High-Performance Matrix Computations: Exploration of StarPU's Runtime

CESISTA Rui[a,1] **and HOUSSENBAY Raoul**[b,2]

[a]*Paris Saclay University*
[b]*Paris Saclay University*

Under the supervision of professor Atte Torri and Oguz Kaya

Abstract—In recent years, the need for efficient computational resource management in high-performance computing has become increasingly critical, particularly in the context of heterogeneous architectures. StarPU, a task-based runtime system, addresses this challenge by providing a unified framework for handling computations across CPUs, GPUs, and other processors. In this paper we will explore the utilization of StarPU, a task-based runtime system, for optimizing matrix multiplication and enhancing computational efficiency across heterogeneous computing architectures. We discuss StarPU's scheduling, data management, and MPI support, highlighting their impact on performance through experimental results. Based on the obtained results we see clearly StarPU's potential in improving resource management and scalability in high-performance computing applications.

keywords—*StarPU, High-performance computing, Task-based Runtime Systems*

Contents

1. Introduction

In the field of high-performance computing (HPC), the rise of mixed hardware systems, like CPUs combined with GPUs, demands powerful tools that manage and make the most of these resources. StarPU, a cutting-edge platform developed by Inria, provides a dynamic runtime system that seamlessly integrates CPUs and GPUs, enabling automatic balancing and optimization of computational tasks across diverse hardware components . StarPU's strength lies in its sophisticated task scheduling capabilities, which leverage both static and dynamic strategies to optimize task execution. By abstracting the complexities of hardware-specific programming, it allows developers to focus on algorithmic design rather than low-level optimizations . We will examines the performance of StarPU by use it to perform matrix multiplication, a basic yet crucial operation that benefits from parallel processing. We will show how StarPU's flexible scheduling and data handling significantly boost performance. Our experiments reveal StarPU's strength in adapting to various workloads and setups, proving its value in today's diverse computing landscapes. The following sections will cover the methods we used, our experimental setup, and a detailed analysis of our results.

2. Theory and Comparison of Task-based Runtime Systems

2.1. Task-based Runtime System Theory

Task-based runtime systems have developed from the need to handle complex applications and various types of computer hardware more efficiently. These systems were born out of the necessity to manage the increasing complexity of applications and the diverse nature of hardware architectures more efficiently. Initially driven by academic research and theoretical constructs, the early models laid the groundwork for defining tasks as discrete units of work with clear dependencies, allowing more granular control over execution and data flow.

The breakthrough came with the formalization of these concepts into the Directed Acyclic Graph (DAG), which maps tasks as nodes and their dependencies as edges. This representation enabled runtime systems to intelligently schedule tasks based on their readiness, ensuring optimal resource utilization without the need for manual synchronization. e.g. a task B which works on a result generated.

Directed Acyclic Graph

A type of graph whose nodes are directionally related to each other and don't form a directional closed loop.

Over the years, as multi-core and multi-threaded processors became the norm, these runtime systems adapted and evolved, providing sophisticated mechanisms to use more wisely the full potential of hardware while abstracting complexity for developers. To use a task-based runtime system, developers need to provide a task graph that outlines all the tasks and how they depend on each other. Each task in this graph is defined as a small, distinct unit of computation, complete with specific functions tailored to different hardware architectures, such as CPUs or GPUs. These tasks operate on data stored in memory buffers, and their interaction with these buffers and whether they read, write, or do both to this data helps shape the task graph. For example, tasks that read data form incoming connections in the graph, while those that write data create outgoing connections.

Various programming models can be used to establish this task graph, dictating how the runtime system will execute the tasks based on the data they need. StarPU, for example, employs a Sequential Task Flow (STF) approach. This method involves sequentially

specifying tasks and the data they will process, allowing the runtime system to construct the Directed Acyclic Graph (DAG) by analyzing data usage and access patterns. It can then determine dependencies based on whether tasks share data and how that data is accessed.

Once developers provide the runtime system with descriptions of the tasks and their dependencies, the system takes over the complex operations necessary to run the application. This includes deciding the order of tasks, running them on the right parts of the computer, and moving data where it's needed. This system takes over the complex tasks that developers used to do by hand to make sure the computer runs efficiently. Now, developers can focus more on building the application itself and worry less about the technical details of the computer hardware.

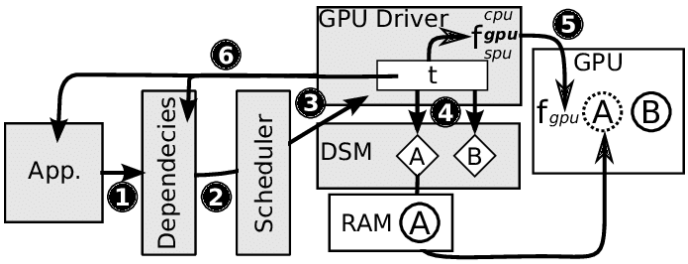

Figure 1. Example of a task execution within StarPU [1].

2.2. StarPU Overview

StarPU is a task-based runtime system that has been particularly designed to manage and optimize computations on heterogeneous machines, such as those combining CPU cores and GPUs. Developed by the Storm team at Inria Bordeaux, [4] was first introduced as part of a PhD thesis by Cédric Augonnet to demonstrate dynamic task scheduling across diverse computing units. Since its inception in 2008, StarPU has grown significantly, supporting not just scientific applications but also serving as a testbed for various research projects on scheduling policies, performance modeling, and more.

The fundamental goal of StarPU is to facilitate the execution of tasks on multiple processing units simultaneously and efficiently. StarPU supports a wide range of architectures, including CPUs, GPUs from both Nvidia and AMD, Intel Xeon Phi processors [3]'s. The adaptability to various hardware is made possible by defining a universal interface that includes instructions for launching computations and handling data transfers for each type of device.

One of the core features of StarPU that significantly simplifies the management of heterogeneous computing resources is its use of data handles to abstract data pointers. This mechanism is crucial for optimizing data placement and movement, which are key factors in improving computational performance on diverse hardware architectures. When using StarPU, developers register each piece of data with the system by creating a data handle, which involves specifying the data's size, structure (whether it is a simple variable, an array, or a matrix), and type (such as float or double). Developers can also specify whether the buffer for this data is already allocated; if not, StarPU can allocate it dynamically as needed. Once registered, StarPU manages all aspects of the data's location transparently. It can track where the data is located, move it across the memory hierarchy to where it's needed for processing.

StarPU also introduces the concept of codelets, which are fundamental to its task scheduling system. A codelet is a reusable structure that encapsulates all the common properties required to execute a specific computation. This includes the computational functions themselves, which can be varied for different types of workers (e.g., specific functions for CPUs or GPUs). The scheduler within StarPU can then select the most appropriate function based on available performance models, ensuring that each task is executed in the most efficient manner possible.

Each task in StarPU is an instantiation of a codelet, which means it is tied to a specific codelet and carries additional, task-specific information such as the data handles needed for the computation, task priority, and any callback functions that should be executed before or after the main task. This structure not only streamlines the execution of tasks but also provides a high degree of flexibility and control over task management.

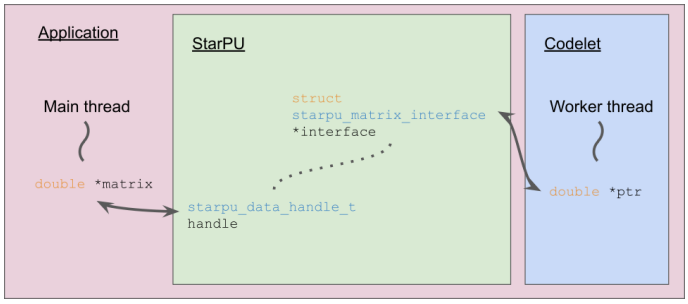

Figure 2. Data Handling in StarPU: Registering and Accessing Matrices with StarPU.

This combination of data abstraction through data handles and structured task management via codelets allows StarPU to effectively coordinate complex computations across a wide array of computing units. It offloads the burden of low-level data management and synchronization from developers, enabling them to focus on optimizing their algorithms and applications.

StarPU can also been use to accommodate distributed applications, making it possible to handle complex computations spread across multiple nodes in a network. This capability is essential for tasks that require large-scale parallel processing across several computers, typically found in research and industrial settings.When setting up distributed applications with StarPU, data is strategically allocated across nodes using data handles, which specify the data's location. This setup allows StarPU to execute tasks directly on nodes where their required data is stored, minimizing unnecessary data movement and improving efficiency. The system operates under a uniform set of instructions across all nodes, with each node processing tasks based on its assigned data.

The system uses a shared Directed Acyclic Graph (DAG) to manage all tasks and their dependencies. StarPU determines which node should execute a task based on where the data resides. If a task requires multiple data elements from different nodes, StarPU chooses to execute the task on the node that minimizes data transfer, optimizing network resource usage. In figure 3 and 4 we can compare how task is been distrubiting by StarPU depending on if it is a distributed setup or a single node setup.

