DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

The Elastic Phase Oriented Programming Model for Elastic HPC Applications

Jophin John

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

The Elastic Phase Oriented Programming Model for Elastic HPC Applications

Elastic Phase Oriented Programming Model für elastische HPC-Anwendungen

Author:	Jophin John
Supervisor:	Prof. Dr. Michael Gerndt
Advisors:	Dr. Isaías A. Comprés Ureña, Andreas Wilhelm
Submission Date:	15.06.2018

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.06.2018 Jophin John

Acknowledgments

First and foremost, I would like to thank God for providing me the knowledge and strength throughout this work. I would like to express my deep and sincere gratitude to my supervisor Prof. Dr. Michael Gerndt for his guidance and support during this thesis work. I am extremely grateful to him for giving me the opportunity to work in this field. I would like to thank Dr. Isaías A. Comprés Ureña for introducing me to this topic and invasive infrastructure as well as for his continous guidance. I would like to extend my deepest thanks to Andreas Wilhelm for his valuable observations regarding the C++ features and also for finding time to read, correct and provide detailed comments about each and every section of the thesis report. I thank him for his patience and suggestions. I am grateful to my father, mother and brother for their continuous and unparalleled support throughout my life.

Abstract

The High-Performance Computing community is steering towards the exascale milestone. To effectively utilize the computing resources, the high performance computing applications should be able to adapt to change in resources. Invasive MPI along with the Invasive Resource manager provides an infrastructure for developing applications that can adapt to the dynamic allocation of resources. Developing an invasive application is a huge challenge for the programmers. One need to consider the possibility of new processes, its interaction with the existing processes, sharing the application data between the existing and new process and finding the suitable entry point for the new processes in the application. These difficulties in making invasive applications became motivation for a phase oriented programming concept and resulted in the Elastic Phase Oriented Programming (EPOP) model. This model treats an application as a collection of multiple phases. A minimalistic EPOP model was implemented in C as a prototype for demonstrating the phase oriented approach. This thesis work extends the EPOP into a fully-fledged programming model with the capabilities to interact with the Invasive MPI and Invasive Resource Manager. This model acts as an abstraction for providing malleability to applications. By using this programming model, a developer can create invasive applications using normal MPI. EPOP model executes the application, profiles it, forwards the performance data of the application to the resource manager for making adaptation decisions and adapts according to the decision made by the resource manager.

Contents

1 Introduction

Increased heat dissipation and subsequent stalling of clock frequencies shifted the processor design paradigm from increasing the performance of a single microprocessor to building multiple processors in a single integrated circuit [16]. The advent of multicore multiprocessor systems brought the parallel programming and parallel programming models into the mainstream.

To effectively utilize the multicore multiprocessor architecture, applications should be programmed to exploit the parallelism. Automatic parallelization and parallel programming are the two common ways to exploit the parallelism in an application. The former approach is not as efficient as the latter because the compiler makes assumptions that are too conservative. If there is enough parallelism in the application, then the programmer can tailor the application in such a way that it can be executed with maximum efficiency and minimum overheads with the aid of suitable programming models and profiling tools. There are different parallel programming models to choose from, depending upon the underlying system architecture and the programming needs. For example, to exploit the parallelism in shared memory systems, one can use OpenMP. CUDA can be used with General Purpose computing on Graphics processing units (GPGPU). Pure parallel programming models, heterogeneous parallel programming models, partitioned global address space model and hybrid, shared distributed memory + GPU models are some of the parallel programming models [6].

A Pure parallel programming model consists of the shared memory and the distributed memory approaches. Pthreads and OpenMP are the two commonly used models for the task-oriented shared memory approach. In the distributed memory approach, message passing is done for the data distribution and synchronization among the processes. MPI is a message passing library commonly used for distributed memory programming. Both shared memory and distributed memory approaches can be combined together to form a new distributed shared memory approach. OpenMP and

MPI are used side by side to exploit the parallelism in shared memory and distributed memory.

In the heterogeneous programming models, along with CPUs, General Purpose GPUs are also used for running the applications. NVIDIA's CUDA, OpenCL, Microsoft's DirectCompute, Intel's Array building blocks are different heterogeneous programming models for GPU programming. The heterogeneous programming is gaining importance in the High Performance Computing field due to the high efficiency of GPU's [1].

The Partitioned Global address space (PGAS) is another model that arose from the fact that shared memory programs are easier to develop than the message passing programs. PGAS utilize the Distributed Shared Memory approach in which a global address space is provided to exploit the data locality thereby increasing the parallel performance of the applications.

The Hybrid programming model combines the different programming models [5]. Combining the distributed memory model (MPI) along with the General purpose GPUs (CUDA) is one such example for the Hybrid programming model.

Since many of the programming models are only dealing with a fixed number of resources which are provided before the execution of a program i.e. static resources, an elastic approach was introduced to dynamically allocate the resources at runtime as per the availability of the resources. The Message Passing Interface (MPI), which is one of the most commonly used distributed memory programming models, supports the dynamic process creation. The application developers are not using these features because of its performance penalty. The standard MPI does not provide the mechanism for the reduction of dynamic processes. To overcome these limitations of MPI in dynamic support, Invasive MPI (IMPI) Library was developed. New operations are added to the Invasive MPI that realizes the dynamic processes and makes it easy to program with minimal latency [3].

Dynamic process creation requires a resource manager which schedules the job according to the availability of the resources, provides additional resources to the compute-intensive applications and take back the resources from the application. The Invasive Resource Manager (IRM) was developed by modifying the SLURM workload manager for dynamic resource management. This invasive resource manager supports the invasive MPI execution [3]. If an application is found to be suitable for the expansion

of resources then new resources are allocated to IMPI.

Creating an application that adapts to change in resources at runtime is a tedious task. Lack of a good programming model for invasive applications makes it even complicated to do invasive programming. Therefore, a new programming model is needed to utilize the dynamic execution of processes and resource allocation offered by IRM and IMPI. Elastic Phase Oriented Programming (EPOP) is a programming model that makes the "invasive programming" easier for developers and also provides the resource manager the flexibility to integrate the resources at runtime.

Elastic Phase Oriented Programming Model is an abstraction for modularizing programs. EPOP is currently a minimalistic C based implementation. This project provides a fully fledged implementation of EPOP using C++. Goals of this master thesis are:

1. Implement a fully fledged EPOP model in C++.
2. Create a developer-friendly syntax.
3. Implement the EPOP drivers to use with the newly designed model.
4. Integrate the EPOP model with the Invasive Resource Manager and Invasive MPI
5. Performance monitoring and profiling of the applications developed using the new model.

2 Motivation

For the invasive resource manager to perform efficiently, applications should be programmed to exploit the elasticity i.e. applications should be able to adapt when new resources are allocated or removed. Programming an application for adapting at runtime is a difficult task for programmers. The programmer needs to be concerned about many aspects (see Section 2.1) while developing such an invasive application.

2.1 Problem Statement - Why is it difficult to do resource aware distributed programming?

To understand the challenges faced by the application developer, let us consider an example application that is using a Jacobi kernel to solve a diagonally dominant linear equation. In a normal non-invasive scenario, a virtual topology of processes is created with a fixed number of processes. The master process divides the data according to the dimension and each process gets its own local data to perform the stencil computation. After the local computation, ghost cells are exchanged to neighbouring processes to synchronise the data along the boundaries. And this data distribution and execution is done for multiple iterations until the application converges. Making the above application into an invasive application can introduce a lot of challenges. The programmer should handle the newly joining processes, the data for the newly joined processes, the outgoing processes, the data from the outgoing processes, the redistribution of data among new and pre-existing processes and also creating a new topology with the new number of processes. The first task of the programmer will be to identify the entry point of the new processes and decide what data should be distributed to it and how to do the distribution. The programmer should probe to check whether new processes are allocated using an IMPI routine. If new processes are added

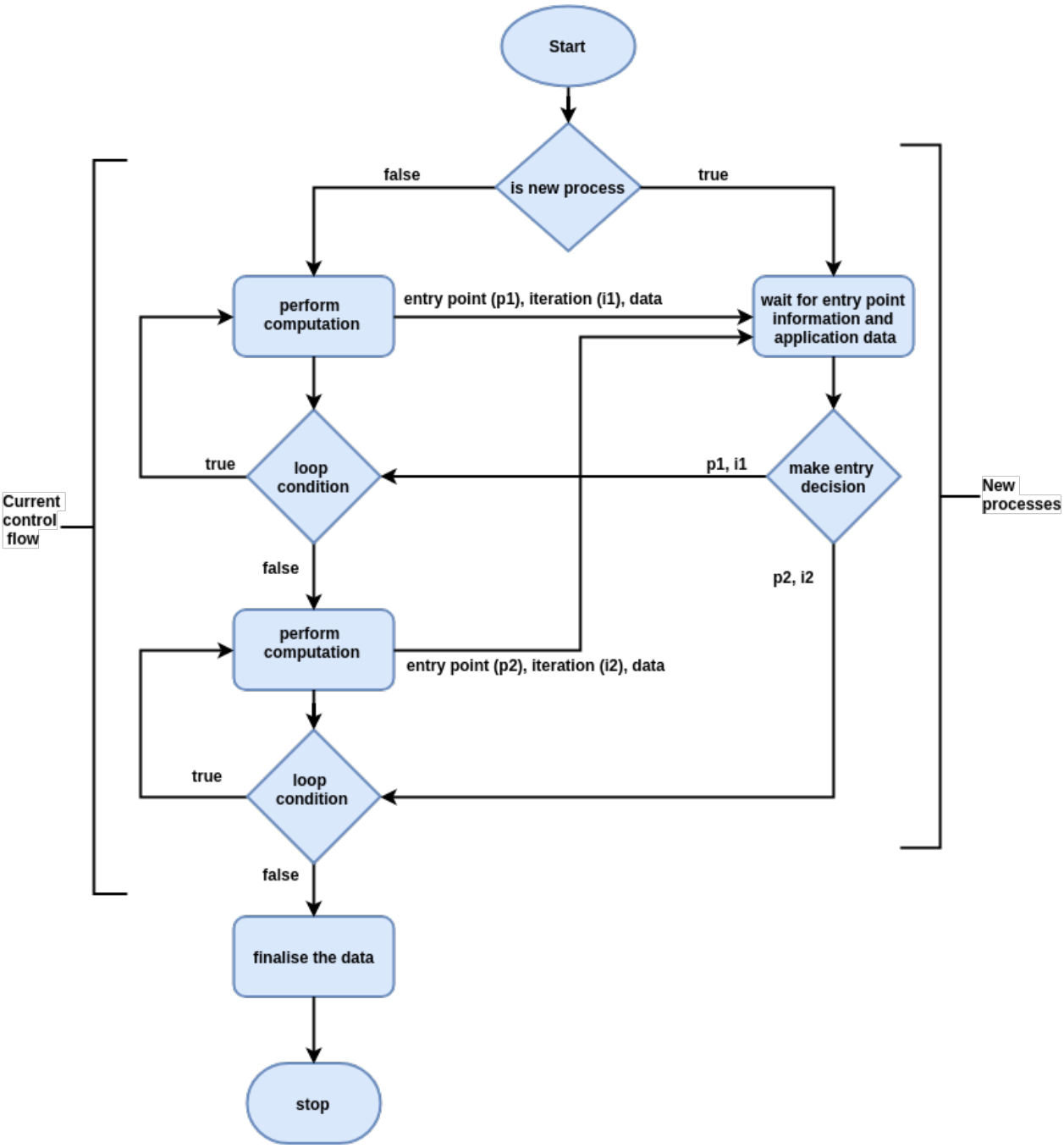

Figure 2.1: Control flow of an Invasive Application

to the application, then corresponding IMPI routine to adapt to the new resources must be called to synchronize the processes. Then, the data should be redistributed among all the processes. In the Jacobi application, after an adaptation, the first task is to change the previous processes mappings and create a new topology with the new dimensions. After that new neighbours should be determined and the data should be redistributed. Finally, the corresponding IMPI routine is called to end the adaptation and to resume the normal execution of the program.

During the adaptation, multiple control statements should be used to differentiate the new processes from the old processes because the newly joined processes should not execute certain part of the program to avoid logical errors. In the Jacobi kernel, the new topology should be formed with the new number of processes and the newly joined processes should wait for the iteration number from the old processes so that it can start its execution from the correct iteration onwards. The newly joined processes should also get all the data required for the computation. During the adaptation, a program can have two different control flows, one for the joining processes and the other for pre-existing processes. They will merge at some point of time. All processes should be synchronized to start working from that point onwards. The joining of new processes to an existing control flow is visualized in Figure 2.1. Similarly, in the case of outgoing processes the data should be redistributed and the exit point in the program from which they can be gracefully removed should be found.

In short, during the development of the invasive application, the programmer is solely responsible for the control flow of the new processes and pre-existing processes, the data distribution and the synchronization of these processes. These difficulties while developing invasive applications became the key motivation for the Elastic Phase Oriented Programming model.

2.2 Solution - Elastic Phase Oriented Programming Model (EPOP)

EPOP treats an application as a collection of different phases. Every normal program has a phase where initialization and distribution of data take place, another phase where computations take place and the final phase where writing results take place.

According to Amdahl's law, speedup of the program is inhibited by the sequential part of the program. In EPOP, these parallelism inhibiting parts of the program are considered as rigid phases and highly parallelizable parts as elastic phases.

EPOP model has a user-friendly syntax that allows a programmer to develop invasive applications with ease. The programmer can specify the phases in which the adaptation should take place and which routine to call once new processes are joined. The programmer is also relieved from including the IMPI routines in their code. EPOP driver will call the IMPI routines for probing, adapting and committing during the execution of the program. The programmer only needs to specify to the driver which routines to be called before adapting i.e as a preparation for adaptation or during adaptation to distribute data. The driver program of the EPOP calls these routines at specified points and controls the program flow. In the Jacobi kernel, the developer can create a phase to do the topology creation, an elastic phase to do the highly parallelizable regions of the code and tag the phase with routines to be called if new processes are added. For example, the programmer can define a function to synchronise the data of pre-existing processes before joining with new processes and ask the driver to call this function just before the adaptation, so that pre-existing processes can prepare for adaptation. The driver program will handle the control flow.

From the perspective of the Invasive Resource Manager, it needs performance data of the application to decide whether the application should be allotted with new resources or to take back the resources. Elastic Phase Oriented Programming (EPOP) model can provide the performance data of the compute-intensive phases, the progress of the application and the structure of the application to the resource manager. As a result, the resource manager can schedule the job efficiently. The resource manager can also pause the EPOP application. The resource manager can send the pause duration and the EPOP driver will pause the application for the provided duration, thereby giving the flexibility to provide resources to other applications.

EPOP is designed using C++11. The EPOP driver runs the application as a shared library, profiles the phases and communicate with the resource manager. EPOP interacts with IMPI library to obtain the profiling information like MPI time. MPI time is the time spent for performing an MPI operation. If profiling is enabled by the user, EPOP provides information such as the number of MPI calls, the time spent for each MPI operations, the time taken for phases and the adaptation time of the application.

3 Related Work

A minimalistic version of Elastic Phase Oriented Programming (EPOP) is previously implemented in C [3]. It is a prototype for demonstrating the phase oriented programming approach. The driver program runs the application phases according to the structure and order defined by the programmer. Our work is an extension of the existing EPOP implementation in a higher level programming language, C++. Our work makes EPOP a fully fledged programming model. This work designs a user-friendly syntax for the declaration of different phases and representation of these phases in C++. Our work provides the driver program with more capabilities, like performance profiling and integration with the resource manager. There is no performance profiling and integration with the resource manager in the previous model. Our EPOP model collects the performance data of the phases and sent it to the resource manager for making better scheduling decisions. The driver program can also pause the execution of the application if requested by the resource manager. It can also provide the profiling information of the application such as the number of MPI calls made, the time spent in MPI calls, the time taken for completion of a phase, the time taken for adaptation and other detailed information. Our model also has frameworks for sending extra information like power value, number of floating point operations. Implementation of the new EPOP model is done using C++11 features like standard function class, unique pointers etc.. Both models use the IMPI constructs for making the program elastic.

IMPI is an extension to the MPI library for making the program invasive. This extension has added four new routines to make the dynamic process creation and reduction [3]. The MPI standard also supports the dynamic creation of processes but with the performance penalty. It does not provide a mechanism for the reduction of dynamic processes. Invasive MPI rectifies these disadvantages of the standard MPI in the dynamic process creation and reduction. Our model is using the IMPI routines in the backend to expand and reduce the dynamic processes. The programmer can achieve the elasticity in the application only using the standard MPI without the need

of calling the invasive specific routines of the IMPI in our programming model. This makes it tolerable for the programmer since he/she does not need to know the syntax and semantics of the IMPI.

Charm++ and Adaptive Message Passing Interface [12], X10 [2] and Parallel Virtual Machine (PVM) [17] are the related works that provides the malleability of the jobs. The main difference between EPOP and these related works are, EPOP is an abstraction of IMPI routines to make the Invasive programming easier. Our work focus on making the standard MPI applications invasive by using the EPOP model while this is not possible with the X10 and PVM. Charm++ and AMPI support the malleability of MPI applications but with more focus on load balancing and virtualization. Our EPOP model focuses on the malleability and performance profiling using IMPI.

