UNIVERSITY OF CALIFORNIA, SAN DIEGO

Tarragon: a Programming Model for Latency-Hiding Scientific Computations

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Pietro Cicotti

Committee in charge:

Professor Scott B. Baden, Chair
Professor Michael Holst
Professor Terrence J. Sejnowski
Professor Allan E. Snavely
Professor Dean M. Tullsen

2011

The dissertation of Pietro Cicotti is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2011

DEDICATION

To My Family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help and support of many people, to whom I owe far more gratitude than I can possibly express here.

First I would like to thank my advisor, Professor Scott B. Baden, for his thoughtful guidance and support during the course of my graduate studies.

I gratefully acknowledge the members of my committee for reading this dissertation and for their valuable remarks. I am particularly indebted to Professor Allan Snavely for his support, encouragement, and for giving me the opportunity to join his research group.

I would like to express my gratitude to Xiaoye Sherry Li for her advice and research support during my summers at the Lawrence Berkeley National Laboratory. While at the lab, I also had the pleasure to share my time with two inspiring researchers and good friends: Tony Drummond and Osni Marques.

During these years, it has been my pleasure to work with my labmates and friends Gregory Balls, Jacob Sorensen, Didem Unat, Alden King, and Nhat Tan Nguyen; to them goes my gratitude for making my time in the lab more enjoyable. I also had the opportunity to work with the PMAC lab members, Laura Carrington, Michael Laurenzano, Mitesh Meswani, Cathie Olschanowsky, Mustafa Tikir, Jiahua He, and Sean Strande; to them goes my gratitude for their help and suggestions.

Special thanks to Alden King, Mustafa Tikir, Cathie Olschanowsky, Laura Carrington, and Alexis Lasheras for proofreading parts of this dissertation.

I want to express my gratitude to my friends, especially those in Italy; life wouldn't be so fun without them.

Finally, I want to express my deepest gratitude to my family, for their endless support. In particular I want to dedicate this dissertation to Laura, my wife, and my daughter Anna.

Chapters 4, in part, is currently being prepared for submission for publication. Pietro Cicotti; Scott B. Baden. The dissertation author is the primary investigator and author of this material.

Chapters 5, in part, is currently being prepared for submission for publication. Pietro Cicotti; Scott B. Baden. The dissertation author is the primary investigator and

author of this material.

Chapters 6, in part, is currently being prepared for submission for publication. Pietro Cicotti; Scott B. Baden. The dissertation author is the primary investigator and author of this material.

This research was supported in part by the NSF contracts ACI0326013 and CCF-1017864, in part by a grant from the NSF's Office of Cyberinfrastructure entitled "Integrated Performance Monitor", and in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. It used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

This research was supported in part by the National Science Foundation through TeraGrid resources provided by the National Center for Supercomputing Applications and the National Institute for Computational Science under grants number TG-ASC070021N and TG-CCR080007. The computations were performed on the Intel 64 Cluster Abe at the National Center for Supercomputing Applications and on Kraken at the National Institute for Computational Sciences (http://www.nics.tennessee.edu/).

VITA

2003	Laurea in Informatica *cum laude*, Università di Bologna, Italy
2008	M.S. in Computer Science, University of California, San Diego
2011	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

Pietro Cicotti and Scott B. Baden, "Tarragon: a Programming Model for Latency Hiding Scientific Applications", *In perparation*

Mitesh Meswani and Pietro Cicotti and Jiahua He and Allan Snavely, "Predicting Disk I/O Time of HPC Applications on Flash Drives", *IEEE Globecom 2010, Workshop on Application of Communication Theory to Emerging Memory Technologies*, December, 2010

Pietro Cicotti and Xiaoye Sherry Li and Scott Baden, "Performance Modeling Tools for Parallel Sparse Linear Algebra Computations", *Proceedings of the International Conference in Parallel Computing (ParCo)*, June, 2009

Pietro Cicotti and Xiaoye Sherry Li and Scott Baden, "LUSim: A Framework for Performance Modeling of Sparse Linear Algebra Computations", *Technical Report, Lawrence Berkeley National Laboratory*, 2009

Pietro Cicotti and Scott Baden, "Asynchronous Programming with Tarragon", *(Short Paper) Proceedings of the 15th IEEE Symposium on High Performance Distributed Computing*, June, 2006

Pietro Cicotti and Scott Baden, "Asynchronous Cellular Microphysiology Simulation", *(Abstract) 12th SIAM Conference on Parallel Processing for Scientific Computing*, February, 2006

Michela Taufer and David P. Anderson and Pietro Cicotti and Charlie L. Brooks III, "Homogenous Redundancy: a Technique to Ensure Integrity of Molecular Simulations Results Using Public Computing", *Proceedings of the 14th Heterogeneous Computing Workshop (HCW2005)*, April, 2005

Pietro Cicotti and Michela Taufer and Andrew Chien, "DGMonitor: a Performance Monitor Tool for Sandbox-based Desktop Grid Platforms", *Journal of Supercomputing*, 2004

Pietro Cicotti and Michela Taufer and Andrew Chien, "DGMonitor: a Performance Monitor Tool for Sandbox-based Desktop Grid Platforms", *Proceedings of the 14th Heterogeneous Computing Workshop*, April, 2004

ABSTRACT OF THE DISSERTATION

Tarragon: a Programming Model for Latency-Hiding Scientific Computations

by

Pietro Cicotti

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Scott B. Baden, Chair

In supercomputing systems, architectural changes that increase computational power are often reflected in the programming model. As a result, in order to realize and sustain the potential performance of such systems, it is necessary in practice to deal with architectural details and explicitly manage the resources to an increasing extent. In particular, programmers are required to develop code that exposes a high degree of parallelism, exhibits high locality, dynamically adapts to the available resources, and hides communication latency.

Hiding communication latency is crucial to realize the potential of today's distributed memory machines with highly parallel processing modules, and technological trends indicate that communication latencies will continue to be an issue as the performance gap between computation and communication widens. However, under Bulk Synchronous Parallel models, the predominant paradigm in scientific computing, scheduling is embedded into the application code. All the phases of a computation are defined and laid out as a linear sequence of operations limiting overlap and the program's ability to adapt to communication delays.

This thesis proposes an alternative model, called Tarragon, to overcome the lim-

itations of Bulk Synchronous Parallelism. Tarragon, which is based on dataflow, targets latency tolerant scientific computations. Tarragon supports a task-dependency graph abstraction in which tasks, the basic unit of computation, are organized as a graph according to their data dependencies, i.e. task precedence. In addition to the task graph, Tarragon supports metadata abstractions, annotations to the task graph, to express locality information and scheduling policies to improve performance.

Tarragon's functionality and underlying programming methodology are demonstrated on three classes of computations used in scientific domains: structured grids, sparse linear algebra, and dynamic programming. In the application studies, Tarragon implementations achieve high performance, in many cases exceeding the performance of equivalent latency-tolerant, hard coded MPI implementations.

The results presented in this dissertation demonstrate that data-driven execution, coupled with metadata abstractions, effectively support latency tolerance. In addition, performance metadata enable performance optimization techniques that are decoupled from the algorithmic formulation and the control flow of the application code. By expressing the structure of the computation and its characteristics with metadata, the programmer can focus on the application and rely on Tarragon and its run-time system to automatically overlap communication with computation and optimize the performance.

Chapter 1

Introduction

1.1 Motivation

In supercomputing systems, architectural changes that increase computational power are often reflected in the programming model. As a result, in order to realize and sustain the potential performance of such systems, it is necessary in practice to deal with architectural details and explicitly manage the resources to an increasing extent. In particular, programmers are required to develop code that exposes a high degree of parallelism, exhibits high locality, dynamically adapts to the available resources, and hides communication latency.

Hiding communication latency is crucial to realize the potential of today's distributed memory machines with highly parallel processing modules, and technological trends indicate that communication latencies will continue to be an issue as the performance gap between computation and communication widens. In fact, while the number of cores per processor continues to increase, specialized processors (often called accelerators) are becoming a common attribute of high performance computing systems[1]. High core counts and accelerators contribute to an increase in the relative performance gap between computation rate and data transfer rate. In addition, the finer granularity required to leverage the parallelism in hardware makes application performance very sensitive to data transfer latency, a slowly improving hardware characteristic [Pat04]. However,

[1] Including the system ranked first, four of the 10 most powerful systems in the world according to the Top500 project use accelerators [Top]

parallel programming models still lack high level abstractions for hiding communication latencies. This dissertation proposes a dataflow programming model that supports latency-hiding scientific computations.

1.2 Foundational Models and the State of the Art

In the context of distributed memory architectures, that is computers with networked computing elements with physically separated memory and operating in separate address spaces, most programming models are inspired by three foundational programming models: Bulk Synchronous Parallel, Dataflow, and Actors.

1.2.1 Bulk Synchronous Parallel

The classical Bulk Synchronous Parallel (BSP) programming model is the dominant model in parallel scientific applications [Les90]. In a classic BSP program, processes execute in parallel operating on a partition of the data and alternating a computation phase with a synchronous communication phase. The underlying parallel computer is a set of processors with local memory connected by a router. BSP has been labeled a *bridging* model to emphasize the intent of defining a unified model driving both software and hardware design. In fact, BSP captures the essence of both distributed memory machines and data-parallel algorithms. Among others, the Message Passing Interface (MPI) and Unified Parallel C (UPC), two well known parallel programming models, are intrinsically BSP models. These models are discussed next.

Message passing is the predominant communication model used in parallel scientific computing. Processes live in a private address space and communicate by exchanging messages. MPI [Mes94], a language-independent standard defined by the MPI Forum [Mes], is generally considered the de-facto standard for message passing on distributed memory machines, and is available on virtually every parallel computer.

In an MPI program, processes execute in Single Program Multiple Data (SPMD) mode, that is, multiple processes execute the same program, and communicate via message passing primitives. In addition to point-to-point communication, MPI also defines several collective communication primitives (e.g. broadcast and barrier), non-blocking

primitives.

By focusing on the semantics of the operations, the MPI standard leaves many implementation details unspecified, allowing for architecture-specific and implementation-specific optimizations. However, MPI primitives are low level and the application developer is required to control both locality and communication, managing constructs such as communication buffers and communication requests. Furthermore, application level performance optimizations, such as overlap of communication with computation, are expressed as part of the algorithmic formulation of the problem.

Hardware support for Remote Direct Memory Access (RDMA) and the implicit overheads of MPI semantics motivated a more efficient communication paradigm: one-sided communication. With one-sided communication processes can access each other's memory independently and without synchronization. Building on the foundations of Active Messages [Tho92], the GASNet networking layer provides an interface to allocate remotely accessible memory and to perform remote memory reads and writes [D. 02]. GASNet has been the target of source-to-source translation to introduce one-sided communication extensions in existing programming languages. In the resulting programming models, processes execute in SPMD mode and have access to a Partitioned Global Address Space (PGAS).

The PGAS family of languages includes UPC, a C extension [UPC05], Titanium, a Java extension [K. 98], and CoArray Fortran, a Fortran extension [Yur04]. The syntax and semantics of PGAS languages differ but the programming model is the same. For example in UPC, which is perhaps the most visible PGAS language, processes executing in SPMD mode access global data structures via one-sided data transfer primitives: *put* and *get*. Though based on a different communication model than message passing, UPC primitives are low level and also require the programmer to manage locality and data transfer details. In addition, asynchronous primitives that enable overlap of communication with computation are not part of the UPC standard. Although certain implementations provide asynchronous communication primitives [D. 04], such extensions are not portable.