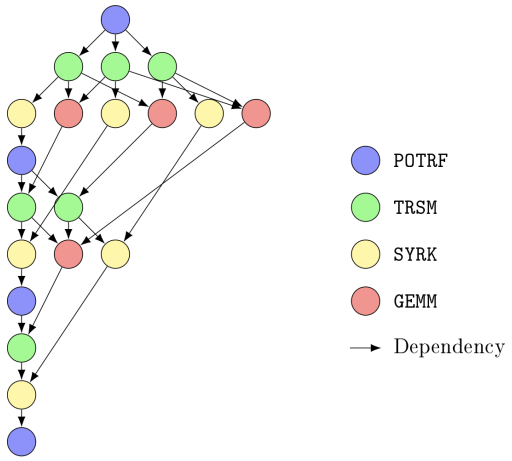

Figure 3. Task-graph of the Cholesky factorization for a matrix divided in 4 × 4 tiles.

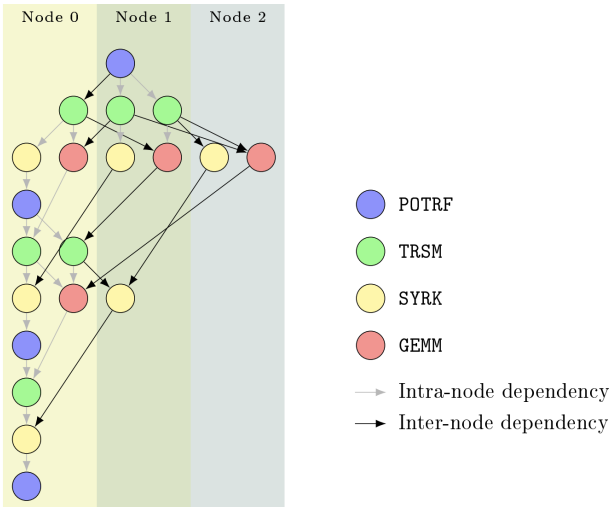

Figure 4. Distribution of the Cholesky Directed Acyclic Graph on 3 nodes.

2.3. Comparative Analysis of Runtime Systems

2.3.1. PaRSEC

As discuss on the previous section StarPU uses Directed Acyclic Graph (DAG) where each task and its dependencies must be explicitly defined. This method tends to be straightforward but can lead to significant memory usage if the number of tasks is large, as the entire graph must be stored and managed by the runtime system.

PaRSEC[2], on the other hand, employs a Parametrized Task Graph (PTG) approach, which offers a more algebraic and compact representation of task dependencies. In PaRSEC's model, tasks are described by their code (usually C functions) and dependencies are managed through algebraic conditions on the input and output data. This lightweight representation does not require all tasks to be instantiated upfront, reducing memory usage as the representation scales with the number of types of tasks, rather than the total number of tasks.

PaRSEC provides more flexibility in thread utilization. Its threads can dynamically switch roles between executing computation tasks and handling communications. This ability to adapt thread roles based on the phase of the application can lead to more efficient use of computational resources, as all threads can perform useful work more consistently.

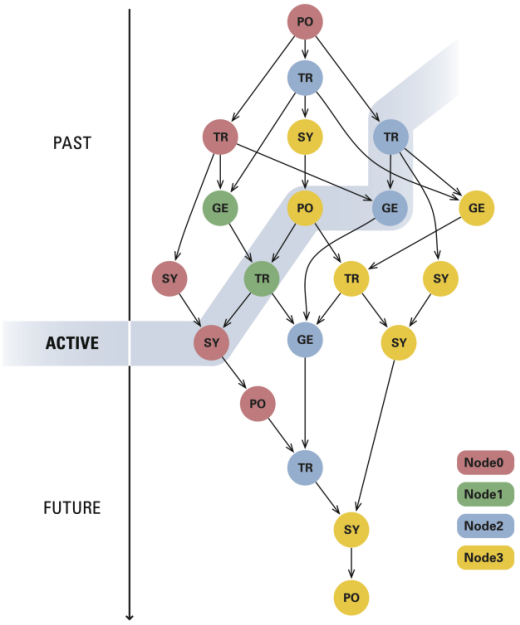

Figure 5. The PaRSEC runtime walks the DAG using a concise representation that instantiates only the relevant tasks at each computing node. Only the active, local tasks need to be stored and considered.

Finally, both StarPU and PaRSEC rely on MPI for managing network communications in distributed applications. However, PaRSEC also supports alternative communication systems like UCX (Unified Communication X) and LCI (Lightweight Communication Interface). UCX provides a low-level communication layer that supports various network interfaces, potentially offering performance benefits in environments where MPI does not align well with application requirements, such as in graph analytics or applications needing strong multithreading support. LCI, being designed specifically for such scenarios, avoids some of the common pitfalls of MPI and is better suited for applications with dynamic communication patterns.

2.3.2. Charm++

Charm++, developed by the Parallel Programming Laboratory at the University of Illinois, utilizes an object-based concurrency model to organize computation around objects, or chares, which contain both data and methods. These chares can dynamically migrate between processors to balance loads, leveraging a message-driven execution system where the arrival of a message triggers method execution, ideally suiting applications with irregular workloads and adaptable load balancing. Charm++ is particularly noted for its automatic, dynamic load balancing that redistributes chares to optimize resource utilization and minimize communication overhead, a boon for large-scale simulations with unpredictable workload shifts.

In contrast, StarPU, which integrates with MPI for network communications management, allows detailed control over data distribution and synchronization across cluster nodes. This model is beneficial for structured parallel applications requiring coordinated task execution and data movement. StarPU's detailed API is tailor-made for environments where various parallel execution strategies are tested, providing developers explicit control over task scheduling and data management.

Meanwhile, Charm++ employs its own communication framework optimized for asynchronous communications and

message-driven execution. This reduces idle time and enhances efficiency on large-scale systems. The framework's design allows developers to concentrate more on application logic rather than on communication details. Charm++ offers a higher-level abstraction through its object-based model, which might be more intuitive for developers not specialized in parallel programming, facilitating a development process more aligned with object-oriented practices.

Both StarPU and Charm++ serve distinct needs in the realm of parallel computing. StarPU is optimal for applications requiring detailed management of heterogeneous computing resources, whereas Charm++ excels in scenarios that demand adaptive load balancing and flexibility in handling dynamically changing workloads.

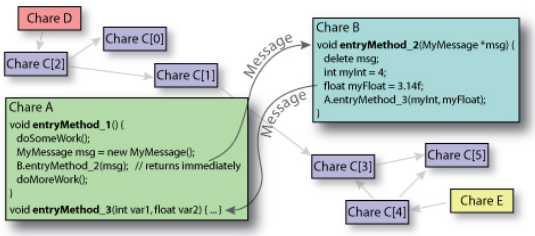

Figure 6. User's View of a Charm++ Application

2.3.3. Many others

- Quark is a task-based runtime system particularly suited for multi-core systems with shared memory, primarily targeting linear algebra applications. It adopts the STF model and significantly influenced the design of StarPU. Howewer it is no longer active.
- Legion, another task-based system, focuses on optimizing data locality to enhance runtime efficiency.
- SuperGlue is a task-based programming model focused on efficient dependency management and high performance in computationally intensive environments.
- OneAPI is a unified programming model initiated by Intel to streamline development and enhance performance across various computing architectures.
- HPX (High Performance ParalleX) is an open-source C++ library designed for scalable parallelism and concurrency, aiming to improve application performance across extensive computing networks.

3. Implementation Details

In the following section, we will go through the implementation of each part of the project. We will introduce step by step the work we have to done each week and explain how we managed to implement it.

First of all the core of our application involves defining and implementing "codelets", which are small, reusable units of computation in StarPU. Codelets are small, reusable units of computation that play a crucial role in achieving efficient parallel computation by allowing tasks to be executed on specific hardware resources. We will outline how these codelets are defined, implemented, and integrated into the application, focusing on both CPU and GPU implementations.

Codelet

A codelet defines characteristics common to a set of tasks.Each codelet is defined with attributes that specify the execution hardware (CPU or GPU), the type of computations it performs, and its data dependencies..

First, the programmer must describe the codelet for the tasks for example to illustrate this we will take as an example the codelet which are needed to perform a gemm operation.

```
struct starpu_codelet get_gemm_cl()
{
    return
    {
    .cpu_funcs = {gemm_1D_tile_cpu<DataType
    >},

#ifdef USE_CUDA

        .cuda_funcs = {cuda_mult<DataType
    >},
        .cuda_flags = {STARPU_CUDA_ASYNC},
#endif
        .nbuffers = 3,
        .modes = {STARPU_R, STARPU_R,
    STARPU_W},
        .name = "gemm",
        .model = &model,
    };
}
```

Code 1. Starpu Codelet.