Charm++ [12] and Adaptive Message Passing Interface [9, 8] supports the malleability of jobs by checkpoint restart along with the task migration and dynamic load balancing. Adaptive message passing interface abstracts the MPI processes as migratable threads and runtime system of charm++ deals with the scheduling and migration of these threads. Standard MPI is expanded to support the Charm++ runtime system. Adaptive MPI follows a message-driven execution model and there is oversubscription due to the threading. EPOP is based on the invasive properties of the IMPI and uses the standard MPI execution model with no oversubscription. EPOP can provide application specific profiling information to the resource manager since the application is treated as a collection of phases. EPOP has the options of sending power and other performance metrics if available while the Charm++ and AMPI does not.

X10 [2] is an object oriented programming language which is an implementation of asynchronous partitioned global address space (APGAS) programming model. It provides java like features and makes application development easier in distributed memory architectures. X10 version 2.5.1 supports the resource elasticity. Our work focuses on MPI applications to exploit the elasticity. Since the X10 is a higher level programming language and has a philosophical difference from the MPI model, MPI applications cannot utilise the invasive behaviour using X10.

Parallel Virtual Machine (PVM) [17] is a software toolset for abstracting the heterogeneous systems as a single parallel distributed computer. It enables the distributed programming using the message passing model. Users can dynamically add the nodes to the system and resource elasticity is achievable in case of a single job in the PVM.

For multiple jobs, coordination with the resource manager is necessary and PVM does not provide it. The difference between PVM and EPOP is that EPOP works closely with the elastic resource manager. Multiple EPOP jobs can be executed and the adaptation decisions can be made by resource manager with the performance data provided by EPOP. EPOP uses IMPI for adaptation and the legacy MPI applications can be converted to the EPOP model with a tolerable amount of work.

4 Background

4.1 Invasive MPI

Invasive MPI is an extension to the MPICH [14] with new functionalities for dynamic process management [3]. The goals of the Invasive MPI was to reduce the latency and the minimal collective latency in the standard mpi dynamic process creation and also to develop a user friendly model for resource aware programming. Four new operations were added to reach the above goal. They are:

1. `int MPI_Init_adapt(int *argc, char **args, int *local_status);`

   `MPI_Init_adapt` signals the resource manager that the application will be adaptive. The signature of this routine is similar to the standard MPI_Init with only one additional `local_status` parameter. This additional parameter is used after `MPI_Init_adapt` operation to identify the type of the processes, i.e whether the process was created as part of the adaptation or the process was created at the start of the application. `MPI_ADAPT_STATUS_NEW` value in the `local_status` parameter implies that the process was created as part of the job creation. `MPI_ADAPT_STATUS_JOINING` signals that the process was created as part of the adaptation. It is necessary to distinguish between these two sets of processes because the newly joined process should immediately call the `MPI_COMM_ADAPT_BEGIN` to take part in the adaptation window. Pre-existing processes should also call the `MPI_COMM_ADAPT_BEGIN` to continue the adaptation.

2. `int MPI_Probe_adapt(int *pending_adaptation, int *local_status, MPI_info *info);`

   `MPI_Probe_adapt` routine is used to check whether there are any resources available for the expansion or resources to be taken back by the resource manager.

The parameters in the probe function provides the information regarding the adaptation. The `pending_adaptation` parameter of the probe operation can return the value `MPI_ADAPT_TRUE` or `MPI_ADAPT_FALSE` to the pre-existing process. If the output is the former then there is an adaptation pending. Otherwise, the processes can continue its normal flow. The second parameter `local_status` gives insight into the type of the adaptation, whether a process is leaving or joining, by using these flags `MPI_ADAPT_STATUS_LEAVING` , `MPI_STATUS_STAYING` or `MPI_STATUS_JOINING`. Newly joined processes will get this local status from the `MPI_Init_adapt` operation and there is no need to call the `MPI_Probe_adapt` operation.

3. ```
   int MPI_Comm_adapt_begin(MPI_Comm *intercomm, MPI_Comm *intracomm,
               int *stayingcount, int *leavingcount, int *joiningcount);
   ```

   This routine is called to begin the adaptation window. Newly joining processes can call the `MPI_Comm_adapt_begin` immediately after the `MPI_Init_adapt` operation. The joining processes get blocked by this operation and wait for all the processes to call this collective operation. For pre-existing processes, if the `local_status` returned by the `MPI_Probe_adapt` operation is leaving or staying, it should call the `MPI_Comm_adapt_begin` to start the adaptation window. Pre-existing processes are notified about the adaptation only after all of the newly joined processes have called the `MPI_Comm_adapt_begin` routine. This operation returns two communicators `intercomm` and `intracomm`. A communicator is an object that is used to describe a group of processes. All the MPI processes are included in the communicator `MPI_COMM_WORLD`. The first communicator, `intercomm` returned by the `MPI_Comm_adapt_begin` routine contains the group of processes that were created as part of the new resource allocation in case of expansion or the processes leaving in case of reduction. The second communicator `intracomm` contains all the processes in the application. In case of expansion of resources, it contains the joining and pre-existing processes and in case of reduction, it contains the group of staying processes.

4. ```
   int MPI_Comm_adapt_commit( ) ;
   ```

   This routine is called to finish the adaptation. After all the processes finish this routine, the `intracomm` returned by the `MPI_Comm_adapt_begin` will become the new `MPI_COMM_WORLD`. The processes with the status `MPI_ADAPT_STATUS_JOINING`

and MPI_STATUS_STAYING will be staying after this operation. Processes with the status MPI_ADAPT_STATUS_LEAVING will be killed during this operation.

The adaptation window between the two operations MPI_Comm_adapt_begin and MPI_Comm_adapt_commit is used to distribute and synchronize the data if needed. Developers can use the two new communicators provided at the MPI_Comm_adapt_begin routine for the data distribution and the repartition of the topology if needed.

To effectively use the above changes in IMPI, there is a need for an Invasive Resource Manager which can dynamically provide the resources to the Invasive MPI. The integration of the Invasive Resource Manager with this invasive MPI is described in the following section.

IMPI and EPOP

EPOP uses IMPI for resource elasticity. EPOP driver uses the above mentioned four additional operations created by Invasive MPI and acts as an abstraction for it. Application developers does not need to call these MPI_Init_adapt, MPI_Probe_adapt, MPI_Comm_adapt_begin or MPI_Comm_adapt_commit operations. These four operations are called automatically by the EPOP driver.

The EPOP driver profiles the application by calculating the MPI time, number of MPI collective and point to point calls were made, the average time spends for the MPI calls in a phase etc. As part of the work, Invasive MPI is extended to provide this profiling information to the EPOP driver. MPI calls made between the start and end of the phases are found out and the time spent for these calls are calculated and is sent to the resource manager for making scheduling decisions. The EPOP informs the IMPI that job is epop by setting the flag in the IMPI.

4.2 Invasive Resource Manager

Invasive Resource Manager is an extension to the scalable workload manager Simple Linux Utility for Resource Management (SLURM). SLURM is a popular open source

resource manager in HPC systems [11]. SLURM contains a configuration file which is used by its components to configure the parameters like the scheduling interval, the number of nodes etc.. SLURM contains a collection of binaries. The following binaries `SLURMCTLD, SLURMD, SLURMSTEPD, SRUN` are important for us.

`SLURMCTLD` or the slurm controller is the centralized component which is concerned with the scheduling, tracking the nodes, allocating the resources. Since the `SLURMCTLD` is the centralized component, all user level interactions are communicated to the controller. `SRUN` is another important binary associated with SLURM. `SRUN` launches the job in the nodes. It takes jobs from the user and asks the `SLURMD` daemon in the nodes to create processes. `SLURMD` daemon is present in each node of the partition. `SLURMD` starts a `SLURMSTEPD` daemon in each node which in turn creates and tracks the local part of the processes.

Invasive Resource Manager extends the SLURM to make it invasive and to integrate the invasive MPI to it. `SLURMCTLD` daemon is not present in the Invasive Resource Manager. It is replaced by the invasive batch scheduler and invasive runtime scheduler. The tasks of the controller are now performed by the invasive batch scheduler and the dynamic scheduling required for the invasive application is implemented by the invasive resource scheduler [3]. Figure 4.1 shows the interaction between the Invasive Resource Manager and the Invasive MPI. A new message type `srun_reallocation_message` and its handlers were created in the scheduler to initiate the adaptation processes. This message contains the node information in which new processes should be created, number of processes to be created and other required metadata for the reallocation. The resource manager can sent reallocation message according to the availability of the resources and by considering the performance metrics of the application.

Node daemons are modified to interact with the IMPI operations - `MPI_Probe_adapt` and `MPI_Comm_adapt_begin`. MPI processes are created by the `SLURMDSTEPDAEMON` which is forked by the `SLURMD` per node. Like in the initial SLURM architecture, there is one `SLURMD` daemon per node and `SLURMDSTEPDAEMON` takes care of the local parts of the process. `SLURMDSTEPDAEMON` creates the mpi processes according to the instruction from the launcher. `SRUN` is extended to send and receive the information from and to the invasive scheduler. `SLURMD` node daemons inform `SRUN` about the status of the adaptation. `SRUN` is modified extensively to accommodate the propagation of control specific information about the nodes and processes.

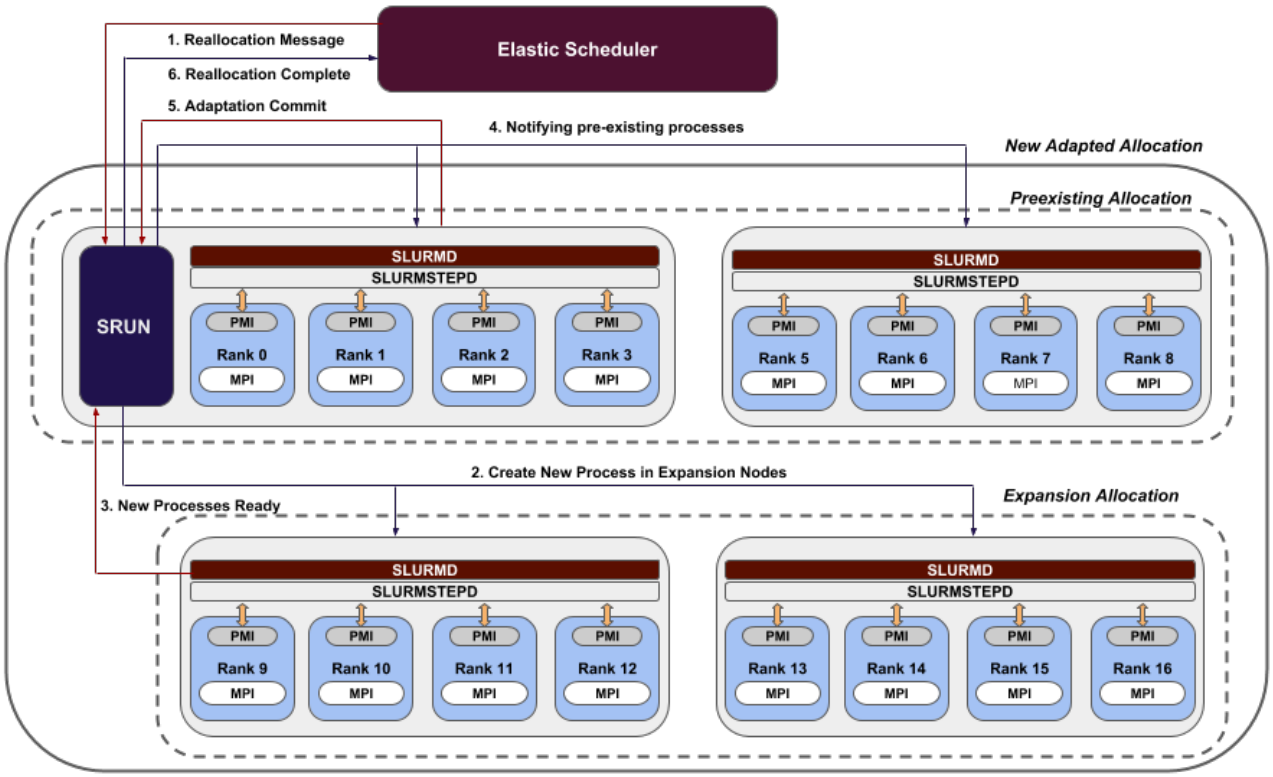

Figure 4.1: Integration of IRM with IMPI [3].

The steps performed during an adaptation can be summarized into six points (shown in the Figure 4.1). The Invasive Resource Manager informs the decision of scheduler to allocate new resources to the launcher `SRUN` by sending a `srun_reallocation_message`. Upon receiving the reallocation message, `SRUN` sends the instruction to the `SLURMD` daemon in each node to take the necessary action i.e if the reallocation message is for expansion, then the `SRUN` instructs the `SLURMD` daemon to create the new processes in the node for the reallocation. After creating the required processes requested by the `SRUN`, `SLURMD` notifies the `SRUN` whenever all the newly created processes call the `MPI_Comm_adapt_begin`. After receiving this notification, `SRUN` informs all the pre-existing processes through the `SLURMD` about the adaptation. The pre-existing processes get this information through the `MPI_probe_adapt` . Now the pre-existing processes can enter the adapt window and can communicate with the newly created processes.

An adaptation is committed only when the `MPI_Comm_adapt_commit` operation is called by all of the participating processes. The leader node of the `SLURMD` notifies the `SRUN` that adaptation is finished. After receiving the adaptation commit information from the daemons, the `SRUN` informs the scheduler that the reallocation is completed. `SRUN` then receives the updated credentials of the participating nodes and its statuses.

Invasive Resource scheduler allocates the node to the application based on the heuristics of the running application. It requests the performance of the application from the `SLURMD` daemons and these daemons collect information from the IMPI. Based on the time information, the looping characteristics of the application, the collective MPI calls and the pattern detection, performance data is collected. This data is then aggregated into the resource manager upon the request and an adaptation decision is based on these collected data. The metric MPI time to compute time is considered for making the adaptation decision. Time balancing and resource filling is applied for elastic backfilling i.e to reduce the idle node counts. Parameters like the current node count, the estimated time of completion, the adaptation time, the resource range are also considered while taking decisions for the backfilling.

IRM and EPOP

Our EPOP model provides the performance data directly to the resource manager. Instead of the resource manager requesting the IMPI and performing the pattern

detection, the EPOP driver provides the required performance data directly for making adaptation decisions. The resource manager is modified to request and receive these data from the EPOP master. Additional handlers are added in the resource manager to identify whether the job is EPOP or not. New functions are added to the resource manager to utilise this information from the driver. `get_epop_mtct_time`, `get_epop_current_phase` etc are some of the added routines. These methods are explained in the following sections.

5 EPOP Design

The design philosophy of the Elastic Phase Oriented Programming model comes from the following three notions.

1. The Amdahl's law states that the speedup of an application is inhibited by its sequential part i.e. the execution time of the parallel part can only be decreased by increasing the number of processes.

2. Every parallel program can be thought of as having three logical parts. The first part can be an initialization part where all the input operations and the data distribution takes place. The second part can be a compute-intensive part where the actual computation takes place which may normally be inside a looping construct. And the last part can be a finalizing part that writes the results.

3. The invasive MPI along with the Invasive Resource Manager is used to make an application adaptive. i.e. new resources can be added to improve the performance if the performance of a compute-intensive application is reduced due to lack of resources or the existing resources can be taken back if an application has no performance improvement.

Adding the third notion makes the application programming complex. A program then needs to have two different control flows whenever the new resources are added. It is the application programmer who is responsible for creating the adaptation window and checking for the new resources. Hence, the programmer needs to logically structure the application as a collection of multiple phases to tackle the addition of new resources. For example, the programmer should find out a suitable entry point for the new processes. Everything above that particular point can be identified as one phase and everything below as another phase. Such a design philosophy can make the invasive programming easier.

Merging the above three notions forms the philosophy for the Elastic Phase Oriented Programming Model. The EPOP model converts the above notions into a working programming model for a resource-aware distributed programming. EPOP treats a program as a collection of multiple phases. According to the Amdahl's law, the parallelism inhibiting serial parts of an application like the initialization part, is not a candidate for parallel performance improvement and can be termed as an init phase or a rigid phase in the EPOP terminology. The compute intensive part of the application, which is a good candidate for malleability and belongs to the parallel part in the Amdahl's law, is called as an elastic phase in the EPOP paradigm. An application is a collection of non-parallelizable and parallelizable phases in the EPOP model. Every invasive application can be designed in such a model.

5.1 Model

The Elastic Phase Oriented Programming Model is an abstraction for modularizing a program. Programmers can divide their code segment into different "phases" and can describe the control flow using the simple syntax introduced by our model. There are four different phases in EPOP and they can be used to develop a resource elastic or a non-resource elastic application. These phases are described below in detail.