1.2.2 Dataflow

Dataflow models were first conceived as architectural support for automatic parallelization [Pau94]. Compilers for dataflow languages produced a graphical program representing instructions and their dependencies. The graph executes on a dataflow computer where instructions are mapped to functional units that communicate according to their data and control dependencies.

In dataflow machines execution is data-driven: the flow of data activates functional units when their operands are available. Despite its inherently parallel execution model, the fine granularity and the tight coupling required between functional units, together with the restrictions imposed on dataflow languages, hindered the adoption of dataflow computers. However, as a programming model dataflow remained attractive for its implicit parallelism.

Large-grain dataflow languages gained traction recently as multi-core architectures pervaded computer architectures. Cilk, for example, is a multithreaded language for task-parallelism [R. 96]. Internally, Cilk activates tasks by maintaining a dependency graph. However, Cilk is intended for shared-memory architectures and does not support performance optimizations. In particular, Cilk is locality oblivious and does not enable overlap of communication with computation.

The Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [Jak09] is a programming model for dense linear algebra calculations designed to match the parallelism offered by multi-core architectures. Even state of the art linear algebra libraries based on MPI, such as ScaLAPACK, do not expose the same degree of parallelism because of their inherently synchronous execution model. The approach of PLASMA is to use a task abstraction to let the user define tasks and their dependencies while the underlying run-time system manages execution and communication transparently.

PLASMA targets numerical linear algebra. Ad-hoc data structures and data distributions provide locality information that the run-time system uses to determine task mapping. The graph structure unfolds dynamically during execution and cannot be analyzed for performance tuning. However, the run-time system may overlap computation with communication.

1.2.3 Actors

The Actors model is a data-driven programming model in which actors are first class objects that communicate by sending messages [Hew73, Gul86]. Actors have a persistent state and execute in response to every message that they receive. When executing, an actor computes a new state; it may send messages to other actors, and it may create new actors.

Charm++ is an object oriented parallel language that implements an actors model [Kal93]. In Charm++ actors are special objects, called *chares*, with *entry* methods supporting *Asynchronous Remote Method Invocation* [Bir84, Obj95]. Entries are like communication primitives: when an entry is invoked, the underlying run-time system creates a message that is sent to the destination chare.

In Charm++ execution is virtualized [Lax02] in the sense that chares execute like virtual processes managed by the underlying run-time system. The run-time system also manages communication and it can overlap communication with computation.

1.2.4 Latency Hiding

While characterized by different design and implementation choices, all BSP models preserve the same computation structure with a communication phase that is part of the control-flow and that lies on the critical path. Despite numerous research efforts [A. 93, Law02, Jos06, Sco98, Cos05, Ant05], in BSP models there is no widely adopted solution to hide latency other than to *manually* develop split-phase communication code. The idea of split-phase communication is simple: initiate the communication as soon as possible (first part of the communication phase) and wait for completion only when required by the implicit data dependencies (second part of the communication phase). Though simple as an idea, split-phase communication requires considerable programming effort because it involves extensive code restructuring. In addition, the restructured code may exhibit a locality oblivious memory access pattern [Pie].

Data-driven programming models, like dataflow and actors, show promise as a way to expose a high degree of parallelism and automatically achieve overlap. In fact, the ability to achieve overlap in data-driven algorithmic formulations has been demon-

strated in ad-hoc implementations of linear algebra kernels [Hus07]. However, current data-driven programming models fail to expose and express the underlying communication pattern and do not present users with high level abstractions that support latency-hiding algorithms.

1.3 Research Contributions

The programming-model proposed in this thesis is Tarragon: a dataflow programming model for latency-tolerant scientific computations [Pie06a, Pie06b, Pie06c]. Tarragon supports a task-dependency graph abstraction in which tasks, the basic unit of computation, are organized as a graph according to their data dependencies. Tasks communicate by transferring data along the edges of the graph and the underlying run-time system manages dataflow execution semantics and data motion. In addition to the task graph, Tarragon uses metadata abstractions to express locality information and scheduling policies [Pau01].

The task-dependency graph and its attributes (e.g. task priority, and task affinity) enable Tarragon programs to decouple programming and correctness concerns from performance concerns; a separation that promotes performance portability. With Tarragon, the application developer defines tasks and dependencies, focusing on the application, rather than on details of the architecture, and exposing the desired granularity of parallelism. Tarragon can exploit such parallelism and overlap communication with computation automatically via its data-driven execution model and underlying scheduler. In addition, the application developer can use Tarragon metadata abstractions to fine tune performance, for example, by establishing scheduling policies that improve overlap.

Tarragon is designed for parallel scientific applications and libraries. Its programming model raises the level of abstraction, but it is geared to performance more than to productivity. Tarragon is not a domain-specific library and it is intended as a substrate providing abstractions for applications and domain-specific library extensions. Tarragon delegates the objective of higher productivity to domain-specific extensions. Such extensions capture the expertise in a scientific domain, increasing productivity through software reuse and, most importantly, by presenting the application developer

with familiar abstractions.

Tarragon has been applied to three classes of computations used in scientific domains: structured grids, sparse linear algebra, and dynamic programming [Col04, Asa06]. The application studies presented in this thesis demonstrate the functionalities of Tarragon and its underlying programming methodology. In addition, each study presents a performance comparison to an equivalent hand-coded latency-tolerant MPI implementation.

The chosen applications exhibit a wide range of characteristics including regular and irregular workloads, compute-bound and memory-bound computation phases, and coarse and fine grain communication. Therefore, the programming methodologies presented in this dissertation are largely applicable to a wide range of computation and communication patterns. However, these programming techniques are mostly relevant for applications designed for distributed memory architectures, whose communication latencies can degrade performance significantly. Tarragon is also relevant for current and near-future heterogeneous architectures based on accelerators whose communication latencies can account for as much time as the computation does [Nha].

1.4 Organization of the Dissertation

This dissertation is organized into seven chapters that present Tarragon's programming model and its implementation, three applications, and a discussion of the research contributions of this thesis.

Chapter 2 presents the programming model of Tarragon and its fundamental abstractions. In particular, Chapter 2 details the execution model and the communication model of Tarragon, and gives programming examples to illustrate the use of Tarragon. Chapter 2 concludes with a review of related work.

Chapter 3 presents the implementation of the Tarragon library and the Application Programmer Interface. Chapter 3 also presents results with a micro-benchmark implemented with Tarragon. The micro-benchmark is used to assess the communication overheads of Tarragon.

Chapters 4, 5, and 6 present applications implemented using Tarragon. The ap-

plications are a finite-difference iterative solver, a direct solver for sparse systems of linear equations, and a string alignment tool, respectively. Each application represents a computational *motif*: a class of computation and data movement patterns (motifs are defined in the classification given by Colella [Col04] and later extended by Asanovic et. al. [Asa06]). The corresponding motifs are: structured grids (numerical methods), sparse linear algebra (numerical methods), and dynamic programming (machine learning and optimization problems). Each Chapter presents implementations and results on two testbeds. In addition, for demonstration purposes, Chapter 4 presents a domain-specific library extension for applications using finite-difference grids.

Chapter 7 summarizes the contributions and the results of this dissertation, concluding with a discussion of future research directions.

References

[A. 93] A. Krishnamurthy and D. E. Culler and A. Dusseau and S. C. Goldstein and S. Lumetta and T. von Eicken and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 262–273, New York, NY, USA, 1993. ACM Press.

[Ant05] Anthony Danalis and Ki-Yong Kim and Lori Pollock and Martin Swany. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 58, Washington, DC, USA, 2005. IEEE Computer Society.

[Asa06] Asanovic, Krste and Bodik, Ras and Catanzaro, Bryan Christopher and Gebis, Joseph James and Husbands, Parry and Keutzer, Kurt and Patterson, David A. and Plishker, William Lester and Shalf, John and Williams, Samuel Webb and Yelick, Katherine A. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[Bir84] Birrell, Andrew D. and Nelson, Bruce Jay. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2:39–59, February 1984.

[Col04] Colella, Phil. Designing Software Requirements for Scientific Computing, 2004.

[Cos05] Costin Iancu and Parry Husbands and Paul Hargrove. HUNTing the Overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society.

[D. 02] D. Bonachea. GASNet Specification, v1.1 . Technical Report CSD-02-1207, U.C. Berkeley , Oct 2002.

[D. 04] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National Lab, Oct 2004.

[Gul86] Gul Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 58–67, New York, NY, USA, 1986. ACM Press.

[Hew73] Hewitt, Carl and Bishop, Peter and Steiger, Richard. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[Hus07] Husbands, Parry and Yelick, Katherine. Multi-threading and one-sided communication in parallel LU factorization. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 31:1–31:10, New York, NY, USA, 2007. ACM.

[Jak09] Jakub Kurzak and Jack Dongarra. Fully Dynamic Scheduler for Numerical Computing on Multicore Processors. Technical Report UT-CS-09-643, University of Tennessee, Knoxville, June 2009.

[Jos06] José Carlos Sancho and Kevin J. Barker and Darren J. Kerbyson and Kei Davis. MPI tools and performance studies—Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 125, New York, NY, USA, 2006. ACM Press.

[K. 98] K. A. Yelick and L. Semenzato and G. Pike and C. Miyamoto and B. Liblit and A. Krishnamurthy and P. N. Hilfinger and S. L. Graham and D. Gay and P. Colella and and A. Aiken. Titanium: A High Performance Java Dialect. *Concurrency and Computation: Practice and Experience*, 10, 1998.

[Kal93] Kalé, L.V. and Krishnan, S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In Paepcke, A., editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[Law02] Lawry, W. and Wilson, C. and Maccabe, A.B. and Brightwell, R. COMB: a portable benchmark suite for assessing MPI overlap. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 472 – 475, 2002.

[Lax02] Laxmikant V. Kalé. The Virtualization Model of Parallel Programming : Runtime Optimizations and the State of Art. In *LACSI 2002*, Albuquerque, October 2002.

[Les90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *CACM*, 33(8):103–111, Aug 1990.

[Mes] Message Passing Interface Forum. Message Passing Interface Forum standards documents, errata, and archives of the MPI Forum.

[Mes94] Message Passing Interface Forum. MPI:A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4):165–414, 1994.

[Nha] Nhat Than Nguyen and Pietro Cicotti and Didem Unat and Scott B. Baden. Latency hiding in Multi-GPU Stencil Computations. *In Preparation*.

[Obj95] Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification (Revision 2.0), July 1995.

[Pat04] Patterson, David A. Latency lags bandwith. *Commun. ACM*, 47:71–75, October 2004.

[Pau94] Paul G. Whiting and Robert S. V. Pascoe. A History of Data-Flow Languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, 1994.

[Pau01] Paul H J Kelly and Olav Beckmann and Tony Field and Scott B Baden. Themis: Component dependence metadata in adaptive parallel applications. In *Parallel Processing Letters*, pages 455–470, 2001.

[Pie] Pietro Cicotti and Scott B. Baden. Tarragon: a Programming Model for Latency Hiding Scientific Applications. *In Preparation*.

[Pie06a] Pietro Cicotti and Scott B. Baden. Asynchronous Cellular microphysiology Simulation. In *12th SIAM Conference on Parallel Processing for Scientific Computing (Abstract)*, San Francisco, California, February 2006.

[Pie06b] Pietro Cicotti and Scott B. Baden. Asynchronous programming with Tarragon. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing (Abstract)*, page 159, New York, NY, USA, 2006. ACM Press.

[Pie06c] Pietro Cicotti and Scott B. Baden. Short Paper: Asynchronous programming with Tarragon. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 375–376, 2006.