Code 1 shows how to do that using StarPU's C API. This codelet provides both a CPU implementation and an CUDA one. It also specifies that the task has two inputs and one output through the nbuffers and modes attributes

Then the " gemm_1D_tile_cpu" and cuda_mult function referenced in the codelet must be defined has show in **Code 2** and **Code 3**.

```
void gemm_1D_tile_cpu(void *buffers[], void
     *cl_args)
{

DataType alpha, beta;
DataType *subA;
DataType *subB;
DataType *subC;
char transA, transB;
int nxC, nyC, nyA;
int ldA, ldB, ldC;

starpu_codelet_unpack_args(cl_args, &alpha,
     &transA, &transB, &beta);

// Get Matrix
beta = 1.0;
subA = (DataType *)(STARPU_MATRIX_GET_PTR(
    buffers[0]));
subB = (DataType *)(STARPU_MATRIX_GET_PTR(
    buffers[1]));
subC = (DataType *)(STARPU_MATRIX_GET_PTR(
    buffers[2]));

// Get size
nxC = static_cast<int>(STARPU_MATRIX_GET_NX
    (buffers[2]));
nyC = static_cast<int>(STARPU_MATRIX_GET_NY
    (buffers[2]));
nyA = static_cast<int>(STARPU_MATRIX_GET_NY
    (buffers[0]));

```



```
// Get Ld
ldA = static_cast<int>(STARPU_MATRIX_GET_LD
    (buffers[0]));
ldB = static_cast<int>(STARPU_MATRIX_GET_LD
    (buffers[1]));
ldC = static_cast<int>(STARPU_MATRIX_GET_LD
    (buffers[2]));

//  Do the blas gemm
blas<DataType>::gemm(transA, transB,
                     nxC, nyC, nyA,
                     alpha,
                     subA, ldA,
                     subB, ldB,
                     beta,
                     subC, ldC);
};
}
```

Code 2. Starpu Codelet.

```
void cuda_mult(void *buffers[], void *_args
    )
{
// Get Matrix, Size and LD as well as any
    others variable are similar than in
    previous example.

    cublasHandle_t handle;
    cublasCreate(&handle);
    cublas<DataType>::gemm(
    handle,

    (transA == 'N' || transA == 'n') ?
    CUBLAS_OP_N : CUBLAS_OP_T,

    (transB == 'N' || transB == 'n') ?
    CUBLAS_OP_N : CUBLAS_OP_T,

    nxC, nyC, nyA,
    alpha,
    subA, ldA,
    subB, ldB,
    beta,
    subC, ldC);
    cublasDestroy(handle);

    cudaStreamSynchronize(
    starpu_cuda_get_local_stream());
}
```

Code 3. Starpu Codelet.

Theses function is mainly divided in four steps:

- Unpack multiplication parameters and matrix dimensions.
- Retrieve pointers to sub-matrices A, B, and C.
- Adjust leading dimensions for correct memory access.
- Perform multiplication using BLAS gemm.

Then the programmer must write the main part of the application which includes initializing matrices, assigning random values, and setting up the execution environment for StarPU. The codelets are then engaged to perform the matrix operations, managed through a series of task insertions that dictate the execution order based on data dependencies

```
    //Initialise matrix
    Matrix<float> matrix1(m, k, bs, bs);
    Matrix<float> matrix2(k, n, bs, bs);
    Matrix<float> result(m, n, bs, bs);

    //Assign random value
    matrix1.fill_random();
    matrix2.fill_random();
    starpu_task_wait_for_all();
    result.fill_value(0);
    starpu_task_wait_for_all();

    //Gemm step
    Matrix<float>::gemm(alpha, matrix1,
    transA, matrix2, transB, beta, result,
    redux, dag);

    starpu_task_wait_for_all();
```

Code 4. Starpu Code To make a gemm multiplication in main.

The procedure for launching the task is completely uniform; it does not know anything about task types as shown in Code 5. It consists of creating the tasks, associating them with the codelet defined in Code 1, setting their dependencies, and finally submitting them.

```

starpu_task_insert(
    &gemm_cl<DataType>,
    STARPU_VALUE, &alpha, sizeof(alpha),
    STARPU_VALUE, &transA, sizeof(transA),
    STARPU_R, A._handle,
    STARPU_VALUE, &transB, sizeof(transB),
    STARPU_R, B._handle,
    STARPU_VALUE, &beta, sizeof(beta),
    STARPU_RW | STARPU_COMMUTE, C._handle,
    0
);
```

Code 5. Starpu Codelet.

3.1. Matrix and Tile

At the core of our project reside the Matrix and Tile classes.These classes are designed to handle matrices that are divided into smaller segments, or "tiles".

The Matrix class is structured to encapsulate the entire matrix and manage its division into tiles. The Matrix class serves as the primary data structure for handling large matrices divided into tiles. It is structured to support operations on matrices by abstracting the complexity of tiled data management. The key components of the Matrix class are:

- Matrix Dimensions: Stores the overall dimensions of the matrix (_height, _width).
- Tile Arrangement: Keeps track of how tiles are organized within the matrix (_hTile, _wTile)
- Tile Storage: Utilizes a vector array (_tiles) to store tiles, facilitating easy access and efficient management of the tiled data structure.

Both the Matrix and Tile classes are designed to integrate seamlessly with StarPU. This integration allows us to:

- **Registration**: Each matrix and its tiles are registered with StarPU, allowing it to manage data movement and task execution efficiently.

- **Computation Handling**: The classes support decomposing computational tasks into smaller sub-tasks that can be independently scheduled and executed across various computing resources.
- **De-allocation**: After computations are complete, resources are properly deallocated.

3.2. 1D Tiling

1D tiling refers to a method of dividing an array or matrix into smaller, more manageable segments or "tiles" along one dimension. It is useful for improving cache efficiency and necessary to enabling parallel execution.

In this method the matrices A and B are decomposed into smaller sub-matrices (tiles), where each tile is processed independently. The resulting matrix C is also stored in a tiled format. The computation proceeds by iterating over the tiles of A and B, performing the necessary multiplication, and accumulating the results in the corresponding tiles of C.

The core computation loop is outlined in the Matrix class:

```
for (uint32_t i = 0; i < C._hTile; ++i)
    for (uint32_t j = 0; j < C._wTile; ++j)
        for (uint32_t k = 0; k < A._hTile;
    ++k)

            Tile<DataType>::gemm(
            alpha, A._tiles[j + k * ldA],
            transA,
            B._tiles[k + i * ldB], transB,
            beta,
            C._tiles[j + i *C._wTile],
            redux, dag);
```

Code 6. Perform 1D Matrix multiplication.

It will then call gemm method inside Tile class which will have the role to submit a task to starpu. As describe inside code 5.

3.3. 2D Tiling

2D Tiling involves arranging objects or pattern in a grid, typically with rows and columns. It manages tiles on 2 dimension. It is useful like 1D Tiling for cache efficiency since we can choose the size of the Tile unlike 1D which only have one either the width or the height of the tile but not both.

Also from the computional loop, we do several time on the same tiles so we need them to stack up so we need them to be able to stack without resetting or changing its value so it is the reason that beta is equal to 1.0 in the gemm code. So do solve that problem we do the C $= \beta xC$ first with

```
// First step of gemm
for(uint32_t i = 0; i < C._hTile * C._wTile
    ; ++i)
{
    Tile<DataType>::init_gemm(beta, C.
    _tiles[i]);
}
```

Code 7. init gemm

This init_gemm call a starpu task with the codelets init_c which does

```
template<typename DataType>
struct starpu_codelet get_gemm_init_c()
{
    return {
        .cpu_funcs= {gemm_init_c<DataType
    >},
        .nbuffers = 1,
        //.modes = {STARPU_RW},
        .modes = {static_cast<
    starpu_data_access_mode> (STARPU_RW|
    STARPU_COMMUTE)},
        .name = "gemm_init_c",
    };
}
template<typename DataType>
static starpu_codelet gemm_init =
    get_gemm_init_c<DataType>();
template <typename DataType>
void gemm_init_c(void *buffers[], void *
    cl_args) {
    DataType* C;
    DataType beta;
    int nxc, nyc;
    C = (DataType *)(STARPU_MATRIX_GET_PTR(
    buffers[0]));
    // Get size
    nxc = static_cast<int>(
    STARPU_MATRIX_GET_NX(buffers[0]));
    nyc = static_cast<int>(
    STARPU_MATRIX_GET_NY(buffers[0]));
    starpu_codelet_unpack_args(cl_args, &
    beta);
    for(int i=0; i < nxc; ++i)
    {
        for(int j=0; j < nyc; ++j)
        {
            C[i * nxc + j] *= beta;
        }
    }

}
```

Code 8. init codelet + method

And then we do the computation so that we can add each time. So those make 1D and 2D result equal when having the same input.

3.4. CUDA Support

To support CUDA we have to implement some CUDA Kernel function that will be referenced in one of our codelet. In our implementation, we use cuBLAS, to perform BLAS operations.

In the project we have declare three of these function:

- void cuda_mult(void *buffers[], void *_args)
- void fill_value_matrix_cuda(void *buffers[], void *cl_args)
- void assert_equal_gpu(void *buffers[], void *cl_args)

We already describe "cuda_mult" in the previous section (Code 3). For the two remaining function. They share the same logic of execution.