5.1.1 Phases in EPOP

The Init, Rigid, Elastic and Branch phases are the building blocks of EPOP. These phases mark the computational blocks of an application as non resource elastic and resource elastic phases. The application developer can declare the phases at the beginning of the application and can create a vector of the phases using the helper methods provided by the EPOP driver. The driver of the EPOP program iterates through this vector and calls the phases according to the phase type. The order of declaring the phases are important since the driver program iterates through the phases automatically in that order. The first phase in EPOP must be an initialization phase.

Init Phase

Init phase is the first phase of every EPOP program. All computations required for the initialization of an application can be provided in this phase. There should be only one initialization phase in an EPOP application and it is called only once by the EPOP driver for each of the processes during the entire lifecycle of an application. All the processes, even the newly allocated processes, should call this initialization phase.

Elastic Phase

Elastic phase is the core concept of this programming model. Elastic phases mark the computational part of the program where resource adaptation is required. An application may contain multiple elastic phases. Each of the elastic phases should contain a code block where the actual computation takes place, a loop condition that specifies the number of times a code block should be executed and an adaptation block. The adaptation block is executed whenever new resources are available or pre-existing resources are taken back. There are separate adaptation blocks for the joining processes, leaving processes and also for the existing processes to prepare for the adaptation. A loop definition is mandatory in this phase since resource adaptation is necessary only for a compute-intensive application. It also determines when to exit from the current phase. The performance data of this phase is taken by the driver and is communicated to the resource manager for taking scheduling decisions. MPI calls made in this phase are recorded by the driver for profiling activities.

Rigid Phase

Rigid phase does not allow resource elasticity. This phase is used to define the code block which should be executed without a resource adaptation. It is used for defining the parallelism inhibiting portions of an application. The part where an application is finalizing its execution is a suitable candidate for a rigid phase. The non-resource elastic applications can be written using this phase. A loop condition can also be used in the rigid phase which specifies the number of times a phase should be executed.

Branch Phase

This phase is used to change the control flow of the application. In a normal scenario, the EPOP driver iterates through the phases in the order of phase creation. If the application needs to jump from one phase to another phase a branch phase can be used. It determines the control flow of the driver. A branch phase can be used for forward jumps as well as backward jumps. It makes the branching decision using the value returned after the execution of a branch phase. This return value is set as the new program counter and the driver starts executing the program accordingly. For example, if a branch phase is declared as the third phase and if it returns the value two, then the driver will jump to the second phase declared in the application and resumes the program execution.

5.2 EPOP Driver

The EPOP driver is in charge of the control flow and passing the data between phases. The EPOP driver loads the provided EPOP program and starts iterates through the phases and executes the phases according to the phase conditions. It implements the functionalities for probing new resources, creating adaptation windows, passing the phase details to the newly joined processes and so on. The working of the driver is as follows:

1. The EPOP driver will call the IMPI initialization function at the beginning of the program. This function marks the job as an invasive job in the resource manager.
2. The EPOP driver calls the code block associated with the init phase.
3. After calling the init phase, the driver checks whether the current process is part of a newly joining processes or it is created as part of a job creation.
4. If the process is created as part of a resource expansion:
    - The driver will immediately call the IMPI routine for starting the adaptation and waits for all the pre-existing processes to call the same IMPI routine.

- After all the processes call the adaptation routine, the driver receives the phase index from the pre-existing EPOP processes so that the new processes can start executing from the same phase as the pre-existing processes. The driver will then call the code block for the expansion if it is provided by the user.

- After that the driver will call the IMPI routine for finishing the adaptation and goes to the phase provided by the pre-existing processes.

5. If the process is created as part of a job creation, the driver will go to the next phase.

6. If the next phase is an elastic phase:

   - the driver will begin the execution by calling the IMPI routine to check for any pending adaptations.

   - If there is no pending adaptation, then the driver will call the code block associated with the elastic phase and check for the loop condition after finishing the execution. The driver will continue calling the elastic phase until the loop condition becomes false.

   - If there is any pending adaptation, then the driver will call the corresponding IMPI routine for starting the adaptation. If the adaptation type is an expansion of the resources then the newly joined processes will be waiting for the pre-existing processes. Then the driver will send the current phase index to the newly joining processes so that they can start the execution from the current phase. The driver will then call the code block for the expansion if provided by the user. If the adaptation type is a reduction, then the driver will call the code block for the reduction if provided by the user. After calling the corresponding code block, the driver will call the IMPI routine for finishing the adaptation and continue the phase execution with the new number of processes. The driver will then check for the loop condition. The driver will continue calling the elastic phase until the loop condition becomes false.

   - The driver will call the next phase when the loop condition becomes false.

7. If the next phase is a rigid phase,
    - The driver will call the code block associated with it.
    - If there is a loop condition, then the driver will check the loop condition and call the rigid phase until the loop condition becomes false.
    - The driver will call the next phase if there is no loop condition or the loop condition is false.
8. If the next phase is a branch phase,
    - The driver will call the code block associated with the branch phase. The return value from this execution determines the branching.
    - The driver will jump to the index contained in the return value. This can be a forward or backward jump.
    - If the return value is the same phase index as the branch phase then the driver will continue its normal execution and call the next phase.

This is the application side of the driver. Another part of the driver deals with the performance profiling. It is the EPOP master which obtains the performance data from the IMPI, aggregates the data from all of the processes and sends it to the resource manager. The EPOP master is the process with the MPI Rank 0. It provides the phase specific information to users if the profiling is enabled.

1. At the beginning of the program, the driver gets the job id from the IMPI and the driver registers itself to the resource manager as an EPOP job.
2. Before calling init and rigid phase, the driver records the time and the time taken for executing the phase is calculated and saved after the end of a phase.
3. If the phase is an elastic phase,
    - At the beginning of each iteration of the phase, the driver calls the IMPI method for recording the MPI time and then the driver locally starts a timer to record the time taken for the iteration.

- After the end of the iteration, the time taken for the iteration is found out and the IMPI routine is also called to get the MPI time. If there are more than one iterations, the average time is found out.
- All of the EPOP processes pass their average time or other metrics to the EPOP master.
- The EPOP master saves the value from the other EPOP processes and sends to the resource manager when requested.

4. If the profiling is enabled, then the EPOP master will write the performance metrics at the end of each phase.

5.3 Control Flow and Data

The control flow of an EPOP program is determined by the EPOP driver according to the phase declaration and the branch phase. Different phases are combined in a logical order to form an EPOP program. The order in which the phases are executed is the order in which the phases are declared in an application. A simple EPOP program constitutes an init phase which initializes the code block, then one or more elastic phases and their loop conditions and finally a rigid phase to finalize the computation. Branching between the phases depends on the value returned after executing the branch phase. The driver adjusts the program counter according to the value returned by the branch phase and jumps to it.

If we consider the Jacobi kernel mentioned in the motivation section, it can be divided into five phases. An init phase that initializes the data needed by the application, a rigid phase which creates the topology and Cartesian grid needed for the application and an elastic phase that calls the Jacobi kernel and starts iterating through different resolutions. A branch phase can be used to calculate the solution again by jumping into the elastic phase. The data is distributed among the processes at the beginning of the elastic phase. The adaptation blocks of this elastic phase can be used to pass the next resolution to the newly joined processes or to take the value from the leaving processes. A rigid phase at the end can write the output and finalize the operation. The Figure 5.1 shows the control flow for such an application.

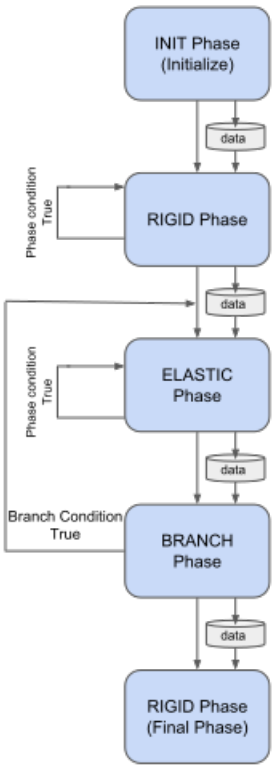

Figure 5.1: Control flow and data in an EPOP program

Data and phases are decoupled like in a functional programming paradigm. All data, whether it is static or allocated, file descriptors and pointers should be created in the data block. No application state is maintained by an EPOP program. Everything associated with the computation should be stored in a block. Data blocks are passed between phases according to the control flow of a program.

5.4 Performance Modelling

The Invasive Resource Manager needs the performance data of an application to decide whether the reallocation is necessary. For the normal IMPI applications, the IRM requests the `SRUN` for the performance data. The `SRUN` requests the `SLURMD` daemons in each node for the data. `SLURMSTEPD` daemons take the required metrics from the IMPI. IRM aggregates the received data from all the participating nodes and finds out the MPI time to compute time ratio. If this ratio is within a specific threshold, the resource manager makes the reallocation decisions. In our EPOP model, the MPI time to compute time ratio is directly sent by the EPOP master to the resource manager. As a result, the IRM does not need to aggregate the data and calculate the ratio.

The EPOP driver measures the phase time at the beginning of each phase and at the end of each iteration of the phase. The average phase time is computed at each phase pass and is stored in the data structure for the performance metrics. The IMPI is extended to obtain the performance data needed by EPOP. Routines are added before the MPI calls to measure the time spent in each call. The total MPI time spent in a phase is returned by the extended IMPI routine. The MPI time counter is reset at the beginning of each phase using the EPOP routine inside the IMPI. Each EPOP process calculates their own MPI time details in each phase. The MPI time to phase time ratio is calculated locally in each of these EPOP processes. The average MPI time to phase time ratio is computed as the phase iteration increases. The EPOP master aggregates this data from the other EPOP process via MPI Reduction and then it is stored in its internal data structure. The frequency of the reduction is determined by a reduction threshold. The reduction threshold is calculated according to the average phase time.

There are two approaches used for sending the performance data to the resource manager. First strategy is a simple polling strategy, which is currently used for all IMPI

jobs now. At the time of a scheduler pass, the resource manager requests the driver for the performance data. Upon request from the resource manager, the driver sends the performance data to the resource manager. In the next scheduler pass, the resource manager makes the scheduling decision with the data received from the EPOP master. Next strategy is implemented to reduce the decision making time. It is a performance data push strategy. Whenever the driver detects a change in the performance data or after the initial execution of the application, the performance data is pushed into the resource manager. So that the resource manager can make the decision in the first scheduler pass itself. As a result the scheduling decision is made faster. The performance data is pushed in the following scenarios:

1. After the first iteration of the elastic phase so that the resource manager can make decisions even at the first scheduler pass.

2. The driver records the previously sent performance data. After calculating the new performance data, it compares the new value with the previous data and if there is a significant difference in both values, then the new data is pushed into the resource manager. i.e. if the previous data was .7896 and new data is .9867, then the performance data is pushed. If there is no change in data, then the driver will only send the performance data upon receiving the request from the resource manager.

3. After the first iteration (or according to the reduction threshold) of the elastic phase after every adaptation.

At the beginning of the EPOP program, the EPOP master packs the structure of the program along with the job number and the master's IP address and is sent to the resource manager. The resource manager is extended to accept the connection from the EPOP master and the received data is stored for later use. The resource manager marks the corresponding job id as EPOP in its job pointer. The job pointer in the resource manager contains the information of a job such as the allocated nodes, whether the job is elastic, the performance data and so on.

5.5 Pausing an application

Another design concept of EPOP is to pause the program execution. A mechanism for sleeping the program is implemented in the driver. The Invasive Resource Manager can request the EPOP driver to pause the program for a duration of t seconds. The EPOP master will broadcast the pause message along with the duration received from the resource manager to the other EPOP processes. After the specified duration, the EPOP processes will check for a message from the EPOP master. If the resource manager doesn't send any other message, after the fixed duration, the EPOP master will inform all the other EPOP processes to resume the execution. This strategy is useful if the scheduler wants to execute a time critical job and the required number of nodes are not available. Then the scheduler can request the EPOP programs to go to sleep mode and then give the nodes to the time critical job.

5.6 Profiling

The EPOP driver profiles the phases in the program. If the profiling is enabled by a user, the EPOP driver provides the detailed information regarding a phase such as types of MPI calls that were made, number of calls, the time spent for each MPI call in a phase, the average MPI time of a phase, the computation time of a phase, the time taken for adaptation and invasive related operations, the overhead induced by the IMPI probe and the adapt operations. This information is written into a log file. A user will get a real-time performance analysis of the current phase of an application by inspecting the log file. The probing interval of the EPOP driver can be set by the user.

5.7 Integration with IMPI and IRM

The Figure 5.2 shows the architecture of an EPOP application. The EPOP process that has the MPI rank of 0 is termed as EPOP master. The EPOP master is in charge of all the communication between the resource manager and EPOP. Invasive Resource Manager is extended to communicate with the EPOP model and to store the information

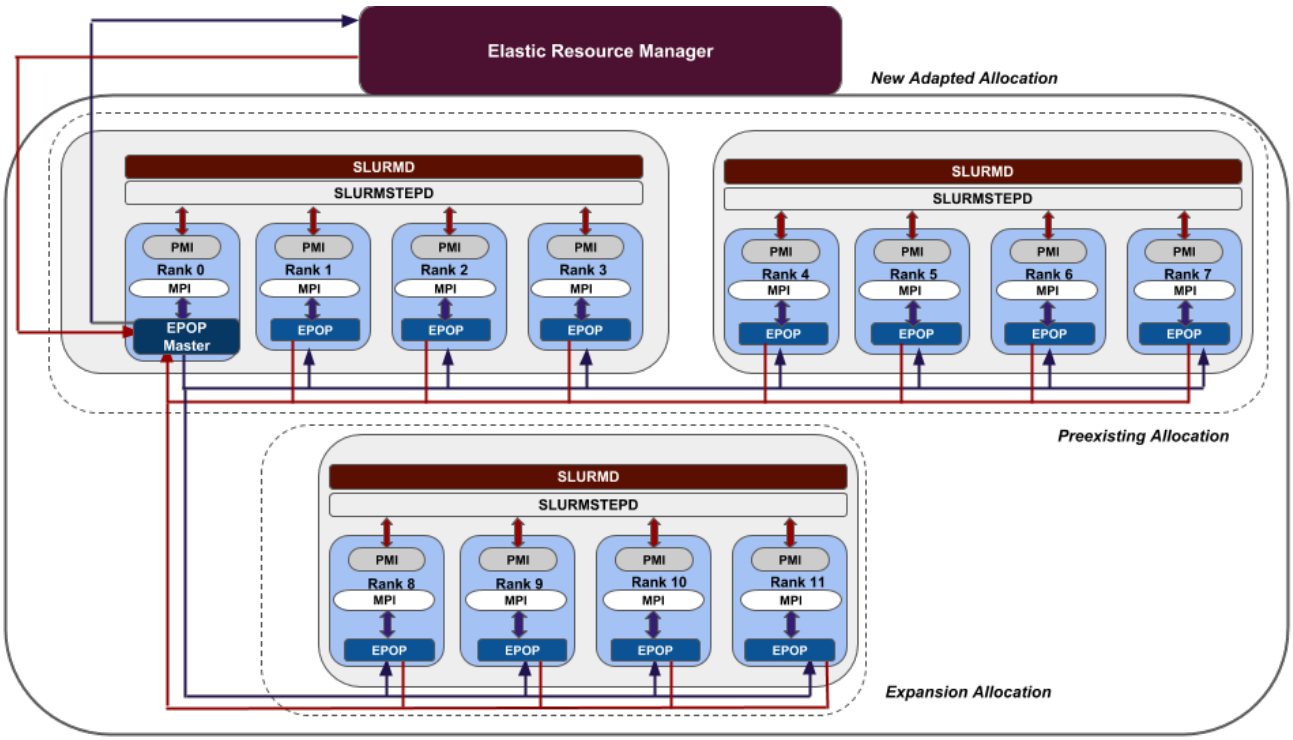

Figure 5.2: Integration of EPOP with IMPI and IRM

received from the EPOP master. All of the EPOP processes are interacting with the MPI library for obtaining the performance data. The IMPI is modified to handle the communication from the EPOP driver. The EPOP master broadcasts the message from the resource manager to all other EPOP processes using the MPI collective operations. The performance data from each process is aggregated at the EPOP master.

The communication between the resource manager and invasive MPI is described in detail in Section 4.2. In the Figure 5.2, the red lines represent all of the communication made to the EPOP master and the dark blue lines represent all of the communication made by the EPOP master. All the EPOP processes interact with IMPI for sending and receiving the profiling information. It is shown in the Figure 5.2 as a double-headed arrow.

The order and type of the communication between the EPOP master, IRM and IMPI is as follows.

1. At the beginning of the application, the EPOP master asks the IMPI for its job id through an extension function added in the IMPI.

2. After receiving the job id, the EPOP master sends the structure of the EPOP program to the resource manager. The structure includes the details such as the total number of phases, the type of phases, the looping information of the phases and the IP address of the EPOP master. Since the EPOP master can be executed in any node, the IP address must be sent to the resource manager so that the resource manager can communicate with the EPOP master.

3. Upon receiving the structure of the EPOP application, the resource manager stores this information and tags the job pointer associated with it as an EPOP job.

