, while the other two are hybrid MPI+OmpSs-2 leveraging tasks and the TAMPI library. We briefly describe each one below:

- `01.heat_mpi.bin`: A straightforward **MPI-only** implementation using **blocking MPI primitives** (MPI_Send and MPI_Recv) to send and receive the halo rows. The computation of blocks and exchange of halos inside each rank is completely sequential.
- `02.heat_itampi_ompss2_tasks.bin`: A hybrid **MPI+OmpSs-2** version leveraging **TAMPI** that performs both computation and communications using **tasks** with **data dependencies**. It instantiates a task to compute each of the blocks inside each rank and for each of the timesteps. It also creates a sending and receiving tasks to exchange the block halo rows for each of the boundary blocks. The execution of tasks follows a **data-flow model** because tasks declare the dependencies on the data they read/modify. Moreover, communication tasks call **non-blocking MPI primitives** and leverage the **non-blocking mechanism of TAMPI** (`TAMPI_Iwait`), so communications are fully non-blocking and **asynchronous** from the user point of view. Communication tasks issue non-blocking communications that are transparently managed and periodically checked by TAMPI. These tasks do not explicitly wait for their communication, but they delay their completion (asynchronously) until their MPI communications finish.
- `03.heat_tampirma_ompss2_tasks.bin`: An implementation similar to `02.heat_itampi_ompss2_tasks.bin` but using **MPI RMA operations** (`MPI_Put`) to exchange the block halo rows. This program leverages the MPI active target RMA communication using the **MPI window fences** to open/close RMA access epochs. It uses the **TAMPI** library and the new integration for the `MPI_Win_ifence` synchronization function. In this way, we use `TAMPI_Iwait` to bind the completion of a communication task to the finalization of a `MPI_Win_ifence`. Therefore, the opening/closing of RMA access epochs is completely non-blocking and asynchronous from the user point of view. We assume the calls to `MPI_Put` are non-blocking. Finally, as an optimization, we register **multiple MPI RMA** windows for each rank to allow **concurrent** communications through the different RMA windows. Each RMA window holds a part of the halo row that may belong to multiple logical blocks. Each communication task exchanges the part of the halo row assigned to a single MPI window.

Using NAM in Heat benchmark

In this benchmark, we use the NAM memory to periodically save the computed matrix. The idea is to save the different states (snapshots) of the matrix during the execution in a persistent NAM memory region. Then, another program could retrieve all the matrix states, process them and produce a GIF animation showing the evolution of the heat during the whole execution. Notice that we cannot use simple RAM memory for that since the matrix could be huge and we may want to store tens of matrix snapshots. We also want the possibility of storing it in a persistent way, so other programs can process the stored data. Moreover, the memory should be easily accessible by the multiple MPI ranks or their tasks in parallel. The NAM memory fulfills all these conditions and ParaStationMPI allows accessing NAM regions through standard MPI RMA operations.

During the execution of the application and every few timesteps (specified by the user), the benchmark saves the whole matrix into a specific NAM subregion. Each timestep saving a matrix snapshot uses a distinct NAM subregion. These subregions are placed one after the other, consecutively, but without overlapping. Thus, the total size of the NAM region is the size of the whole matrix multiplied by the number of times the matrix will be saved. However, the NAM memory region is allocated using the Managed Contiguous layout (`psnam_structure_managed_contiguous`). This means that the rank 0 allocates the whole region but each rank acquires a consecutive memory subregion where it will store its blocks' data for all the spanshots. For instance, the NAM allocation will first have all the space for storing all snapshots of the blocks from rank 0, followed by the space for all snapshots of blocks from rank 1, and so on. Notice that the NAM subregions are addressed by the rank it belongs to, simplifying the task of saving and retrieving the snapshots.

When there is a timestep that requires a snapshot, the application instantiates multiple tasks that save the matrix data into the corresponding NAM subregion. Each MPI rank creates a task for saving the data of each matrix block into the NAM subregion. These communication tasks do not have any data dependency between them, so they can run in parallel writing data to the NAM region using regular `MPI_Put`. Ranks only write to the subregions that belong to themselves, never in other ranks' subregions. Even so, all `MPI_Put` calls must be done inside an RMA access epoch, so there must be one fence call before all the `MPI_Put` calls and another one after them to close the epoch for each of the timesteps with snapshot. Here is where we use the new function `MPI_Win_ifence` together with the TAMPI non-blocking support. In this way, we can fully taskify both synchronization and writing of the NAM window, keeping the data-flow model, and without having to stop the parallelism (e.g., with a `taskwait`) to perform the snapshots. Thanks to the task data dependencies and TAMPI, we cleanly include the snapshots in the application's data-flow execution, as regular communication tasks with dependencies.

