Final thesis

Optimized Composition of Parallel Components on a Linux Cluster

by

Anas Al-Trad

LIU-IDA/LITH-EX-A—12/066—SE

2012-11-27
Final Thesis

Optimized Composition of Parallel Components on a Linux Cluster

by

Anas Al-Trad

LIU-IDA/LITH-EX-A—12/066—SE

2012-11-27

Supervisor: Mudassar Majeed
Examiner: Christoph Kessler
ABSTRACT

We develop a novel framework for optimized composition of explicitly parallel software components with different implementation variants given the problem size, data distribution scheme and processor group size on a Linux cluster. We consider two approaches (or two cases of the framework). In the first approach, dispatch tables are built using measurement data obtained offline by executions for some (sample) points in the ranges of the context properties. Inter-/extrapolation is then used to do actual variant-selection for a given execution context at run-time.

In the second approach, a cost function of each component variant is provided by the component writer for variant-selection. These cost functions can internally lookup measurements’ tables built, either offline or at deployment time, for computation- and communication-specific primitives.

In both approaches, the call to an explicitly parallel software component (with different implementation variants) is made via a dispatcher instead of calling a variant directly.

As a case study, we apply both approaches on a parallel component for matrix multiplication with multiple implementation variants. We implemented our variants using Message Passing Interface (MPI). The results show the reduction in execution time for the optimally composed applications compared to applications with hard-coded composition. In addition, the results show the comparison of estimated and measured times for each variant using different data distributions, processor group and problem sizes.
Acknowledgements

I would like to thank my examiner Christoph Kessler for his support and ideas. Special thanks to my supervisor Mudassar Majeed for his support, ideas and for the nice company during every stage in this thesis. I would like also to express my deep gratitude to Usman Dastgeer for sharing his valuable help.

I am also thankful to the members of National Supercomputer Center in Linköping (NSC) for providing me access to Neolith Cluster for running the experiments and for their support.

I am very grateful to my family and friends for their support during this thesis. Special thanks to my parents, Ibrahim Al-Trad and Mariam Alwidian, for their encouragement, support and patience during my study.

I am also very grateful to my beloved wife, Amena Al-Tradat, for her love, support, patience and inspiration.
## Contents

1 Introduction

1.1 Motivation .................................................. 1
1.2 Problem Definition and Proposed Solution ................. 2
1.3 Contributions ............................................... 3
1.4 Outline ....................................................... 4

2 Background

2.1 Parallel Computing ........................................ 5
   2.1.1 Parallel Computers .................................... 5
   2.1.2 Parallel Programming Models ......................... 5
   2.1.3 Parallel Execution Styles ............................. 6
2.2 Message Passing Interface (MPI) .......................... 6
   2.2.1 MPI Communicator ..................................... 7
   2.2.2 MPI point-to-point and Collective Communications 7
2.3 Software Composition ...................................... 7
   2.3.1 Software Components .................................. 7
   2.3.2 The Software Composition Concept ................... 9
   2.3.3 Optimized Composition ............................... 10
2.4 Auto-tuning ................................................ 10
## Contents

3 The Proposed Framework
- 3.1 Overview ........................................ 12
- 3.2 Component Interfaces .......................... 12
- 3.3 Dispatcher ..................................... 14

4 Case Study
- 4.1 Matrix Data Structure ....................... 15
- 4.2 Optimized Composition: Semi-Performance-aware ...... 16
  - 4.2.1 ScALAPACK Distribution Schemes .......... 16
  - 4.2.2 Selected Variants ......................... 18
  - 4.2.3 The Dispatcher and The Dispatch Table .... 22
- 4.3 Optimized Composition: Performance-aware .......... 23
  - 4.3.1 Implemented Data Distribution Schemes .... 23
  - 4.3.2 Implemented Variants ................... 26
  - 4.3.3 Estimating the Computation Time .......... 28
  - 4.3.4 Estimating the Communication Time ....... 28
  - 4.3.5 Overall Time Estimation ................. 29
  - 4.3.6 Dispatch Function ....................... 30

5 Experimental Setup and Evaluation .................. 31
- 5.1 Optimized Composition: Semi-Performance-aware ...... 31
  - 5.1.1 Experimental Setup ....................... 31
  - 5.1.2 Experimental Evaluation .................. 33
- 5.2 Optimized Composition: Performance-aware .......... 37
  - 5.2.1 Experimental Setup ....................... 37
  - 5.2.2 Experimental Evaluation .................. 38

6 Discussion and Conclusions ....................... 44
Contents

7 Related Work ........................................... 46
8 Future Work ............................................ 48
Chapter 1

Introduction

1.1 Motivation

Software complexity increases due to many factors such as changing customer requirements, or the need to reduce the development cost by automating the manual tasks [12]. Component-based software development (CBSD) is one of the most accepted approaches by developers to manage such complexity and to construct scalable, flexible, reliable, and large software systems by reusing pre-defined and pre-tested partial units [21]. This trend was expected by McIlroy's paper [9] in which the author states and encourages the idea of mass-produced software components.

Nowadays, we can find many mass-produced software components. This vast existence of such components creates a need to know which one to compose/use with an application (e.g., which one has the lowest execution time given a certain problem size). One solution [5][4] is to make components performance-aware, by annotating performance-related meta-data with components. Usually, there is no unique performance-aware component that is optimal for all possible values of given context properties (for example, problem size, number of processes, and data distribution schemes), which results in a need, in this case, for an optimized composition technique based on performance. Current component technology works well in sequential and concurrent object-oriented programming but lacks performance aware composition of components [5]. However, in the High Performance Computing (HPC) domain, performance aware composition is necessary for building complex and scalable systems.

As an example, consider that an application developer has to compose a matrix multiplication parallel component (where it can be executed on mul-
tiple processes) with an application for a distributed MPI platform. Then, one possibility is to use third party libraries like Scalable Linear Algebra PACKage (ScaLAPACK) [34] that has a parallel matrix multiplication subroutine available. In this case, Application developer is responsible for organizing the set of MPI processes as Cartesian grid and distribute the matrices among this grid, which results in multiple implementation configurations (variants) (i.e., for different grids, or different distributions of matrices). The developer cannot know in advance which variant is the best in order to achieve an optimized composition for any given execution context. This is because of non-trivial inter-play between implementation details of an algorithm (cache behavior, data locality, vectorization etc.) and underlying hardware architecture (cache hierarchy, memory and network bandwidth, SIMD etc.). Furthermore, the problem becomes even more complex if we consider the applicability of available implementation variants as some variants may only work for certain execution contexts. One notable example is implementation variants that work only for certain problem sizes (e.g., multiples of cache-line sizes etc.).

Because of above mentioned issues, our goal, in this thesis, is to achieve an optimized composition that automatically will select the best variant. In this thesis, we propose a novel framework for optimized composition of different parallel components for MPI clusters. This framework can be consulted at runtime to call the expected best performance variant instead of using hard-coded composition of variants where one variant is statically selected and composed with the application.

1.2 Problem Definition and Proposed Solution

We define a component as a module with an interface for composition. If there is no implementation to a component, then it is an abstract component. The different implementations of a component are called component variants. An abstract component can be substituted by a component variant. In this work, we refer to a component variant as variant. We define also context properties (or component call properties) as values or pre-conditions that are evaluable before a call is composed (bound) to its callee (variant) definition. The call properties can numerously vary, however, in this work, they are chosen to be the problem size, the MPI group size (number of processes) and different data distributions for operand data (e.g., how the input matrices of a matrix multiplication algorithm are distributed among given MPI processes). Thus, the problem of optimized composition could be stated as follows:

1Optimized depends upon one's objective function which is 'reduction of execution time' for the scope of our thesis project.
1.3. CONTRIBUTIONS

For each function call to an abstract component, the composer should choose the best variant from a set of implemented variants, based on particular criteria and given call properties.

The criterion that is used to find the expected best variant is performance (i.e., reducing execution time). To solve this, we develop a novel framework (see chapter 3 for details) for composing explicitly parallel MPI components on a Linux cluster environment. We exploit a single call in an application to such components and conduct optimized composition on multiple implementation variants in order to reduce the execution time.

Two approaches have been investigated:

- In the first approach, different configurations of ScalAPACK have been specified and selected as semi-performance-aware variants\(^2\). In addition to these variants, the variants replicating A matrix and replicating B matrix and serial (i.e., only one processor is involved in the multiplication operation) variants are also considered.\(^3\)

- In the second approach, different performance-aware implementation variants (i.e., different algorithms) written in MPI are used. In this case, the variant designer is required to provide the time estimation functions for the implementation variants, given the communication primitives’ execution time tables computed offline or at deployment time.

In both approaches, the call to an explicitly parallel software component (with different implementation variants) is made via a dispatcher instead of calling a variant directly. Then, the dispatcher calls the expected best variant of the parallel component for given context property values after looking up into a dispatch table built offline or at deployment time, or after executing time estimation functions of the variants of the performance-aware parallel component in order to find the expected best performing one.

1.3 Contributions

The main contributions of our work are as follows:

- We do an optimised composition case study on MPI Linux platform.

\(^2\)Semi-performance-aware variant is a variant that has a method for measuring its execution time (profile method) in its performance-aware interface (see chapter 3).

\(^3\)Assuming that the matrix multiplication operation is \(A \times B = C\), replicating means that all processes have a copy of the entire matrix.
CHAPTER 1. INTRODUCTION

• We propose two different techniques for (semi) automatic optimised composition. Moreover, experimental results are used to compare the effectiveness of both approaches.

• We investigate and document the potential issues when using time-estimation functions in an actual MPI distributed environment. We discuss possible issues with writing estimation functions and how certain measurement data about computation and communication can help in this regard.