4. The scheduler needs the performance metrics to make the reallocation decision. The scheduler checks if the job is EPOP and if it is an EPOP job then the scheduler will sent a request for the performance metrics to the EPOP master. The performance metrics can be anything from the MPI time to compute time, the phase time, the power or the floating point operations. The scheduler sends the request message type as MTCT for the MPI time to compute time ratio or TIME for the phase time.

5. Upon receiving the performance metrics request from IRM, the EPOP master checks for the type of metrics requested and the requested metrics is sent along with the current phase index.

6. The EPOP master pushes the performance metrics to IRM in some scenarios even without the request from IRM. For example, if the EPOP master sees a significant change in value of the performance metrics compared to the previous value, then the performance metrics is pushed.

7. The IRM stores the performance metrics and uses it in the next scheduler pass to decide about the reallocation.

8. The EPOP master aggregates the phase time and the MPI time to compute time ratio from all the other EPOP processes via MPI collective calls. The performance data is aggregated at the end of each phase iteration or according to a reduction threshold.

9. Each of the EPOP processes calls the corresponding MPI function to obtain the MPI time after the end of each phase iteration. If the profiling is enabled by the user, additional information like the number of MPI calls and the type of MPI calls were made are also obtained by the EPOP driver.

10. The IMPI operations for the adaptation probing, creating the adaptation window and committing the adaptation window are called by the EPOP driver. In case of the newly joined EPOP processes, the driver calls the init phase followed by the `MPI_Comm_adapt_begin` routine to start the adapt window and then waits for all the other processes to join the adaptation. In pre-existing processes, after the each iteration of an elastic phase, the driver calls the `MPI_Probe_adapt` routine to check for a pending adaptation. If there is a pending adaptation, then the EPOP driver calls the `MPI_Comm_adapt_begin` routine to be part of the adapt window.

11. After all the processes enters the adapt window, the EPOP master sends the current phase index to all the joining processes so that they can reach the correct phase immediately after an adaptation and can start the computations. In the adapt window, the elastic phase specific functions for adaptation provided in the application are called.

12. If the adaptation type is a reduction, then the corresponding elastic phase specific function for reduction is called by the driver. Leaving processes will be killed after calling this IMPI routine `MPI_Comm_adapt_commit` and the staying processes continue the execution.

13. After calling the elastic phase specific functions provided in the application, the drivers of both pre-existing and newly joining processes call the routine `MPI_Comm_adapt_commit` to finalize the adaptation.

6 EPOP Implementation

6.1 Programming Languages

EPOP is implemented with C++11 features like unique pointers, `std::function` class etc. The object oriented concept of C++ along with the features like unique pointers, lambda functions, vectors and support for MPI make C++ the right choice for EPOP. C programs can directly be used in our model with few changes. By using our EPOP model, legacy MPI applications can be converted into an invasive application with a tolerable amount of work from the developer side. The changes made to accommodate EPOP in the Invasive Resource Manager and the extensions made to the MPI for profiling is implemented in C. The network modules are implemented in C.

6.2 Phases

Phases are the core concept of the EPOP programming model. The driver program needs to iterate through the phases one by one in the given order and executes the code block associated with the phase. In order for the driver program to iterate through these phases swiftly, it should be of similar type. Since each phase is different from one another conceptually and have different functionalities, the object oriented concepts of C++ must be exploited to design the phases.

The phases can be represented as classes and the driver program can iterate through its objects. Though some phases share some common attributes, the behaviour of the phases are entirely different. Since there are multiple phases, the object should be identified by the driver program to call its corresponding methods. This problem can be solved by polymorphism and inheritance (see Figure 6.1). Creating a superclass Phase

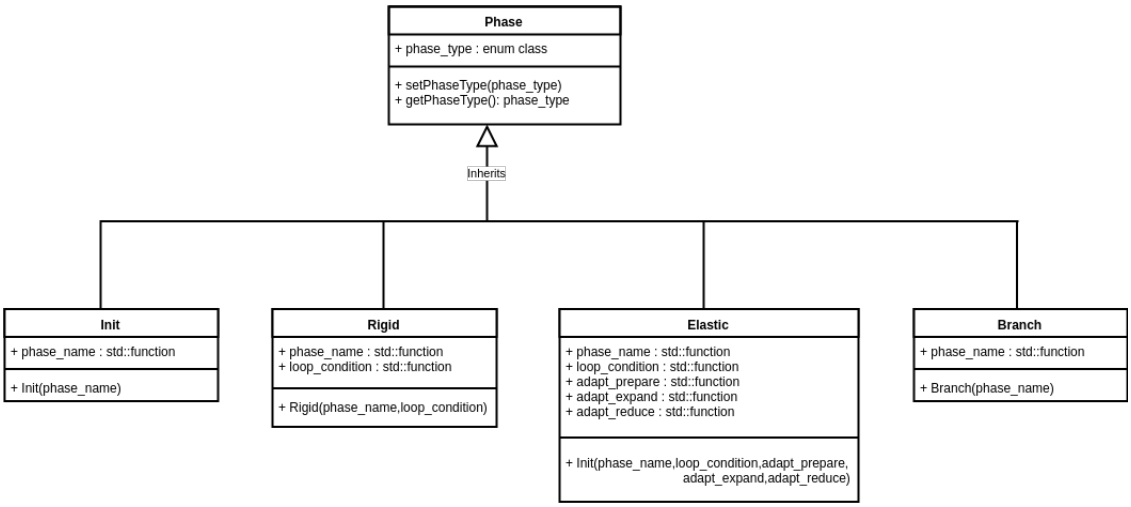

Figure 6.1: UML class diagram of the EPOP Phases

which has the type member variable to identify the phase type and all the subclasses which represent the different phases can inherit this superclass Phase. The objects of the superclass can be created and can be cast to the required phase subclass by checking the `phase_type` member of the superclass. As a result, the driver can iterate through the objects and can check its type using the member variable of the superclass. Then the object can be cast to the corresponding phase subclass and the methods associated with that phase can be called. Figure 6.1 shows the UML class diagram of the phases implemented in our EPOP model.

The classes, inheritance and polymorphism are abstracted from the developers. The application developer does not need to create the objects of the phases. The developers can define the phases with a set of helper methods (see next subsection). But it is necessary that the application developer knows the signature of the functions in the phases. For example, in the elastic phase object, `adapt_expand` member is executed by the driver whenever an expansion adaptation occurs. The argument passed by the driver to the member `adapt_expand` is the number of joining processes, the number of leaving processes, new communicator etc. The application developer needs to write this function with a fixed order and number of arguments.

6.2.1 Init Phase

The Init phase is implemented as a subclass of the superclass Phase (see Figure 6.2). The Init phase has one member variable `phase_name` of the type `std::function`. This `std::function` can store as well as invoke the lambda expressions or functions. Lambda expressions are anonymous function objects which can be used to write inline functions that are not candidates for reuse. `std::function` can also store the pointers to the member functions. `phase_name` is used to point to the function that defines the initialization code block. The constructor of the Init phase class sets the value of its member variable. A set of helper methods abstract the object creation and the member variable assignments from the application programmer. The programmer can use the helper method `setInit()` to set the value of the member variable. Syntax of the `setInit()`

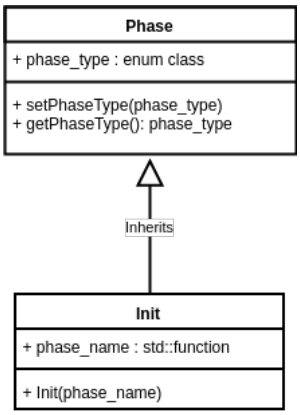

Figure 6.2: UML class diagram of the Init Phase

helper method is as follows:

```
void setInit(std::function);
```

The signature of the Init phase function should be:

```
void phase_function_name (int argc, char** argv, void** data, int rank,
                                          int size, int local_status);
```

The order and the number of parameters in the function are fixed but the Init phase function name can be any valid C++ identifier. The driver calls the Init phase function with the command line arguments, along with the MPI rank, `size` and `local_status` as the function parameters. `local_status` is provided by `MPI_Init_adapt()` to the driver and the driver passes it to the application program through the phase function. The phase function written by the developer should be able to handle these arguments. MPI initialization routine is called by the driver before calling the Init phase function. For an invasive application, `MPI_Init_adapt()` should be called instead of `MPI_Init()`. As mentioned earlier, all IMPI related operations are implemented by the driver in the programming model.

This phase is mandatory and the application execution will fail if not defined. Every EPOP program should have an Init Phase. The developer can write the initialization code as a function or a lambda expression and pass it as an argument to the helper method `setInit()`. `setInit()` will create the object of the Init class. A sample source code for declaring an init phase is shown in Listing 6.1.

```
auto init_phase = [](int argc, char** argv, void** data,
                     int rank, int size, int local_status) -> void {
// code in initialization block
}
setInit(init_phase);
```

Listing 6.1: Declaration of an Init Phase

6.2.2 Rigid Phase

A Rigid phase can have a loop condition. To accommodate the loop condition, a Rigid class has two member variables. One is for the rigid phase block and the other is for the loop condition. Both the member variables are of the type `std::function`. The driver executes the target of the rigid phase member variable and checks for the loop condition. The loop condition is optional and will only execute if it is present otherwise, it iterates into the next phase. If there is a loop condition, the return value from the loop condition is used to decide whether to execute the phase again or to go to the next phase. A Loop condition can return true or false in this model.

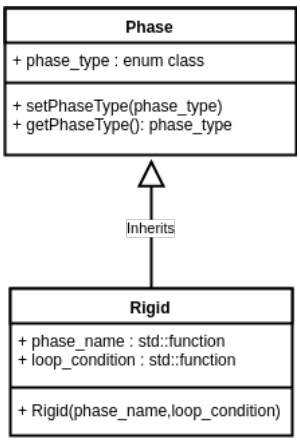

Figure 6.3: UML class diagram of the Rigid Phase

A Rigid phase is optional and an application can be written without the Rigid phase. The developer can write the code for the block and can use the helper method `setRigid()` to set the rigid phase. Since this phase has two member variables, the helper method takes two arguments. The syntax of the helper method `setRigid()` is as follows:

```
void setRigid(std::function phase_function, std::function loop_condition);
```

The loop condition is optional and can be avoided while calling the helper method. The signature for the Rigid Phase function and the loop condition is as follows:

```
void phase_function_name (void ** data);
bool loop_condition (void ** data);
```

The number and type of the arguments and return types are fixed for both functions but the name can be any valid C++ identifier. The driver passes the data to the phase function as a void pointer and the application developer can cast it into the required datatype (see section Data and Control Flow). The return type of the phase function is void since it does not need to interact with the driver.

A Sample source code for declaring a Rigid phase is shown in Listing 6.2.

```
auto rigid_phase = [](void** data) -> void {
// code in rigid block
}

auto rigid_loop_condition = [](void** data) -> bool {
// decision making logic
// return true or false
}

setRigid(rigid_phase,rigid_loop_condition);
setRigid(rigid_phase); // if the phase is to be executed only once
```

Listing 6.2: Declaration of a Rigid Phase

6.2.3 Elastic Phase

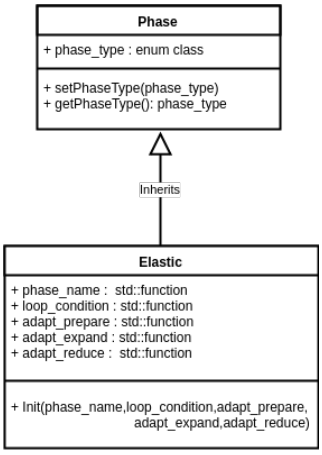

Figure 6.4: UML class diagram of the Elastic Phase

Like a rigid phase, an elastic phase has `std::function` member variables for defining the code for the phase and also for the loop condition. Both members are mandatory and their absence will lead to the exit of the application. The signature for the Elastic Phase function and the loop condition is as follows:

```
void phase_function_name (void ** data);
bool loop_condition (void ** data);
```

Unlike the rigid phase, the Elastic phase should deal with adaptation. To accommodate the adaptation and to make the invasive programming easier, three additional member variables of type `std::function` are added. Each member variable will assist the developer at different stages of the adaptation.

The first member variable is `adapt_prepare`. It is to call the block of code that needs to be executed just before an adaptation. The reason behind this member is that in some scenarios, the application needs to synchronize the data before the adaptation happens. So in such scenarios, the `adapt_prepare` method can be used to do the preparation for adaptation. This block will only be called by the driver if the `MPI_Probe_adapt` function returns positive for the adaptation. This block will be exclusively called by all the pre-existing processes. The syntax of the `adapt_prepare` function is as follows:

```
void  adapt_prepare(void** data);
```

This is an optional function and the driver will call this routine only if it is provided by the application developer.

The second member variable is `adapt_expand`. This is used for defining the code block that should be called by both the pre-existing processes and the newly joined processes when the adaptation begins. This block will only be called by the driver if the adaptation type is an expansion. This code block can be used to send the current iteration number or the file offset index or any data to the newly joined processes to synchronize all the processes in the program. The syntax of the `adapt_expand` function is as follows:

```
void  adapt_expand(void** data, int local_status, int staying_count,
                 int joining_count, MPI_Comm Intercomm, MPI_Comm Newcomm);
```

The signature of the function includes the parameters for the values like `local_status`, `joining_count`, `staying_count`, communicators for the old and new processes. The parameter `local_status` provide the information about the processes such as whether

it is a newly joined process or an old process. `joining_count` provides the number of newly joining processes in this expansion. `staying_count` is the number of the pre-existing processes in the expansion operation. The `Intercomm` is the MPI `communicator` solely consisting of the newly joined processes. This `communicator` can be used for the communication with the newly joined processes by the pre-existing processes. The `Newcomm` MPI `communicator` contains the new rank and order of the processes after the adaptation. This `communicator` includes both the newly joined and pre-existing processes. It will be the new `MPI_COMM_WORLD` once an adaptation is finished. By using this `communicator`, the application can do data distribution among all the processes in the adaptation window. The `adapt_expand` function is optional and an elastic phase can be defined without this routine. The driver will only call this function if available.

`adapt_reduce` is the third member variable. It is used to define the code block that will be called by the driver if the adaptation type is a reduction. This block can also be used for the data distribution and synchronization among all the processes as per the needs of the application. The syntax of the `adapt_reduce` function is as follows:

```
void  adapt_reduce(void** data, int local_status, int staying_count,
                   int leaving_count, MPI_Comm Intercomm,MPI_Comm Newcomm);
```

The signature of the functions `adapt_expand` and `adapt_reduce` differs only slightly. Instead of the joining count in `adapt_expand` the `adapt_reduce` gets the leaving count of the processes. The `local_status` of the process informs whether that the process is staying or leaving. The `staying_count` provides the number of processes that will remain once an adaptation occurs. The `Intercomm` communicator contains the group of leaving processes. The `adapt_reduce` function is also an optional function like the other two member variables.

Like in the init phase and rigid phase, there is a helper method provided to declare the elastic phase. The syntax of the `setElastic()` helper function is as follows:

```
void setElastic(std::function phase_function, std::function loop_condition,
                    std::function adapt_prepare, std::function adapt_expand,
                                                 std::function adapt_reduce);
```

The `adapt_prepare`, `adapt_expand` and `adapt_reduce` functions are optional and can

be excluded while calling the helper method. The default value for the optional member variables are set as `nullptr`. A sample usage is shown in Listing 6.3:

```
auto elastic_phase = [](void** data) -> void {
// code in elastic block
}
auto loop_condition = [](void** data) -> bool {
// decision making logic
// return true or false
}
auto adapt_prepare = [](void** data) -> void {
// data distribution or preparation before adapt
}
auto adapt_expand = [](void** data,int local_status, int staying_count,
                                int joining_count, MPI_Comm Intercomm,
                                           MPI_Comm newcomm) -> void {
// data distribution if expansion
}
auto adapt_reduce = [](void** data, int local_status, int staying_count,
                                int leaving_count, MPI_Comm Intercomm,
                                           MPI_Comm newcomm) -> void {
// data distribution if reduction
}
// declaring Elastic Phase if all methods are present
setElastic(elastic_phase, loop_condition, adapt_prepare, adapt_expand,
                                                        adapt_reduce);
//declaring Elastic Phase if all methods except adapt_prepare are present
setElastic(elastic_phase, loop_condition, nullptr, adapt_expand,
                                                        adapt_reduce);
//declaring Elastic Phase if all the optional methods are not present
setElastic(elastic_phase, loop_condition);
```

Listing 6.3: Declaration of an Elastic Phase

6.2.4 Branch Phase

This class has only one member variable of type `std::function`. This function variable determines whether to jump to a branch or to continue the normal iteration.