The following pseudo-code shows how the saving of snapshots work in `02.heat_itampi_ompss2_tasks.bin`:

```
void solve() {
   int namSnapshotFreq = ...;
   int namSnapshotId = 0;

   for (t = 1; t <= timesteps; ++t) {
       // Computation and communication tasks declaring
       // dependencies on the blocks they process
       gaussSeidelSolver(..all blocks in current rank..);

       if (t % namSnapshotFreq == 0) {
           namSaveMatrix(namSnapshotId, namWindow, ...);
           ++namSnapshotId;
       }
   }
   #pragma oss taskwait
}
```

The function above is the main procedure that executes all Heat application's timesteps applying the Gauss-Seidel method. This function is executed by all MPI ranks and each one works with their corresponding blocks from the matrix. In each timestep, the `gaussSeidelSolver` function instantiates all the computation and communication tasks that process the rank's blocks and exchanges the halo rows with the neighboring ranks. These tasks declare the proper input/output dependencies on the blocks they are reading/writing. Every some timesteps, the algorithm calls `namSaveMatrix` in order to perform a snapshot of the data computed after computing that timestep. Notice that `namSaveMatrix` will have to instantiate tasks with input dependencies on the blocks in order to perform the snapshot in the correct moment of the execution. Notice also that each snapshot is identified by the `namSnapshotId`, which will be used to know where the snapshot data should stored inside the NAM region. After all tasks from all timesteps have been instantiated, the application calls a taskwait to wait for the completion of all computation, communication and snapshot tasks.

```
void namSaveMatrix(int namSnapshotId, MPI_Win namWindow, ...) {
   // Compute snapshot offset inside NAM region
   int snapshotOffset = namSnapshotId*sizeof(..all blocks in current rank..);

   // Open RMA access epoch to write the NAM window for this timestep
   #pragma oss task in(..all blocks in current rank..) inout(namWindow)
   {
       MPI_Request request;
       MPI_Win_ifence(namWindow, 0, &request);
       TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
   }

   // Write all blocks from the current rank to NAM subregions concurrently
   for (B : all blocks in current rank) {
       #pragma oss task in(..block B..) in(namWindow)
       {
           MPI_Put(/* source data */   ..block B..,
                   /* target rank */   currentRank,
                   /* target offset */ snapshotOffset + B,
                   /* target window */ namWindow);
```

```
        }
    }

    // Close RMA access epoch to write the NAM window for this timestep
    #pragma oss task in(..all blocks in current rank..) inout(namWindow)
    {
        MPI_Request request;
        MPI_Win_ifence(namWindow, 0, &request);
        TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
    }
}
```

The function above is the one called periodically from the main function. This function instantiates the tasks that will perform the snapshot of the current rank's blocks into the corresponding NAM memory subregion. The first step is to compute the offset of the current snapshot inside the NAM region using the snapshot identifier. Before writing to the NAM window, the application must ensure that a MPI RMA access epoch has been opened in that window. That is what the first task is doing. After all blocks are ready to be read (see the task dependencies), the task can run and execute an `MPI_Win_ifence` to start the opening of the epoch generating an MPI request, and after that, the task binds its completion to the finalization of the request by calling `TAMPI_Iwait`. This last call is non-blocking and asynchronous, so the fence operation may not be completed after returning. The task can finish its execution but it will not complete until the fence operation finishes. Once it finishes, TAMPI will automatically complete the task and make the successor tasks ready. The successor tasks of the fence task are the ones that perform the actual writing of data to the NAM memory calling `MPI_Put`. All blocks can be saved in the NAM memory in parallel through different tasks. The source of the `MPI_Put` is the block itself (in regular RAM memory) and the destination is the place where the block must be written in the NAM memory. After all writer tasks have finished, the task responsible for closing the MPI RMA access epoch in the NAM window will be able to start. This one will behave similarly to the opening task.

Notice that all tasks declare the proper dependencies on the matrix blocks and the NAM window to guarantee the correct order of execution. Thanks to these data dependencies and the TAMPI non-blocking feature, we can cleanly add the execution of the snapshots into the task graph, to be executed asynchronously, and being naturally interleaved with the other computation and communication tasks. Finally, it is worth noting that the writing of blocks to the NAM memory is done in parallel, trying to efficiently utilize the CPU and network resources of the machine.

Requirements

The requirements of this application are shown in the following lists. The main requirements are:

- The **GNU** or **Intel®** Compiler Collection.
- The **ParaStationMPI** installation supporting **multi-threading** and featuring the **libNAM** integration that allows access to NAM memory regions through MPI RMA windows.
- The **Task-Aware MPI (TAMPI)** library which defines a clean **interoperability** mechanism for MPI and OpenMP/OmpSs-2 tasks. It supports both blocking and non-blocking MPI operations by providing two different interoperability mechanisms. Downloads and more information at ?https://github.com/bsc-pm/tampi.
- The **OmpSs-2** model which is the second generation of the **OmpSs** programming model. It is a **task-based** programming model originated from the ideas of the OpenMP and StarSs programming models. The specification and user-guide are available at ?https://pm.bsc.es/ompss-2-docs/spec/ and ?https://pm.bsc.es/ompss-2-docs/user-guide/, respectively. OmpSs-2 requires both **Mercurium** and **Nanos6** tools. Mercurium is a source-to-source compiler which provides the necessary support for transforming the high-level directives into a parallelized version of the application. The Nanos6 runtime system provides the services to manage all the parallelism in the application (e.g., task creation, synchronization, scheduling, etc.). Downloads at ?https://github.com/bsc-pm.
- The NAM software allowing access to NAM memory.

Building & Executing on DEEP

The instructions to build and execute the Heat benchmark with NAM checkpointing will appear here soon.

References

- ?https://pm.bsc.es/ompss-2
- ?https://github.com/bsc-pm
- ?https://github.com/bsc-pm/tampi
- ?https://en.wikipedia.org/wiki/Gauss-Seidel_method
- ?https://pm.bsc.es/gitlab/DEEP-EST/apps/Heat