• Last but not the least, we present that our technique can actually provide superior performance in some cases to a pure static choice by automatic dynamic selection for each execution context.

1.4 Outline

The rest of the thesis is outlined as follows: In chapter 2, we give some background about parallel computing and software composition. More precisely, we give an introduction to parallel execution styles and parallel programming models. Also in the same chapter, we give an introduction to software components and software composition concepts including optimized composition and a brief introduction to auto-tuning. Chapter 3 describes the proposed framework in details. Chapter 4 presents the details of the case study. Experiments and results are presented and discussed in Chapter 5. Chapter 6 provides discussions and conclusions about the two approaches investigated in the thesis. Chapter 7 presents related work. Finally, Chapter 8 presents possible future work.
Chapter 2

Background

2.1 Parallel Computing

Parallel computing has already become mainstream [10]. It is mainly used to solve computation problems faster (than a single computing unit) and to solve problems with heavy computational complexity such as galaxy simulation and weather forecasting where they cannot be solved on a single computing unit in reasonable and acceptable time.

2.1.1 Parallel Computers

A parallel computer can be a multiprocessor or a multicomputer. In the former case, two or more processors are tightly connected together with a shared memory (possibly with different levels of caches). In the latter case, two or more computers are loosely connected together using a high speed inter-connection network. A cluster is an example of such a parallel computer where many nodes are loosely connected together by a high speed network. Each node in a cluster is by itself a standalone parallel computer (usually a multiprocessor having its own shared memory). Such multicomputer machines are also called distributed memory computers.

2.1.2 Parallel Programming Models

In the design of sequential algorithms, programmers assume that there is a single processor that is connected to a main memory which can be accessed randomly. This programming model is called Random Access Ma-
chine (RAM) model. However, for parallel computers there are many programming models that are proposed and implemented so that programmers can use those models to program parallel computers. One of these proposed models is Parallel Random Access Machine (PRAM) which is an extension to the RAM model [3]. The shared memory model resembles the PRAM model in some aspects. However, they differ in resolving conflicting accesses to the shared memory [3]. In PRAM, it is assumed that all memory accesses are carried out in one unit of time and conflict resolution is deterministic by the hardware, while in shared memory the responsibility of resolving concurrent memory access conflicts is delegated to the programmer and hardware[3].

Many other models exist, such as Bulk Synchronous Parallel (BSP) model, or Data Parallel Computing models. For more information, the reader can refer to Kessler et al. survey article in [3]. The model used in this research is the Message Passing model. This model assumes that processors can communicate with each other by sending and receiving messages regardless of the network structure used between the processors [3]. An example of such a model is MPI (Message Passing Interface) standard.

2.1.3 Parallel Execution Styles

Fork-join and SPMD (Single Program Multiple Data) are the most significant parallel execution styles.

- In fork-join style, at the beginning and end of an execution of a program only one process or thread is being executed, and many processes/threads can be spawned dynamically at certain points in the program.
- In SPMD style, a fixed number $p$ of processes/threads are executed from the beginning of program execution till the end and no new parallel processes are spawned.

2.2 Message Passing Interface (MPI)

An implementation of the MPI standard is a message passing parallel programming library that is incorporated into programming languages such as C or Fortran. An MPI program executes in an SPMD parallel execution style. In this section, we describe only the important and basic topics related to this thesis work.
2.3 SOFTWARE COMPOSITION

2.2.1 MPI Communicator

The communicator object (MPI_Comm) determines the scope at which its contained processes can communicate. Each process contained in a communicator has a unique identifier called rank. The rank of an MPI process acts as an identifier of that process within that communicator object. Besides other usage, ranks are used to identify the source and the destination processes when sending a message. MPI_COMM_WORLD is the default communicator which includes all MPI processes. The MPI_COMM_DUP function creates a new communicator with same processes as the input communicator but has a different communication context.

2.2.2 MPI point-to-point and Collective Communications

Point-to-point communication occurs between any two MPI processes. In this type of communication, the sender process executes a send operation (for example MPI_Send) and the receiving process executes the corresponding receive routine (MPI_Recv).

A collective communication operation involves all processes in a given communicator object. In this work, we have used the following collective communication primitives: MPI_Bcast, MPI_Scatter, MPI_Gather, and MPI_Reduce. Figure 2.1 illustrates these subroutines where the reduce operator for MPI_Reduce is MPI_SUM. In MPI_Bcast, each process has a copy of the value (a0) of the root process (p0). MPI_Scatter scatters the root process (p0) data to all other processes. In MPI_Gather, each process sends its value to one process (p0) which stores them in rank order. When the reduce operation is MPI_SUM, MPI_Reduce adds the values of all processes and store the result at the root process.

2.3 Software Composition

2.3.1 Software Components

The most widely used and accepted definition for software component is excerpted from [7]:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."
CHAPTER 2. BACKGROUND

Each component has a model that describes its appearance and its detailed meta information like location, lifetime, language, etc. Among many other things, it also describes binding points, binding times, and interfaces.

Figure 2.2 shows a black-box component (i.e, a component where its internal details are hidden and not changed (or used) by a composer (e.g., programmer)) and its provided interfaces. Other types of components are white-box component where the internal details are visible to a composer, and grey-box component where parts of the implementation are subject for adaptation by a composer.

The different implementations of a component are called component variants. Ericsson et al. [21] define a component variant as a component substitutable for an abstract component (a component without an implementation). A parallel software component is a component that contains independent malleable tasks where a malleable task is a computational unit that can be executed on multiple processors [4]. A component that implements a specific performance interface is called performance-aware component [5].
2.3. SOFTWARE COMPOSITION

Matrix Multiply Component

Figure 2.2: A black-box component that provides Multiply and Profile Interfaces.

Figure 2.3: Blackbox Composition

2.3.2 The Software Composition Concept

Composition refers basically to connecting two or more components together to form a large composite construct [17]. The main goal of composition is to reuse existing components in order to reduce the development time and facilitate the construction of complex and large software systems.

Components are composed together using a composition technique and a composition language, where the composition technique describes how and when (compile, link, deployment, connection, runtime) the components are glued or merged together, while the composition language manages and describes the composition process. A composition technique, a composition language and a component model (section 2.3.1) represent a software composition system [29].

Figure 2.3 shows a black-box composition which is very similar to the CORBA composition system. As in CORBA the caller (client) calls the callee (server) through a mediator which binds the caller to the callee. The figure shows a caller where the binding is delegated to a dispatcher which binds the caller with one callee from several callees. Other types of composition like white-box composition and grey-box composition are not used in this thesis.

More precisely and specifically, the composition can be described as bind-
CHAPTER 2. BACKGROUND

ing of a call to its definition [21]. This binding could be, for example, at compilation time, or dynamically at run time such as in polymorphism (where the binding is carried out based on object types using runtime lookup). In contrast to polymorphism, call context properties can have any value that can be extracted from the call and is accessible (i.e. known) before the binding time [21]. For instance, such properties can be the effective parameters (their types or values, they are parameters that affect the non-functional properties of functions like execution time), and the number of processors allotted for the called component. Also, the context properties could be fields or subtypes of a parameter data structure.

2.3.3 Optimized Composition

The optimization goal can be performance, the resources used, the energy consumed, or a hybrid of these criteria. In case of multiple goals, many issues arise like the trade-off across goals, the prioritization between the goals, and solving the conflicts across these goals. One optimization goal that is required on Clusters is performance (i.e. execution time). As mentioned in Section 1.2, choosing the expected best implemented variant according to a specified optimization goal is the optimized composition problem.

2.4 Auto-tuning

The basic idea of auto-tuning is to use searching and/or machine learning to achieve automatic optimization on target platforms. As library developers or software developers usually do not know the target platform, they try to build libraries or softwares that are automatically adaptable. For example, ATLAS [23] is a software that generates libraries which are tuned to the target platform (where it implements Basic Linear Algebra Subprogram (BLAS) [31]). In order to provide portable performance across many platforms, ATLAS basically uses a code generation approach to generate the optimized routine for the target architecture. For example, it searches for the best blocking factor and for the best unrolling factor for a given matrix multiplication computation. It bases its search on the target platform specification like L1 cache and number of registers.

There are mainly two approaches to autotuning; one is called model-driven optimization [27]. This approach can be applied by compilers where tunable parameters are determined by analytical models [27]. On the other hand, large number of parameterized variants for a given algorithm are generated and executed on a certain platform in order to find the best performance one [27]. This approach is called empirical autotuning. Our work on the optimized composition for semi-performance-aware parallel components
2.4. AUTO-TUNING

is an example of empirical autotuning where problem size and number of MPI processes are the context properties that are used to determine the tuning parameters (here, the implementation variant to be selected). Furthermore, we do not consider the intra-variant tuning (i.e., local parameters of a variant such as block size, unrolling factor, etc.).
Chapter 3

The Proposed Framework

3.1 Overview

Figure 3.1 shows the framework of optimized composition of parallel components. At the top level, there is an application that has several parallel component calls. For each call, the composition process is carried out where instead of calling one of the $m$ available variants directly, the application consults a dispatcher. This process is carried out at run-time.

The Offline Profiler builds dispatch data information at deployment time by calling the Profile Interface of the variants. However, performance-aware variants have a different interface called Time Interface to indicate that they are performance-aware. In our work, we consider two special cases of this framework: i) a parallel abstract component where its variants are considered as semi-performance-aware variants (variants that have only profile methods in their performance-aware interfaces) ii) a parallel abstract component where all variants are performance-aware.

It is important to note that a variant is considered as a black-box variant even though its internal details might be visible. In other words, the composition type is black-box composition. The reason is that the internal details of the variant is not changed/touched at the composition.