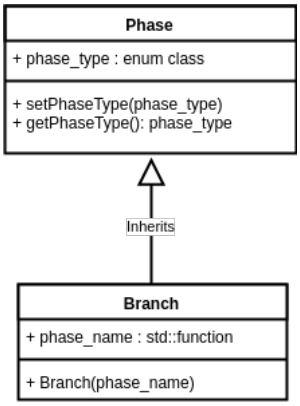

Figure 6.5: UML class diagram of the Branch Phase

The signature of the branch phase function variable is different from the other phases. The syntax of the branch phase is as follows:

```
int branch_phase(int pc, void** data);
```

The first parameter of the branch phase function is the program counter, which provides the current phase index. For example, if the order of the declaration of the phases are init phase, rigid phase, elastic phase, branch phase and rigid phase, then the current phase index will be 3 i.e. the index of the branch phase in the order of the declaration. The branching function can alter this index after making the branching decision and can return the new program counter. Upon receiving the index, the driver will set it as the new program counter and will continue the execution of the program from that index onwards. `setBranch()` is the helper function to declare a branch phase. The syntax of the helper function is as follows:

```
void setBranch(std::function);
```

A sample source code for declaring the branch phase is shown in Listing 6.4.

```
auto branch_phase = [](int pc, void** data) -> int {
//branching decision and return new pc to driver
}
setBranch(branch_phase);
```

Listing 6.4: Declaration of a Branch Phase

6.3 Driver Program

The EPOP programs are compiled as shared libraries and loaded by the EPOP driver at runtime. The EPOP driver deals with the execution and profiling of the application and also the communication with the Invasive Resource Manager. The location of the shared library should be provided while launching the driver program. The EPOP driver can be launched by the `mpiexec` launcher or `SRUN` launcher. The syntax for launching the EPOP application is as follows:

```
mpiexec -n 2 epop --enable-profiling=true sharedlibrary.so parameters
srun -n 2 epop --enable-profiling=true sharedlibrary.so parameters
```

`"epop"` is the binary for the EPOP driver. `"--enable-profiling"` parameter can be used to turn on the profiling done by the driver.

The developer should create the EPOP application in a custom function instead of the `main` function. The signature of function should be:

```
extern "C" phase_vector epop_program();
```

`extern "C"` should be written to avoid the name mangling[4]. C++ implements the name mangling to facilitate the function overloading and also for variable scoping in different namespaces. The driver loads the application as a library and the name mangling can result in symbol lookup errors. The signature and name of the function should not be changed. The driver will call the `epop_program()` function in the shared

library to load an EPOP application. The application should return the `phase_vector` as the return type which will contain the vector of the declared phases. The user can call the `createPhaseVector()` helper method in the return statement to return `phase_vector`. The declaration of the phase vector is as follows:

```
std::vector<unique_ptr<Phase>> phase_vector;
```

`phase_vector` contains the unique pointers of the objects of the superclass Phase. Unique pointer allows the explicit ownership transfer and once the phases are declared, the driver should get the ownership of the objects. Phase vector holds the objects of all the phases and the EPOP driver uses this phase vector for execution of the program.

6.3.1 Working of the Driver

A pseudocode for the driver program is in Listing 6.5. The iteration through the phases is shown. Profiling and communication if included can make the psuedocode complex and hence it is not included. Figure 6.6 shows the flow chart of the driver program.

```
Load the EPOP program and get the phase_vector
MPI_Init_adapt(&argc,&argv,&local_status); // Initializing as iMPI job
cast the first element in phase_vector into Init class object in
in->phase_name(&argc,...); //calling init phase
if(local_status = MPI_ADAPT_STATUS_JOINING){
    MPI_Comm_adapt_begin(); // Start adaptation
    Receive pc from epop master
    cast the phase_vector.begin()+pc object into Elastic class object e
    e->adapt_expand(&data,...);
}
else{
    pc = 1
}
for (auto i = (phase_vector.begin()+pc) ; i != phase_vector.end(); ++i){
    switch(i->getPhaseType()){
        case phase_type::RIGID : {
            cast Phase class object i into Rigid class object r
```

```
            do{
                r->phase_name(&data); // calling rigid phase
            }while(r->loop_condition);
            pc++;
            break;
        }
        case phase_type::ELASTIC : {
            cast Phase class object i into Elastic class object e
            do{
                MPI_Probe_adapt(&adapt,...); //Probing for adaptation
                if(adapt == MPI_ADAPT_TRUE){
                    e->adapt_prepare(&data,...);
                    MPI_Comm_adapt_begin(); // Start adaptation
                    if (epop_master and adaptation_type = expansion) {
                        Send pc to the newly joining process
                    }
                    if (adaptation_type = expansion)
                        e->adapt_expand(&data,...);
                    if (adaptation_type = reduction)
                        e->adapt_reduce(&data,...);
                    MPI_Comm_adapt_commit(); // Finishes the adaptation
                }
                e->phase_name(&data);   // calling elastic phase
            }while(e->loop_condition);
            pc++;
            break;
        }
        case phase_type::BRANCH : {
            cast Phase class object i into Branch class object r
            pc = b->phase_name(pc,&data); // calling branch phase
            break;
        }
}
```

Listing 6.5: Psuedocode of the driver program

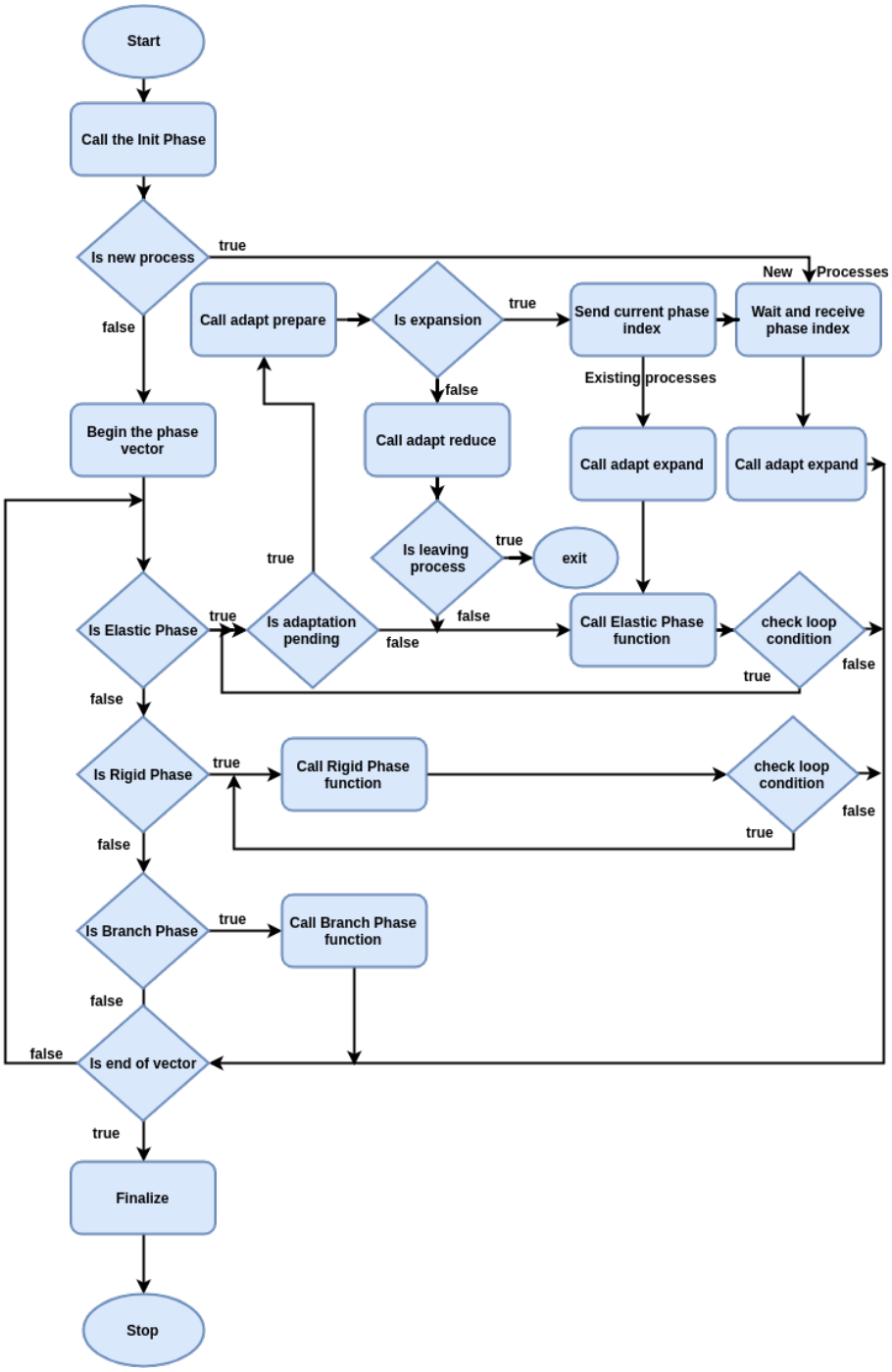

Figure 6.6: Flow chart of the Driver program

As soon as the EPOP application is launched, the driver program will load the shared library and call the `epop_program()` function which returns the `phase_vector`. Then the driver program will call the `MPI_Init_adapt` to inform the resource manager that the application is an invasive application. The driver program will start iterating through the `phase_vector`.

Since the first phase of every EPOP program will be an Init phase, the driver program will cast the Phase object into the Init object and calls the member function `phase_name` associated with the Init phase. The `local_status` from the `MPI_Init_adapt` is used to determine whether the current process is created as part of the adaptation or as part of the job creation.

1. If the process is created as part of an adaptation i.e when the `local_status` set by `MPI_Init_adapt` routine is `MPI_ADAPT_STATUS_JOINING`, then the driver immediately calls the `MPI_Comm_adapt_begin()` for starting the adaptation. Now, the newly joined EPOP processes will wait for all the pre-existing EPOP processes to call this routine. After all the processes call this routine, the newly joined EPOP processes will wait for the program counter (current phase index) from the pre-existing processes. The EPOP master of the pre-existing processes will send their phase index. After receiving the phase index, the EPOP driver casts the element of the `phase_vector` at the received phase index into the Elastic phase. Then the driver calls the `adapt_expand` method of the Elastic phase class if available. The `adapt_expand` method can be used by the application for data distribution and other application-related tasks. Then the EPOP driver calls the `MPI_Comm_adapt_commit()` operation to finish the adaptation. The newly joined EPOP processes will start iterating the `phase_vector` from the received phase index, call the Elastic phase function in that index and join the pre-existing processes in the computation.

2. If the process is created as part of an initial application launch, the driver program will continue iterating through the `phase_vector`. The driver program checks the type of each vector element using the member function `getPhaseType()` of the superclass Phase (see the Design section UML diagram).

    - If the vector element is a Rigid phase, then the driver will cast the Phase object into the Rigid phase object and calls the rigid phase function. After calling the phase function, the driver will call the loop condition method

associated with the rigid phase if present. The driver will then execute the rigid phase function until the loop condition returns false. If there is no loop condition or the loop condition returns false, then the driver will continue to the next element of the `phase_vector`.

- If the vector element is an Elastic phase, then the driver will cast the Phase object into an Elastic phase object. The driver then calls the `MPI_Probe_adapt` function to check for any pending adaptations. This method returns the adaptation status. If the adaptation status is `MPI_ADAPT_TRUE` then the driver will call the three methods associated with the adaptation in the Elastic phase class. The driver will call the `adapt_prepare` method first if it is defined in the application. Then the driver of the pre-existing processes will call the `MPI_Comm_adapt_begin()` routine. Now all the processes have entered the adaptation window. If the adaptation type is expansion then the EPOP master will send the current phase index to the newly joined processes. The EPOP drivers of the newly joined processes will be waiting to receive the phase index. Then the expansion related operation `adapt_expand` will be called by the EPOP drivers of all the pre-existing processes and the newly joined processes. If the adaptation type is a reduction, then the EPOP driver will call the `adapt_reduce` method if it is defined in the application. After calling this routine, the driver of the pre-existing processes calls the `MPI_Comm_adapt_commit()` to finish the adaptation. Calling this IMPI routine will kill all the leaving processes if the adaptation type is a reduction or merges the joining processes with the pre-existing processes if the adaptation type is an expansion. Then the drivers of the pre-existing processes call the Elastic phase function. After that, the driver calls the loop condition function of the Elastic phase and continue executing the Elastic phase until the loop condition returns false.

- If the vector element is a Branch phase, then the driver will cast the Phase object into the Branch object and calls the function associated with the object. The return value of this function is taken as the new phase index and the next iteration will start from that index onwards.

6.3.2 Data flow

The EPOP driver passes the data between the phases as a void pointer because all application contains different user specific data. The driver program passes this data to all the methods. Inorder to access the data between the phases, the application developer needs to declare the variables inside a `struct`. Then the application developer can reference the structure to the void pointer passed by the driver to the Init phase function. The driver then passes the same pointer to the next Phases and the developer can cast it into his/her own `struct` in each phase. The listing 6.6 below shows the sample usage.

```
typedef struct {
int rank;
int size;
} sample_data; // Declaring a structure to store the data
auto init_phase = [](int argc, char** argv, void** data, int rank,
                            int size, int local_status) -> void {
//creating the structure variable
sample_data *local_data = (sample_data*)malloc(sizeof(sample_data));
local_data->rank = rank;
local_data->size = size;
// code in initialization block
*data = local_data; // Passing the structure to the driver
}

auto elastic_phase = [](void** data) -> void {
// Casting the void pointer to the structure
sample_data *local_data = (sample_data*)(*data);
cout<<local_data->rank;
// code in elastic block
}
}
```

Listing 6.6: Data flow between the Phases

6.3.3 A Simple EPOP Program Structure

Listing 6.7 shows the source code which will provide insight on how to write a simple EPOP application.

```
typedef struct {
int rank; int size;
}sample_data;

extern "C" phase_vector epop_program(){
auto init_phase = [](int argc, char** argv, void** data, int rank,
                            int size, int local_status) -> void {
sample_data *local_data = (sample_data*)malloc(sizeof(sample_data));
local_data->rank = rank;
local_data->size = size;

// code in initialization block
*data = local_data;
}
auto elastic_phase = [](void** data) -> void {
sample_data *local_data = (sample_data*)(*data);
// code in elastic block
}
auto loop_condition = [](void** data) -> bool {
// decision making logic
// return true or false
}
auto adapt_prepare = [](void** data) -> void {
// code for adaptation preparation
}
auto adapt_expand = [](void** data, int local_status, int staying_count,
                                  int joining_count, MPI_Comm Intercomm,
                                            MPI_Comm newcomm) -> void {
// code for adaptation expansion
}
auto adapt_reduce = [](void** data, int local_status, int staying_count,
```

```
                                int leaving_count, MPI_Comm Intercomm,
                                          MPI_Comm newcomm) -> void {
// code for adaptation reduction
}
auto rigid_phase = [](void** data) -> void {
sample_data *local_data = (sample_data*)(*data);
// code in rigid block
}
auto rigid_loop_condition = [](void** data) -> bool {
// decision making logic
// return true or false
}
setInit(init_phase);
setElastic(elastic_phase, loop_condition, adapt_prepare, adapt_expand,
                                                        adapt_reduce);
setRigid(rigid_phase,rigid_loop_condition);
return  createPhaseVector();
}
```

Listing 6.7: A simple EPOP application

6.4 Representation of the Performance information

The performance information is stored as a structure element `epop_perf` shown in Listing 6.8. This structure stores the information such as the job id, the number of phases and also a pointer to the structure `phase_perf` which stores the phase information and performance metrics of each phase. The structure `phase_perf` contains the phase type, information about the loop, the time taken for each phase, the average time per phase, the global average time for that phase, the MPI time, the number of times the phase is called, the power information etc.. Lots of information are stored regarding each phase and can be seen in the listing. After getting the request from the resource manager, the required contents of this structure is packed and sent back. In some scenarios, the driver pushes the performance data to the resource manager even without the request from the resource manager.

```
typedef struct {
   double job_time;
    int job_id;
    int number_of_phases;
    phase_perf *info;
}epop_perf;

typedef struct{
    char phase_type;
    int is_loop;
    double call_count;
    double phase_time;
    double total_phase_time;
    double average_phase_time;
    double average_mpi_time;
    double average_mtct;
    double mtct;
    double mpi_time;
    double total_mpi_time;
    double average_phase_power;
    double phase_power;
    double global_average_phase_time;
    double global_average_mpi_time;
    double global_average_mtct;
}phase_perf;
```

Listing 6.8: Structure that represents the Phase information in EPOP

6.5 Message types and Communication

The message types of the EPOP model are declared as `enum` and is shown in Listing 6.9. They are sent to and received from the Invasive Resource Manager.

```
typedef enum
{
        STRUCTURE = 22000,TIME,POWER,MTCT,PAUSE
}message_type;
```

Listing 6.9: Message type in EPOP

STRUCTURE is the enum type used to denote the structure of an EPOP program. The enum message_type is started with the value 22000 to differentiate it from the enum types of the Invasive Resource Manager. The EPOP master will send the structure of the program to the resource manager at the beginning of the application. STRUCTURE message contains the general information such as the job id, the total number of phases, the IP address of the EPOP master and the phase details such as the type of phases, whether a loop is present in the phase. After getting a request from the resource manager, the driver checks for the type of the request in the message type. If the MTCT (MPI time to compute time ratio) is requested by the Invasive Resource Manager, then the driver will call the corresponding packing function to pack the MTCT ratio of the phases along with the current phase index. All of the performance data are stored inside the structure epop_perf. According to the type of the message, the required information is packed and sent to the resource manager. The syntax of the packing function is as follows:

```
pack_metrics(epop_perf phaseinfo,int message_type,int current_phase_index);
```

The message from the EPOP master to the Invasive Resource Manager contains a fixed message header and a corresponding message body according to the message type (see Listing 6.10). The header contains the size of that message, the current job id, the type of the message, the total number of phases in the application, the current phase index, the length of the body of the message, the IP address of the EPOP master and the size of the IP address. The message type can be STRUCTURE, MTCT, TIME or POWER. The body of the message is varied according to the type of the message. If the message type is STRUCTURE then the body of the message contains the different phase information like the phase type and whether a loop is present in the phase. If the message type is TIME then the body contains the time value for all the executed phases and current phase in the application. The messages sent by the EPOP master are converted into the

network byte order to deal with the endianness.

```
struct header {
    uint32_t size;
    uint32_t job_id;
    uint32_t type;
    uint32_t number_of_phases;
    uint32_t current_phase_index;
    uint32_t number_of_elements;
    uint32_t ip_address_size;
    char *ip_address;
};

struct body {
    uint32_t *values;
    uint64_t *doublevalues;
};
```

Listing 6.10: Representation of the header and body of the performance data response message

The EPOP master gets the IP address of the Invasive Resource Manager from the `SLURM` configuration file. After getting the IP address, the driver registers itself with the Invasive Resource Manager by sending the `STRUCTURE` of the EPOP application. The corresponding unpack function is implemented in the Invasive Resource Manager to handle the incoming message from the EPOP master. The Invasive Resource Manager registers the EPOP job after receiving the `STRUCTURE` message and stores the received IP address. The received job id from the EPOP master is compared with the list of job ids and sets the `is_epop` variable of the job pointer to true. It is done to request the performance data and other metrics from the EPOP driver whenever a job is an EPOP job.