[R. 96] R. D. Blumofe and C. F. Joerg and B. C. Kuszmaul and C. E. Leiserson and K. H Randall and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[Sco98] Scott B. Baden and Stephen J. Fink. Communication overlap in multi-tier parallel algorithms. In *Proc. of SC '98*, Orlando, Florida, November 1998.

[Tho92] Thorsten von Eicken and David E. Culler and Seth Copen Goldstein and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM Press.

[Top] Top 500. http://top500.org.

[UPC05] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, May 2005.

[Yur04] Yuri Dotsenko and Cristian Coarfa and John Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2004. IEEE Computer Society.

Chapter 2

Programming Model

2.1 Overview

The programming model proposed in this thesis is based on a dataflow abstraction. The abstraction hides most of hardware and software low-level details while it requires programmers to decompose the problem into tasks and express dependencies between the tasks. Exposing parallelism is still the programmer's duty, as in most widely adopted programming models [Asa06, HPC, Pet], although parallel execution and concurrency are implicit and their control transparent to the programmer.

Tarragon's programming model is a coarse-grain dataflow model. In classical dataflow models a program implicitly defines a graph in which nodes are instructions and edges connect nodes such that the result of an instruction becomes an operand of another instruction [Jac80, Jac75, J. 85, Pau94, Arv90]. The *firing rule* is straightforward: an instruction executes when all its operands are available. Data is transferred along the edges, rather than stored in memory, and there is no program counter as the graph execution manages control flow. In Tarragon the nodes of the graph are coarse-grained objects, called *tasks*. Tasks of arbitrary complexity can be defined and can preserve their state across executions. The *firing rule* is user-defined and the data is stored in memory. The edges carry data in the form of *messages*.

Tarragon tasks are similar to *actors* [Hew73, Gul86]. In the *Actors* model, actors are objects that communicate by passing messages. In response to a message, an actor processes the message and possibly sends more messages. Actors provide encapsulation

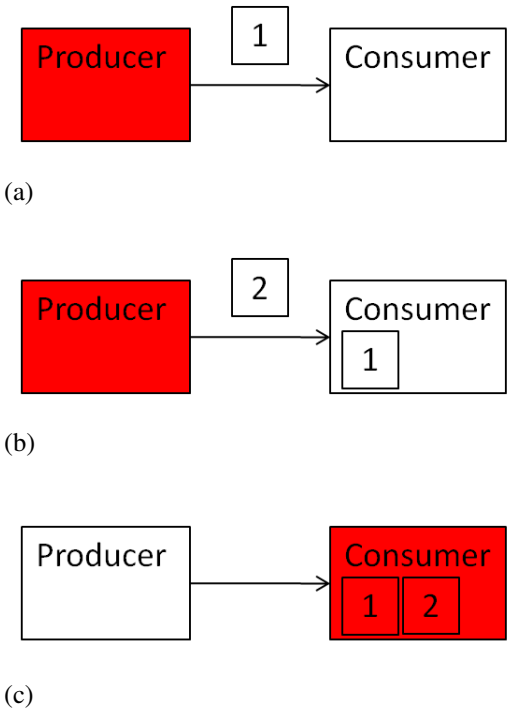
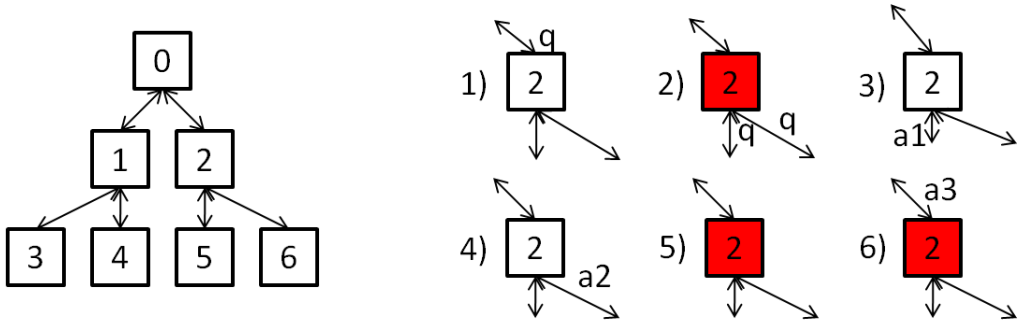


Figure 2.1: Producer-consumer. The producer task sends messages executes and sends two messages to the consumer. The first message has weight 1 (Figure 2.1a), the second has weight 2 (Figure 2.1b). The consumer executes after receiving two messages.

by containing a private state and a set of operations. Similarly, Tarragon tasks have a state and they process data according to their state and functionality. The computation that a task performs can therefore be a function of both its state and the data received.

Several differences distinguish Tarragon's model from dataflow and actors. Mainly, Tarragon's model differs in that it uses an explicit task graph. The graph also accommodates the use of metadata which can be used to optimize and tune performance. In particular, the task graph may be annotated with attributes, such as priorities, to drive scheduling decisions and improve performance.

The example in Figure 2.1 illustrates a simple graph with two tasks working in a producer-consumer relationship. The edge connects the tasks and is oriented toward the consumer indicating that data flows in that direction. The producer executes first and produces two messages: the first with *weight* 1, as illustrated in Figure 2.1a, and the second with *weight* 2, as illustrated in Figure 2.1b. Then it stops executing. In



(a) (b)

Figure 2.2: Map-reduce network, shaped as tree (Figure 2.2a). The steps of node 2 are illustrated in Figure 2.2b: first the node receives the query (1), then it forwards the query to its children nodes and processes the query (2), receives the partial answers from the children (3,4), combines the answers with the local answer (5), and finally it sends the combined answer to the parent node.

this example, the firing rule for the consumer is defined such that it executes only after receiving a cumulative workload of at least *weight* 2. The consumer executes only after receiving both messages, as illustrated in Figure 2.1c. In general, firing rules can take into account characteristics of the data, in addition to the number of messages and their source.

This example illustrates other differences between Tarragon and dataflow models and actors. The producer sends more than one message, and it does so within the same task execution. In fact, in the example, the first message is sent immediately, before the second message is even produced. In Tarragon, tasks can send messages multiple times and at any time during execution, like actors. In contrast, in dataflow each instruction produces one result, possibly duplicated, which is transferred only after the instruction is completed. In addition, in dataflow and actors there is one predefined firing rule: execute when all the operands are available (one message in the case of actors), whereas Tarragon supports user-defined firing rules.

Consider another example, a map-reduce network, illustrated in Figure 2.2. In a map-reduce network a tree of tasks operate on a dataset in a divide-and-conquer fashion. A problem is distributed (*mapped*) to the nodes of the tree, from the root towards the

leaves, and the nodes compute a partial solution locally; then the result is *reduced* as a combination of all the partial results, which move up from the leaves towards the root. A classical dataflow implementation requires two trees: a map tree and a reduce tree. The map tree is made of instructions that produce partial results. The reduce tree is made of instructions that receive partial results from the map tree and from the children nodes, and that produce partial results. The resulting graph is more complex than the one illustrated in Figure 2.2a. This tree can be implemented in Tarragon because depending on their state tasks can perform different operations. In this example, tasks execute two times: first when they receive the problem, which they forward to their children before computing the local solution, and then when they receive the solutions from the children tasks, which they combine with the local solution before sending the combined solution to the parent task. The sequence of steps is illustrated in Figure 2.2b. This capability supports the construction of concise graphs which are easier to define and manipulate, and that also occupy less space in memory. In fact, the size of the graph may be a concern in applications with a very large number of tasks, such as dense linear algebra computations [Jak09]. In addition, the flexibility in defining tasks enables data-parallel formulations in which tasks are associated with a partition of the data and each partition is accessed and modified only by the associated task.

2.2 Execution Model

Tarragon's programming model executes under a three-layer control structure. The three levels resemble the *XYZ levels* of the Phase Abstraction programming model [Sny93]. In Phase Abstractions the X level is the *process level* and represents a sequence of instructions logically grouped together (e.g. a procedure); the Y level is the *phase level* and controls processes that together execute a parallel algorithm; and the Z level is the *problem level* and executes a sequence of phases. The corresponding three levels in Tarragon are the *task level*, the *graph level*, and the *control level* (the levels are listed in Table 2.1). The task level represents the instructions within a task. The graph level controls task execution according to dataflow semantics. Finally, the control level controls the graph abstractions and is responsible for creating and executing graphs.

Table 2.1: Control levels in Phase Abstractions and in Tarragon.

Level	Phase Abstraction	Tarragon
0	Problem (Z)	Control
1	Phase (Y)	Graph
2	Process (X)	Task

The control level initializes Tarragon's run-time system (RTS), creates a graph, and launches the graph's execution. The example in Algorithm 1 illustrates the construction of a ring of *n* tasks. After starting the RTS (line 1), a *Map* is created (line 2). A *Map* defines a logical naming scheme for the tasks of the graph (in this case it can be assumed that tasks names are ids in [0..n-1]). Then, a graph containing *n* tasks is allocated (line 3). At this point, all the tasks have been allocated but no dependencies are defined. The loop (lines 4-6) connects all the tasks; the resulting graph is a ring as illustrated in Figure 2.3. Finally, the graph is initialized and executed (lines 7-8). The program terminates by finalizing the RTS (line 9).

Algorithm 1 Ring Program

```
1: Tarragon_initialize()
2: Map ring(n)
3: Graph graph(ring)
4: for all task∈graph do
5:     task.connect(ring.next(task))
6: end for
7: Tarragon_initialize_graph(graph)
8: Tarragon_execute_graph(graph)
9: Tarragon_finalize()
```

The RTS supports the graph level of execution abstraction. Once the tasks of the graph are defined and connected, execution is transparently carried out by the RTS, which manages task scheduling and data motion, orchestrating the dataflow abstraction. The *execute_graph* function call (line 8) encapsulates graph execution. At the graph level, the order in which tasks are executed is constrained by dependencies, but among

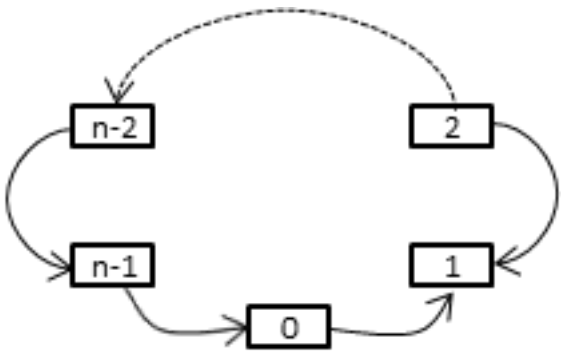


Figure 2.3: Ring with *n* tasks.

partially ordered tasks[1], the exact ordering is determined by Tarragon's RTS. As dependencies are satisfied, tasks become ready to execute and eventually they are executed by the RTS. When multiple tasks are ready for execution, they run in parallel as available resources allow.

The number of tasks should reflect a logical partition of the problem and should be independent of the number of processor cores. In fact, the number of tasks should be large compared to the number of cores, to enable the RTS to leverage the benefits of processor virtualization [Lax02] (e.g overlap). In contrast to the Single Program Multiple Data model of execution, in which each process is an instance of a program (usually executing with exclusive access to a core), Tarragon's RTS schedules task execution by mapping tasks to cores dynamically, treating tasks as virtual processes. Virtualization also enables scheduling policies that improve performance, for example, by overlapping execution with communication; while tasks execute the RTS transfers information along the edges of the graph. In addition, the Application Programmer Interface (API) enables the programmer to control scheduling and mapping of tasks to processing elements. For example, programmers can define task priorities and affinities using graph metadata.

