Wikiprint Book

Title: Accessing Network Attached Memory (NAM) from Tasks using TAMPI

Subject: DEEP - Public/User_Guide/TAMPI_NAM

Version: 25

Date: 02.05.2024 10:31:11

Table of Contents

Accessing Network Attached Memory (NAM) from Tasks using TAMPI

Table of contents:

Quick Overview

In this page we show how to access Network Attached Memory (NAM) from tasks in hybrid task-based MPI+OmpSs-2 applications. The NAM memory regions can be accessed through MPI RMA windows thanks to the extensions on the RMA model interface provided by ParaStation MPI. The applications can allocate MPI windows that are linked to NAM memory regions and perform RMA operations on those NAM regions, such as `MPI_Put` and `MPI_Get`. Following the standard MPI RMA fence synchronization mode, applications can open access epochs on those regions by calling `MPI_Win_fence`, then call the desired RMA operations, and finally, close the epoch with another call to `MPI_Win_fence`. Moreover, this NAM support has been integrated into the OmpSs-2 tasking model, allowing accessing NAM regions from OmpSs-2 tasks efficiently and safely through the Task-Aware MPI (TAMPI) library.

In the following sections, we describe the TAMPI library and how hybrid task-based applications should access NAM memory regions. Finally, we show the Heat equation benchmark as an example of using this support to save periodic snapshots of the computed matrix in a NAM allocation.

Task-Aware MPI (TAMPI)

The most common practice in hybrid applications is combining MPI and OpenMP in a fork-join approach, where computation phases are parallelized using OpenMP and communication phases are serialized. This is a simple approach but it usually does not scale as well as the MPI-only parallelizations. A more advanced technique is the taskification of both computation and communications phases using the task data dependencies to connect both task types and guarantee a correct execution order. This strategy allows applications to parallelize their workloads following the data-flow paradigm principles, where communications are integrated into the flow of tasks. However, MPI and OpenMP were not designed to be combined efficiently since MPI only defines threading levels to safely interact with other parallel programming models.

The Task-Aware MPI library was recently proposed to overcome these limitations. TAMPI allows the safe and efficient taskification of MPI communications, including all blocking, non-blocking, point-to-point, and collective operations. The TAMPI library allows calling MPI communication operations from within tasks. On the one hand, blocking MPI operations pause the calling task until the operation completes, but in the meantime, the tasking runtime system may execute other tasks.

On the other hand, non-blocking operations can be bound to a task through two API functions named `TAMPI_Iwait` and `TAMPI_Iwaitall`, which have the same parameters as the standard `MPI_Wait` and `MPI_Waitall`, respectively. These two TAMPI functions are non-blocking and asynchronous and can be used to bind the completion of the calling task to the finalization of the MPI requests passed as parameters. The calling task will not release its dependencies until (1) the task finishes its execution and (2) all bound MPI operations complete. Its successor tasks will be ready to execute only once its dependencies are released. We must highlight that a task can call those TAMPI functions several times, binding multiple MPI requests during its execution. Moreover, TAMPI offers a wrapper function for each non-blocking MPI operation, which performs the standard non-blocking operation, and then, it automatically calls `TAMPI_Iwait` with the resulting MPI request. The `TAMPI_Isend` and `TAMPI_Irecv` are examples of these wrappers.

More information about the TAMPI library and its code is available at ?https://github.com/bsc-pm/tampi.

Accessing NAM through ParaStationMPI and TAMPI

The main idea is to allow tasks to access data stored in NAM regions efficiently and potentially in parallel, e.g. using several tasks to put/get data to/from the NAM. The mechanism to access NAM regions is provided by the ParaStation MPI and is based on the MPI RMA model. ParaStation MPI allows allocating NAM regions as MPI RMA windows, so that they can be accessed remotely by the ranks that participate in those windows using the standard `MPI_Put` and `MPI_Get` RMA operations. This support is based on the fence synchronization mode of the MPI RMA model. This mode works as follows: (1) all ranks participating in a given window have to open an access epoch on that window calling `MPI_Win_fence`, (2) they perform the desired RMA operations on the NAM window, and (3) they close the epoch with another call to `MPI_Win_fence`. This mode of operation can be integrated into a hybrid task-based application by instantiating a task to open the epoch on the NAM window with an MPI fence, followed by multiple concurrent tasks that write or read data to/from the NAM window, and finally, another task closing the access epoch with another fence. Notice that tasks can define the corresponding dependencies on the window to ensure this order of execution.