- Initially, the function extracts the necessary parameters from the input arguments, such as the target matrix and the value to be assigned to each element.
- We will then perform the operation in our case fill the matrix or perform an equality verification.
- Finally, we ensure that all GPU operations are synchronized to maintain data consistency. We also handles the cleanup of GPU resources

```

```



```
void fill_value_matrix_cuda(void *buffers[], void *cl_args)
{
  DataType *A;
  DataType value;
  int nxA, nyA, ldA;
  // unpack arguments
  starpu_codelet_unpack_args(cl_args, &value);

  // Matrix info
  A = (DataType *)(STARPU_MATRIX_GET_PTR(buffers[0]));
  nxA = static_cast<int>(STARPU_MATRIX_GET_NX(buffers[0]));
  nyA = static_cast<int>(STARPU_MATRIX_GET_NY(buffers[0]));
  ldA = static_cast<int>(STARPU_MATRIX_GET_LD(buffers[0]));

  // Cuda stream and Cublas  handle
  // cudaStream_t stream;
  cublasHandle_t handle;
  cublasCreate(&handle);

  //  Fill matrix using cuda
  cuextensions<DataType>::fill(value, nxA, nyA, A,
                               ldA, starpu_cuda_get_local_stream());

  // synchronize stream
  cudaStreamSynchronize(starpu_cuda_get_local_stream());

  // destroy resource
  cublasDestroy(handle);
}
```

Code 9. CUDA Function to fill a matrix.

```

void assert_equal_gpu(void *buffers[], void *cl_args)
{
  DataType *A, *B;
  int nxA, nyA, ldA, ldB;
  A = (DataType *)(STARPU_MATRIX_GET_PTR(buffers[0]));
  nxA = static_cast<int>(STARPU_MATRIX_GET_NX(buffers[0]));
  nyA = static_cast<int>(STARPU_MATRIX_GET_NY(buffers[0]));
  ldA = static_cast<int>(STARPU_MATRIX_GET_LD(buffers[0]));
  B = (DataType *)(STARPU_MATRIX_GET_PTR(buffers[1]));
  ldB = static_cast<int>(STARPU_MATRIX_GET_LD(buffers[1]));

  cublasHandle_t handle;
  cublasCreate(&handle);

  bool is_equal = cuextensions<DataType>::test_equals(nxA, nyA, A, ldA,
                                                      B, ldB,
                                                      starpu_cuda_get_local_stream());
  // synchronize stream
  cudaStreamSynchronize(starpu_cuda_get_local_stream());

  // destroy resource
  cublasDestroy(handle);

  if (is_equal)
    std::cout << "Equal\n";
  else
    std::cout << "Not Equal \n";
}
```

Code 10. CUDA Function to perform matrix equality verification.

3.5. MPI Support

To correctly offer MPI Support with StarPU. We have to make sure that the initialization of both MPI and StarPU environments at the start of the application is correctly realized. This dual initialization is essential to ensure that all computing nodes are synchronized and prepared for distributed task execution:

```
ret = starpu_mpi_init_conf(&argc, &argv, mpi_init, MPI_COMM_WORLD, NULL);

STARPU_CHECK_RETURN_VALUE(ret, "starpu_mpi_init_conf");

starpu_mpi_comm_rank(MPI_COMM_WORLD, &rank);

starpu_mpi_comm_size(MPI_COMM_WORLD, &size);

if (size < 2)
{
    if (rank == 0)
    {
      std::cout << "Need 2 process to work\n";
      starpu_mpi_shutdown();
    }
    if (!mpi_init)
      MPI_Finalize();
    return -1;
}
```

Code 11. MPI initialisation using StarPU

Small Note

To ensure that our implementation correctly adapts to different environments, we employ preprocessor logic to handle conditional compilation. This approach is essential when we aim to utilize specific features such as MPI, CUDA,SIMGRID...

These conditional blocks ensure that the compilation process selectively incorporates features relevant to the user's specific needs and available system resources.

Once we correctly realize that we have to distribute the data correctly accross each node. In our implementation, StarPU handles data registration and synchronization across nodes using MPI,

ensuring that each node operates on the correct segment of data. This is accomplished by registering data with both StarPU and MPI.

To illustrate that we will take as an example how we allocate some tile using MPI and StarPU.

```
Tile<DataType>::Tile(uint32_t width,
    uint32_t height, int rank, bool
    belonging, uint32_t belong, uint32_t
    tag)
{
    // Put size of the matrix
    _width = width;
    _height = height;

// Using starpu malloc we alloc the matrix
#ifdef USE_MPI
    starpu_malloc((void **)&_tile, _width *
     _height * sizeof(DataType));
        starpu_matrix_data_register(&
    _handle, -1, (uintptr_t) nullptr,
    _width, _width, _height, sizeof(
    DataType));
    }
    starpu_mpi_data_register(_handle, tag,
    (int)belong);
#endif
}
```

Code 12. Tile constructor.

- The function starpu_malloc is utilized to dynamically allocate memory for each tile of the matrix directly in the main memory of the computing node handling the tile.
- The function starpu_matrix_data_register is then used to register the allocated matrix with StarPU.This function associates the matrix with a starpu _data_handle_t (_handle), which is used by StarPU to track and manage the life cycle of the data.
- Finally, starpu_mpi_data_register integrates the data management with the MPI environment, associating it with a specific MPI tag and the rank of the node that "owns" this data (belong).

By allocating in these way we are not only allocated memory and manage data locally but also prepare the data for synchronized access across multiple nodes.

Finally at the conclusion of the computational tasks, it is imperative to properly shut down the MPI and StarPU environments using **starpu_mpi_shutdown()** command.

3.6. Reduction

To enable reduction with starpu we need to init the handle and give the reduction method to starpu. We do it in the matrix.cpp in the gemm function. So that we set several time the reduction method to the same handle.

```
for(uint32_t i=0; i< C._hTile*C._wTile;++i)
{
    starpu_data_set_reduction_methods(C.
    _tiles[i]._handle, &sum_matrix_cl<
    DataType>, &init_cl<DataType>);
}
```

Code 13. Declaration Reduction handle

init_cl is the init method and sum_matrix_cl is the reduction method.

```
template <typename DataType>
struct starpu_codelet get_init_c(){
    return {
        .cpu_funcs = {init_c<DataType>},
        .nbuffers = 1,
        .modes = {STARPU_W},
        .name = "init_c",
    };
}
template <typename DataType>
static starpu_codelet init_cl = get_init_c<
    DataType>();

template <typename DataType>
struct starpu_codelet get_sum_matrix_cl()
{
    return{
        .cpu_funcs = {sum_matrix<DataType
    >},
        .nbuffers = 2,
        .modes = {static_cast<
    starpu_data_access_mode>(STARPU_RW|
    STARPU_COMMUTE), STARPU_R},
        .name = "sum_matrix",
    };
}
template <typename DataType>
static starpu_codelet sum_matrix_cl =
    get_sum_matrix_cl<DataType>();
template <typename DataType>
void sum_matrix(void *buffers[], void *
    cl_args)
{
    DataType *A;
    DataType *B;
    int nxA, nyA;
    A = (DataType *)(STARPU_MATRIX_GET_PTR(
    buffers[0]));
    B = (DataType *)(STARPU_MATRIX_GET_PTR(
    buffers[1]));

    // Get size
    nxA = static_cast<int>(
    STARPU_MATRIX_GET_NX(buffers[0]));
    nyA = static_cast<int>(
    STARPU_MATRIX_GET_NY(buffers[0]));

    // Get Ld
    for(int i=0; i < nxA; ++i)
        for(int j=0; j < nyA; ++j)
            A[i * nxA + j] = A[i * nxA + j]
     + B[i * nxA +j];
}
template <typename DataType>
void init_c(void *buffers[], void *cl_args)
    {
    DataType* C;
    int nxc, nyc;
    C = (DataType *)(STARPU_MATRIX_GET_PTR(
    buffers[0]));
    // Get size
    nxc = static_cast<int>(
    STARPU_MATRIX_GET_NX(buffers[0]));
    nyc = static_cast<int>(
    STARPU_MATRIX_GET_NY(buffers[0]));

```



```
    for(int i=0; i < nxc; ++i)
        for(int j=0; j < nyc; ++j)
            C[i * nxc + j] = static_cast<
    DataType>( 0.0);
}
```

Code 14. Reduction Codelet and Methoc

Also when we need to call the Reduction method for the starpu task we need instead of putting STARPU_RW, STARPU_REDUX.

```
starpu_task_insert(&gemm_cl<DataType>,
    STARPU_VALUE, &alpha, sizeof(alpha),
    STARPU_VALUE, &transA, sizeof(transA),
    STARPU_R, A._handle,
    STARPU_VALUE, &transB, sizeof(transB),
    STARPU_R, B._handle,
    STARPU_VALUE, &beta, sizeof(beta),
    STARPU_REDUX, C._handle,
    0);
```

Code 15. Starpu Task Reduction

Also there was a problem with the reduction. To use it we needed STARPU_REDUX but in the codelet it was STARPU_RW|STARPU_COMMUTE which was not compatible so solution when we need it we don't specify it buffers.

```
template <typename DataType>
struct starpu_codelet get_gemm_cl() {
    return {
        .cpu_funcs = {gemm_1D_tile_cpu<
    DataType>},
#ifdef USE_CUDA
        /* CUDA implementation of the
    codelet */
        // .where = STARPU_CUDA,
        .cuda_funcs = {cuda_mult<DataType
    >},
        .cuda_flags = {STARPU_CUDA_ASYNC},
#endif
        /* the codelet manipulates 3
    buffers that are managed by the DSM */
        .nbuffers = 3,
        //.modes = {STARPU_R, STARPU_R,
    static_cast<starpu_data_access_mode> (
    STARPU_RW|STARPU_COMMUTE)},
        //.modes = {STARPU_R, STARPU_R,
    STARPU_REDUX},
        .name = "gemm",
  };
}
```

Code 16. Solve Reduction problem

4. Implementation of Optional Features

4.1. Out-of-core Support

Normally when using StarPU we may store more data than the RAM can hold with STARPU_MAIN_RAM to register data. So the Out of core support can make a new memory node on a disk and use it. StarPU will manage automatically the memory if the memory placement of the data is not specified and will redirect it onto the disk.