3.2 Component Interfaces

Components are connected together via their interfaces [7]. Listing 3.1 shows an example of the interfaces for performance-aware and semi-performance-
3.2. COMPONENT INTERFACES

Composed Parallel App. (using components C1..Cn)

Start → Call C1 → Call C2 → Call Cn → End

Call C1(param1, param2, .. param k)

1) call dispatcher
2) lookup
3) call best variant

Component C1
(with m variants)

Dispatch Table

V1 → V2 → V3 → Vm

Update Dispatch Table

Offline Profiler for C1

Deployment Time

Run Time

Figure 3.1: The Framework of Optimized Composition
Chapter 3. The Proposed Framework

aware abstract components for matrix multiplications. This example will be used in Chapter 4 and is presented here to illustrate the general concept in more detail. In both interfaces, the multiply method represents the call interface which is used by the dispatcher. The profile interface is represented by an execute method that is a connection point between the offline profiler and the variants to build a dispatch table. The time method represents the performance-aware interface (i.e. Time Interface).

Listing 3.1: The Component’s Interfaces.

class MatMul{
    public:
    virtual Matrix* multiply(Matrix *A, Matrix *B, Matrix *C) = 0;
};
class SemiPerformanceAwareMatMul : public MatMul{
    public:
    virtual double execute(Matrix *A, Matrix *B, Matrix *C) = 0;
};
class PerformanceAwareMatMul : public SemiPerformanceAwareMatMul{
    public:
    virtual double time(Matrix *A, Matrix *B) = 0;
};

3.3 Dispatcher

The Dispatcher does two things:

1. It performs lookup in the dispatch table.
2. It calls the expected best variant found.

Dispatching consists either of looking up a dispatch table or of a search process for the best variant. If all variants are performance-aware the dispatcher calls each variant’s Time interface. Otherwise, it is a normal table lookup process, where, if the required entry is not present directly in the table, an interpolation technique is used.

The dispatcher is usually generated from the interfaces, see e.g. the PEPPHER composition tool [28]. In this work we use hand-generated dispatchers for simplicity.
Chapter 4

Case Study

As case study we consider here optimized composition of parallel matrix multiplication variants on a Linux cluster using MPI.

4.1 Matrix Data Structure

In order to keep track of the operand matrices’ elements among the processes, each process should know:

- The distribution type of each matrix.
- The number and topology of processes.
- The global number of rows and columns of each matrix.
- The local number of rows and columns owned by each process for each matrix.

The matrix data structure is specified in Listing 4.1. Each process has a copy of this data structure where local_rows and local_cols variables are initialized according to the distribution scheme. The Distribution_scheme is a data structure that specifies the distribution scheme of the matrix (see Section 4.3.1). Finally, the context variable is a MPI_Comm object copied by the MPI_Comm_dup subroutine which represents the MPI communication context in which the matrix is distributed.
4.2 Optimized Composition of Semi-Performance-aware Parallel Variants

4.2.1 ScaLAPACK Distribution Schemes

2D-Block-Cyclic distribution scheme is the general distribution scheme where a matrix is distributed in two dimensions among processes initialized as a Cartesian grid. ScaLAPACK [34] uses this scheme for their matrix multiplication algorithms. It can be briefly described as follows:

1. Processes and matrix A are divided into a 2-dimensional process grid and 2-dimensional sub-matrices respectively.
2. The division of processes is done by using Cblacs_gridinit routine of Basic Linear Algebra Communication Subprograms (BLACS) [30].
3. For the division of matrix A, two block/tile sizes MB and NB are considered where the rows of the matrix are divided into groups of size MB and the columns into groups of size NB.
4. The resultant sub-matrices are distributed across the process grid in a cyclic manner.

Figure 4.1 shows an example of a (2×2) process grid and a (9×9) matrix and shows how the sub-matrices are distributed when MB = 2 and NB = 2. In this example, the process(0,0) has a 5 × 5 submatrix while the process(1,1) has a 4 × 4 submatrix. The local rows and local columns of both submatrices are depicted in the figure.

1BLACS is the communication layer of ScaLAPACK which is used to ease the programming of linear algebra applications and to provide more portability. Cblacs_gridinit routine specifies how given processes are mapped into a Cartesian BLACS process grid.
4.2. OPTIMIZED COMPOSITION: SEMI-PERFORMANCE-AWARE

Matrix A (9X9)

\[
\begin{array}{cccccc}
A_{00} & A_{01} & A_{04} & A_{05} & A_{08} \\
A_{10} & A_{11} & A_{14} & A_{15} & A_{18} \\
A_{40} & A_{41} & A_{44} & A_{45} & A_{48} \\
A_{50} & A_{51} & A_{54} & A_{55} & A_{58} \\
A_{80} & A_{81} & A_{84} & A_{85} & A_{88} \\
A_{22} & A_{23} & A_{26} & A_{27} \\
A_{32} & A_{33} & A_{36} & A_{37} \\
A_{62} & A_{63} & A_{66} & A_{67} \\
A_{72} & A_{73} & A_{76} & A_{77} \\
\end{array}
\]

Process Grid (P x Q)

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 1 & m & k & \end{array}
\]

Figure 4.1: 2D-Cyclic Distribution Scheme

Table 4.1: Special distribution schemes (for \(A(m \times n)\) matrix and \(P \times Q\) process grid

<table>
<thead>
<tr>
<th>Dist. scheme</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D-Cyclic</td>
<td>(m_b = m, n_b = 1, 2D \text{ process grid} )</td>
</tr>
<tr>
<td>2D-Block</td>
<td>(m_b = m/P, n_b = m/Q, 2D \text{ process grid} )</td>
</tr>
<tr>
<td>2D-Block-Cyclic</td>
<td>(1 &lt; m_b \leq m/P, 1 &lt; n_b \leq n/Q, 2D \text{ process grid} )</td>
</tr>
<tr>
<td>Column-Cyclic</td>
<td>(m_b = n, n_b = m/P, P = 1, Q = \text{nprocs} )</td>
</tr>
<tr>
<td>Column-Block</td>
<td>(m_b = m/n_b, n_b = n/Q, P = 1, Q = \text{nprocs} )</td>
</tr>
<tr>
<td>Row-Cyclic</td>
<td>(m_b = 1, n_b = n, P = \text{nprocs}, Q = 1 )</td>
</tr>
<tr>
<td>Row-Block</td>
<td>(m_b = m/P, n_b = m, P = \text{nprocs}, Q = 1 )</td>
</tr>
<tr>
<td>Row-Block-Cyclic</td>
<td>(1 &lt; m_b \leq m/P, n_b = n, P = \text{nprocs}, Q = 1 )</td>
</tr>
<tr>
<td>Replicated</td>
<td>each process has a copy of the whole matrix</td>
</tr>
<tr>
<td>Mono</td>
<td>process(0,0) has the matrix alone</td>
</tr>
</tbody>
</table>

However, choosing how the processes are initialized and how the matrix is divided (values of \(MB\) and \(NB\)) leads to a distribution scheme that can be considered as a special case of this general scheme. For example, if the grid in Figure 4.1 is initialized into \((4 \times 1)\) and we choose \(MB = 2\) and \(NB = 9\) then we get the Row-Block distribution scheme. Table 4.1 shows different distribution schemes when considering special values for the grid and for \(NB\) and \(MB\) block sizes. It is important to note that the Replicated distribution scheme cannot be considered as a special case of that general distribution scheme.

It is important to note also that all distribution schemes (except Replicated) represent a possibly slightly un-balanced distribution of data where all processes do not have the same amount of data. This un-balanced distribution is eased when initializing the grid into a square grid and the block sizes \(MB\) and \(NB\) are equal to 1. In addition, this un-balance is negligible if the matrix dimensions are much larger than the process grid dimension.
CHAPTER 4. CASE STUDY

4.2.2 Selected Variants

In this approach we assume that all variants take matrices $A$, $B$, and $C$ as input parameters and compute the operation $C = A \ast B$, where these input matrices are in Mono distribution scheme. However, these matrices can internally be translated within a variant to its required distribution schemes.

The Serial Variant

The implementation of this variant is shown in Listing 4.2. We leverage the optimized implementation of Basic Linear Algebra Subprogram (BLAS) [31] subroutine $gsl\_blas\_dgemm$ written by GNU Scientific Library [32]. Before calling the multiplication, it assumes that $A$, $B$, and $C$ matrices are in Mono distribution and all have the same context.