Finally, at the task level, tasks execute as virtual processes, unaware of the underlying levels. A task becomes ready to execute when its firing rule is satisfied. Once a task begins executing, it runs to completion. It cannot be preempted and it cannot wait for communication events. These conditions simplify the task code which can be

[1] A partial order on a set defines an ordering on the elements although not all the elements of the set are comparable; that is, the order relation is not defined on all the pairs of elements.

designed as a self-contained sequential program, free of scheduling and concurrency concerns.

2.3 Communication Model

In Tarragon, communication between tasks is expressed by connecting tasks with directed edges. During graph execution, tasks can move data along the edges of the graph. The graph is therefore a representation of the communication pattern.

Communication is one-sided in the sense that the destination task is not actively involved in the communication. As with Active Messages [Tho92], data arrival triggers a handler function execution. The handler injects data into the task and triggers task readiness according to the firing rule that it encodes. Since sending data is an asynchronous operation and there is no receive operation, a task never blocks on communication and it is therefore guaranteed that, when executing, tasks are actively computing. In this way processor virtualization is implemented without preemption yet it is guaranteed that cores are utilized efficiently.

Tarragon has an intuitive communication cost model in which locality is loosely related to the mapping of tasks to processes. The cost of sending data is a function of the amount of the data sent and whether the receiving task exists within the same address space. Tarragon transparently manages communication within process boundaries, where it can take advantage of shared memory, as well as between nodes. Therefore, there is a distinction between local tasks, which live within the same address space, and remote tasks, which do not. Process boundaries are not exposed as graph attributes and are not part of the abstraction, although the API provides the means to obtain locality information. Whether process boundaries match physical shared-memory nodes boundaries is a run-time configuration and it is not exposed to the programmer.

2.4 Tarragon Abstractions

The programming model of Tarragon presents several abstractions to the programmer. These abstractions, which are summarized in Table 2.2, are implemented as

classes of Tarragon Application Programmer Interface (API). Each abstract class is a template that the programmer must customize to define an application specific subclass. The remainder of this Section gives a brief description of these abstractions and the abstract and concrete classes of the API. All the classes are discussed in greater detail in Chapter 3, which illustrates the implementation details.

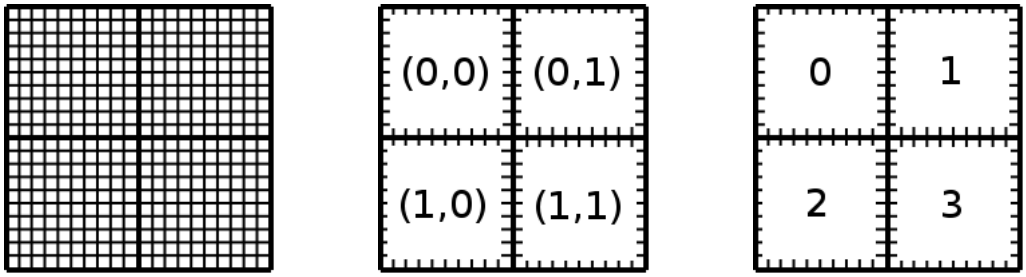
Table 2.2: Fundamental abstractions of Tarragon.

Abstraction	Abstract Class	Concrete Sub-Class	Description
Graph	Graph	VectorGraph	distributed vector of tasks
Task Naming	Map	IdentityMap	identity mapping
Node	Task	RingTask	example: task of the ring
Edge	Dependency	OutDependency	outgoing dependency
Message	Message	BufferedMessage	contiguous memory message

2.4.1 Map and Graph

In Tarragon a problem is *decomposed* by partitioning the computation into tasks. The partitioning should reflect a logical separation of operations, data, or both. Operations and data are two natural sources of parallelism with a distinction leading to two forms of parallelism: *task parallelism* and *data parallelism*. Task-parallelism exists when distinct groups of operations can be executed concurrently. In Tarragon each group of operations could be naturally defined as a task and the dependencies between tasks could be defined to both enable data transfers between tasks and to ensure the correct order of execution. Data-parallelism exists [2] when identical sequences of operations can be applied concurrently on multiple data. In Tarragon, individual data could be associated to a different task and the tasks would differ only in the data operated on. The choice of which form of parallelism to exploit in an application depends on the algorithmic formulation of the problem. The two forms of parallelism can coexist in Tarragon.

[2] Data-parallelism in this context has coarser granularity in comparison to the granularity of vector instructions.



(a) (b) (c)

Figure 2.4: Map of tasks. A 2-dimensional mesh is decomposed into four quadrants (Figure 2.4a). The quadrants are identified by the pair (row, column) and enumerated from left to right, top to bottom (Figure 2.4b and Figure 2.4b).

The application programmer is responsible for decomposing the application and choosing the appropriate level of granularity, two activities that require deep knowledge of the application. In contrast, Tarragon manages and abstracts the low-level details of data motion and execution. In this way Tarragon helps to separate application specific concerns from performance concerns. In addition, because of processor virtualization, the granularity of the decomposition and the number of tasks are logically independent of the number of physical processing elements. The decomposition strategy can reflect the logical structure of the computation. In contrast, in most SPMD model implementations, the decomposition strategy must reflect partitions and processing elements to guarantee efficient execution.

Tarragon tasks are uniquely identified by a task id that can be assigned according to a naming scheme that provides a logical interpretation of the name. For example, for problems characterized by regular domain decomposition it is often convenient to define tasks as if they were arranged on a multi-dimensional mesh. To support such mappings between multi-dimensional coordinate systems and tasks, Tarragon provides the *Map* class.

Map is an abstract base class. A *Map* describes the task structure underlying a graph. Programmers define customized *Map* subclasses, implementing a mapping function from a multi-dimensional coordinate system to a set of task ids. As an example, Figure 2.4 shows a problem that is regularly decomposed in a 2-dimensional partition resulting in 4 tasks. The tasks are logically arranged as a 2-dimensional coordinate sys-

tem preserving the position of the associated partitions. The mapping is then defined in terms of the coordinates as illustrated in Figure 2.4c. The corresponding *Map* encapsulates the implementation of a function $m : Coordinates \rightarrow Task$ and its inverse. In the case of the ring example (previously introduced with Algorithm 1), the mapping defaults to the identity function defined on the set $[0..n-1]$:

$$Map\,map = new\,Identity(n).$$

All the tasks of a graph are defined as an instance of class *Graph*. A *Graph* is a container of tasks. When instantiating a *Graph* object, users also associate the tasks with a map.

*Graph*s can be characterized by different allocation strategies and data structures. *Graph* is an abstract class and Tarragon allows ad-hoc *Graph* implementations. However, Tarragon also provides a concrete implementation, called *VectorGraph*, that implements a distributed vector template. To instantiate a graph of tasks, in the case of the ring example, it is therefore sufficient to create a *VectorGraph* as follows:

$$Graph\,graph = new\,VectorGraph < RingTask > (map).$$

2.4.2 Tasks and Dependencies

Once a graph is instantiated all the tasks must be connected according to their dependencies. In Tarragon only connected tasks can communicate and programmers must declare how tasks exchange data and interact. By defining dependencies, programmers define also synchronization and precedence between tasks. For example, defining a *directed acyclic graph* imposes a partial ordering on tasks. The partial order is also imposed on execution: two tasks that have a dependence cannot execute concurrently and but they will execute according to their ordering.

Tasks are connected using the *connect* method as illustrated in the ring example in Algorithm 1 (line 5):

$$task.connect(map.next(task)).$$

When two tasks are connected, a *Dependency* object is automatically instantiated. Connected tasks can then use the dependency as a communication channel. The argument of

the method *connect* is the value returned by the *next* function, which is the id of the following task in the ring (e.g. $(id+1) \, mod \, n$). At the end of the loop in Algorithm 1 (lines 4-6), the ring is formed and each task has a reference to its incoming and its outgoing dependencies. The outgoing dependency is an *OutDependency* object accessible only by the source task. An *OutDependency* differs from a *Dependency* in that it may be used to send data via a *put* method. The *put* method is a non-blocking operation that sends a *Message* along a graph edge to the destination task. Eventually, the run-time system (RTS) delivers the message, but there is no receive operation: messages are processed by a special method of the task, called *vinject*, that encapsulates the firing rule for that task and the application-specific data delivery mechanism. Low level details of the data transfer and message dispatch are hidden and dealt with by the RTS.

The *Task* class defines the task in terms of its operations and its behavior. The *Task* class is a template that programmers customize to define application-specific subclasses; in particular, when extending *Task*, programmers override two methods: *vexecute* and *vinject*. The RTS invokes *vinject* to deliver messages, and invokes *vexecute* to execute tasks. The subclass of *Task* used in the ring example is *RingTask*.

Algorithm 2 RingTask::vinject(message)

```
1: →EXEC
2: msg = message
```

The *vinject* method encapsulates a firing rule and a data delivery mechanism. Algorithm 2 shows the *vinject* implementation of *RingTask*. In this case the firing rule is straightforward: always execute when a message is received (line 1). The *_state* variable is an attribute that the RTS monitors and it is used to encode relevant changes in the state of the task. In this case, the task has become ready for execution and eventually the RTS will invoke its *vexecute* method. The data delivery mechanism is also straightforward in this case: a reference to the message is set.

Typical firing rules are not so simple and require inspecting the actual state of the task, including whether or not other messages have been received since the last execution. Also, the delivery mechanism may involve sophisticated data structures and even memory copies although it is recommended that *vinject* be kept as simple as possible to avoid overloading the RTS with application-specific work. Examples of more elaborate

Algorithm 3 RingTask::vexecute()

```
1:  if id==0 then
2:      if (trips==0) then
3:          →DONE
4:      else
5:          dependency.put(msg)
6:          →WAIT
7:      end if
8:      trips = trips-1
9:  else
10:     dependency.put(msg)
11:     trips = trips-1
12:     if trips==0 then
13:         →DONE
14:     else
15:         →WAIT
16:     end if
17: end if
```

cases are presented in Chapters 4, 5, 6.

By changing the value of *_state*, *vinject* and *vexecute* control transitions of the observable state of a task. In fact, a task is a state machine and, most notably, *vinject* may cause transitions from waiting to ready to execute (from *WAIT* to *EXEC*), and *vexecute* may cause transitions from executing to waiting or completed (from *EXEC* to *WAIT* or *DONE*). The RTS transfers control to a task by invoking *vexecute*. Within *vexecute*, a task executes at the task level of control, as defined in Section 2.2. At this level the task executes to completion and produces the results that it will pack into messages and *put* on the outgoing edges. *vexecute* encapsulates most of the application code. In the ring example, the application sends a message around for a predefined number of times (the *trips* variable). In the ring implementation of *vexecute*, as presented in Algorithm 3, a small distinction is made between the first task and the others: the first task has to count the trips executed before sending a message whereas the other tasks can forward the message immediately. In addition, the first task executes one time more than the others in order to receive the message that completes the last trip.

2.4.3 Messages

In Tarragon, a *Message* is a data object that can be transferred along edges. *Message* objects are defined as data that can be transformed into a serialized sequence of bytes, allowing the data to be transferred between tasks residing in different address spaces. Programmers can define new types of messages together with the methods to be used for serializing the data to be sent, and then reconstruct the message from the data received (i.e. objects and elaborate pointer based structures).