So to it we have taken alot of step: and modified our code to support it First of all we need to export some information to the environment variable:

```
export STARPU_DISK_SWAP=/tmp
export STARPU_DISK_SWAP_BACKEND=unistd
export STARPU_DISK_SWAP_SIZE=200
```

Code 17. Environment variable Declaration.

- With STARPU_DISK_SWAP being the directory where the memory is stored.
- With STARPU_DISK_SWAP_BACKEND is choosed.
  There are different type of backend for example unistd which permit only caching in the kernel, stdio caching for kernel and the library libc, unistd_o_direct where there are no caching , leveldb, or hdf5
- STARPU_DISK_SWAP_SIZE which is the size of the memory of our disk

After to create a disk for our memory management of StarPu we need to do:

```
int disk = starpu_disk_register(
&starpu_disk_unistd_ops,
(void*) "disk_tmp_mpi", 4096 * 4096 * 10);
```

Code 18. Starpu Disk Declaration.

starpu_disk_register take 3 argument, the type de backend, the directory and the size.

And the last step data registration:
For the Tile we do that for CPU/CUDA:

```
template <typename DataType>
Tile<DataType>::Tile(uint32_t width,
    uint32_t height)
{
    // Put size of the matrix
    _width = width;
    _height = height;

    // Using starpu malloc we alloc the
    matrix
    starpu_malloc((void **)&_tile, _width *
     _height * sizeof(DataType));
    //starpu_matrix_data_register(&_handle,
     STARPU_MAIN_RAM, (uintptr_t)_tile,
    _width,_width, _height, sizeof(DataType
    ));
    // Used when out of core support
    starpu_matrix_data_register(&_handle,
    -1, (uintptr_t)nullptr, _width,_width,
    _height, sizeof(DataType));
}
```

and that for MPI:

```
template <typename DataType>
Tile<DataType>::Tile(uint32_t width,
    uint32_t height, int rank, bool
    belonging, uint32_t belong  ,uint32_t
    tag)
{
    // Put size of the matrix
    _width = width;
    _height = height;
```

```
    // Using starpu malloc we alloc the matrix
    #ifdef USE_MPI
    starpu_malloc((void **)&_tile, _width * _height * sizeof(DataType));
    if(!belonging)
    {
        starpu_matrix_data_register(&_handle, -1, (uintptr_t)nullptr, _width,_width, _height, sizeof(DataType));
    }
    else
    {
        //starpu_matrix_data_register(&_handle, STARPU_MAIN_RAM, (uintptr_t)_tile, _width,_width, _height, sizeof(DataType));
        // Used when out of core support is here
        starpu_matrix_data_register(&_handle, -1, (uintptr_t)nullptr, _width,_width, _height, sizeof(DataType));
    }
    starpu_mpi_data_register(_handle, tag,(int)belong);
    #endif
}
```

So instead of putting STARPU_MAIN_RAM and the DataType* we just put that to -1 nullptr to precise that we let StarPu manage the Memory of the handle.

It was quite difficult to make it work because on our version of StarPU the disk of StarPU didn't work even though there is those function declared in the include of StarPU in the local directory but to fix it we needed to change version.

```
# Install StarPU
cd $SRC_DIR;
git clone --recurse-submodules https://gitlab.inria.fr/starpu/starpu.git;
cd starpu;
git fetch --all --tags --prune;
git checkout starpu-1.4;
./autogen.sh;
mkdir build;
cd build;
../configure --prefix=$INSTALL_DIR/starpu --disable-opencl --disable-build-doc --disable-build-examples --disable-build-test;
make -j;
make install -j;
```

4.2. Parallel worker Support

Parallel workers, also known as clusters, are introduced as a solution to the granularity problem in parallel computing. Rather than dynamically splitting tasks, resources are aggregated to process coarse-grained tasks concurrently. This approach relies on scheduling contexts to effectively handle various types of parallel tasks. The framework leverages two levels of parallelism within a Directed Acyclic Graph (DAG): DAG parallelism and internal task parallelism. Each task in the DAG may contain internal parallelism, such as OpenMP. Integration of multiple runtime systems, such as StarPU for managing DAG parallelism and another runtime (e.g., OpenMP) for managing internal parallelism, poses a challenge. The key lies in establishing an interface between these runtime systems to enable StarPU to aggregate cores within a machine (creating parallel workers), on which parallel tasks like OpenMP tasks can run efficiently and in a contained manner.
So to use Parallel workers we need to declare it first:

```
struct starpu_parallel_worker_config * parallel_workers;
parallel_workers = starpu_parallel_worker_init(
        HWLOC_OBJ_SOCKET,
        STARPU_PARALLEL_WORKER_POLICY_NAME, "dmdas",
        STARPU_PARALLEL_WORKER_PARTITION_ONE,
        STARPU_PARALLEL_WORKER_NEW,
        STARPU_PARALLEL_WORKER_TYPE, STARPU_PARALLEL_WORKER_OPENMP,
//      STARPU_PARALLEL_WORKER_TYPE, STARPU_PARALLEL_WORKER_INTEL_OPENMP_MKL,
        STARPU_PARALLEL_WORKER_NB, 2,
        STARPU_PARALLEL_WORKER_NCORES, 1,
        0);

/* Code */

starpu_parallel_worker_shutdown(parallel_workers);
```

- STARPU_PARALLEL_WORKER_POLICY_NAME is for the type of scheduler
- STARPU_PARALLEL_WORKER_OPENMP which is the default when choosing the type of workers
- STARPU_PARALLEL_WORKER_NB The number of worker per cores
- STARPU_PARALLEL_WORKER_NCORES the number of cores.

and to use it on a task we use

```
starpu_task_insert(&gemm_cl<DataType>,
    STARPU_VALUE, &alpha, sizeof(alpha),
    STARPU_VALUE, &transA, sizeof(transA),
    STARPU_R, A._handle,
    STARPU_VALUE, &transB, sizeof(transB),
    STARPU_R, B._handle,
    STARPU_VALUE, &beta, sizeof(beta),
    STARPU_RW|STARPU_COMMUTE, C._handle,
#ifdef STARPU_OPENMP
    STARPU_POSSIBLY_PARALLEL, 1,
#endif
    0);
```

Since we use STARPU_POSSIBLY_PARALLEL to say either or not we can parallelized our codelets.
Also in the gemm function I just use :

```
blas<DataType>::gemm(transA, transB,
                     nxC, nyC, nyA,
                     alpha,
                     subA, ldA,
                     subB, ldB,
```



```
                        beta,
                        subC, ldC);
```

Since we use openBLAS for the LAPACK usage and openBLAS if there any thread available for OpenMP it will divide it equally to all thread. And I suppose that it will be the case for gemm.
But if it was necessary to do the gemm we can do it by hand like that

```
unsigned i, j, k;
#pragma omp parallel for default(none) \
    shared(subC, subB, subA, nyC, nxC, nyA,
     alpha, beta, ldA, ldB, ldC)  \
    private(i, j, k)
    for(i = 0; i < nyC; ++i)
    {
        for(j=0; j < nxC; ++j)
      {
            //subC[j + i * ldC] *= beta;
          for (k = 0; k < nyA; ++k)
          {
              subC[j + i * ldC] += alpha *
    subA[j + k * ldA] * subB[k + i*ldB];
          }
      }

    }
```

Installation for making Parallel Workers works:

```
# Install StarPU
cd $SRC_DIR;
git clone --recurse-submodules https://
    gitlab.inria.fr/starpu/starpu.git;
cd starpu;
git fetch --all --tags --prune;
git checkout starpu-1.4;
./autogen.sh;
mkdir build;
cd build;
../configure --prefix=$INSTALL_DIR/starpu-
    parallel --disable-opencl --disable-
    build-doc --disable-build-examples --
    disable-build-test --enable-parallel-
    worker;
make -j;
make install -j;
```

And add a compile definition to CMAKE

4.3. Hierarchical DAG Support

Using Hierarchical DAG make us use task to make other task. So recursively make task one after another. So that is called bubblification, we make a bubble graph via proposing the version of our DAG. And it depend of what does you program do to divide it properly between people.