6.6 Infrastructure changes

6.6.1 Extensions to Invasive MPI

IMPI is extended in this work to accommodate the profiling of the application and the performance metrics gathering. The present IRM uses the metrics MPI time to compute time ratio for making the scheduling decisions. In the EPOP programming model, the EPOP master sends the structure of the application apriori and the performance metrics whenever the resource manager requests it. To obtain the MPI time, every MPI call should be tracked and the time spent in the MPI operation should be found out. For implementing this functionality, the IMPI should be extended.

The following changes have been made in IMPI. An EPOP module is added to IMPI for the handling of the performance metrics data. `MPID_start_epop_mpi_time` routine and `MPID_update_epop_mpi_time` routine are the two functions implemented inside the IMPI for computing MPI time. Before each MPI call, `MPID_start_epop_mpi_time` function is called to mark the start time. After finishing the MPI call, the function `MPID_update_epop_mpi_time` is called to mark the finish time of the application, to find the difference between the start time and the finish time and add the difference to the MPI time counter. For profiling purposes, the `MPID_start_epop_mpi_time` also records the MPI call name and the number of times a particular MPI operation is invoked. A specific time counter and a call counter per MPI operation is maintained for that. Whenever the MPI call is finished, the time counter and the call counter associated with the particular MPI call is updated. The Invasive Resource Manager provides the job id to the IMPI for the adaptation purposes. This job id is copied into the EPOP module by the `MPID_set_epop_job_id(job_id)` routine which is called inside the `MPI_Init_adapt` routine.

The four new MPI operations are added to IMPI and can be directly accessed by the EPOP driver. These operations are used for obtaining the profiling information. They are added as `MPI_T` operations which is an interface for the tools introduced in MPI 3.0 [15].

1. `MPI_T_set_epop_mpi_time()`

   This function is implemented to set the MPI time counter to 0. The MPI time

is calculated for each iteration of the phase. After the end of an iteration, the counter should be set to 0 again. This routine is called at the beginning of each phase iteration by the EPOP driver.

2. `MPI_T_get_epop_mpi_time();`

   This function returns the total time spent for the MPI operations. The driver calls this routine at the end of each iteration of the phase and stores the return value.

3. `MPI_T_get_epop_job_id()`

   This operation is added to obtain the job id from the EPOP module inside the IMPI. The EPOP master can use this job id to register itself as an EPOP job inside the Invasive Resource Manager.

4. `MPI_T_set_epop_profiling();`

   This MPI routine is added to enable profiling. The EPOP driver can enable the profiling by giving the parameter as 1 and disable by setting it as 0. If enabled, this routine will track all the MPI calls made by each EPOP rank. The information such as the number of MPI calls made per MPI operation, the total time spent in each operation and the name of the MPI operation are recorded. At the end of the program, all these information will be provided to the user. User can use this information to fine tune the application. A sample profiling output will be of the form:

   Operation: MPI_Send Total calls: 23 Total time taken: 10secs Rank: 1 Phase: Elastic Phase Percentage of MPI_Send Time/Total MPI Time: 20%

6.6.2 Extensions to Invasive Resource Manager

Integration with the Invasive Resource Manager is essential for the EPOP model. The EPOP master communicates the performance data to the resource manager which in turn uses it for making the adaptation decisions. The Invasive Resource Manager is extended to receive the data from the EPOP master and asks the EPOP master for performance data whenever a job is EPOP. At present, the Invasive Resource Manager

gets the performance data from IMPI. For an EPOP job, to make a better scheduling decision, the EPOP master can provide more accurate data since the EPOP driver has a global view of the application than the automatic profiling done in the IMPI by IRM. The EPOP driver knows the starting of a heavy computation phase and can provide a more accurate MTCT ratio.

The resource manager needs to request the performance data from the EPOP driver for the EPOP jobs. For a job to be annotated as EPOP, the EPOP driver needs to register a job as an EPOP job in the job pointer of the Invasive Resource Manager. As mentioned in the previous section, the EPOP master will send the master's IP address, the EPOP program structure and the job id, whenever an EPOP job is executed. A communication module has been added to the resource manager to receive these messages from the EPOP driver and to unpack it. If the message type is `STRUCTURE`, then the EPOP module in the resource manager sets the job id, the IP address of the EPOP master and allocates memory for the performance data to be stored.

The invasive scheduler iterates through the jobs and analyse their performance data for making adaptation decisions. In order to take the performance data from the driver in case of an EPOP job, the scheduling algorithm is modified. In the job pointer, a new flag `is_epop` is added. `is_epop` is set by the resource manager for the job id registered by the EPOP master. If a job is EPOP, the resource manager changes its normal flow. In Normal MPI jobs it requests IMPI for the performance data but for the EPOP jobs, the resource manager requests the EPOP master. A new message type is needed for requesting the performance data from the EPOP master.

A new message type `epop_driver_msg_t` is added to the Invasive Resource Manager to send the performance data request to the EPOP master (see Listing 6.11).

```
typedef struct epop_driver_msg {
        uint32_t message_type;
        uint32_t message_value;
} epop_driver_msg_t;
```

Listing 6.11: Structure of the performance data request message

`message_type` is the type of the message which IRM requests (explained in Section 6.5). `message_value` is necessary only if the `message_type` is `PAUSE`, in that case the message value will be the time duration in seconds for pausing the application. A message

handler `epop_driver_message` is implemented in the resource manager to send the performance data request with the `message_type` and `message_value` . The syntax of the message handler is as follows:

```
epop_driver_message(message_type,message_value,address_of_epop_master)
```

`get_epop_ip_address(int job_id)` is implemented in IRM to obtain the IP address of the EPOP master, and a message is sent to request the required performance metrics. Sample usage is:

```
epop_driver_message(PAUSE,60,get_epop_ip_address(job_id));
```

The above message will ask the EPOP master to pause the execution of all the EPOP processes for 60 seconds.

The invasive scheduler while iterating through the job pointers will pass the job id to the EPOP module to check whether the current job is EPOP. If the current job is EPOP, then the resource manager will call the handler function `is_perf_data_available` to check whether there is any performance data received from the driver. If it returns false, then the resource manager calls the `epop_driver_message` with `message_type` as `MTCT`. After getting the response from the driver, the EPOP module sets the content of the `perf_data_available` flag as true. In the next scheduler pass, after taking the current performance data, the scheduler resets the `perf_data_available` flag using the handler method `reset_is_perf_data_available()` . The previous performance data is stored in the EPOP module for calculating the `MTCT Trend` which is needed by the invasive scheduler. Since there can be multiple elastic phases, the scheduler should request the performance data of the current phase only. `current_phase_index` is passed by the EPOP driver in every message as part of the message header. The Invasive Resource Manager can use this information for the progress evaluation also. If the program is in the last rigid phase, then the resource manager can conclude that the application will be finished soon and can use this information for better scheduling of the remaining jobs in the job queue.

Modification of Scheduler Algorithm

As part of this work, the scheduler is modified to accommodate the newly added EPOP modules. The modifications made inside the invasive resource scheduler is shown in the listings 6.12 and 6.13. Listing 6.12 shows the modification made in the performance data request part of the scheduler.

```
//marking the EPOP job in the job pointer
job_ptr->is_epop = get_epop_job_id(job_ptr->job_id);

//checking for the performance data for jobs
if(!job_ptr->performance_data_available){

    // checking whether a job is EPOP or not
    if(job_ptr->is_epop){

        // checking whether performance data is available or not
        if (!is_perf_data_available){

            //Requesting the MPI time to Compute time data from the master
            epop_driver_message(MTCT,1,get_epop_ip_address());
        }
        else{

            //Setting the flag that performance data is available
            job_ptr->performance_data_available = 1;
        }

    }
    else{
        //for normal impi jobs
    }
}
```

Listing 6.12: Performance data request from the scheduler

Decision making based on the received data

Listing 6.13 shows the modifications done in the decision making part of the scheduler code.

```
if(job_ptr->is_epop){
  int is_perf_data_available = get_is_perf_data_available();
    if (!is_perf_data_available){
        // Do not change the resource allocation
        continue;
    }
  int phase_index = get_current_phase_index();
  double current_mtct_epop = get_current_mtct_time(phase_index);
  double trend_mtct_epop = get_previous_mtct_time(phase_index);
  reset_is_perf_data_available();
  if(current_mtct_epop >= MTCT_upper_threshold
              || trend_mtct_epop >= MTCT_upper_threshold){
      // reduce the resources
  }
  else if(current_mtct_epop < MTCT_lower_threshold
      || trend_mtct_epop < MTCT_lower_threshold){
      // expand the resources
  }
}
else{
   // for jobs other than EPOP
}
```

Listing 6.13: Scheduler decisions based on the performance data

`MTCT_upper_threshold` and `MTCT_lower_threshold` along with the `MTCT_Trend` is used to make an adaptation decision [3]. The general idea behind this is, if an application takes a significant amount of time for MPI operations compared to the actual computation time then it means that the application is spending a lot of time for communication between the processes. In such a scenario i.e. when the `MTCT` is high, the resource expansion won't make any change since the expansion will only increases

the communication time. So the resources can be shrunk to reduce the communication time. If the communication time is less and computation time is high, then the resource expansion should be done .ie. when the `MTCT` will be low. `MTCT_upper_threshold` and `MTCT_lower_threshold` can be fine-tuned according to the application. The `MTCT` from the EPOP is used in the scheduler if the job is an EPOP job. `MTCT_Trend` is the average of all the previously obtained `MTCT` values.

7 Evaluation

7.1 Evaluation Setup

The evaluation is performed on the supercomputer "SuperMUC" [13] managed by the Leibniz Supercomputing Centre. The results are taken in the Phase 1 thin nodes based on the Sandy Bridge EP [10] processer. The interconnection between nodes is done with the Infiniband FDR10 standard. The jobs are submitted in the general queue. The number of nodes used in this evaluation is 33 (minimum node count for a job in general queue) and each node contains two processor with 8 cores each.

The Invasive Resource Manager and the Invasive MPI are compiled in the login node of the SuperMUC. Whenever a job is submitted, the list of nodes allocated for the job is provided by the loadleveler in the SuperMUC. To set up the Invasive Resource manager in the SuperMUC, special scripts [3] were used to create a virtual cluster with the allocated nodes. In the virtual cluster, the Invasive Resource Manager is set as the resource manager and IMPI as the MPI version. The launcher SRUN is used to submit jobs to the Invasive Resource Manager.

For the evaluation, the applications were run on the virtual cluster created with 33 nodes (528 processes). The EPOP driver is launched by the SRUN launcher in the virtual cluster. The driver then opens the shared library provided in the launch command and starts execution of the application.

7.2 Application

An invasive application is needed to demonstrate the working of EPOP model. Two applications were made Invasive to test the programming model. The first one is a simple application to calculate the value of Pi with around 100 lines of code. This application is used to compare the previous EPOP model with the new model and also to benchmark the loading time and phase declaration time. The second application is the heat simulation using a 2D Jacobi iterative solver. It is used to evaluate the adaptation of EPOP model to the dynamic decisions made by the resource manager and the communication between the Invasive Resource Manager and EPOP. This application has around 900 lines of code. Both applications were MPI and converted to EPOP model.

Heat simulation using a 2D Jacobi solver is a complex application. At the beginning of the application, the master process initializes the grid. According to the number of processes, the dimension to divide the grid is determined. Using the MPI routines, a virtual topology is formed with the dimension and the neighbours of a process are found out. The master process then divides the grid according to the dimension and then the divided data is passed to the corresponding processes. Each process gets a part of the data in the grid. The master process also takes its own part of the grid. After the data distribution, all the processes start solving their part of the grid using the Jacobi kernel. Jacobi kernel is called iteratively. After each iteration, the elements at the border should be sent to the neighbouring processes to synchronize the data. After calculating the heat propagation in a grid with the provided initial resolution, the master again starts initializing the grid of a higher resolution according to the input file. In this evaluation, the application is run with the resolutions 1200X1200 till 25200x2500 and executed in a loop to see the adaptation behaviour.

This application was converted into an invasive application using the following four phases:

1. Init Phase - This phase contains the initialization of the application like reading the configuration file, deciding the number of iterations etc.

2. Rigid Phase - This phase is used to calculate the dimensions to create a virtual topology.

3. Elastic Phase - This phase contains :
   - An Elastic phase code block that contains the Jacobi kernel that is called iteratively for all the resolutions.
   - A loop condition to increment the resolutions and to determine when to exit from the Elastic phase.
   - The data to be sent to the newly joining process in case of an adaptation.
4. Branch Phase - This phase is used to jump into elastic phase to recompute the values again.

7.3 Programming Model

For all the following evaluations, the application was started with 32 number of nodes and the job is run for one hour in the SuperMUC. To evaluate the working of the programming model, three scenarios need to be considered.

1. Evaluate the resource adaptation. Executing the EPOP application with a predetermined resource expansion and reduction schedule from the resource manager can demonstrate the adaptive behaviour of the EPOP model. This scenario is evaluated in Subsection 7.3.1
2. Evaluate the resource adaptation based on the performance data from EPOP. This can be done in two ways:
   - Method 1: The first method is to provide the performance data to IRM upon request only. This scenario is evaluated in subsection 7.3.2
   - Method 2: The second method is to push the performance data proactively to IRM whenever there is a change in performance metrics. This scenario is evaluated in Subsection 7.3.3

   This also verifies that IRM receives the performance data from EPOP and can make a decision based on the received data.

7.3.1 Adaptation of a 2D Jacobi - Fixed schedule

To show the working of the programming model and driver program, the scheduling strategy in the Invasive Resource Manager was changed to a fixed order. The order for the fixed scheduling was 1, 2, 4, 8, 4, 16, 32, 32 and 1 nodes. At first, the application is launched with 32 nodes and the resource manager performs the scheduling based on the above schedule. This schedule covers all the scenarios like reducing from maximum nodes to minimum nodes, doubling the resources, no adaptation, reduction by half and expansion to the maximum. From the successful execution of this schedule by the EPOP driver, we can conclude that the driver can adapt to any resource changes proposed by the resource manager.