In the ring example, a message of type *BufferedMessage* is used. A BufferedMessage is a contiguously stored message that does not require serialization. In the example it is assumed that one such message is instantiated during the initialization of task 0:

$$msg = make\_message < BufferedMessage > (size).$$

2.4.4 A Working Example

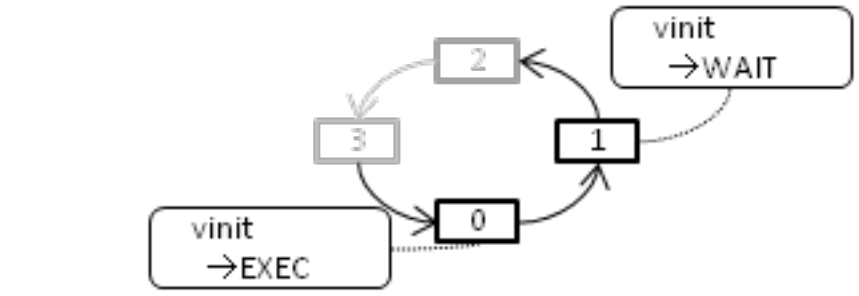
To summarize, this section illustrates the ring example describing all the steps that tasks and the run-time system (RTS) take during execution. The example is based on the illustration in Figure 2.5, which represents a 4 task ring. The illustration focuses on tasks 0 and 1, which execute different instructions; the remaining tasks execute the same sequence of instructions that task 1 executes.

The illustration shows initialization and execution of the tasks. Figure 2.5a illustrates the initialization. During the initialization, the RTS invokes *vinit* on every task that is in *INIT* state. All the tasks, in this example, default to the *INIT* state when instantiated, and the RTS invokes *vinit* on all of them. Then, *vinit* sets the state to *WAIT* for all the tasks but one: task 0. Task 0 is the task that starts the communication ring by sending the first message and its state is set to *EXEC*.

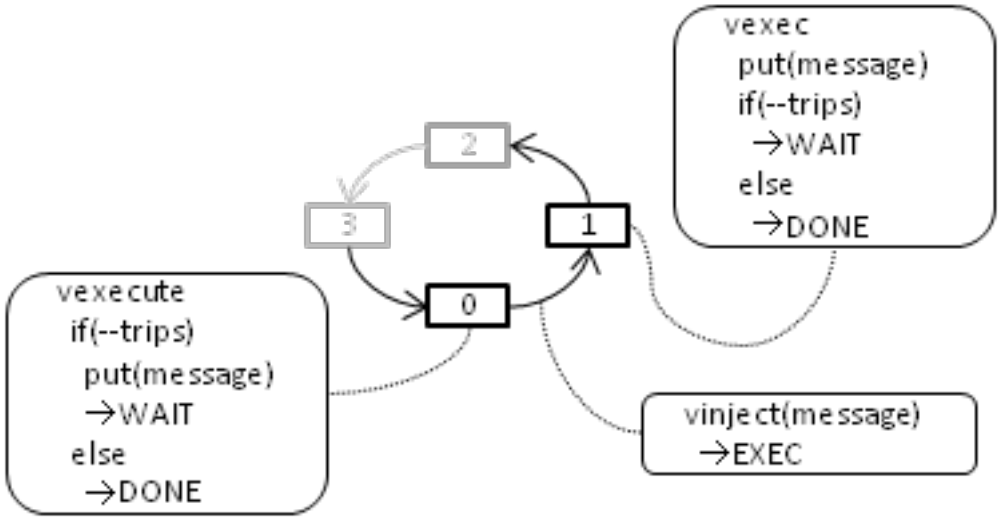
Figure 2.5b illustrates the execution of the tasks. When execution begins, the only task that is ready to execute is task 0. When it executes, task 0 sends the message to task 1, then enters the *WAIT* state. *vinject* delivers the message to task 1, and sets its state to *EXEC*. Then, since task 1 can now be executed, the RTS executes task 1. When executing, task 1 forwards the message and sets its state to either *WAIT* or *DONE*, depending on whether more messages are expected or not. The message travels around the ring, while all the other tasks repeating the instructions that task 1 executed, and eventually reaches task 0. At this point, if there are trips left to do, task 0 continues as in the previous iteration, otherwise it is *DONE* and execution terminates.

2.5 Related Work

The Candidate Type Architecture (CTA) is a parallel machine model that captures basic architectural features of distributed memory machines [Law86]: a collection of von Neumann machines (i.e. a processor with local memory), connected by a communication network. CTA emphasizes the physical separation in distributed memory machines and supports programming models, such as Phase Abstraction [Sny93,Gai98], and cost models, such as LogP and LogGP [Dav96, Alb95]. The three levels of execution of Phases Abstraction, the XYZ levels, identify three levels of control: processor,



(a)



(b)

Figure 2.5: Ring execution. Tasks, which are represented by numbered square blocks, are connected by edges forming a ring, and the RTS executes the virtual methods, represented by blocks with round corners. During the initialization, which is illustrated in Figure 2.5a, the RTS invokes the *vinit* method. Then, during the execution, the RTS invokes *vinject* to deliver messages, and *vexecute* to execute tasks.

phase, and global; the phase and the global level introduce synchronization and coordinated execution on the processor and the phase level, respectively.

Synchronization is emphasized in the Bulk Synchronous Parallel (BSP) model [Les90]. BSP unifies the design of the architecture and of the programming model: the machine is a set of processors, periodically synchronized, that are connected by a router capable of delivering point-to-point messages, and that execute processes alternating computation and communication phases. Although today's machines have multiprocessor nodes, the essence of the model still applies and BSP is a widely adopted programming style in current parallel programming models. For example, programs written using the Message Passing Interface (MPI) [Mes94] are generally BSP. In MPI, processes execute in SPMD mode and communicate by exchanging messages, typically alternating between communication and computation phases. In MPI, locality is controlled by partitioning and mapping data to processes; communication is explicitly managed by the programmer, within the control flow of the computation.

Partitioned Global Address Space (PGAS) languages [UPC05, Yur04, K. 98], though based on a different communication model, are also generally BSP. For example, in Unified Parallel C (UPC), perhaps the most successful PGAS language to date, processes execute in SPMD mode and communicate via one-sided communication primitives. With one-sided communication, processes can access each other's memory, independently and without synchronization. Despite the decoupling between processes that one-sided communication facilitates, some form of synchronization between processes is necessary to preserve data dependencies; therefore, BSP remains the programming model of choice as it simplifies dependency management.

Hiding communication latency in BSP programming models requires careful code restructuring to implement *split-phase* communication. Split-phase communication is achieved by breaking up the communication phase into an initiation and a completion phase, using non-blocking primitives. For example, MPI provides non-blocking primitives making it possible to initiate a send (or a receive) and then wait for completion later, using a *wait* primitive. In between the two phases the computation advances to the extent allowed by data dependencies, and it is overlapped with communication. Similarly, non-blocking versions of one-sided communication primitives can be defined.

For example, the Berkeley UPC implementation supports non-blocking primitives [D. 04], however the UPC standard does not include non-blocking primitives and such extensions are not portable.

Split-phase communication requires a considerable programming effort because it involves extensive code restructuring. Not only must communication be split in two phases, but the computation must be rearranged accordingly. Operations whose data dependencies are satisfied within the same process are extracted and executed between the two communication phases; the remaining part is executed after the second phase, that is when communication satisfies the remaining dependencies. As a result, the memory access pattern which is determined by data dependencies may lead to low data locality and therefore poor performance [Bad01, Pie].

Dataflow models were proposed as an alternative processor design for speeding up computation via parallel execution [Jac80, Jac75, Pau94]. A dataflow computer has a communication network that automatically transfers data packets between specialized functional units. Dataflow machines are inherently parallel and programs written in a dataflow language achieve high parallelism automatically. However, in order to facilitate translation to dataflow execution, dataflow languages impose some restrictions.

The first dataflow model was a *static* dataflow model: only one packet was allowed on an edge of the dataflow graph and operands of functional units were trivially matched by arrival order [Jac80]. The implementation proposed supported the Value-oriented Algorithmic Language (VAL) [McG82]. VAL observed the single assignment rule: once a variable is assigned a value, it retains such value throughout its scope. VAL major weaknesses included lack of I/O facilities and lack of recursion as well as other restrictions [Pau94].

Some restrictions were relaxed in *dynamic* dataflow models using *tagging*: packets are tagged and functional units execute when operands with matching tags are received [J. 85, Arv90]. Tagging increases parallelism and efficiency allowing out-of-order communication and enabling functional units to be reused (e.g. in loops and procedures) and recursion. Id [Arv73] and Streams and Iterations in a Single Assignment Language (SISAL) [McG83] are example of languages for dynamic dataflow machines. However, limitations in the programming model, such as the single assignment rule, and in the

hardware, such as its memory model and scalability [Cul92], limited wide acceptance of dataflow models.

To overcome such limitations the Large-Grain Dataflow (LGDF) model was proposed. LGDF combined imperative languages, which can execute on traditional von Neumann architectures, with dataflow semantics [Bab84]. LGDF is a compromise between dataflow and traditional sequential programming models in that it relies on modularity as a decomposition mechanism. Modules in LGDF are statically connected program chunks forming a dataflow graph.

More recently, large-grain dataflow languages gained traction on multi-core architectures for their ability to expose fine grain parallelism. Cilk is a multithreaded C language extension. In a Cilk program, special asynchronous functions are executed in parallel creating branches on the stack [R. 96, Mat98]. Functions are then efficiently scheduled by Cilk's run-time system which treats frames as tasks and unfolds a Directed Acyclic Graph (DAG) of tasks. Cilk is best suited for shared-memory architectures. Its execution is based on a shared activation stack which is expensive to support on distributed memory architectures. In addition, the programming model of Cilk is locality oblivious: tasks are mapped to processors dynamically but without taking locality into account.

The problem of locality in shared-memory architectures is addressed in SMARTS [Suv99] and OSCAR [Kas98]. Both SMARTS and OSCAR infer a task graph from loops that operate on special data structures. The idea is that the underlying run-time system can schedule tasks that operate on the same data as soon as possible in order to increase temporal locality. The two differ mainly in that OSCAR creates the graph statically, at compile-time, and is therefore more limited in the type of analysis it can perform. Neither SMARTS nor OSCAR is designed for distributed memory machines and nor do they take communication latencies into account.

The Parallel Linear Algebra Software for Multicore Architectures (PLASMA) library applies dataflow ideas to support dense linear algebra operations [Jak09, Son09]. With PLASMA, tasks are defined by the application, at run time, as function calls with special arguments. Such arguments express dependencies to other tasks: as their value is defined, dependencies are satisfied and the corresponding tasks are automatically gen-

erated, executed, and destroyed.

PLASMA is intended for dense linear algebra kernels that expose a high degree of parallelism and it aims at achieving performance and scalability by supporting fine grain task parallel algorithms. Since the size of the resulting graph for the problems addressed can easily exceed physical memory capacity, the task graph is dynamically unfolded by a run-time system that maintains a window of instantiated tasks. Consequently, only a small portion of the graph is available at any time preventing graph analysis for optimizations. Data motion between tasks is automatically managed by the run-time system, which sends and receives data as required, possibly overlapping communication with computation.

The Actors programming model also offered abstractions for fine-grain concurrency [Hew73, Gul86]. Actors is a data-driven programming model, based on message passage semantics, in which actors are objects activated by messages; when activated, an actor may change state, send messages to other actors, or create new actors. Charm++ is an implementation of actors [Kal93]. In Charm++ actors are special objects, called *chares*, communicating by invoking special methods, called *entry*. Invoking an entry corresponds to asynchronously sending a message to the target chare. In fact, entries are invoked through an Asynchronous Remote Method Invocation mechanism [Kal93] implemented using messages. The underlying run-time system manages data motion and, when delivering messages, it activates chares by invoking the corresponding entry.

The data-driven execution model of actors and the asynchronous semantics of entry invocations enable concurrent execution. In addition, Charm++ enables overlap of computation with communication. Ideally, a number of chares larger than the number of available processors is defined and because of the inherent decoupling between execution and communication, the run-time system can activate chares while communication takes place.