To use DAG recursively we need to partition the Data before dividing it into task and then unpartition it . For example in matrix.cpp gemm use:

```
for(uint32_t i=0; i< C._hTile*C._wTile;++i)
{
    starpu_data_partition(C._tiles[i].
    _handle, &block_filter_for_matrix);
}
for(uint32_t i=0; i< A._hTile*A._wTile;++i)
{
    starpu_data_partition(A._tiles[i].
    _handle, &block_filter_for_matrix);
    starpu_data_partition(B._tiles[i].
    _handle, &block_filter_for_matrix);
}

/* Code */

for(uint32_t i=0; i< C._hTile*C._wTile;++i)
{
    starpu_data_unpartition(C._tiles[i].
    _handle, STARPU_MAIN_RAM);
}
for(uint32_t i=0; i< A._hTile*A._wTile;++i)
{
    starpu_data_unpartition(A._tiles[i].
    _handle, STARPU_MAIN_RAM);
    starpu_data_unpartition(B._tiles[i].
    _handle, STARPU_MAIN_RAM);
}
```

with block_filter_for_matrix in codelets.hpp

```
static starpu_data_filter
    block_filter_for_matrix =
{
    .filter_func =
    starpu_matrix_filter_block,
    .nchildren = PARTS,
};
```

And PARTS= 4. So we divide the matrix into 4 equal part to do gemm on them so we use the function recursive task:

```
template <typename DataType>
void recursive_task_func(void *buffers[],
    void *cl_arg)
{
    assert(0);
    return;
}

template <typename DataType>
int is_bubble(struct starpu_task *t, void *
    cl_arg)
{
    (void)t;
    (void) cl_arg;
    return 1;
}

template <typename DataType>
void gemm_gen_dag(struct starpu_task *t,
    void *cl_arg)
{
    DataType alpha, beta;
    char transA, transB;
    int i;
    struct gemm_bubble_task_arg<DataType> *
    value = (struct gemm_bubble_task_arg<
    DataType>*)  cl_arg;
    starpu_data_handle_t subA = (
    starpu_data_handle_t) value->A;
    starpu_data_handle_t subB = (
    starpu_data_handle_t) value->B;
```

```
    starpu_data_handle_t subC = (starpu_data_handle_t) value->C;
    alpha = static_cast<DataType>(value->alpha);
    beta = 1.0;
    transA = static_cast<char>(value->transA);
    transB = static_cast<char>(value->transB);
    for(i=0; i < PARTS; ++i)
    {
    int indexA = (i/2) * 2;
    int indexB = (i%2);
    #ifndef USE_MPI
    starpu_task_insert(&gemm_cl<DataType>,
    STARPU_VALUE, &alpha, sizeof(alpha),
    STARPU_VALUE, &transA, sizeof(transA),
    STARPU_R, starpu_data_get_sub_data(subA, 1, indexA),
    STARPU_VALUE, &transB, sizeof(transB),
    STARPU_R, starpu_data_get_sub_data(subB, 1, indexB),
    STARPU_VALUE, &beta, sizeof(beta),
    STARPU_RW|STARPU_COMMUTE, starpu_data_get_sub_data(subC, 1, i),
    0);
     starpu_task_insert(&gemm_cl<DataType>,
    STARPU_VALUE, &alpha, sizeof(alpha),
    STARPU_VALUE, &transA, sizeof(transA),
    STARPU_R, starpu_data_get_sub_data(subA, 1, indexA+1),
    STARPU_VALUE, &transB, sizeof(transB),
    STARPU_R, starpu_data_get_sub_data(subB, 1, indexB+2),
    STARPU_VALUE, &beta, sizeof(beta),
    STARPU_RW|STARPU_COMMUTE, starpu_data_get_sub_data(subC, 1, i),
    0);
    /*
    MPI Part
    */
    #endif
    }
   free(value);
}
```

Here in the code I used indexA and indexB to do recursively a 2x2 gemm multiplication and use the data structure for stocking data for the recursive function.

```
template<typename DataType>
struct gemm_bubble_task_arg
{
    DataType alpha;
    char transA;
    starpu_data_handle_t A;
    char transB;
    starpu_data_handle_t B;
    DataType beta;
    starpu_data_handle_t C;
};
```

And make a codelet for it to be used in StarPu task

```
template <typename DataType>
struct starpu_codelet get_recursive_gemm()
{
    return {
        .cpu_funcs = {recursive_task_func<DataType>},
        .bubble_func = is_bubble<DataType>,
        .bubble_gen_dag_func= gemm_gen_dag<DataType>,
        .nbuffers = 0,
        //.modes = {STARPU_R,STARPU_R, static_cast<starpu_data_access_mode> (STARPU_RW|STARPU_COMMUTE)},
        .name = "recursive_block",
    };
}

template<typename DataType>
static starpu_codelet recursive_gemm = get_recursive_gemm<DataType>();
```

bubble_func is used for testing if we bubble.And bubble_gen_dag_func for the recursive function to generate.

And to use Hierarchical DAG we need to do:

```
 gemm_bubble_task_arg<DataType> *dag= nullptr;
dag = new gemm_bubble_task_arg<DataType>{};
dag->A = A._handle;
dag->B = B._handle;
dag->C = C._handle;
dag->alpha = alpha;
dag->beta = beta;
dag->transA = transA;
dag->transB = transB;
starpu_task_insert(&recursive_gemm<DataType>,
            STARPU_BUBBLE_GEN_DAG_FUNC_ARG, dag,
            0);
```

STARPU_BUBBLE_GEN_DAG_FUNC_ARG is used for the argument used in the recursive function.

It was quite hard to implemented it because in the current version of StarPU all the bubble function has been replaced by recursive task and it took times to find it. Also to compile bubble graph we need to compile StarPU with –enable-bubble and it is not written in the part extension of Hierarchical DAG. So to use Hierarchical DAG we used:

```
# Install StarPU
cd $SRC_DIR;
#git clone --recurse-submodules https://gitlab.inria.fr/starpu/starpu.git;
cd starpu;
git fetch --all --tags --prune;
git checkout starpu-1.4;
./autogen.sh;
mkdir build;
cd build;
../configure --prefix=$INSTALL_DIR/starpu --disable-opencl --disable-build-doc --disable-build-examples --disable-build-test --enable-bubble --enable-parallel-worker;
make -j;
make install -j;
```



```
make clean;
```

5. Experiments and Benchmarks

5.1. Benchmarking methodology

The benchmarks were conducted on a high-performance computing system equipped with Intel® Xeon® Silver 4214R CPUs. Here are the key specifications of the system used for the benchmarking process:

- **CPU Model and Capabilities**: Intel® Xeon® Silver 4214R CPU @ 2.40GHz, optimized for reliability and performance with a turbo boost speed up to 3.5 GHz.
- **Core Configuration:** Each of the two sockets on the server hosts 12 cores, with hyper-threading enabled, providing a total of 48 logical cores across the system.
- **Memory Hierarchy**: The CPUs are supported by a robust memory infrastructure, including 768 KiB of L1 cache per socket, 24 MiB of L2 cache, and a 33 MiB L3 cache.
- **NUMA Nodes**: The system is divided into two NUMA nodes.

5.2. Results

5.2.1. Tiling 1D vs 2D

For CPU only

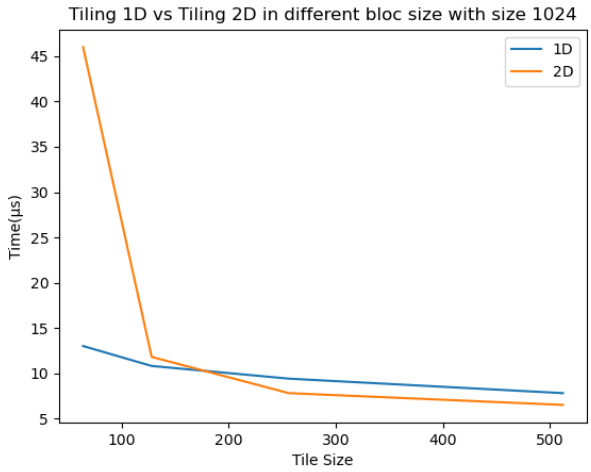

Figure 7. Tiling 1D vs 2D with matrice size 2024

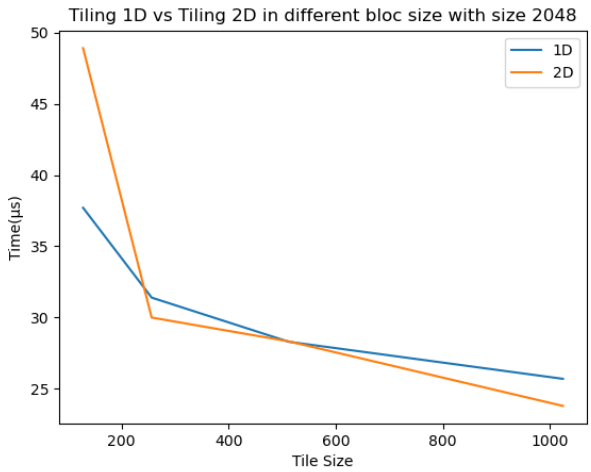

Figure 8. Tiling 1D vs 2D with matrice size 2024

For CPU+CUDA only

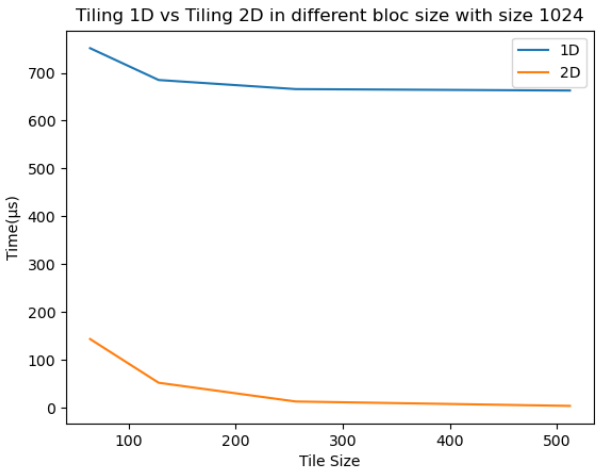

Figure 9. Tiling 1D vs 2D with matrice size 2024

5.2.2. Different Scheduler comparison

Choosing the correct scheduler is crucial for optimizing the performance of an application. The effectiveness of a scheduler directly impacts the application's overall efficiency, execution speed, and resource utilization.