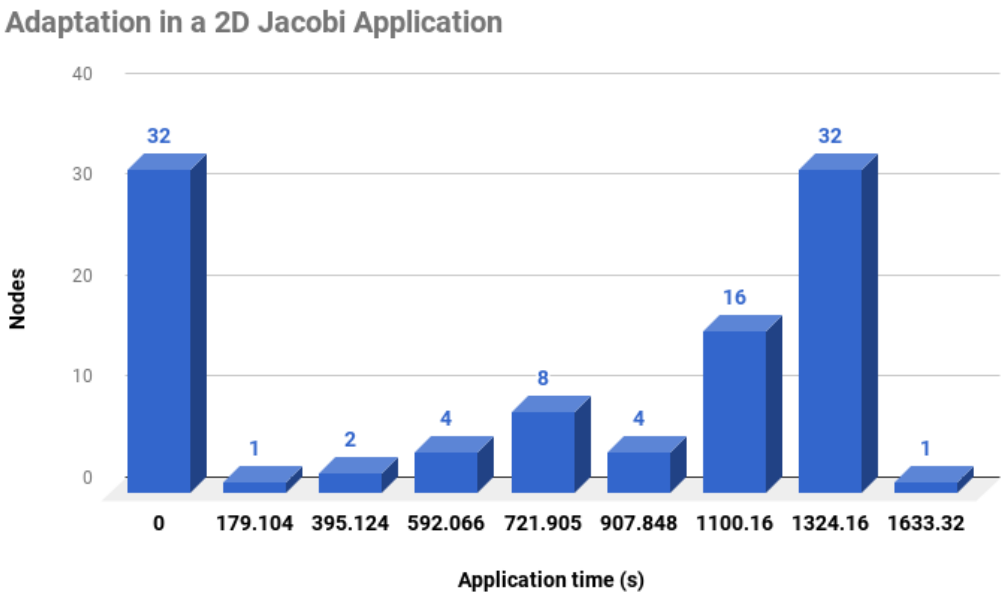

Figure 7.1: Adaptation of a 2D Jacobi with a fixed expansion and reduction schedule

Figure 7.1 shows the adaptation of the application performed as a result of the fixed schedule. At time point 0 in the graph, we can observe that the application has 32 nodes. At 180 seconds, we can observe the first adaptation in which the number of

nodes was reduced to 1. The time interval between the two scheduler pass is set as 180 seconds in the SLURM configuration file. Every 180 seconds the scheduler makes the adaptation decision according to the fixed schedule. In the next three scheduler passes, it makes the adaptation decision of doubling the resources according to the fixed schedule (see time points 395.124, 592.006 and 721.905 s) and increase the total node count to 8.

In the fifth adaptation (time point 907.848), the scheduler halves the resources to 4. At time point 1100.16 in Figure 7.1 the scheduler decides to quadruple the resources to 16 and then doubles it to 32 in the following scheduler pass. In the fixed schedule, 7th and 8th adaptation has the same number of nodes, so the scheduler does not make any resources changes and hence no adaptation. As a result, we can observe in the graph a time difference of around 300 seconds between the 7th and 8th adaptation. At 1633 seconds, the scheduler reduces the resources to 1 node again according to the fixed schedule.

Since our application is a normal MPI application using the EPOP model and the EPOP driver is making all the necessary IMPI calls to perform the adaptation, by observing Figure 7.1, we can conclude that the EPOP programming model is making an application invasive with the help of the Invasive Resource Manager.

7.3.2 Adaptation of a 2D Jacobi - Profiling method 1

In this method, the profiling information from EPOP is used by the Invasive Resource Manager to perform the scheduling. The EPOP driver sends the performance metrics only upon request from IRM.

Figure 7.2 shows the invasive behavior of the application while using the first profiling method. The x-axis of the graph represents the application time and y-axis of the graph represents the number of nodes allocated to the application. The scheduler makes the adaptation decision with the performance data provided from the EPOP model.

The performance data trend of the application in the same period is plotted in Figure 7.3. The x-axis of this graph provides the application time and the y-axis represents the time in seconds. Analysis of both graphs will provide a clear picture of

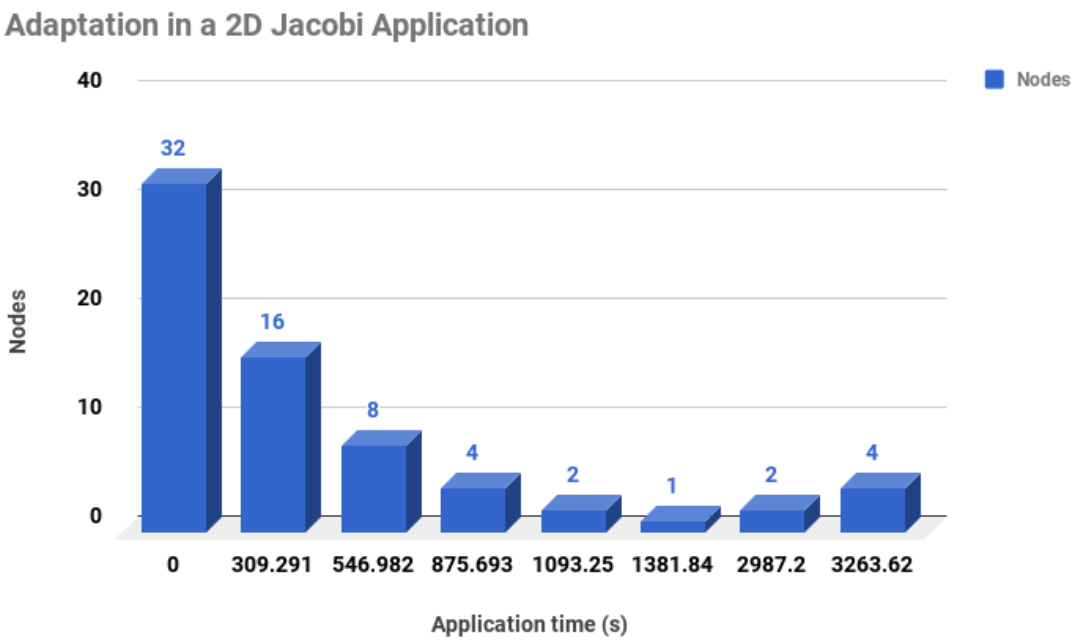

Figure 7.2: Adaptation of a 2D Jacobi using Profiling Method 1

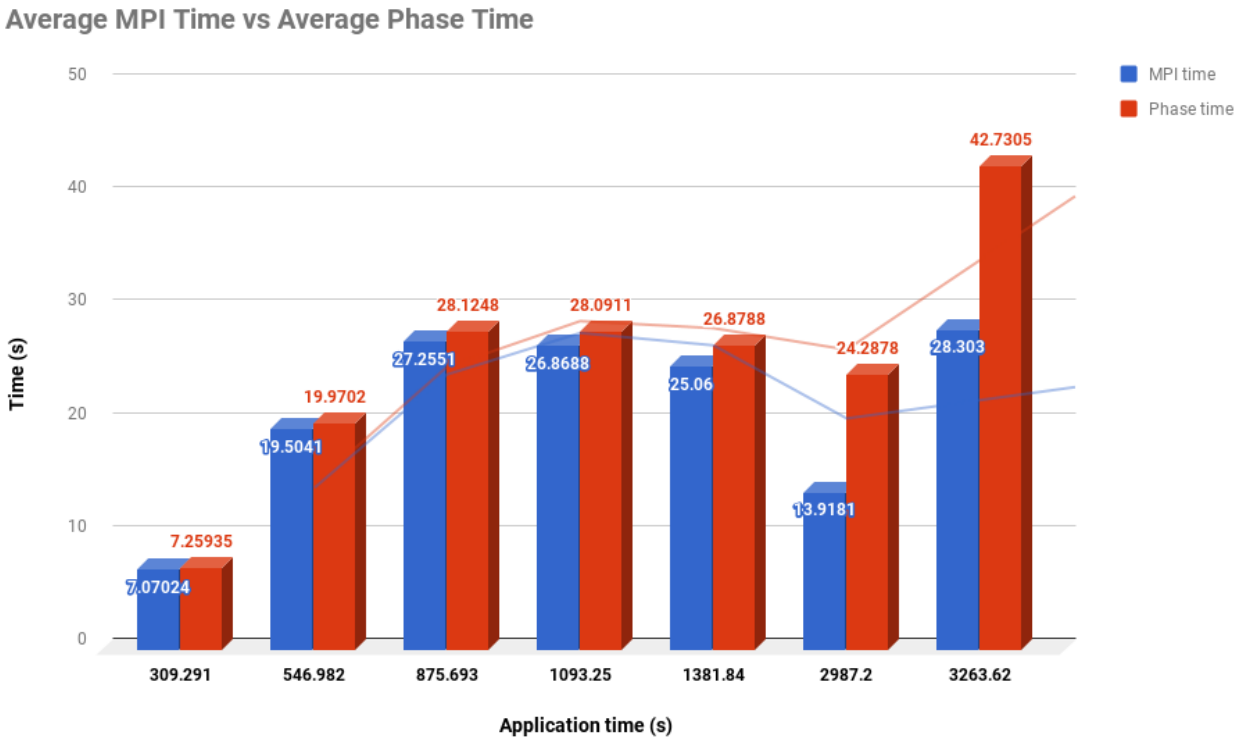

Figure 7.3: MPI time vs Phase time in adaptation of a 2D Jacobi using Profiling Method 1

how the performance data is used to make scheduling decisions. The application was launched with 32 nodes. We can observe this from Figure 7.2.

At time point 0, the application has 32 nodes. After some point of time, the scheduler makes the decision to shrink the number of resources from 32 nodes to 16 nodes by using the performance data obtained from the EPOP model. The EPOP driver sends the ratio of the average MPI time to the average phase time i.e. `MTCT` ratio to perform the scheduling decision. `MTCT` ratio is configured with .9 as (`MTCT_upper_threshold` and .6 as (`MTCT_lower_threshold`) in IRM. The application's resources were shrunk to half in 309 seconds after the application launch. Looking at this time point in the performance metrics graph (second time point in the Figure 7.3), where the first reduction occurs, we can see that the average MPI time (7.0724 s) and the average phase time (7.25935 s) of the application is almost the same which implies that the communication time is almost same as the computation time of the application. The `MTCT` ratio (MPI Time/Phase Time) is 0.9765 in this point. According to the scheduling logic (see 6.6.2), if `MTCT` ratio is greater than `MTCT_upper_threshold`, the scheduler decides to reduce the number of nodes to efficiently utilise the resources.

From 310th second onwards, the application continues the execution with 16 nodes. At 546th second in the plot, again a reduction is performed (see the Figure 7.2). If we look at the same time point in the Figure 7.3, we can see that the average MPI time and the average phase time is almost the same. As a result, the scheduler has to reduce the number of resources by half. The resource manager always reduces and expands the resources by a factor of two in the present IRM implementation. As we can observe from both graphs, as the scheduler reduced the resources, the difference between the MPI time and the phase time increases i.e communication time among processes are reducing and the computation time is increasing.

At 1381 seconds, the scheduler reduced the number of nodes to a single node. This reduction of the application to a single node has increased the average MPI time to phase time difference significantly. We can observe it in the 2987 second on Figure 7.3. The MPI time is 13.981 seconds and the Phase time 24.2878 seconds and the resulting `MTCT` ratio is 0.573048 which is less than the `MTCT_lower_threshold` in IRM. As a result, the scheduler made the decision to expand the number of resources in the next scheduler pass. We can observe the expansion of resources in the 2987th second on Figure 7.2. Even after the expansion, the time difference between the MPI time and

phase time remains high which means that more time is spent on the computation than the communication and there is a scope of performance improvement with the expansion of the resources. Therefore the scheduler again doubles the resources (see Figure 7.2) at 3263 seconds.

The trendline in the graph of the MPI time versus phase time of the application (Figure 7.3) provides a clear picture of how the scheduling decision is made. As the resources are reduced, the MPI time decreases, that is less time is spent on the communication between the processes. So the MPI time to phase time ratio decreases. The scheduler decides to reduce the resources until the ratio is below a certain threshold (`MTCT_upper_threshold`). This can be observed in the trendlines between 0 to 2987 seconds. As the distance between both trendline increases, the resources are reduced in each adaptation. At 2987 seconds, when the distance between both trendlines is large enough i.e. when the MPI time to phase time ratio crosses a certain threshold (`MTCT_lower_threshold`) the scheduler decides to expand the resources to make the computation faster.

The analysis of the resource adaptation graph along with the performance metrics graph indicates that the implemented programming model is making the application invasive and the performance data from the model is successfully used by the scheduler for resource adaptation. Listing 7.1 shows the log entry of the resource manager while making the adaptation decisions. We can see that the resource manager receives the data and makes the adaptation decision using that data. Listing 7.2 shows the log entry of the resource manager when it receives the structure of the EPOP application. By this we can conclude that the integration with the resource manager is successfull.

```
-----------------------EPOP RETURNING perf_data_available 1
-----------------------EPOP RETURNING CURRENT PHASE INDEX 2
-----------------------EPOP RETURNING CURRENT MTCT RATIO  0.896405
-----------------------EPOP RETURNING MTCT TREND  0.896405
-----------------------EPOP RETURNING perf_data_available 1
::: IN EPOP IRS_HEURISTIC ::: trend_mtct_epop 0.896405
metric average is 0.852075 ::
irtsched:  Job 2 marked for reduction: idle_node_count increased to 31;
RRV[0]=2
```

Listing 7.1: Log file of the Invasive Resource Manager

```
Type 22000 job_id 2 Num Phases 4 current_phase_index 0
number_of_elements 8 ip_address 10.5.73.38
Phase 1 : Init Phase
No Loop in this Phase
Phase 2 : Rigid Phase
No Loop in this Phase
Phase 3 : Elastic Phase
Loop in this Phase
Phase 4 : Branch Phase
No Loop in this Phase
```

Listing 7.2: Log file of the Invasive Resource Manager

7.3.3 Adaptation of a 2D Jacobi - Profiling method 2

In the previous method, the Invasive Resource Manager requests the performance data and the EPOP driver provides the data whenever a request is received. The scheduler is configured to make scheduling decisions once every 180 seconds (this can be changed using the parameters `SchedulerParameters,sched_min_interval` in the configuration file). At the first pass, the scheduler checks whether the performance data is available for a job and if it is not available, then the scheduler requests the performance data. Only at the next pass, the scheduling decision is made with the performance data obtained with the previous request. That is, the scheduler uses one pass for requesting the performance data and the other pass for making decision. As a result every adaptation decision is delayed by `sched_min_interval` seconds (180 seconds in this scenario).

To speed up the scheduling process, EPOP driver pushes the data proactively upon certain scenarios (see Chapter 5). This method is proved to be very efficient as per the test results on the SuperMUC. The same Jacobi application with the same configuration was started with 32 nodes. Figure 7.4 shows the adaptation decision made based on this method. Figure 7.5 is the comparison of adaptation decision made by the scheduler with profiling methods 1 and 2.

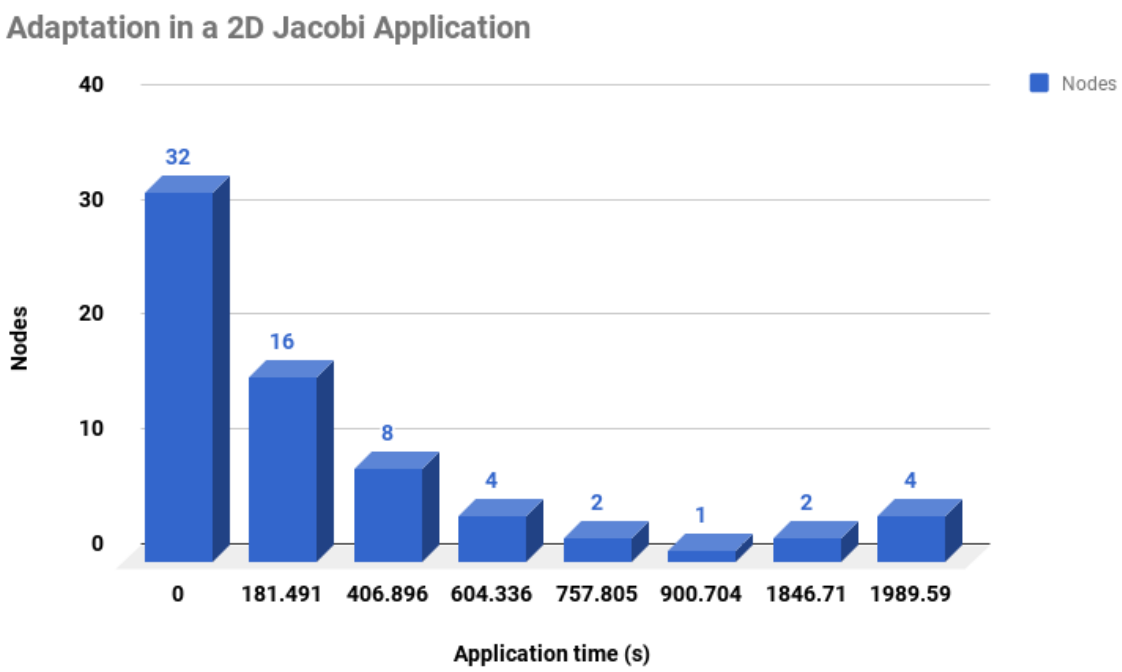

Figure 7.4: Adaptation of a 2D Jacobi using Profiling Method 2

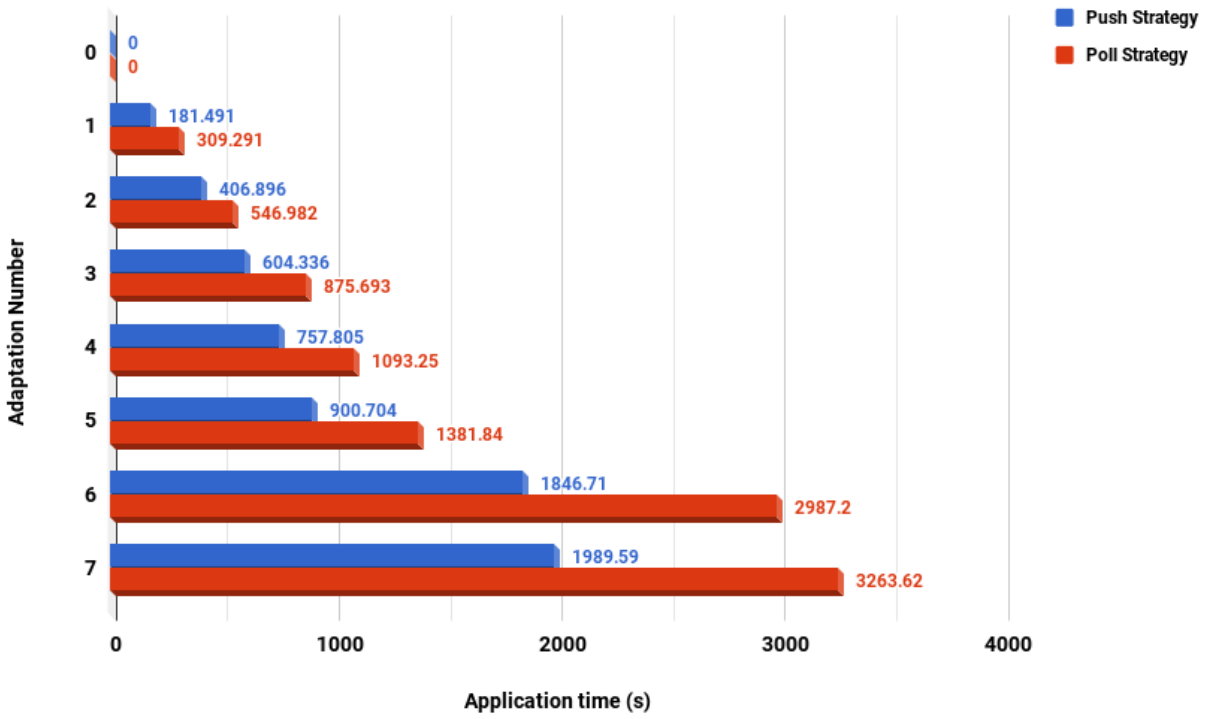

Figure 7.5: Effects of two profiling methods in adaptation of a 2D Jacobi

For comparison, let's refer profiling method 2 as push method and profiling method 1 as poll method. In push method, the first resource adaptation occurs at 181 seconds while in the normal poll method it took about 309 seconds. This difference is same as the scheduler pass interval. In the push method, the EPOP driver sends the message immediately after the execution of first iteration of the phase so that the scheduler can make decision at the first pass itself. While in the poll method, the EPOP driver waits for the message request from IRM to sent the performance data. So the scheduler can only make the same adaptation decision only in the second pass of this iteration.