In Charm++ the dataflow structure is embedded in the control flow and it is unveiled only during program execution. The lack of an explicit control flow was considered a limitation of Charm++ [L. 04], especially from a software engineering point of view, because it increased the complexity of extending and combining existing programs. To overcome such limitations and with the goal to achieve better productivity it

was introduced Charisma++ [L. 04]. Charisma++ is a high level *orchestration* language that can be used to define chare arrays, and describe operations and data dependencies between operations. Orchestration files are then translated to Charm++ and combined with Charm++ code. The benefit lies in the ability to express a *collective* view of the computation. However, Charisma++ cannot support run-time optimizations based on the chares structure because the structure is lost during the translation.

Computer architecture continues to evolve and programming models continue to adapt to such evolution. The High Productivity Computing System program [Don08] funded the design and development of two additional PGAS languages: Chapel [Bra] by Cray and X10 [Cha05] by IBM (a third HPCS language called Fortress, by Sun, was dropped in 2006 [For]). Features of Chapel and X10 are designed for special hardware support: Cascade and PERC are the systems conceived by Cray and IBM for Chapel and X10 respectively. Currently, both systems are still in development and there is little information about performance achieved and optimization techniques supported. However, the fine granularity and the asynchronous nature of certain language constructs suggest that implementations will automatically overlap computation with communication.

Table 2.3 summarizes the characteristics of the most influential implementations of the foundational parallel programming models.

Table 2.3: Summary of related programming model implementations.

Implementation	Model	Architecture	Memory Model	Locality	Data Motion	Dependency Definition	Performance Optimizations
VAL	DF	DF	None	N/A	Implicit	Implicit	None
SISAL	DF	DF	None	N/A	Implicit	Implicit	None
LGDF	DF	Shared Memory	Shared Memory	N/A	Implicit	Explicit	None
MPI	BSP	Distributed Memory	Distributed Memory	Explicit	Explicit Message Passing	Implicit	Split-Phase Communication
UPC	BSP	Distributed Memory	PGAS	Explicit	Explicit One-sided	Implicit	Split-Phase Communication
Cilk	DF	Distributed Memory	Shared Memory	N/A	Implicit	Implicit	None
SMARTS	DF	Shared Memory	Shared Memory	Implicit	Implicit	Implicit	Locality-Aware Scheduling
OSCAR	DF	Shared Memory	Shared Memory	Implicit	Implicit	Implicit	Locality-Aware Scheduling

Table 2.3: (continued)

Implementation	Model	Architecture	Memory Model	Locality	Data Motion	Dependency Definition	Performance Optimizations
PLASMA	DF	Distributed Memory	Distributed Memory	Implicit	Implicit	Explicit	Automatic Overlap
Charm++	Actors	Distributed Memory	Distributed Memory	N/A	Implicit	Implicit	Automatic Overlap
Charisma++	Actors	Distributed Memory	Distributed Memory	N/A	Implicit	Explicit	Automatic Overlap

2.6 Discussion

Tarragon introduces a novel programming model based on dataflow semantics. Tarragon relies on data-driven execution to expose parallelism and create opportunities for hiding communication latencies. The proposed model borrows principles and ideas from dataflow and actors, but it also differs from existing programming models in several ways.

Tarragon is less restrictive than classic dataflow models; for example, in Tarragon, tasks can be very complex and have a persistent state. In addition, one of the novel features of Tarragon design is that it supports user defined firing rules. For example, Tarragon supports static dataflow semantics but also dynamic dataflow semantics. Users may encode *tagging* and inspect the metadata content when a message is delivered via *vinject*. Encapsulation is achieved by defining class hierarchies rather than modules, such as in Large Grain Data Flow. In addition, the graph in Tarragon is not statically encoded in the program, as in classical dataflow programming models; the graph in Tarragon is constructed dynamically and is an object that can be analyzed and manipulated at run-time.

Tarragon's model has characteristics of actors: tasks have a state, perform operations in response to messages, and may send messages to other tasks. In fact, actors is essentially a data-driven model in which actors are unary functions firing at every message they receive. Charm++'s execution model uses remote method invocation to encapsulate data motion and execution, whereas Tarragon decouples data motion from execution, and execution is regulated by custom firing rules and scheduling policies that can improve performance. In addition, Charm++'s execution model may incur scheduling overhead and cache pollution. For example, when significant computation can be performed only after a number of messages have been received, all the messages but the last will cause a short execution. In contrast, Tarragon's model promotes short message handlers, encapsulated in *vinject*, and the RTS tries to avoid inefficiencies by executing a task only when it is ready to execute.

In BSP models, such as MPI and UPC, mapping partitions of a problem to computing resources is usually a matter of defining how processes are laid out on processor cores. The decomposition of the computation, which matches how operations and

data are distributed across processes, also defines locality. On the other end, dataflow and actor models, such as Cilk and Charm++, use task abstractions that free the programmer from restrictions on the decomposition, but that also hide locality information. PLASMA is an exception as it adopts well-defined data structures and data distributions for dense matrices.

Tarragon also provides a task abstraction. In addition, it enables fine control on mapping and exposes locality. In Tarragon, task mapping is resolved in two phases. The first phase is part of the creation of the graph, when tasks are distributed over processes. In this phase applications may control the distribution of tasks ensuring load balancing and locality. In this way, Tarragon relies on the programmer to decide the appropriate task granularity and the optimal workload distribution. The second phase is part of task execution and is dynamically carried out by the RTS. In this phase the mapping problem is implemented as a scheduling problem. While in general this phase is automatically carried out by the RTS, the programmer can control scheduling and mapping to cores using task attributes, that is metadata (i.e task priority and task affinity).

Tarragon's approach to preserving locality with scheduling is similar to the approach taken in SMARTS and OSCAR. Tarragon's RTS attempts to prioritize tasks to favor temporal locality. However, Tarragon's approach is general and is not limited to dependencies inferred from predefined data structures, as in SMARTS and OSCAR. Rather, Tarragon scheduling responds at run-time to execution order and graph dependencies. In addition, Tarragon application developers have the ability to create scheduling policies by defining task affinities and priorities, and they may use this ability to improve locality in ways that are application specific.

Unlike SMARTS and OSCAR, Tarragon is designed for distributed memory machines and exposes two-tiers of memory: distributed and local memory. Shared-memory boundaries are not exposed in the graph, but Tarragon provides such locality information through its API: application developers can query Tarragon to discover which tasks exist within the same address space. In this way, while the graph executes on a single-tier memory hierarchy, Tarragon makes it possible to implement dual-tiered algorithms that improve performance by leveraging shared memory [Sco98].

In other programming models there is no explicit representation of the commu-

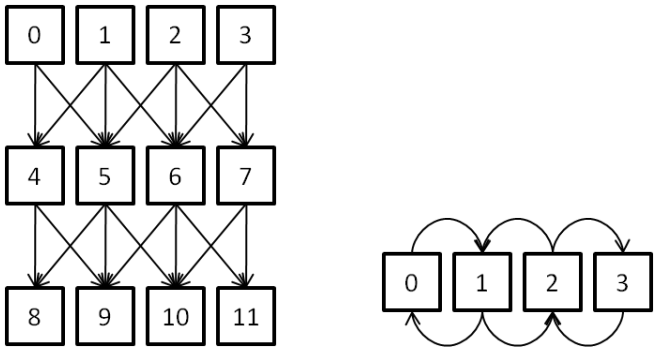


Figure 2.6: Iteration space unrolling with tasks. In Figure 2.6a three iterations unroll creating with new tasks for each iteration. In Figure 2.6b iterations unroll as part of the encoded state of the tasks.

nication pattern as in Tarragon. Tarragon's philosophy is that graph analysis can be used to improve performance and therefore, it is convenient to represent the task-dependency graph in memory. In PLASMA the problem of managing a large graph is solved by unfolding the graph dynamically as the computation progresses, making it harder to analyze the graph. The generality of Tarragon's model supports techniques for defining a graph such that the size of the graph is greatly reduced. For example, representing the entire iteration space of a data-parallel iterative computation as a DAG requires storing a very large graph. For each iteration and for each data element there must be a node in the DAG. Even worse, if the number of iterations is not known, it would not be possible to create such graph before the computation unfolds the iteration space completely. Tarragon overcomes these limitations by defining the iterations number as part of the state of the tasks. In this way, the graph matches the data space and the communication pattern in each iteration. The example is illustrated in Figure 2.6. Figure 2.6a illustrates three iterations over a four element data array such that for each iteration there are four tasks updating the associated data element. In contrast, Figure 2.6b illustrates a possible representation in Tarragon with only four tasks that execute three times each.

In BSP models, the order of the instructions implicitly defines a static schedule of operations. All the phases of the computation are defined and laid out as a linear sequence of operations. The order is determined by the programmer's perception of what

is the most elegant and perhaps the most efficient arrangements of the operations. Tarragon forces the programmer to explicitly outline the units of the computation and their data dependencies. By doing so Tarragon relieves the programmer from the responsibility to define the schedule. Tarragon schedules execution dynamically, at run time, adapting to the workload and automatically overlapping communication with computation.

In BSP models, hiding communication latencies requires tedious and error prone split-phase coding. In addition, by disrupting the sequential flow of execution, split-phase coding may have a negative impact on performance. Conversely, Tarragon's data-driven execution model supports automatic overlap of communication with computation without the need to resort to split-phase coding. All that is required is enough parallelism to ensure that work is available while communication takes place. By virtualizing processors, Tarragon's abstractions enable decompositions that meet this requirement and Tarragon metadata can help improve overlap further through application-specific policies. Through graph and metadata analysis the RTS discovers and prioritizes tasks with dependencies crossing shared-memory boundaries reducing the likelihood that data-dependencies prevent a task from executing because it is waiting on data. The schedule is modified without having to change the application.

References

[Alb95] Albert Alexandrov and Mihai F. Ionescu and Klaus E. Schauser and Chris Scheiman. LogGP: incorporating long messages into the LogP model one step closer towards a realistic model for parallel computation. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, New York, NY, USA, 1995. ACM Press.

[Arv73] Arvind, K. P. Gostelow, W. Plouffe. An asynchronous programming language and computing machine. Technical Report TR-114-a, Dept. of Information and Computer Science, Univ. of California, Irvine, December 1973.

[Arv90] Arvind, K. and Nikhil, Rishiyur S. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39:300–318, March 1990.

[Asa06] Asanovic, Krste and Bodik, Ras and Catanzaro, Bryan Christopher and Gebis, Joseph James and Husbands, Parry and Keutzer, Kurt and Patterson, David A. and Plishker, William Lester and Shalf, John and Williams, Samuel Webb and Yelick, Katherine A. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[Bab84] Babb, R.G., II. Parallel Processing With Large-Grain Data Flow Technique. *Computer*, 17(7):55–61, July 1984.

[Bad01] Baden, Scott and Shalit, Daniel. Performance Tradeoffs in Multi-tier Formulation of a Finite Difference Method. In Alexandrov, Vassil and Dongarra, Jack and Juliano, Benjoe and Renner, RenÃl'nd Tan, C., editor, *Computational Science ICCS 2001*, volume 2073 of *Lecture Notes in Computer Science*, pages 785–794. Springer Berlin / Heidelberg, 2001.

[Bra] Bradford L. Chamberlain and Steven J. Deitz and Mary Beth Hribar and Wayne A. Wong. Chapel: The Cascade High Productivity Language.