Listing 4.2: The Serial Variant (Inherited from SemiPerformanceAwareMatMul Class).

```cpp
Matrix\* Serial::multiply (Matrix\* A, Matrix\* B, Matrix\* C){
    if ( !( (A->context==B->context) && (A->context==C->context) )
        && (B->context==C->context) )
        return 0;
    int m = A->global_rows;
    int k = A->global_cols;
    int n = B->global_cols;
    gsl_matrix_view AA = gsl_matrix_view_array(A->data, m, k);
    gsl_matrix_view BB = gsl_matrix_view_array(B->data, k, n);
    gsl_matrix_view CC = gsl_matrix_view_array(C->data, m, n);
    // the following subroutine computes $C = x*A + y*C$, where $x$ and $y$ here are 1.0 and 0.0, respectively.
    gsl_blas_dgemm (CblasNoTrans, CblasNoTrans, 1.0, &AA.matrix, &BB.matrix, 0.0, &CC.matrix);  
    return C;
}
```

ReplicatedA Variant

This component variant requires that matrix $A$ is replicated across all processes while matrix $B$ is distributed using the Column-Block scheme (see section 4.3.1). The resultant matrix $C$ is also distributed Column-Block. Each process will execute the serial matrix multiplication subroutine $gsl\_blas\_dgemm$ on its own local data. Listing 4.3 demonstrates the variant's pseudocode.
4.2. OPTIMIZED COMPOSITION: SEMI-PERFORMANCE-AWARE

Listing 4.3: The ReplicatedA Variant Pseudocode (Inherited from SemiPerformanceAwareMatMul Class).

```cpp
Matrix* ReplicatedA::multiply(Matrix *A, Matrix *B, Matrix *C)
{
    ...
    // change the distribution of the A matrix to Replicated
    and B matrix to Col-Block
    AA->init_alloc(m, k, Replicated, A->context);
    BB->init_alloc(k, n, col_block, A->context);
    distributor.change_distribution(A, AA);
    distributor.change_distribution(B, BB);
    // initialize matrix CC to Col-Block
    CC->init_alloc(m, n, col_block, AA->context);
    // call matrix multiplication on the local data of AA, BB,
    // and CC matrices
    gsl_blas_dgemm(...);
    // change the distribution of CC matrix to Mono
    distributor.change_distribution(CC, C);
    return C;
}
```

This variant may consume much memory, thus it is important to apply a memory consumption constraint. To illustrate this, consider, for example, a relatively large matrix (e.g., $14000 \times 14000$) and assume that the available physical memory for each node at a certain cluster is 16GB. MPI_Bcast routine can efficiently broadcast this matrix across process group in short time especially when the group size is small (e.g., 8). Thus, ReplicatedA variant can be as fast as other parallel variants that consume less memory and have communication routines within their computation algorithms. In this case, ReplicatedA requires around 88% of the total physical memory of that node. If an application is using more than 12% of the available physical memory, then calling this variant takes higher time than what is measured in the dispatch table where it is built offline.

If we assume that the user has $A$, $B$, and $C$ matrices in Mono distribution scheme, then Equation 4.1 represents an upper bound for the node memory consumption$^2$:

$$[mk(1 + \min(p, NPN)) + (m + k)(n + \lceil \frac{n}{p} \rceil \times \min(p, NPN))] \times \text{sizeof(double)}.$$  \hspace{1cm} (4.1)

Where $NPN$ in the equation refers to the number of processes per node, \(\min\) is the minimum function of two values, $m$, $k$, and $n$ are matrices dimensions, and $p$ is the group size. The term $mk(1 + \min(p, NPN))$ refers to the amount of memory needed to replicate input matrix $A$ at one node, while

$^2$We don’t consider the time because we use empirical time modeling for ScaLAPACK variants. Analytical time model will be given for similar variants in section 4.3.
CHAPTER 4. CASE STUDY

the second term is for the memory needed for changing $B$ and $C$ matrices to the Col-Block distribution schemes.

ReplicatedB Variant

This variant is similar to ReplicatedA, however, it assumes that matrix $B$ is replicated among all processes, while matrices $A$ and $C$ are distributed into Row-Block scheme. Listing 4.4 demonstrates the Pseudocode of this variant.

```
Listing 4.4: The ReplicatedB Variant (Inherited from SemiPerformanceAwareMatMul Class).

Matrix* ReplicatedB::multiply (Matrix *A, Matrix *B, Matrix *C) {
    ...
    // change the distribution of the A to row_block and B to Replicated
    AA->init_alloc(m, k, row_block, A->context);
    BB->init_alloc(k, n, Replicated, A->context);
    distributor.change_distribution(A, AA);
    BB->init_alloc(k, n, Replicated, A->context);
    distributor.change_distribution(B, BB);
    // initialize matrix C to Row-Block
    CC->init_alloc(m, n, row_block, AA->context);
    // call matrix multiplication on the local data of AA, BB, and CC
    gsl_blas_dgemm (...);
    // change the distribution of CC matrix to Mono in C
    matrix
distributor.change_distribution(CC, C);
    return C;
}
```

If we assume that $A$, $B$, and $C$ matrices are in Mono distribution schemes then Equation 4.2 represents an upper bound for the node memory consumption:

$$[nk(1 + \min(p, NPN)) + (n+k)(m + \left\lceil \frac{m}{p} \right\rceil \times \min(p, NPN))] \times \text{sizeof(double)}. \tag{4.2}$$

Where $NPN$ in the above equation refers to the number of processes per node, $\min$ is the minimum function of two values, $m$, $k$, and $n$ are matrices dimensions, and $p$ is the group size. The term $nk(1 + \min(p, NPN))$ refers to the amount of memory needed to replicate $B$ input matrix, while the second term is for the memory needed for changing $A$ and $C$ matrices into the Row-Block distribution schemes.
Table 4.2: Different ScaLAPACK Configurations

<table>
<thead>
<tr>
<th>Variant no.</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( P \geq Q, \ mb = k, k_B = 1 )</td>
</tr>
<tr>
<td>2</td>
<td>( P &gt; Q, \ mb = m/P, k_B = k/Q, k_B = k/P, nb = n/Q )</td>
</tr>
<tr>
<td>3</td>
<td>( P &gt; Q, \ mb = m/(2P), k_B = k/(2Q), k_B = k/(2P), nb = n/(2Q) )</td>
</tr>
<tr>
<td>4</td>
<td>( P &lt; Q, \ mb = k, k_B = nb = 1 )</td>
</tr>
<tr>
<td>5</td>
<td>( P &lt; Q, \ mb = m/P, k_B = k/Q, k_B = k/P, nb = n/Q )</td>
</tr>
<tr>
<td>6</td>
<td>( P &lt; Q, \ mb = m/(2P), k_B = k/(2Q), k_B = k/(2P), nb = n/(2Q) )</td>
</tr>
<tr>
<td>7</td>
<td>( P = 1, Q = m/poc, \ mb = m, k_B = k, nb = k, nb = 1 )</td>
</tr>
<tr>
<td>8</td>
<td>( P = m/poc, Q = 1, \ mb = 1, k_B = k, k_B = 1, nb = n )</td>
</tr>
<tr>
<td>9</td>
<td>( P = m/poc, Q = 1, \ mb = m/(2P), k_B = k, k_B = k/(2P), nb = n/(2Q) )</td>
</tr>
<tr>
<td>10</td>
<td>( P = m/poc, Q = 1, \ mb = m/(2P), k_B = k, k_B = k/(2P), nb = n )</td>
</tr>
<tr>
<td>11</td>
<td>( P = 1, Q = m/poc, \ mb = m, k_B = k/Q, k_B = k, nb = n/Q )</td>
</tr>
<tr>
<td>12</td>
<td>( P = m/poc, Q = 1, \ mb = m/P, k_B = k, k_B = k/P, nb = n )</td>
</tr>
</tbody>
</table>

Selected ScaLAPACK Variants

ScaLAPACK uses the Parallel BLAS (PBLAS) [33] matrix multiplication subroutine `pdgemm` which implements the parallel matrix multiplication algorithm. Before calling a ScaLAPACK routine, four basic steps have to be carried out by the user of this library:

1. Initialize the process grid,
2. Distribute the matrices on the process grid,
3. Call that ScaLAPACK routine, and finally
4. Release the process grid.

It’s the user’s responsibility to distribute the data to the processes. Basic Linear Algebra Communication Subprograms (BLACS) provides the subroutine `Cp_gemm2d` for a generic 2D-Block-Cyclic distribution scheme. Because the number of processes and the problem size are not known until runtime, the best configuration for initializing the process grid and distributing the data cannot be determined before runtime.

Table 4.2 shows different configurations for ScaLAPACK where each configuration represents a variant. We assume that the number of processes is initialized into a \((P \times Q)\) Cartesian grid using the `Chlacs_gridinit` subroutine of BLACS. The block sizes \( mb, k_B \) are used by the 2D-Block-Cyclic distribution scheme for matrix \( A \), while \( k_B, nb \) are the block sizes used with the matrix \( B \) distribution. The block sizes for matrix \( C \) are \( mb \) and \( nb \).
4.2.3 The Dispatcher and The Dispatch Table

The Dispatch Table

The dispatch table is a multi-dimensional array where each element consists of a data structure type as in Listing 4.5. \texttt{var} is a pointer to the expected best variant object. While the \texttt{time} variable represents the average time spent in executing that variant given a certain context specified from the entry's index in the table.

\begin{verbatim}
Listing 4.5: The Dispatch Table Element.
struct Dispatcher_element {
    MatMul *var;
    double time;
};
\end{verbatim}

The Dispatcher

In this approach the different variants are executed offline at different sample points to get the actual measurements on a particular platform. Then, those measurements are saved to a file where it contains the variants' id. After adding the variants to the dispatcher, then the dispatcher builds the dispatch table with pointers to the variants by reading the saved file (see Listing 4.6). Once the dispatch table is constructed, the application can consult the \texttt{dispatcher} for the expected best variant given a certain call context by calling the dispatch method. For example, if the application wants to compute: \texttt{multiply(A,B,C)}; then a call to dispatcher is shown in Listing 4.6 (C++ code).

\begin{verbatim}
Listing 4.6: Using the dispatcher.
Dispatcher_MatMul dispatcher_mm;
SemiPerformanceAwareMatMul *serial, ... , *scalapack12;
dispatcher_mm.add(serial);
... 
dispatcher_mm.add(scalapack12);
dispatcher_mm.build_disp_table("disp_table_MM.data");
... 
(dispatcher_mm.dispatch(A,B,C)).var->multiply(A,B,C);
\end{verbatim}

Given \(A, B,\) and \(C\) matrices, the dispatcher first looks up the dispatch table for the expected best variant. If the required element is not present in the table, then it uses \textit{multivariate linear interpolation} on the \texttt{time} variable.
4.3 Optimized Composition of Performance-aware Parallel Components

If a user has knowledge about internal implementation details of a variant and the target platform, it is possible to actually write a time estimation function for that variant. In this case, the dispatcher can refer to the time estimation of each variant to find the expected best variant instead of sampling the whole variant execution. In this section, we address the implemented special distribution schemes, the implemented performance-aware variants, and how to estimate the computation and communication times.