We can see that every adaptation decision is made faster in the push method. The time taken for seven adaptations was around 1990 seconds in the push method while the poll method took almost double the time around 3263 seconds. Figure 7.5 shows the time taken for both strategies to perform same number of adaptations. It is evident from the graph that by using the push method the scheduler was able to make the same adaptation decisions by saving up to 20 minutes. i.e. scheduler made scheduling decisions 48.5% faster with the push method.

The number of messages sent by the EPOP driver to IRM in both the methods is same. For the normal poll method the driver sent 13 performance data messages upon request from the resource manager to reach seven adaptations. The driver only sent the same number of messages to perform the seven adaptations in the push method. In short, we can conclude that this method used by the EPOP driver is much efficient for sending the performance data than the request and response method implemented currently at the scheduler.

7.4 Benchmark

A simple EPOP application with three Phases (Init, Elastic and Rigid) is created using the previous and current EPOP Models. In both models, EPOP loads the application as a shared library. To analyze the overhead induced by the new design and syntax of EPOP against the previous model, a micro-benchmarking is done for the time taken to load the shared libraries and for the creation of the phases. Google micro benchmark [7] is used for benchmarking both models. The benchmarking was performed on a single node in the SuperMUC.

7.4.1 Loading the shared library

Listing 7.3 shows the benchmarking result of loading the shared library in the previous and current EPOP models. The benchmarked function name, wall clock time, CPU time and the number of iterations are shown in the result. The benchmark column shows the name of the benchmarked function. `Current_EPOP` and `Previous_EPOP` are the names given to the new EPOP model and the previous EPOP model respectively in the benchmarking application. The iterations column denotes the number of times the code was executed to reach this benchmarking result. The average time of all the iterations is taken and shown in the columns Time and CPU.

It can be seen that the previous EPOP model [3] was slightly faster in loading the application than the new EPOP model. There was a slight time difference of 0.002664 milliseconds with the new and old model in loading a three phase application. This increase in time is due to the overhead for creating the Phases in the new model. The benchmarked time for Phase creation is discussed in the next subsection. Considering the new syntax and functionalities, a difference of 0.002664 milliseconds is negligible.

```
Run on (32 X 2701 MHz CPU s)
CPU Caches:
  L1 Data 32K (x16)
  L1 Instruction 32K (x16)
  L2 Unified 256K (x16)
  L3 Unified 20480K (x2)
***WARNING*** CPU scaling is enabled, the benchmark real time measurements
 may be noisy and will incur extra overhead.
-----------------------------------------------------------------
Benchmark              Time           CPU       Iterations
-----------------------------------------------------------------
Current_EPOP        0.004705 ms       0.004700 ms     148602
Previous_EPOP       0.001196 ms       0.001193 ms     589702
```

Listing 7.3: Google Benchmark for loading the shared libraries

Listing 7.4 shows the time taken for loading the shared library of the 2D Jacobi solver (`EPOP_Heat`)and a simple Pi application (`EPOP_Pi`) . `EPOP_Heat` took more time to load the libraries because it has four phases.

```
Run on (32 X 2701 MHz CPU s)
CPU Caches:
 L1 Data 32K (x16)
 L1 Instruction 32K (x16)
 L2 Unified 256K (x16)
 L3 Unified 20480K (x2)
**WARNING*** CPU scaling is enabled, the benchmark real time measurements
may be noisy and will incur extra overhead.
-----------------------------------------------------------
Benchmark            Time           CPU                        Iterations
-----------------------------------------------------------
EPOP_Heat         0.005408 ms       0.005403 ms    121886
EPOP_Pi           0.004968 ms       0.004963 ms    141651
```

Listing 7.4: Google Benchmark for loading the shared libraries in multiple applications

7.4.2 Phase creation

The benchmarking of the creation of the phases is shown in Listing 7.5. The prefix `Previous` is used before the phase creation routines in the previous EPOP model and the prefix `Current` is used for the routines of the new EPOP model.

The first entry in the listing, `Previous_Init_Phase` is the time taken for creating the Init phase in the previous model. 0.000110 millisecond was taken for the Init phase creation. It is faster than the `Current_Init_Phase` creation time which took around 0.000509 milliseconds. This increase in time is due to the vector operation and the creation of class in the new model. This was necessary to make the syntax user-friendly.

Listing 7.6 shows the declaration of an Init Phase in both previous and current models. The previous model needed 9 lines of code for the declaration of a single phase, while the new EPOP model requires only a single line of code. The previous model required the developers to manually allocate the memory for the phases, define the number of phases and provide the type of phases. In the new model, all these things are performed automatically by the driver. This design decision to make a simpler

model increased the phase declaration overhead. In order to declare a phase in a single line, the `setInit` helper method creates the required phase class object, cast it into the superclass Phase and push it into the vector for later use (See Chapter 6). This leads to the increase in the phase creation time.

```
Run on (32 X 2701 MHz CPU s)
CPU Caches:
  L1 Data 32K (x16)
  L1 Instruction 32K (x16)
  L2 Unified 256K (x16)
  L3 Unified 20480K (x2)

***WARNING*** CPU scaling is enabled, the benchmark real time measurements
may be noisy and will incur extra overhead.
-----------------------------------------------------------------
Benchmark                    Time           CPU        Iterations
-----------------------------------------------------------------
Previous_Init_Phase          0.000107 ms    0.000107 ms    6533871
Previous_Rigid_Phase         0.000105 ms    0.000105 ms    6628674
Previous_Elastic_Phase       0.000105 ms    0.000105 ms    6663495
Previous_Branch_Phase        0.000103 ms    0.000103 ms    6785065
Current_Init_Phase           0.000487 ms    0.000486 ms    1443918
Current_Rigid_Phase          0.000730 ms    0.000729 ms     928319
Current_Elastic_Phase        0.001312 ms    0.001311 ms     538147
Current_Branch_Phase         0.000773 ms    0.000772 ms    1000000
```

Listing 7.5: Google Benchmark for creating the Phases

Like in the Init phase, the phase creation time for Rigid, Elastic and Branch phases are lesser in the previous model (See Listing 7.5). From the developer and design perspective, a difference of 0.0004 - 0.001 milliseconds is negligible with an added advantage of a better syntax. Also from the application performance perspective, a difference of 0.0004 - 0.001 milliseconds does not influence the overall performance of the application.

Listing 7.7 shows the declaration of Rigid phase in the previous and new models.

```
Previous EPOP:

        program_t *program = (program_t *)malloc( sizeof(program_t) );
        (program)->size = 1;
        (program)->elements = malloc( 1 * sizeof(program_element_t) );
        program->elements[0].type             = INIT;
        program->elements[0].init             = init;
        program->elements[0].phase_adapt      = NULL;
        program->elements[0].phase_exec       = NULL;
        program->elements[0].loop_condition   = NULL;
        program->elements[0].branch_condition = NULL;

Current EPOP:
        setInit(init);
```

Listing 7.6: Declaring an Init Phase in Previous and Current Models

```
Previous EPOP:
        program->elements[0].type             = RP;
        program->elements[0].init             = NULL;
        program->elements[0].phase_adapt      = NULL;
        program->elements[0].phase_exec       = finalize;
        program->elements[0].loop_condition   = NULL;
        program->elements[0].branch_condition = NULL;

Current EPOP:
        setRigid(finalize,NULL);
```

Listing 7.7: Declaring a Rigid Phase in Previous and Current Models

7.4.3 A simple Invasive application

A simple invasive application to calculate Pi was implemented in both IMPI and EPOP to evaluate the overhead induced by EPOP on a running application. Both applications

were run on a single node (16 processes) in the SuperMUC to check the overall effect of the new model in the application performance. This is not performed using the benchmarking tool.

A performance overhead of 0.06 second was introduced by the new model. This was due to the profiling and the communication of the application structure to the resource manager which is lacking in the normal IMPI model.

```
Application time EPOP Model: 1.1614775 seconds

Application time Normal IMPI Application: 1.09488875 seconds
```

The overhead induced for a simpler user-friendly syntax, the communication with the resource manager and profiling is less than `0.065` seconds.

8 Conclusion

8.1 Goals achieved

As stated in the Introduction section of this thesis, there were five main goals. This section gives an overview of the goals achieved and the future work needs to be done. The first three goals were dealing with the design of the programming model. They are:

1. Implement a fully fledged EPOP model in C++.
2. Create a developer-friendly syntax.
3. Implement the EPOP drivers to use with the newly designed model.

These three goals were achieved with this work. A new fully fledged EPOP model in C++ was designed and implemented using classes and vectors. As a result, a developer-friendly syntax was achieved. Instead of writing 6-9 lines of code for declaring a phase in the previous EPOP model, a developer can create the phase in a single line of code with our new model. With the use of polymorphism and inheritance along with the `std::function` class, this was made possible with a minimum overhead of around 0.0004 to 0.0011 milliseconds for different phase creation. Since the new model requires a new driver to execute the application, a driver program was created to execute the applications designed with the new model.

A simple application that calculates the value of Pi and a complex 2D Jacobi iterative solver for heat simulation were converted into the new model and tested with the driver. All of the applications were executed successfully in the superMUC. To show the invasive behaviour of the EPOP driver, IRM was used to perform a fixed schedule by allocating the resources to the EPOP job in a predetermined schedule (i.e performing

reduction and expansion of resources in a fixed order) and analyzed the effect on the driver. The driver program reduced and expanded the resources according to the allocation made by the IRM (see Chapter 7). In short, we can conclude that the three design goals were achieved with a minimum overhead (0.0001 to 0.001 milliseconds) compared to the previous model.

The remaining two goals were to create the infrastructure for accommodating the EPOP model in the Invasive Resource Manager and Invasive MPI since old EPOP model does not interact with IRM and IMPI. The goals were:

4. Integrate the EPOP model with the Invasive Resource Manager and Invasive MPI

5. Performance monitoring and profiling of the applications developed using the new model.

The fourth goal was to integrate the EPOP model with the IRM and IMPI. A communication module was added in the EPOP driver to interact with the IRM and a new EPOP module was added in IRM to handle the messages from the EPOP driver. Now, the new EPOP model can mark the job as an EPOP job and can pass the structure of the application to IRM. The existing scheduling strategy used the automatic profiling module in the IMPI to schedule the job. Since the EPOP driver gets a more accurate view of the application than the automatic profiling, there is a need for monitoring the performance and profiling the application.

This leads to the fifth goal of performance monitoring and profiling the applications run using the EPOP model. A profiling module was added in IMPI for EPOP jobs which provides the information as per the request of the EPOP driver. To utilize profiling and performance data gathering done by the new model, the IRM scheduler was modified to request the performance data from the EPOP driver whenever a job is an EPOP job. As a result, IRM used the data from the driver to make scheduling decisions. The invasive scheduler allocated resources and taken back the resources by analyzing the data from the EPOP driver and the EPOP driver adapted to the reduction and expansion of resources according to the scheduler allocation. In short, we can conclude that the new model collected the performance data, sent it to IRM, IRM made some scheduling decision with the received data and the EPOP driver successfully adapted to the scheduling decision i.e The above two goals were achieved (see Chapter 7).

Other than the above five goals, an infrastructure for sending the POWER information of a phase, the number of floating point operations per second and pausing mechanism were also implemented.

8.2 Future work

In the present EPOP model, the profiling information like MPI time and phase time were only recorded and passed it to the resource manager. But the framework for extending the type of information was already added in this model. In the future, the energy characteristics of the application can be analyzed. The power usage information of phases can be recorded and can be sent to the resource manager. If the scheduler is modified to make decisions based on the power usage of the application, then this infrastructure can be used to communicate the information with the driver.

The mechanism for pausing the application was also implemented and tested. IRM sents message to pause the application along with duration and the EPOP drivers sleeps the application for the specified duration. This is very useful in scenarios like if any time critical job is submitted, the EPOP jobs can be instructed to sleep till the end of the time critical job and can resume the execution later or when the electricity usage of supercomputer at a certain time is going to cross the approved limit then the EPOP jobs can be asked to pause the execution. This infrastructure can be used in the future if necessary modifications are made in the scheduler.

The EPOP driver loads the application as a shared library and executes the application. In future, a wrapper for the compiler like `mpicc` can be developed for elastic phase oriented programming. It can be integrated with C++ compiler and can compile the elastic applications with ease. This will make the model much simpler. Since the compiler is lower level than the driver program, this model will be very efficient.

List of Figures

Listings

Bibliography

[1] B. Betkaoui, D. B. Thomas, and W. Luk. "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing." In: *Proceedings of the International Conference on Field-Programmable Technology, FPT 2010, 8-10 December 2010, Tsinghua University, Beijing, China*. 2010.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. "X10: An Object-oriented Approach to Non-uniform Cluster Computing." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. 2005.

[3] I. A. Comprés Ureña. "Resource-Elasticity Support for Distributed Memory HPC Applications." Dissertation.

[4] cppreference. *Language linkage*. `https://en.cppreference.com/w/cpp/language/language_linkage`.

[5] M. Daga, Z. S. Tschirhart, and C. Freitag. "Exploring Parallel Programming Models for Heterogeneous Computing Systems." In: *2015 IEEE International Symposium on Workload Characterization*. 2015.

[6] J. Diaz, C. Muñoz-Caro, and A. Niño. "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era." In: *IEEE Transactions on Parallel and Distributed Systems* (2012).

[7] Google. *Google Micro Benchmark*. `https://github.com/google/benchmark`.

[8] C. Huang, G. Zheng, L. Kalé, and S. Kumar. "Performance Evaluation of Adaptive MPI." In: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2006.

[9] C. Huang, G. Zheng, and L. V. Kalé. *Supporting Adaptivity in MPI for Dynamic Parallel Applications*. Tech. rep. 2007.

[10] Intel. *Intel Xeon Processor*. https://ark.intel.com/products/64583/Intel-Xeon-Processor-E5-2680-(20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI).

[11] M. A. Jette, A. B. Yoo, and M. Grondona. "SLURM: Simple Linux Utility for Resource Management." In: *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. 2002.

[12] L. V. Kale and S. Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based on C++." In: *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*. 1993.

[13] Leibniz-Rechenzentrum. *SuperMUC Petascale System*. https://www.lrz.de/services/compute/supermuc/.

[14] MPICH. *MPICH High-Performance Portable MPI*. https://www.mpich.org/.

[15] wiki mpich. *MPI Tools Interface*. https://wiki.mpich.org/mpich/index.php/Tool_interfaces_(MPI-T),_MPICH_parameters_and_instrumentation.

[16] P. E. Ross. *Why CPU Frequency Stalled*. https://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled.

[17] S. V. S. "PVM: A framework for parallel distributed computing." In: *Concurrency: Practice and Experience* ().