In order to support this taskification, the fences should be managed by the TAMPI library to perform it efficiently and safely. We have extended the ParaStation MPI to provide a new non-blocking function called MPI_Win_ifence that performs that starts a fence operation on a specific window and generates an MPI request to check its completion later. This MPI request can be then naturally handled by the TAMPI library using the `TAMPI_Iwait` function, as shown below.

```
// Open RMA access epoch to write the NAM window
#pragma oss task inout(namWindow)
{
   MPI_Request request;
   MPI_Win_ifence(0, namWindow, &request);
   TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
}

// Write to NAM region concurrently
for (...) {
   #pragma oss task concurrent(namWindow)
   MPI_Put(..., namWindow);
}

// Close RMA access epoch to write the NAM window
#pragma oss task inout(namWindow)
{
   MPI_Request request;
   MPI_Win_ifence(0, namWindow, &request);
   TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
}
```

In this example, the first task opens an access epoch on the NAM window using the new `MPI_Win_ifence`. This function starts a fence operation, generates an MPI request and returns immediately. This request is then handled by the TAMPI library through the `TAMPI_Iwait`, which bind the completion of the task to the finalization of the fence operation. The task can continue executing immediately without blocking and finalize its execution. Once the fence operation finalizes, the task will complete automatically and its successor task will become ready. The successors are the tasks that perform `MPI_Put` operations on the window in parallel. Notice that the dependencies of these tasks allow them to execute all `MPI_Put` in parallel, always after the window epoch has been opened. After all `MPI_Put` tasks have executed, the last task can run and close the window epoch.

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 `heat` subfolder at the ?https://gitlab.version.fz-juelich.de/DEEP-EST/ompss-2-benchmarks 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 save the computed matrix periodically. The idea is to save different states (snapshots) of the matrix during the execution in a persistent NAM memory region. Then, another program could retrieve all the matrix snapshots, process them and produce a GIF animation showing the heat's evolution throughout the execution. Notice that we cannot use regular RAM for that purpose because the matrix could be huge, and we may want to store tens of matrix snapshots. We also want to keep it persistently so that other programs could process the stored data. Moreover, the memory should be easily accessible by the multiple MPI ranks or their tasks in parallel. The NAM memory satisfies all these requirements, and as previously stated, ParaStationMPI allows accessing NAM allocations through standard MPI RMA operations. We only implement the NAM snapshots in the TAMPI variants `02.heat_itampi_ompss2_tasks.bin` and `03.heat_tampirma_ompss2_tasks.bin`.

The Heat benchmark allocates a single MPI window that holds the whole NAM region, which is used by all ranks (via the `MPI_COMM_WORLD` communicator) and throughout the execution. Every few timesteps (specified by the user), it saves the whole matrix into a specific NAM subregion. Each timestep that saves a matrix snapshot employs a distinct NAM subregion. These subregions are placed one after the other, consecutively, without overlapping. Thus, the entire NAM region's size is the full matrix size multiplied by the number of times the matrix will be saved. Even so, we allocate the NAM memory region using the Managed Contiguous layout (`psnam_structure_managed_contiguous`). This means that rank 0 allocates the whole region, but each rank acquires a consecutive memory subset, where it will store its blocks' data for every snapshot. For instance, the NAM allocation will first have 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. By using that layout, NAM subregions are rank-addressed using the rank it belongs to, simplifying the saving and retrieving of snapshots.

Then, when a timestep 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 writing (copying) the data of each of its matrix blocks into the NAM subregion. These communication tasks do not have any data dependency between them, so they can write data to the NAM using regular `MPI_Put` in parallel. Ranks only write to their own subregions, never in other ranks' subregions. Nevertheless, we must call all MPI_Puts inside an MPI RMA access epoch, so there must be a window fence call before all the MPI_Puts, and another one after them to close the epoch, for each of the snapshot timesteps. Here is where we leverage the new function `MPI_Win_ifence` along 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 closing the parallelism for doing the snapshots (e.g., with a `taskwait`) . Thanks to the task data dependencies and TAMPI, we cleanly include the snapshots in the application's data-flow execution as any other regular communication task with dependencies.

The following pseudo-code shows how the saving of snapshots works 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 the Heat's timesteps applying the Gauss-Seidel method. All MPI ranks run this function concurrently, each one working 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` that will issue tasks 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 at the right moment of the execution, i.e. after the `gaussSeidelSolver` tasks from that timestep. Notice also that we identify each snapshot with the `namSnapshotId`, which we use to know where we should store the snapshot data inside the NAM window. This taskwait at the end of the algorithm is the only one in the whole execution; we do not close the parallelism anywhere else.

```
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(0, namWindow, &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..) concurrent(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(0, namWindow, &request);
       TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
   }
}
```

The function above is the one called periodically from the primary procedure. It instantiates the tasks that will perform the snapshot of the current rank's blocks into their corresponding NAM memory subregions. 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 an MPI RMA access epoch has been opened on that window. That is what the first task is doing. After all the blocks have been computed in that timestep and are ready to be read (notice its task dependencies), the first task will run and execute an `MPI_Win_ifence` to start the window epoch's opening. This MPI function generates an MPI request and serves as parameter of the subsequent call to `TAMPI_Iwait`, which binds the current task's completion to the finalization of the MPI request. 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 its successor tasks ready. The successors of the fence task are the ones that perform the actual writing (copying) of data into the NAM memory by calling `MPI_Put`. All blocks can be saved in the NAM memory in parallel by different tasks. The source of the `MPI_Put` is the block itself (in regular RAM), while the destination is the place where the block should be written inside the NAM region. After all writer tasks finish, it is the turn for the task that closes the MPI RMA access epoch on the NAM window. This one should behave similarly to the one that opened the epoch.

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

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**, featuring the **libNAM** integration that allows access to NAM memory regions through MPI RMA windows, and supporting the new request-based `MPI_Win_ifence` function.
- 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 snapshots 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://gitlab.version.fz-juelich.de/DEEP-EST/ompss-2-benchmarks