4.3.1 Implemented Data Distribution Schemes

In section 4.2.1, we described the ScalAPACK data distribution schemes including the generic ScalAPACK 2D-Block-Cyclic scheme and its special distribution schemes. In these distribution schemes, some schemes (e.g., Row-Block and Col-Block) suffer from serial bottleneck where the remaining data is assigned to one process. For example, consider a $A(11 \times 11)$ matrix where it is required to be distributed as Row-Block. If the number of processes is equal to 4, then $\lfloor 11/4 \rfloor = 2$. In this case, each process have 2 data elements and the remaining data is 3 data elements. If this remaining data is given to one process, it results in one process having 5 data elements. In this section, we present data distribution schemes which avoid the serial bottleneck and hence provide well balanced data distribution.

---

3For simplicity we haven’t considered the current distribution of A and B matrices as context properties. This is left as future work.
Listing 4.7: The Distribution Data Structures.

```cpp
class Distributor {
  public:
    void change_distribution(Matrix *source, Matrix *destination);
    double time(Matrix *source, Matrix *destination);
};
```

Listing 4.7 shows the distribution data structures, where the different distribution schemes are explained in the following subsections. The `Distribution_scheme` data structure specifies how the matrix is distributed among the grid of processors. Not all schemes are implemented yet. Figure 4.2 highlights the implemented distribution schemes with dark boxes, and redistributions by solid lines. The light boxes and dashed lines refer to parts which are left for future work. Users can also provide more redistribution implementations, for example, from `Row-Block` to `Col-Block`.

The `change_distribution` member function of a `distributor` object can be used to exchange data from one distribution scheme into another for a given input matrix. The `time` member function is used to get an estimation of the redistribution time. Only the implemented redistributions can be used (see Figure 4.2). For example, if a variant requires to change the distribution of a matrix from `Row-Block` to `Col-Block` then the distributor will change that matrix first to `Mono` and then to the required distribution. The estimated time is the time taken in redistributing to `Mono` plus the time taken to redistribute to the required distribution scheme.

**Mono Distribution Scheme**

The entire matrix is located on the root (assumed to be rank 0).
4.3. **OPTIMIZED COMPOSITION: PERFORMANCE-AWARE**

![Diagram of distribution schemes]

**Replicated Distribution Scheme**

All processes have a copy of the matrix.

**Row-Block-Wise Distribution Scheme**

A matrix with dimension \((m \times k)\) is divided into blocks where blocks of \(\lceil \frac{m}{P} \rceil\) rows and \(k\) columns are assigned to the processes ranked from 0 until \(m\%P - 1\) and blocks of \(\lfloor \frac{m}{P} \rfloor\) rows and \(k\) columns are assigned to the processes from \(m\%P\) until \(P - 1\). This type of distribution does not exhibit the mentioned **serial bottleneck**, because there will be no remaining data instead processes ranked from 0 to \(m\%P - 1\) have only one element more than the processes ranked from \(m\%P\) until \(P - 1\). For example, if we take the mentioned example at the beginning of this section, we have two blocks \([11/4] = 3\) and \([11/4] = 2\). Processes ranked from 0 to \(11\%4 - 1 = 2\) take 3 elements while the last process (ranked 3) takes 2 elements. Recall that the previous example gives 2 elements for 3 processes and one process has 5 elements.

**Column-Block-Wise Distribution Scheme**

In this distribution scheme, a matrix with dimension \((m \times k)\) is divided into blocks where blocks of \(\lfloor \frac{k}{P} \rfloor\) columns and \(m\) rows are assigned to the processes ranked from 0 until \(k\%P - 1\) and blocks of \(\lfloor \frac{k}{P} \rfloor\) columns and \(m\) rows are assigned to the processes from \(k\%P\) until \(P - 1\). This type of distribution also does not suffer from the **serial bottleneck**.
4.3.2 Implemented Variants

All variants take matrices $A$, $B$, and $C$ as input and compute the operation $C = A \times B$, where these matrices are distributed in one of the implemented schemes.

Serial Variant

This variant is same as what was described in Section 4.2.2. The only difference is that, for this approach, it is inherited from the `PerformanceAwareMatMul` class. The pseudocode of this variant resembles the one in the previous approach, however, the redistribution of $A$ and $B$ matrices to the required distribution schemes is included.

ReplicatedA and ReplicatedB Variants

These variants are same as those introduced in Section 4.2.2. Again, the difference in this case is that these variants are inherited from the `PerformanceAwareMatMul` class. Also, these variants implement the redistribution (if required) of $A$ and $B$ matrices.

Row-Col-Col (RCC) Variant

This variant requires that the $A$ matrix is distributed `Row-Block` wise among the working group (the group of processors that are selected to compute the matrix multiplication). The Matrix $B$ is distributed `Column-Block` wise and the resultant matrix $C$ is distributed `Column-Block` (i.e., each process is responsible for computing one `Column-Block` of the $C$ matrix). To achieve this, each process broadcasts its local block of matrix $A$ and receives the blocks from all other processes. The pseudo-code of this algorithm is depicted in Listing 4.8.
4.3. OPTIMIZED COMPOSITION: PERFORMANCE-AWARE

Listing 4.8: Row-Col-Col (RCC) Variant pseudo-code.

```c
for (p = 0; p < nprocs; ++p) {
    if (my_rank == p) {
        broadcast(myAblock, p);
        // computing the local part of C.
        multiply(myAblock, myBblock, myCblock);
    } else {
        recv(temp, p);
        multiply(temp, myBblock, myCblock);
    }
}
```

Row-Col-Row (RCR) Variant

Matrix $A$ is distributed Row-Block matrix $B$ Column-Block and each process computes one Row-Block of the resultant matrix $C$. This variant is the same as the previous one except that in this case, each process broadcasts its $B$ local sub-matrix block in order to compute one row block of matrix $C$. The pseudo-code of the algorithm is given in Listing 4.9.

Listing 4.9: Row-Col-Row (RCR) Variant pseudo-code.

```c
for (p = 0; p < nprocs; ++p) {
    if (my_rank == p) {
        broadcast(myBblock, p);
        multiply(myAblock, myBblock, myCblock);
    } else {
        recv(temp, p);
        multiply(myAblock, temp, myCblock);
    }
}
```

Col-Row-Mono(CRM) Variant

If matrix $A$ is $(m \times k)$ and distributed using the Column-Block scheme, matrix $B$ is $(k \times n)$ and distributed using the Row-Block scheme, then each process computes a partial sum of the $(m \times n)$ $C$ matrix. The entire matrix is calculated by using MPI Reduce with the MPI_SUM operation. The pseudo-code of the algorithm is shown in Listing 4.10.

Listing 4.10: Col-Row-Mono(CRM) Variant pseudo-code.

```c
multiply(myAblock, myBblock, myCblock);
MPI_Reduce(C->data, m*n, MPI_DOUBLE, MPI_SUM, 0, mpi_communicator);
```
4.3.3 Estimating the Computation Time

As each working process will compute at least one or more elements of matrix $C$, we can base our computation time estimation on the (average) time spent in calculating one element of matrix $C$. This average time value is referred to as $\alpha$. The total computation time is the total number of $C$ elements multiplied by $\alpha$. The main factor that affects this value is the value of $k$ which is the number of $A$ columns and the number of $B$ rows. Thus, $\alpha$ is considered to be a function of $k$. In this approach, a small table at deployment time is built by executing the serial component multiple times with different values of $k$. This table can be looked up using a $\text{time\_alpha}$ function.

We do not measure the time of each individual floating point operation, but we take the calculation of $\alpha$. In this way, we get a representative value of the time for the overall computation that includes possibly the cache effects as well. As measuring the time for an individual floating point operation may sometimes involve cache effects, taking average time for a certain number of operations provides better approximation.

4.3.4 Estimating the Communication Time

Multiple models exist for estimating the point-to-point communication time. Such models are $\text{LogGP}$ [1], $\text{PLogP}$ [26], and $\text{Hockney}$ [24]. The user can use one of these models to estimate the communication time. However, in this work, we consider actual measurements and apply the same approach for point-to-point communication as we did in estimating the computation time.

Although collective communications are based on multiple point-to-point communications, there exists no specific model which can be used to accurately estimate the communication time. One of the reasons for this is that collective communications can use multiple algorithms and there is no optimal algorithm used for a particular collective communication for a broad range of message sizes and numbers of processes and network topologies. One solution is to use actual measurements for every collective communication sub-routine used. In this approach, a 2-dimensional table is built at deployment time, containing the message size and the number of processes, for each type of collective communication. The variant-writer can refer to these tables when writing the $\text{time}$ function of a variant by calling the $\text{time\_x(m, p)}$ function (where $x$ refers to the name of the collective communication). Table 4.3 shows the MPI communication routines considered and their corresponding $\text{time\_x}$ names.
Table 4.3: Estimated MPI Communication Routines

<table>
<thead>
<tr>
<th>MPI Communication</th>
<th>time_x name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Bcast</td>
<td>time_mpi_bcast</td>
</tr>
<tr>
<td>MPI_Scatterv</td>
<td>time_mpi_scatterv</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>time_mpi_gatherv</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>time_mpi_reduce</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>time_mpi_send</td>
</tr>
</tbody>
</table>

4.3.5 Overall Time Estimation

For any implementation variant, the overall execution time (given by Equation 4.3) includes the time taken for distributing the matrices $A$ and $B$, denoted as $\text{time}_{\text{distribute},A}$ and $\text{time}_{\text{distribute},B}$ respectively, as well as the time taken for executing its parallel algorithm, denoted as $\text{time}_{\text{algorithm}}$ (i.e., $\text{time}_{\text{computation}} + \text{time}_{\text{communication}}$).

$$\text{time}_{\text{variant}} = \text{time}_{\text{distribute},A} + \text{time}_{\text{distribute},B} + \text{time}_{\text{algorithm}} \quad (4.3)$$

The time function on the right side of Equation 4.3 takes matrix $A$ and matrix $B$ as arguments and calculates the corresponding times with the help of tables generated at deployment time. The accuracy of the time function depends on how accurate the variant designer writes the time function given the right side of Equation 4.3, as well as the accuracy of computation and communication time tables (e.g., sample points in the measurement tables, interpolation technique used).