Scheduler

A scheduler in StarPU is responsible for dynamically assigning tasks to various computational resources, such as CPUs, GPUs, and other accelerators, based on their availability and the task's requirements.

StarPU provides several built-in schedulers, each designed for different scenarios and workloads. For example, some schedulers might prioritize minimizing data transfers between devices, while others might focus on load balancing to prevent any single resource from becoming a bottleneck. Selecting an inappropriate scheduler could lead to suboptimal performance, such as increased execution times, underutilization of resources, or excessive energy consumption.

The choice of scheduler highly depend on the application's workload characteristics and the underlying hardware architecture. For instance, applications with highly interdependent tasks might benefit from schedulers that can effectively manage task dependencies and data locality.

StarPU also allows for the development of custom schedulers tailored to specific needs, enabling developers to fine-tune task management according to the unique demands of their applications.

To generate our benchmark, we exclusively utilize five default StarPU schedulers.

- **LWS :** Utilizes work-stealing to balance load while prioritizing local memory to minimize data transfers.
- **DM :** Aims to minimize task completion time by dynamically distributing tasks based on their execution profiles.
- **DMDA** : Enhances DM by also considering data transfer times and locality, suitable for heterogeneous systems.
- **Eager** : Immediately assigns tasks to the least loaded workers, focusing on quick dispatch over optimizing data transfers.
- **Prio :** Manages tasks based on priorities, ensuring higher-priority tasks are processed first.

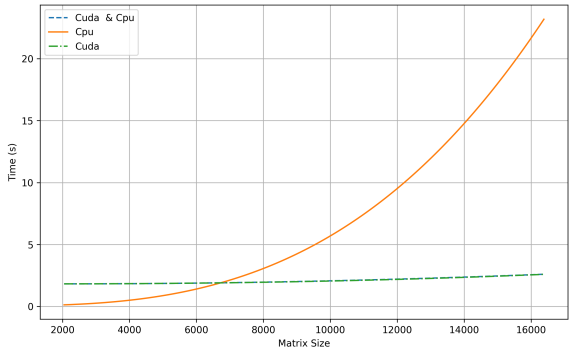

Figure 10. DM Scheduler.

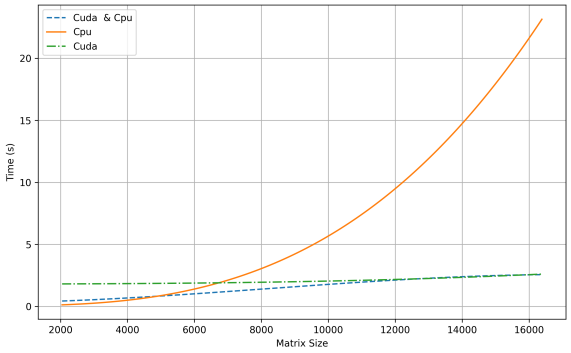

Figure 13. PRIO Scheduler.

From the results we've obtain, the default StarPU scheduler, LWS (Local Work Stealing), works well in systems that have both a CPU and a GPU. However, when only CPUs are used, all the scheduling policies tend to perform similarly, showing no significant differences in their outcomes. This indicates that the advantages of moving tasks around don't stand out as much without the added capabilities of a GPU.

Among the various schedulers, the DM (Distributed Memory) scheduler seems to be the best option.

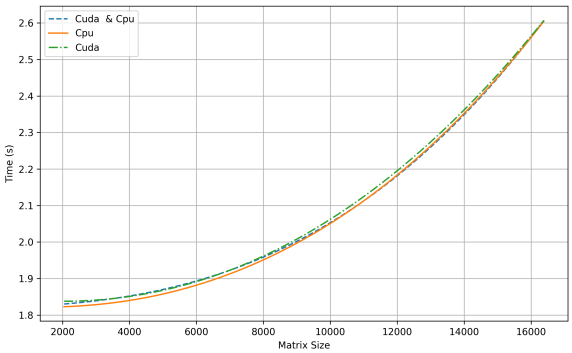

Figure 11. DMDA Scheduler.

5.2.3. Effects of tile size

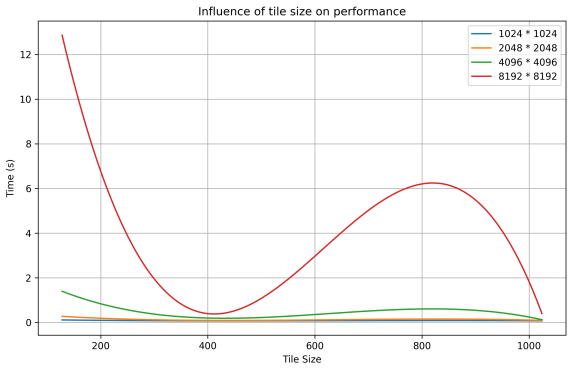

Figure 14. Influence of tile size on performance

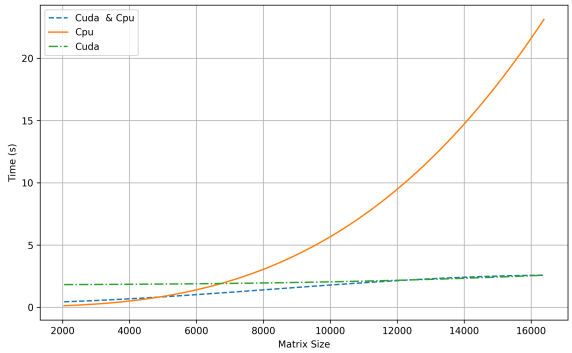

Figure 12. LWS Scheduler.

Choosing the right tile size for matrix multiplication in high-performance computing is crucial for optimal system performance and resource utilization. The tile size affects how a matrix is split into smaller parts or tiles, which are then processed separately. Getting the tile size right can significantly impact processing speed, memory usage, and workload distribution.

When matrices are divided into very small tiles, the system might spend too much time managing many small tasks, which can lead to inefficient use of the computing power. On the other hand, very large tiles might not fully take advantage of the system's ability to do many things at once, and larger tiles can cause issues with memory access that slow down the system.

The benchmark result indicating that the optimal tile size is up to 512x512.

5.2.4. Bus transfer

Table 1. Bus transfer

Matrix dimension	Total transferred	Speed
1024 * 1024	0.0352 GB	14.4157 MB/s
2048 * 2048	0.1406 GB	55.8492 MB/s
4096 * 4096	0.5625 GB	199.3685 MB/s
8192 * 8192	2.2500 GB	517.3247 MB/s
16384 *16384	9.0000 GB	632.0774 MB/s

Note: Memory transfer from NUMA to CUDA

5.2.5. MPI

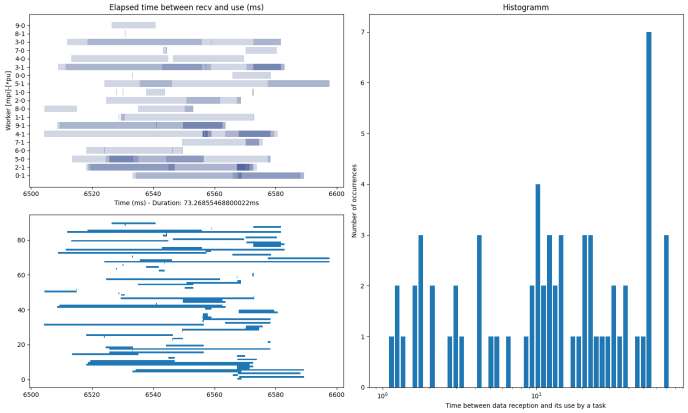

Figure 15. LWS Scheduler Data Receive.

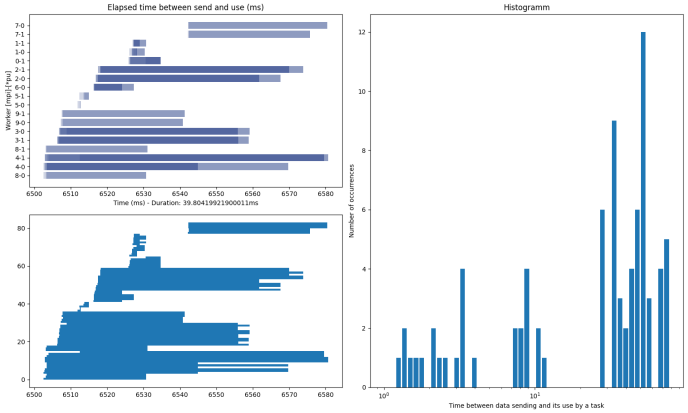

Figure 16. LWS Scheduler Data Send.

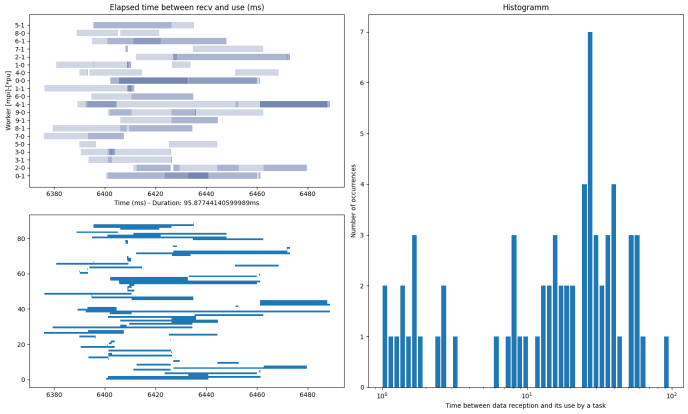

Figure 17. DMDA Scheduler Data Receive.