[Cha05] Charles, Philippe and Grothoff, Christian and Saraswat, Vijay and Donawa, Christopher and Kielstra, Allan and Ebcioglu, Kemal and von Praun, Christoph and Sarkar, Vivek. 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*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[Cul92] Culler, David E. and Schauser, Klaus Erik and von Eicken, Thorsten. Two Fundamental Limits on Dataflow Multiprocessing. Technical Report UCB/CSD-92-716, EECS Department, University of California, Berkeley, Dec 1992.

[D. 04] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National Lab, Oct 2004.

[Dav96] David E. Culler and Richard M. Karp and David Patterson and Abhijit Sahay and Eunice E. Santos and Klaus Erik Schauser and Ramesh Subramonian and Thorsten von Eicken. LogP: a practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996.

[Don08] Dongarra, J. and Graybill, R. and Harrod, W. and Lucas, R. and Lusk, E. and Luszczek, P. and McMahon, J. and Snavely, A. and Alam, S. and Campbell, ROY and others. DARPAâĂŹs HPCS Program: History, Models, Tools, Languages. *Advances in Computers: High Performance Computing*, 72:1, 2008.

[For] Fortress Project.

[Gai98] Gail A. Alverson and William G. Griswold and Calvin Lin and David Notkin and Lawrence Snyder. Abstractions for Portable, Scalable Parallel Programming. *IEEE Trans. Parallel Distrib. Syst.*, 9(1):71–86, 1998.

[Gul86] Gul Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 58–67, New York, NY, USA, 1986. ACM Press.

[Hew73] Hewitt, Carl and Bishop, Peter and Steiger, Richard. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[HPC] HPCS Application Analysis and Assessment. http://www.highproductivity.org/kepner-HPCS.htm.

[J. 85] J. R Gurd and C. C Kirkham and I. Watson. The Manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[Jac75] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, New York, NY, USA, 1975. ACM Press.

[Jac80] Jack B. Dennis. Data Flow Supercomputers. *Computer*, 13(11):48 – 56, Nov 1980.

[Jak09] Jakub Kurzak and Jack Dongarra. Fully Dynamic Scheduler for Numerical Computing on Multicore Processors. Technical Report UT-CS-09-643, University of Tennessee, Knoxville, June 2009.

[K. 98] K. A. Yelick and L. Semenzato and G. Pike and C. Miyamoto and B. Liblit and A. Krishnamurthy and P. N. Hilfinger and S. L. Graham and D. Gay and P. Colella and and A. Aiken. Titanium: A High Performance Java Dialect. *Concurrency and Computation: Practice and Experience*, 10, 1998.

[Kal93] Kalé, L.V. and Krishnan, S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In Paepcke, A., editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[Kas98] Kasahara, Hironori and Yoshida, Akimasa. A data-localization compilation scheme using partial-static task assignment for Fortran coarse-grain parallel processing. *Parallel Comput.*, 24:579–596, May 1998.

[L. 04] L. V. Kalé and Mark Hills and Chao Huang. An Orchestration Language for Parallel Objects. In *Proceedings of Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 04)*, Houston, Texas, October 2004.

[Law86] Lawrence Snyder. Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–317, 1986.

[Lax02] Laxmikant V. Kalé. The Virtualization Model of Parallel Programming : Runtime Optimizations and the State of Art. In *LACSI 2002*, Albuquerque, October 2002.

[Les90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *CACM*, 33(8):103–111, Aug 1990.

[Mat98] Matteo Frigo and Charles E. Leiserson and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.

[McG82] McGraw, James R. The VAL Language: Description and Analysis. *ACM Trans. Program. Lang. Syst.*, 4:44–82, January 1982.

[McG83] McGraw, J.R., et. al. Streams and Iteration in a Single-Assignment Language, Language Reference Manual, Version 1.1. Technical Report M-146, Lawrence Livermore National Laboratory, July 1983.

[Mes94] Message Passing Interface Forum. MPI:A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4):165–414, 1994.

[Pau94] Paul G. Whiting and Robert S. V. Pascoe. A History of Data-Flow Languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, 1994.

[Pet] Peter Kogge at. al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems.

[Pie] Pietro Cicotti and Scott B. Baden. Tarragon: a Programming Model for Latency Hiding Scientific Applications. *In Preparation*.

[R. 96] R. D. Blumofe and C. F. Joerg and B. C. Kuszmaul and C. E. Leiserson and K. H Randall and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[Sco98] Scott B. Baden and Stephen J. Fink. Communication overlap in multi-tier parallel algorithms. In *Proc. of SC '98*, Orlando, Florida, November 1998.

[Sny93] Snyder, Lawrence. Foundations of practical parallel programming languages. In Volkert, Jens, editor, *Parallel Computation*, volume 734 of *Lecture Notes in Computer Science*, pages 115–134. Springer Berlin / Heidelberg, 1993.

[Son09] Song, Fengguang and YarKhan, Asim and Dongarra, Jack. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11, New York, NY, USA, 2009. ACM.

[Suv99] Suvas Vajracharya and Steve Karmesin and Peter Beckman and James Crotinger and Allen Malony and Sameer Shende and Rod Oldehoeft and Stephen Smith. SMARTS: exploiting temporal locality and parallelism through vertical execution. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 302–310, New York, NY, USA, 1999. ACM Press.

[Tho92] Thorsten von Eicken and David E. Culler and Seth Copen Goldstein and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM Press.

[UPC05] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, May 2005.

[Yur04] Yuri Dotsenko and Cristian Coarfa and John Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2004. IEEE Computer Society.

Chapter 3

Design and Implementation

3.1 Overview

Tarragon's programming model poses several requirements on the underlying implementation: the implementation must support explicitly-defined task graphs, task graphs must be annotated with attributes, such as task priority and affinity; and, there must be a run-time component that executes the graph, orchestrating task execution and managing data motion.

Tarragon is implemented as a C++ library. The library includes classes that define the building blocks of the task graph; and the run-time system (RTS) that executes the task graph. The application programmer interface (API) of the library is logically divided into the *Core API* and the *Extended API*. The Core API defines the interface to the RTS and the fundamental classes that support the dataflow abstraction: *graph*, *task*, *dependency*, and *map*. The Extended API defines additional classes providing a useful set of ready-to-use structures and serving as an illustrative example of how to build domain-specific library extensions (DSE) and applications.

Tarragon's software architecture is layered (Figure 3.1): the API creates the dataflow abstraction on top of the underlying layers, hiding low level machine-dependent details, and increasing performance portability.

Tarragon is designed for distributed memory machines with shared-memory nodes, i.e. multi-core nodes. It relies on a communication substrate and a threading substrate. Although the current implementation uses MPI [Mes94] and PThreads [Nic96],

Tarragon does not depend on specific communication and threading libraries and could be ported to different ones, for example, GASNet and OpenMP.

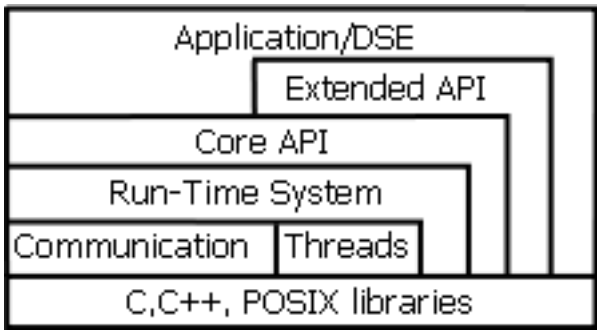


Figure 3.1: Software architecture of Tarragon. The levels indicate software levels that also corresponds to different levels of abstraction, starting from the bottom, which is the system libraries level, to the top, which is the application level.

This Chapter uses *UML class diagrams* [Fow03] to illustrate the structure of the API. In a class diagram, a class is represented by a box containing the name of the class and the members of the class: fields, for data members, and methods, for function members. Members are preceded by an access modifier: the plus sign indicates a public member accessible by all other classes. Field names are succeeded by the their type. Method names appear with the arguments list enclosed between parenthesis and succeeded by their return type. Underlined methods are *static* and do not refer to an instance of the class but to the class itself. A UML class diagram can also represent relationships between classes: *inheritance*, *composition*, and *reference*. *Inheritance* indicates a "*is a*" relationship and is represented by a line with a closed arrowhead. *Composition* indicates a "*is part of*" relationship and is represented by a line with a filled diamond. *Reference* is a "*refers to*" relationship and is represented by a dashed line. Figure 3.2 illustrates symbols of a UML class diagrams and their semantics.

3.2 Run-Time System

As discussed in Chapter 2, the execution model of Tarragon is layered and consists of a *control* level, a *graph* level, and a *task* level (see Table 2.1). Tarragon's RTS

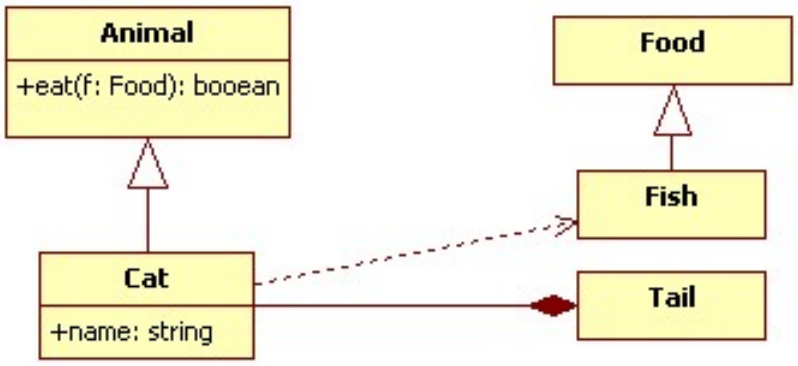


Figure 3.2: UML class diagram: symbols and semantics. In the diagram, *Animal* is a base class that defines the public method *eat*, which takes an argument of type *food* and returns a boolean value. *Cat* "is a" *Animal*, and has a public *name* field of type string. In addition, *Cat* "references" class *Fish*, its favorite subclass of *Food*. Class *Tail* defines an object that "is part of" *Cat*. A "reference" implies that a class knows and uses about another class, whereas "is part of" is a stronger relation in which an object is a part of another object and their existence is strongly related.

executes at the control level and supports the graph level and the task level by managing graph execution and data motion. In the current implementation Tarragon uses MPI for communication and for launching the computation. Therefore, the control level is realized as set of physical processes[1] that execute the RTS in Single Program Multiple Data (SPMD) mode: multiple processes execute the same program.

The RTS is started through its interface, the *Tarragon* class, which is illustrated in Figure 3.3 and summarized in Table 3.1. There must be only one RTS instance in each Tarragon process. To ensure that only one instance of the RTS is created, Tarragon implements the *singleton* design pattern [Gam02]. By construction, a singleton cannot be instantiated more than once because its constructor is not public and the RTS is instantiated only once through a static initialization method, as illustrated in Algorithm 4 (line 1). The instance of the RTS is then referenced via the *tarragon* method (line 2).

[1]Throughout this chapter, a physical process indicates an operating system process and will be referred to as process. In contrast, a task is a virtual process and will be referred to as task. Many tasks can live within a process.

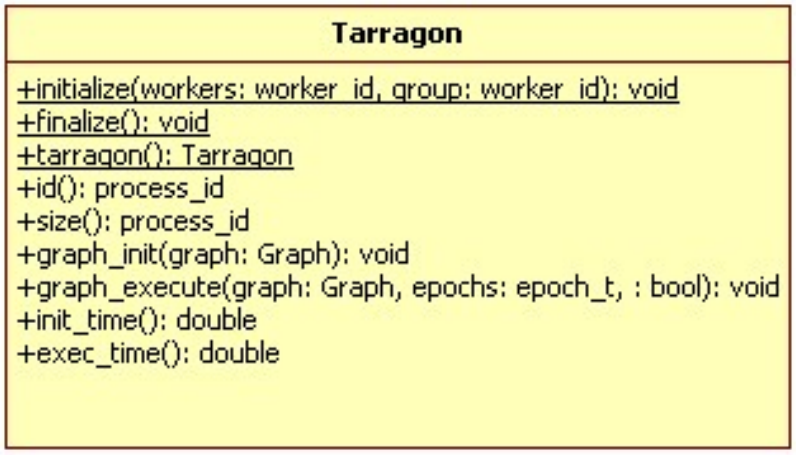


Figure 3.3: Class diagram of class Tarragon. The class defines the API to the RTS and enables users to control the RTS, and to execute a task graph.