For the distribution time, the user can refer to the $\text{distributor.time}$ function which takes as arguments the source and the destination matrices and returns the estimated time. This function uses the communication primitives tables generated at the deployment time. For an example, we consider the derivation of the time function for the Replicated $B$ variant. Other derivations are very similar to the approach discussed below. In this variant, the Row-Block distribution of matrix $A$ is required, so if matrix $A$ is in Mono distribution scheme then $\text{time}_{\text{distribute},A}$ will give the time taken for redistributing matrix $A$ from Mono to Row-Block distribution, given by Equation 4.4. Similarly for matrix $B$, the Replicated distribution is required, so if matrix $B$ is in Mono distribution scheme then $\text{time}_{\text{distribute},B}$ is given by Equation 4.5.

$$\text{time}_{\text{distribute},A} = \text{time}_\text{mpi_scatterv}(m \times k, P) \quad (4.4)$$
\begin{equation}
time_{\text{distribute}, B} = \text{time\_mpi\_bcast}(k \times n, P)
\end{equation}

Because all computations are carried out in parallel, the time\_algorithm function is considered as the computation time spent by the first process as shown in Equation 4.6 (note that time\_communication = 0 and first process computes $\left\lfloor \frac{m}{P} \right\rfloor \times k$ elements of the C matrix):

\begin{equation}
time_{\text{algorithm}} = \left\lfloor \frac{m}{P} \right\rfloor \times k \times \text{time\_alpha}(k)
\end{equation}

By substituting Equations 4.4, 4.5 and 4.6 into Equation 4.3, we get the overall estimated time for executing the variant. In Equation 4.3, we haven’t considered the time required for synthesizing the result matrix $C$ on the root MPI process. This choice is justified by the fact that normally matrix multiplication is used as a kernel in a large application and keeping data distribution intact may benefit when doing further computations on that data.

### 4.3.6 Dispatch Function

In contrast to the dispatcher discussed in section 4.2.3, it executes all variants’ time functions given a certain sample point instead of looking up a table. The variant with the lowest expected execution time is called. When the number of variants are large (i.e., hundreds of variants), the dispatcher exhibits runtime overhead compared to the previously mentioned ones (which has a table lookup overhead). However, this overhead is negligible compared to the algorithmic complexity of the variants used.
Chapter 5

Experimental Setup and Evaluation

In this chapter we evaluate both approaches with some experiments. In the first section, we evaluate the optimized composition of semi-performance-aware variants while the second approach is evaluated in the next section. A later chapter provides discussion and conclusions about the two approaches.

5.1 Optimized Composition of Semi-Performance-aware Parallel Components

5.1.1 Experimental Setup

We have considered the Serial, ReplicatedA, and ReplicatedB variants of section 4.2.2 as well as the 12 different configurations of ScaLAPACK (shown in table 4.2). The memory consumptions per node for ReplicatedA and ReplicatedB variants are bounded to be 1GB which is calculated by Equation 4.1 and Equation 4.2, respectively. Therefore, both variants are not executed if this limit is exceeded and their profile executions are recorded as INT_MAX when the variants are executed. We have applied such memory constraint on these variants to:

- demonstrate the flexibility of our approach as it can also work in cases where some variants have applicability constraints, i.e., some variants work only for certain execution contexts (or under certain execution requirements fulfilled).
32  CHAPTER 5. EXPERIMENTAL SETUP AND EVALUATION

Table 5.1: The Dispatch Table Dimensions (The Range of Values for \( m, k, n, \) and \( p \))

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m, k, n )</td>
<td>5, 10, 100, 500, 1000, 2000, 3000, 5000</td>
</tr>
<tr>
<td>( p )</td>
<td>2, 4, 8, 16, 32, 64</td>
</tr>
</tbody>
</table>

Table 5.2: The Set of Partial Evaluation Points (i.e., only \( m, k, \) and \( n \); \( p \) is not included)

<table>
<thead>
<tr>
<th>( m \cdot k \cdot n )</th>
<th>Sample id</th>
<th>( m )</th>
<th>( k )</th>
<th>( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>216 ( \cdot 10^9 )</td>
<td>1</td>
<td>6000</td>
<td>6000</td>
<td>6000</td>
</tr>
<tr>
<td>57.6 ( \cdot 10^9 )</td>
<td>2</td>
<td>1600</td>
<td>6000</td>
<td>6000</td>
</tr>
<tr>
<td>57.6 ( \cdot 10^9 )</td>
<td>3</td>
<td>6000</td>
<td>1000</td>
<td>6000</td>
</tr>
<tr>
<td>57.6 ( \cdot 10^9 )</td>
<td>4</td>
<td>6000</td>
<td>6000</td>
<td>1000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>25.6 ( \cdot 10^9 )</td>
<td>25</td>
<td>400</td>
<td>1000</td>
<td>400</td>
</tr>
<tr>
<td>25.6 ( \cdot 10^9 )</td>
<td>26</td>
<td>1600</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>64 ( \cdot 10^9 )</td>
<td>27</td>
<td>400</td>
<td>400</td>
<td>400</td>
</tr>
</tbody>
</table>

• to have more selection opportunities (or performance variation) between the selected variants.

In order to simplify building and looking up the dispatch table, we assume that we have an application where input matrices \( A(m \times k) \), \( B(k \times n) \), and the resultant matrix \( C(m \times n) \) are in Mono distribution scheme. Table 5.1 shows the ranges of the sizes of the input matrices and the number of processes (i.e., values of \( m, k, n, \) and \( p \)). Therefore, the dispatch table built offline is a 4-dimensional table and its size is 36KB (\( 8^3 \times 6 \times \) dispatch_table_element_size bytes, where the dispatch element size is 12 bytes).

We performed experiments on a machine with 805 nodes, each having two 2.33 GHz Xeon quad-core processors and 16 GB RAM; running under CentOS5 and OpenMPI implementation of the Message Passing Interface. The nodes are interconnected by Infiniband.

In the following experiments, we consider a set of evaluation points where \( m, k, \) and \( n \) are generated as follows: we define \( L = 6000, M = 1600, S = 400 \) and then \( m, k, \) and \( n \) is assigned iteratively to one subset of the following set, respectively:

\[
\{\{L,L,L\}, \{L,L,M\}, \{L,L,S\}, \{M,L,L\}, \ldots, \{S,S,M\}, \{S,S,S\}\}
\]

Table 5.2 demonstrates some of these evaluation points.
5.1. Experimental Evaluation

Figures 5.1 and 5.2 show a first comparison of running Serial, ReplicatedA, ReplicatedB, ScALAPACK variants and the composed solution using the dispatcher. The x-axis represents the evaluation points \((m, k, n, \text{ and } p)\) where values of \(m, k,\) and \(n\) are given in Table 5.2. We can observe that dispatcher most often succeeds in selecting the best variant or the one that is very close to the best variant. The figures also represent the average accuracy\(^1\) of the dispatcher solution with respect to the actual best variant at each evaluation point. It is important to note that the serial variant is not included in the comparison shown in Figures 5.1(a) and 5.2(a) because of its high execution time. In addition, ReplicatedA and ReplicatedB are not executed at all evaluation points due to the applicability (memory) constraint discussed earlier. Table 5.3 shows the sum (in seconds) of the execution times for each variant and the dispatcher solution where boldface numbers represent the variant that has the lowest execution times sum. It shows also the speedup gained when using the dispatcher solution over the fastest variant. We notice that the speedup decreases when the number of processes increases. This is due to the following reasons:

- ReplicatedA and ReplicatedB variants become less competitive to ScALAPACK variants when the number of processes is large.
- Only a few ScALAPACK variants are optimal at most of the evaluation points considered in the experiments.

In this case, we have a few variants where the variation in performance is small when the number of processes is large. Therefore, the speedup gain of the dispatcher decreases when the number of processes increases.

It is important to note that the table consists of two sub-tables where:

- the first sub-table shows the results when all evaluation points are considered.
- the second sub-table shows the results when all variants are executed (i.e., when \(m, k,\) and \(n\) values of the evaluation points are small values).

Next, all variants and dispatcher are executed multiple times for different sample points (where different values of \(p\), values of \(m, k,\) and \(n\) are generated randomly). Table 5.4 summarizes the results of dispatcher compared with Serial, ReplicatedA, ReplicatedB, and ScALAPACK variants. It is important to note that a few experiments do not compare some ScALAPACK variants

\(^1\)The accuracy is computed by dividing the execution time of the actual best variant, with the execution time of the variant selected by the dispatcher.
Figure 5.1: Dispatcher vs Serial, ReplicatedA, ReplicatedB and ScALAPACK variants when $p=16$ (Average accuracy is 96.1%)
5.1. OPTIMIZED COMPOSITION: SEMI-PERFORMANCE-AWARE

![Graph 1](image1.png)

(a)

![Graph 2](image2.png)

(b)

Figure 5.2: Dispatcher vs Serial, ReplicatedA, ReplicatedB and ScaLA-PACK variants when $p=32$ (Average accuracy is 95.7%)
CHAPTER 5. EXPERIMENTAL SETUP AND EVALUATION

Table 5.3: The execution times sum (in seconds) for Dispatcher, Parallel and Serial variants using the evaluation set given in Table 5.2.