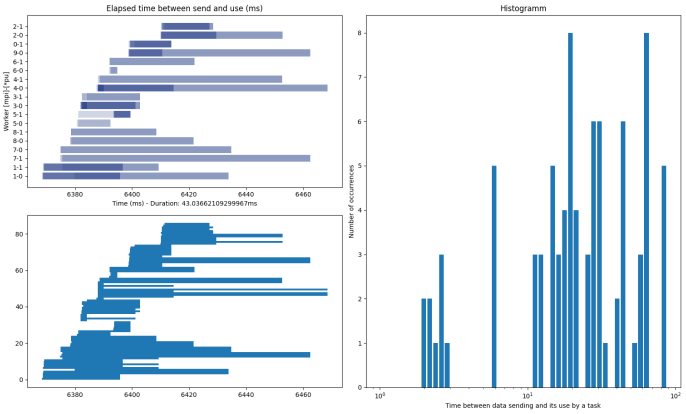

Figure 18. DMDA Scheduler Data Send.

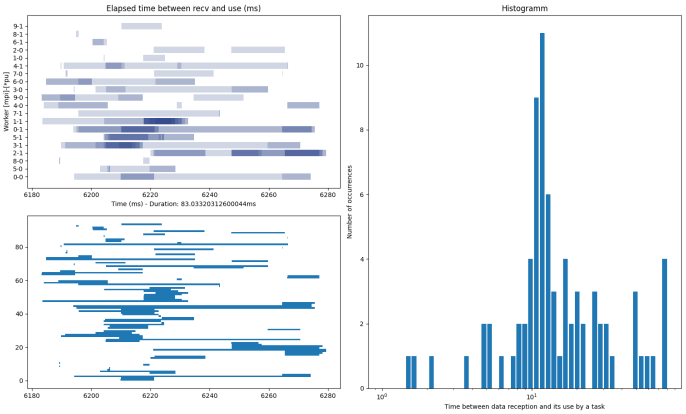

Figure 19. PRIO Scheduler Data Receive.

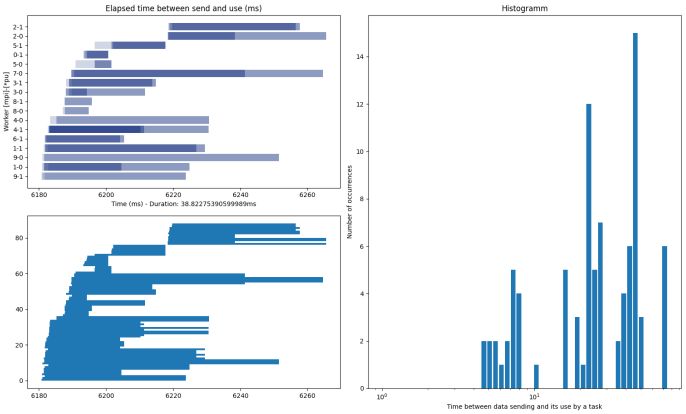

Figure 20. PRIO Scheduler Data Send.

- For the LWS Scheduler our findings indicate that this approach provides a consistent means of handling data transfers. The time taken for sending and receiving data is uniformly distributed, suggesting that LWS avoids bottlenecks effectively by not overloading any single node and surprisingly perform correctly.

- DMDA shows variability in performance. We observed spikes in data transmission, indicating occasional high-volume data transfers. Despite these peaks, the majority of data remains localized, enhancing access speeds and reducing the need for frequent data exchanges across nodes.

- The Prio approach show a histogram-like pattern in data transmission, with periods of increased data movement

followed by reductions. Such a pattern suggests that Prio may temporarily accumulate data transfers as higher priority tasks preempt others, leading to bursts of high activity followed by quieter periods.

5.3. Monitoring Activity

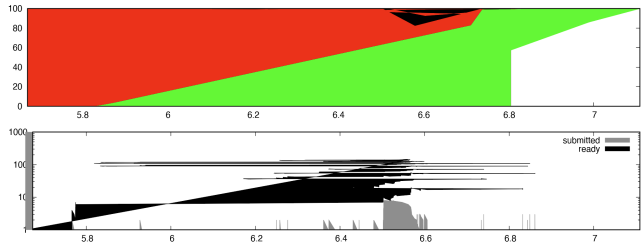

Figure 21. LWS Scheduler.

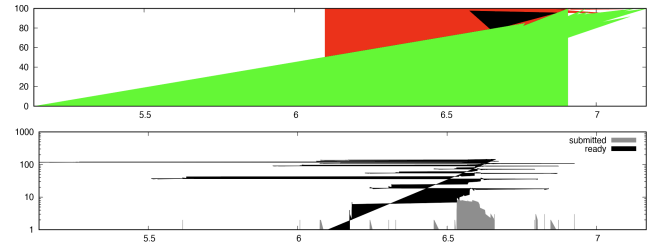

Figure 22. Eager Scheduler.

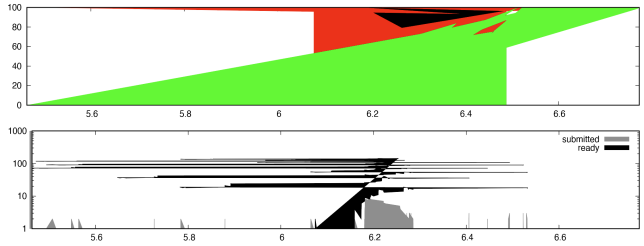

Figure 23. PRIO Scheduler.

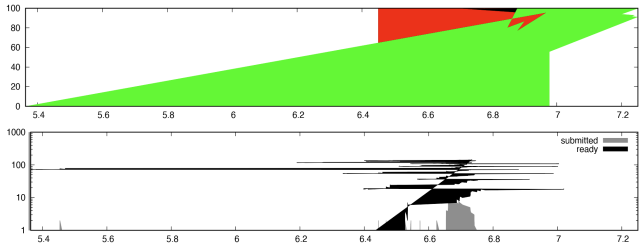

Figure 24. DM Scheduler.

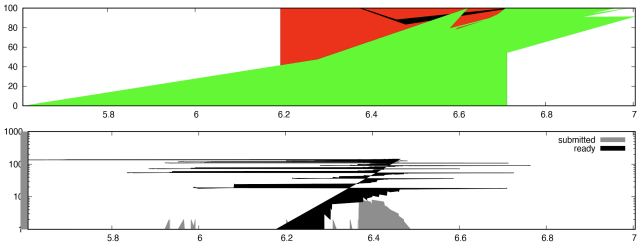

Figure 25. DMDA Scheduler.

Throughout our benchmarking process, we observed that the LWS Scheduler exhibited great performance, despite initial doubts and compiler warning about its effectiveness compared to other alternatives.

Our understanding deepened when we plotted the activity monitor for each scheduler, revealing significant overhead and potential granularity issues for LWS Scheduler.

Conversely, DMDA and DM schedulers performed similarly, with comparable execution times for kernel operations on the processing unit. This consistency suggests that both schedulers effectively distributed tasks across available resources without significant deviations in performance.

One noteworthy observation concerns the Prio scheduler, where the processing unit experienced blocking due to a lack of pending tasks. This occurrence hints a potential issues with parallelism, where the scheduler may not effectively exploit available computational resources to maintain continuous processing.

6. Conclusion

The main conclusion of this study confirms the feasibility of developing matrix multiplication algorithms for heterogeneous computing systems using almost sequential code. This method significantly simplifies the programmer's task by eliminating the complex, low-level communication protocols usually necessary in other task programming runtime systems.

We have also gained a clearer understanding of why StarPU enjoys popularity among the French research community and recognized the importance of task-based programming models. In particular, the STF model is notable for its scalability and its ability to enhance portability and ease of maintenance.

While the performance metrics already achieved are already impressive, the work conducted during this TER identifies significant potential for further improvements. Specifically, for matrix sizes exceeding 2$\hat{1}$6, the processing times are still too long, highlighting an essential area for further development and optimization within this framework.

References

[1] R. N. Cédric Augonnet Samuel Thibault, *StarPU: A Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines*. INRIA, 2010. [Online]. Available: https://inria.hal.science/inria-00467677/document.

[2] A. D. George Bosilca Aurelien Bouteiller, *PaRSEC: A Programming Paradigm Exploiting Heterogeneity for Enhancing Scalability*. IEEE, 2013. [Online]. Available: https://www.netlib.org/utk/people/JackDongarra/PAPERS/ieee_cise_submitted_2.pdf.

[3] B. Christodoulis Selva, *An FPGA Target for the StarPU Heterogeneous Runtime System*. INRIA, 2018. [Online]. Available: https://inria.hal.science/inria-00525540/document.

[4] S. D. Team, *Starpu handbook for starpu 1.4.5*, 2024, pp. 1–867. [Online]. Available: https://files.inria.fr/starpu/doc/starpu.pdf.

We extend our gratitude to Professor Atte Torri and Oguz Kaya for their guidance and insights, which were invaluable throughout this TER. ☻