(a) Considering all evaluation points

| p | Serial | Rep A | Rep B | v1 | v2 | v3 | v4 | v5 | v6 | v7 | v8 | v9 | v10 | v11 | v12 | Disp. | speedup |
|---|-------|-------|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-------|--------|
| 32 | 8.1   | 4.1   | 3.16  | 3.2 | 3.6 | 4.2 | 3.6 | 4.5 | 3.5 | 3.6 | 4.5 | 3.6 | 4.5 | 3.6 | 4.5   | 3.6    | 1.38   |

(b) Considering evaluation points when all variants are executed

| p | Serial | RepA | RepB | v1 | v2 | v3 | v4 | v5 | v6 | v7 | v8 | v9 | v10 | v11 | v12 | Disp. | speedup |
|---|-------|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-------|--------|
| 32 | 8.1   | 4.1  | 3.16 | 3.2 | 3.6 | 4.2 | 3.6 | 4.5 | 3.5 | 3.6 | 4.5 | 3.6 | 4.5 | 3.6 | 4.5   | 3.6    | 1.38   |

because their randomly generated values of m, k, and n are less than the dimension of the process grid. The table shows two types of experiments. The first type occurs when the problem size is bounded with a certain value as in sub-tables 5.4(a) to 5.4(c). The second one occurs when considering a relatively large range of the problem sizes as in sub-table 5.4(d). As we can see from the first three sub-tables, the dispatcher has, at some experiments, an execution times sum that is slightly higher than one or two of the variants. For example, as shown in 5.4(a), the serial variant is a good choice most of the time when m, k, and n are less than 500. The same observation is also valid in the next two sub-tables. This phenomenon is due to the interpolation technique used and the size of the dispatch table built. In addition, it is due to the fact that when using a limited problem size range, there are few performance variations between variants and hence one variant can dominate the performance metric. Finally, we can see that the ReplicatedA and ReplicatedB variants have higher execution times sum when the number of processes is increased due to increase in communication cost.

The last set of experiments is when m, k, and n are less than 5000. In this case, a wide range of problem sizes are considered. It is important to note that serial, ReplicatedA, and ReplicatedB variants are omitted from the comparison for the same reason mentioned earlier. Table 5.4(d) shows promising results where the dispatcher has very low total execution times sum compared to the fastest ScALAPACK variants. One of the reasons is that ScALAPACK does not consider serial, ReplicatedA, or ReplicatedB variants which can perform better at certain sample points. Because of the wide range in problem sizes, the variation in performance between variants becomes prominent and the dispatcher detects and exploits this phenomenon. The table shows also the speedup gained when using dispatcher over using the variant that has the lowest total sum. Again, this speedup decreases when the number of processes is increased.

We believe that ScALAPACK conducts its own optimization to select the best algorithm given certain problem size and process grid. This belief is
5.2 Optimized Composition of Performance-aware Parallel Components

5.2.1 Experimental Setup

We have considered the variants described in Section 4.3.2. The variants implemented are executed at different sample points to compare the actual measurements with the estimated values. The variants are executed using different values of \( p \). Table 5.2 demonstrates the values of \( m, k, \) and \( n \) used.

Table 5.4: Sum of the execution times (in seconds) over different random evaluation points, for the Dispatcher, Parallel and Serial variants after running them multiple times (where \( m, k, \) and \( n \) are generated randomly)

\[ m \leq 500, k \leq 500, \text{ and } n \leq 500 (m \cdot k \cdot n \leq 125 \cdot 10^6); \text{ the number of randomly generated values for } m, k, \text{ and } n \text{ is } 100 \]

\[ m \leq 2000, k \leq 2000, \text{ and } n \leq 2000 (m \cdot k \cdot n \leq 8 \cdot 10^6); \text{ the number of randomly generated values for } m, k, \text{ and } n \text{ is } 100 \]

\[ 2000 \leq m \leq 5000, 2000 \leq k \leq 5000, \text{ and } 2000 \leq n \leq 5000; \text{ the number of randomly generated values for } m, k, \text{ and } n \text{ is } 60 \]

\[ m \leq 5000, k \leq 5000, \text{ and } n \leq 5000; \text{ the number of randomly generated values for } m, k, \text{ and } n \text{ is } 70 \]
Experiments test the dispatcher with respect to the variants used. The experiments are executed on the same machine described in section 5.1.1.

5.2.2 Experimental Evaluation

Comparison of estimated and measured execution times

The results for comparison of estimated and measured times are shown in Figures 5.3 and 5.4. In each graph, the horizontal axis shows the evaluation points (Table 5.2) and the vertical axis shows the time in seconds. The estimated and measured time graphs are labeled at each graph. We observe that for all variants, the computed time is close to the measured time. The graphs show the accuracy of the estimation for each evaluation point as well as the average accuracy. The serial variant has the highest accuracy compared to all other variants because of the absence of the communication routines and hence the estimation consists of predicting only the computation time. The ReplicatedA variant has the lowest accuracy. The estimation accuracy is because of the approach used, as well as due to the memory allocation/deallocation and sub-matrices operations (like setting and/or getting them to/from temporary buffers) in the computation algorithms of the variant. It is important to note that in this comparison, the memory constraint on ReplicatedA and ReplicatedB variants are lifted.
Figure 5.3: Estimated vs Actual measurements of the variants
CHAPTER 5. EXPERIMENTAL SETUP AND EVALUATION

Figure 5.4: Estimated vs Actual measurements of the variants cont.

Testing the Dispatcher

We have executed the variants and the dispatcher on the same evaluation points used (as in Table 5.2 when \( p = 12 \) and \( p = 48 \)). Figure 5.5 shows the results and average accuracy of the dispatcher with respect to the parallel variants. It is obvious from the figure to see that, at most of the eval-
5.2. **OPTIMIZED COMPOSITION: PERFORMANCE-AWARE**

At evaluation points, the dispatcher has the lowest execution time. This means that the dispatcher often succeeds in selecting the expected best variant and has never selected the worst variant at a particular evaluation point. It is important to note that variations in performance between the different variants become more visible when the number of processes increases. Table 5.5 summarizes the benefit of using the dispatcher over any variant and highlights the speedup obtained. The table shows the execution times sum of the variants in seconds after executing them on all evaluation points. We can observe that the dispatcher has the lowest sum compared to the fastest variant which is ReplicatedB.

The last experiment executes the dispatcher and the variants with random values of $m$, $k$, and $n$ and with different values of $p$. In this experiment, we apply the 1GB memory constraint on ReplicatedA and ReplicatedB variants. The final result of this experiment is summarized in Table 5.6. We observe a reduction in time when using the dispatcher over using any single variant. Also, it is obvious to see that most of the time ReplicatedB is selected. This is due to the existence of few variations between the variants. However, the dispatcher is able to detect and exploit such variations and hence performs better overall than the fastest variant.
CHAPTER 5. EXPERIMENTAL SETUP AND EVALUATION

Dispatcher vs Parallel Variants (p=12)

(a) when p=12 (Average Accuracy 97.2%)

Dispatcher vs Parallel Variants (p=48)

(b) when p=48 (Average Accuracy 97.8%)

Figure 5.5: Dispatcher vs Parallel variants
5.2. **OPTIMIZED COMPOSITION: PERFORMANCE-AWARE**

Table 5.5: Execution time sum over all evaluation points of the Dispatcher, Parallel and Serial variants (in seconds) after running them using the evaluation set, given in Table 5.2.

<table>
<thead>
<tr>
<th>P</th>
<th>Dispatcher</th>
<th>RepA</th>
<th>RepB</th>
<th>RCC</th>
<th>RCR</th>
<th>CRM</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>18.3</td>
<td>25.1</td>
<td>21.0</td>
<td>24.2</td>
<td>24.2</td>
<td>28.4</td>
<td>1.15</td>
</tr>
<tr>
<td>48</td>
<td>11.4</td>
<td>19.9</td>
<td>14.3</td>
<td>25.2</td>
<td>23.9</td>
<td>22.9</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Table 5.6: Execution time sum of the Dispatcher and other variants (in seconds) after running them multiple times (where $m$, $k$, and $n$ are generated randomly).

(a) $(m \ast k \ast n <= 125 \ast 10^6)$; the number of randomly generated values for $m$, $k$, and $n$ is 700

<table>
<thead>
<tr>
<th>P</th>
<th>Dispatcher</th>
<th>Serial</th>
<th>ReplicatedA</th>
<th>ReplicatedB</th>
<th>RCC</th>
<th>RCR</th>
<th>CRM</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.91</td>
<td>2.76</td>
<td>2.71</td>
<td>2.38</td>
<td>2.36</td>
<td>2.45</td>
<td>2.38</td>
<td>1.16</td>
</tr>
<tr>
<td>12</td>
<td>2.10</td>
<td>2.87</td>
<td>2.54</td>
<td>2.79</td>
<td>2.61</td>
<td>3.18</td>
<td>3.47</td>
<td>1.21</td>
</tr>
<tr>
<td>48</td>
<td>2.68</td>
<td>3.51</td>
<td>3.16</td>
<td>3.43</td>
<td>4.84</td>
<td>4.83</td>
<td>5.38</td>
<td>1.18</td>
</tr>
</tbody>
</table>

(b) $(125 \ast 10^6 <= m \ast k \ast n <= 8 \ast 10^9)$; number of randomly generated values for $m$, $k$, and $n$ is 300

<table>
<thead>
<tr>
<th>P</th>
<th>Dispatcher</th>
<th>Serial</th>
<th>ReplicatedA</th>
<th>ReplicatedB</th>
<th>RCC</th>
<th>RCR</th>
<th>CRM</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>59.86</td>
<td>149.95</td>
<td>62.85</td>
<td>61.27</td>
<td>66.23</td>
<td>66.23</td>
<td>67.32</td>
<td>1.02</td>
</tr>
<tr>
<td>12</td>
<td>36.54</td>
<td>139.85</td>
<td>41.74</td>
<td>38.40</td>
<td>46.60</td>
<td>45.48</td>
<td>50.99</td>
<td>1.05</td>
</tr>
<tr>
<td>48</td>
<td>10.35</td>
<td>43.22</td>
<td>12.51</td>
<td>11.66</td>
<td>16.51</td>
<td>12.75</td>
<td>17.56</td>
<td>1.07</td>
</tr>
</tbody>
</table>

(c) $(8 \ast 10^9 <= m \ast k \ast n <= 216 \ast 10^9)$; number of randomly generated values for $m$, $k$, and $n$ is 30

<table>
<thead>
<tr>
<th>P</th>
<th>Dispatcher</th>
<th>Serial</th>
<th>ReplicatedA</th>
<th>ReplicatedB</th>
<th>RCC</th>
<th>RCR</th>
<th>CRM</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>56.8</td>
<td>*</td>
<td>*</td>
<td>90.7</td>
<td>70.5</td>
<td>90.5</td>
<td>1.04</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>48.8</td>
<td>*</td>
<td>*</td>
<td>90.7</td>
<td>91.1</td>
<td>66.6</td>
<td>1.34</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6

Discussion and Conclusions

As seen from the experiments, it is not always optimal to execute matrix multiplication in parallel. In many cases, serial execution is the best choice especially when the number of floating point operations is small (i.e. the values of $m \times k \times n$ is small). Such a choice cannot be determined at development (or compile) time unless the problem size is fixed statically. One faces the same difficulty when dealing with choice provided by multiple parallel algorithms and different data distribution schemes. Our two approaches provide systematic ways to expect the best choice at runtime.

In the first approach, we built dispatch tables by empirical executions at sample points where many ScaLAPACK variants are used. ScaLAPACK offloads developers from selecting the parallel algorithm but requires them to choose the appropriate data distribution scheme as well as the initialization of the process grid. Therefore, many different configurations (variants\(^1\)) can be specified. For this reason, we used 12 different ScaLAPACK configurations (variants) with distinct data distribution and process grid. In experiments, we have shown how incorporating serial, ReplicatedA and ReplicatedB variants besides ScaLAPACK variants, positively affects the performance. In other words, these variants perform better than the ScaLAPACK variants at many sample points which results in a dispatcher that has a good speedup gain over the fastest ScaLAPACK variants.

In the second approach, we implement our own parallel variants to investigate the time estimation of communications and computations costs. The results demonstrate the benefit of using the dispatcher instead of static selection of one variant even though some of the variants do not have highly accurate time estimations.

\(^{1}\)we consider each configuration a potential variant.
Both approaches potentially allow usage of many variants, and using more variants might yield better results. The first approach requires a dispatch table to be built. It can be built offline or at deployment time and this building process consumes much time even when building small tables. However, building small dispatch tables and using basic extra/interpolation techniques can yield good results for structured computations such as matrix multiplication. Our first approach allows using variants whose execution time estimation cannot be determined (such as third party libraries where the internal implementations are not visible). Our second approach can be applied only if developers are able to write execution time estimation functions for variants.

There are no specific machine models for estimating MPI collective communication routines. Our technique in the second approach is by building primitive communication time tables offline or at deployment time. The primitive communication time tables store measured time data for each basic communication pattern. Then, these tables are consulted at runtime for the time estimation given a certain call context. In this way, we have a good and acceptable accuracy where the dispatcher is able to expect the best variant. In addition, building small tables for computation-specific primitives (such as the average computation of one element of the resultant matrix C) instead of considering the time of the individual floating point operation for estimating the computation time, gives a good and acceptable accuracy as well.
Chapter 7
Related Work

In [5][4], a framework for performance-aware composition of explicitly parallel components, based on a dispatch-table mapping of a call to an expected best implementation variant and schedule, is presented. The approach relies upon statically generated dispatch and schedule tables that can be used during the runtime to look up the best implementation variant and schedule for a given call context (problem and processor group size). The dispatch table is built using an interleaved dynamic programming method at the deployment time. The framework is applied for sorting algorithms on a shared-memory system. In our case, we apply our framework on a different application (matrix multiplication algorithms) on a distributed memory system, considering different distribution schemes. Also, the dispatch table is generated by probing profile interface of variants at the deployment time or offline.

Another mechanism for portable data layout and algorithm selections via automatic use of statistical modelling is given in [11]. For each variant, a dynamically generated model based on profiling information is used to predict the performance. In [11], two high level libraries are built that use automatically calibrated statistical models to select the best implementation and respective optimized parameters that leads to performance portability for different platforms and workloads. This statistical model approach requires the library designer to provide a list of terms from which the statistical models are computed. However, in our framework, the designer has to wrap each variant with a profile interface where a dispatch table is generated afterwards, or all variants are wrapped with a performance-aware interface.

[21] presents formal definitions and specifications of the concepts in the context of composition and optimization. It also addresses the technique of solving the optimized composition problem based on the non-functional
properties of the components. In [13], the authors address the dynamic composition issues by carrying out a profiling approach at the deployment time. However, the optimization approach is for implementation variants and for different data structure representations. At the run-time, the best algorithm variant and the best data representation are selected for a particular call.

In [17], Lau and Rana provide a survey of common software composition approaches. In recent times, the component composition problem for modern architectures gained the interest of many research groups. Several research projects tackle this multi-faceted problem in the context of modern heterogeneous systems, containing multi-core CPUs and one or more programmable accelerators (GPUs, FPGAs etc.).

PEPPHER (PErformance Portability and Programmability for HEterogeneous many core architectures) is an EU FP7 (2010-2012) project that addresses the performance portability and programmability problem for modern heterogeneous architectures. In essence, it relies on information obtained at runtime about data-locality and system workload, as well as performance feedback obtained via historical executions [2] and offline training [18] to make the composition decisions.

PEPPHER does not propose any new programming model but instead supports composition of parallel components written in existing parallel programming models. Our work is different from the PEPPHER project, primarily in its focus on distributed (multi-node) homogeneous MPI platforms whereas PEPPHER mainly targets shared memory single-node heterogeneous systems (i.e. no MPI components).

PetaBricks [14] proposes a custom programming model (with implicit parallelism) and a runtime system. It heavily relies on auto-tuning methods and can optimise for a different optimisation criterion than performance (e.g., numerical accuracy). Merge [19] uses the classical MapReduce pattern [15] to provide a high-level abstraction and support component composition. Elastic computing [16] provides the notion of an elastic function which is the same as a component in our case. The selection for optimal variants (elastic function) for a given invocation is guided by performance profiles computed statically. All these techniques actually propose a new programming model which restricts their applicability to existing code-base without significant code-porting effort.
Chapter 8

Future Work

As a future work, one can develop an approach to consider global optimization of multiple calls (instead of local optimization where only one binding point is considered) to component variants and provide a schedule algorithm to execute independent calls in parallel. In this case, extending the dispatcher functionality to expect the best number of processes given problem size and data distribution schemes becomes essential.

In addition, we need to improve the time estimation for larger problem sizes. Also, one can extend the dispatch table built by considering the different redistribution schemes of the matrices and hence providing more flexibility. Another future extension is to test both approaches for some other parallel components and with several implementation variants in scientific computing with more aspects like stride size, recursive parallel components as well as trying dynamic and non-dedicated environments. In this thesis work, we considered homogeneous nodes where time estimations of computation operations on different dedicated nodes are equal. This limitation of homogeneous and dedicated nodes can be removed in possible future extensions of this work.
Bibliography


[31] BLAS (Basic Linear Algebra Subprograms): http://www.netlib.org/blas/ last access [2012-07-04].

BIBLIOGRAPHY

[33] Parallel Basic Linear Algebra Subprograms (PBLAS):
http://www.netlib.org/scalapack/pblas_ref.html last access [2012-07-04].

[34] Scalable Linear Algebra PACKage (ScALAPACK):
http://www.netlib.org/scalapack/ last access [2012-07-04].
We develop a novel framework for optimized composition of explicitly parallel software components with different implementation variants given the problem size, data distribution scheme and processor group size on a Linux cluster. We consider two approaches (or two cases of the framework).

In the first approach, dispatch tables are built using measurement data obtained offline by executions for some (sample) points in the ranges of the context properties. Inter-/extrapolation is then used to do actual variant-selection for a given execution context at run-time.

In the second approach, a cost function of each component variant is provided by the component writer for variant-selection. These cost functions can internally look up measurements' tables built, either offline or at deployment time, for computation- and communication-specific primitives.

In both approaches, the call to an explicitly parallel software component (with different implementation variants) is made via a dispatcher instead of calling a variant directly.

As a case study, we apply both approaches on a parallel component for matrix multiplication with multiple implementation variants. We implemented our variants using Message Passing Interface (MPI). The results show the reduction in execution time for the optimally composed application compared to applications with hard-coded composition. In addition, the results show the comparison of estimated and measured times for each variant using different data distributions, processor group and problem sizes.
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Anas Al-Trad