Each instance has a unique id that identifies the process and that can be retrieved via the *id* method, whereas the total number of RTS instantiated, that is the number of Tarragon processes, can be retrieved via the method *size*.

Algorithm 4 Tarragon Program

```
1: Tarragon::initialize()
2: Tarragon rts=Tarragon::tarragon()
3: Tarragon::graph_init(graph)
4: Tarragon::graph_execute(graph)
5: Tarragon::finalize()
```

Each instance of the RTS is composed of a *scheduler* in charge of scheduling task execution and communication requests, and a pool of *workers* that execute the tasks. Workers and the scheduler are threads of execution within the RTS and ideally each thread is bound to a core exclusively. However, it is possible to create more workers than available cores or to assign workers to the same core where the scheduler runs. The RTS can be configured to run in single-threaded mode. In this case, a single thread of execution alternates between scheduling and executing tasks. The number of pro-

Table 3.1: Methods of class *Tarragon*. The class *Tarragon* defines the interface to the run-time system.

Method	Description
initialize	initialize the RTS
finalize	finalize the RTS
tarragon	reference to the RTS
size	number of RTSs
id	RTS id
graph_init	initialize the graph
graph_execute	execute the graph
init_time	graph initialization wallclock time
exec_time	graph execution wallclock time

cesses per shared-memory node is defined when the application is launched. Different configurations change the way cores are matched to the components of the RTS. Different configurations can result in different performance depending on how a configuration suits the application needs and the architecture of the machine. Figure 3.4 illustrates different configuration examples.

Processes executing at the control level construct the graph in parallel. Once the graph is complete, it is initialized and executed (lines 3-4). Initialization and execution occur at the graph level and the task level of execution, respectively. The graph level of execution is characterized by a higher degree of parallelism: the RTS activates workers to execute ready tasks. During initialization, the RTS establishes communication channels between processes according to the communication pattern described by the graph. Then, it activates all the tasks ready for initialization; when activated, tasks execute at the task level. When initialization completes, the application returns to the control level until the graph is executed.

When the graph is executed, the RTS executes the graph level and the task level of execution, activating tasks according to dataflow semantics. Finally, when the execution completes, the application continues at the control level. At that level, it can

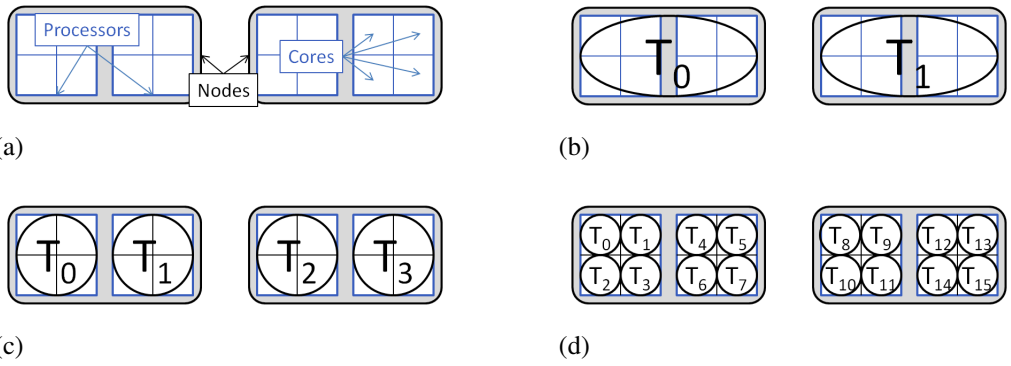


Figure 3.4: Run-time system configurations. Using the two nodes with two quad-core processors each, illustrated in Figure 3.4a, three possible configurations are illustrated in Figure 3.4b, Figure 3.4c, and Figure 3.4d. In Figure 3.4b there is a Tarragon process per node, in Figure 3.4c there is a Tarragon process per processor, and in Figure 3.4d there is a Tarragon process per core.

re-execute the graph, or repeat the construction-initialization-execution process with a different graph. Before terminating, each application must finalize the RTS (line 5). Finalization ensures that the RTS releases all the resources acquired directly, such as allocated memory, and indirectly through the communication and threading substrates. Figure 3.5 illustrates the phases that characterize execution in a typical Tarragon application. Figure 3.5 also reveals barriers that synchronize processes when graph execution is activated and terminated, and when the RTS is initialized and finalized.

Tarragon implements two timing methods that can be used to time initialization and execution: *init_time* and *exec_time*. *init_time* returns the wallclock time of executing *graph_init*, whereas *exec_time* returns the wallclock time of executing *graph_execute*.

The RTS contains a scheduler and a pool of workers, as illustrated in Figure 3.6. While workers execute tasks, the scheduler controls graph execution and manages queues of tasks that are ready to execute(called *ready queues*), queues of outgoing messages, and queues of pending communication requests. Whenever a communication request is completed the scheduler retires the request. If the request is a *send* request (i.e. put)then the to the source task is notified. If the request is a *receive* request then the scheduler instantiates the corresponding *Message* object and delivers the message to the destination task (i.e. invokes *vinject*). If, as the result of a receive request, the receiving

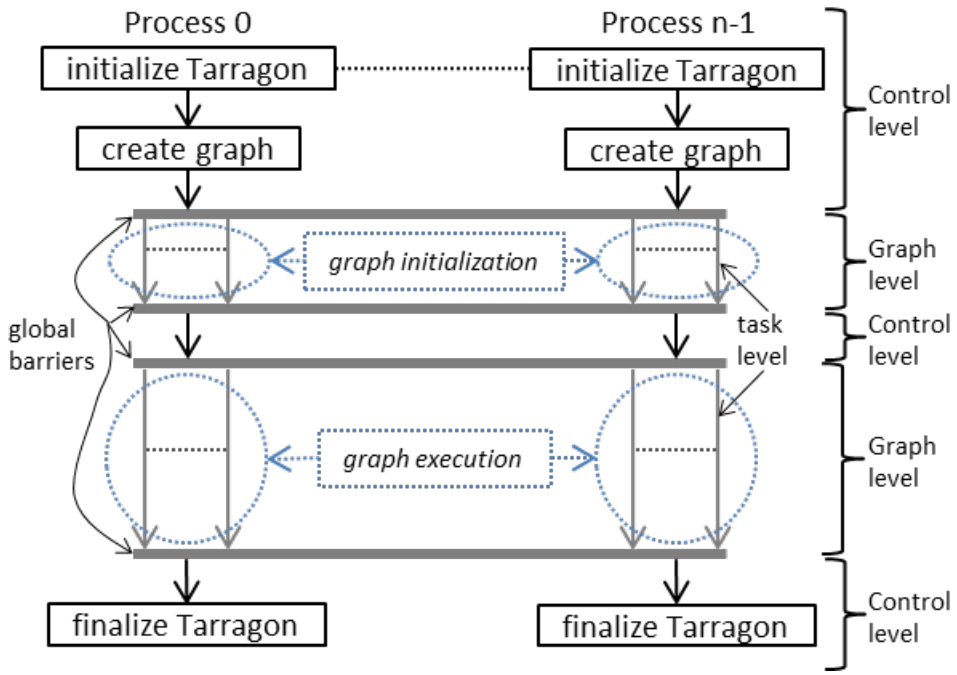


Figure 3.5: Controlflow of a Tarragon application. The execution of a Tarragon application is divided into levels, indicated on the right. Global barrier synchronize the processes when graph level execution begins and ends.

task becomes ready, then the scheduler schedules the task according to its priority and affinity (both priority and affinity are attributes of the task). At this point the task is appended to a ready queue.

The separation between graph level and task level execution is reflected in the provision of two methods of *Task*: *vinject* and *vexecute*, that implement the firing rule, and task execution, respectively. This separation is also reflected in the architecture of the RTS. The RTS invokes *vinject* when a message reaches a task and completes the data transfer. As a result, the task is assigned to a ready queue and executed only if the firing rule is satisfied. Tasks do not necessarily execute in response to every message received, this is determined by the firing rule of the task.

Task scheduling is affected by the scope of ready queues used. Workers in the same process may share a process-wide ready queue, or have a private ready queue. The choice affects how work is balanced within a process, whether tasks can be prioritized or not, and whether tasks can have worker affinity or not. Using a shared queue helps

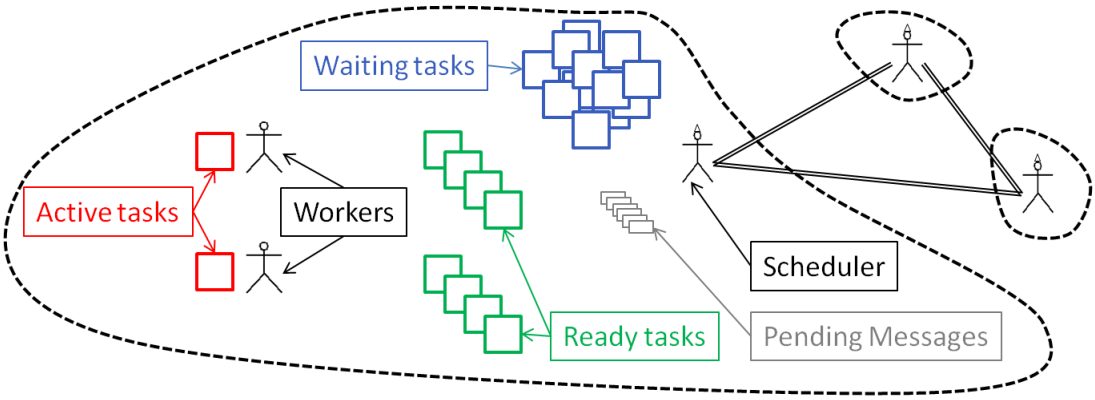


Figure 3.6: Internal structure of the run-time system.

reduce the idle times in presence of load imbalance, but does not support affinity because all the workers obtain tasks from the same queue. On the other hand, by having multiple queues, one per worker, it is possible to specify affinities for a particular ready queue. As a result, though with a shared ready queue a task is assigned to the first available worker, but with affinities a task is always assigned to a specific worker. In this case, tasks assigned to the same worker are executed in prioritized order although the scope of the priority is restricted to the ready queue.

The use of multiple queues also enables *work stealing* [R. 96]. Work stealing allows idle worker threads to steal work from the end of ready queue of busy worker threads within the same address space. However, because stolen tasks are pulled from the end of the queue, work stealing might cause tasks with low priority to execute before tasks with high priority. Table 3.2 summarizes the options and their properties.

Table 3.2: Queue configurations in Tarragon. Marks in each column indicate whether a configuration enables or not the property in the header.

Configuration	Priority	Load Balancing	Affinity
Shared	✓	✓	×
Private	✓	×	✓
Stealing	×	×	✓

Tarragon is able to overlap data transfers and computation. While scheduling

tasks, the RTS also acts as a communication proxy [Lim97, Bad00]. In this way, data transfers take place while the workers execute tasks; computation and data transfer may be concurrent and overlapped.

Overlap of communication with computation may be improved using task prioritization. By analyzing the graph it is possible to prioritize tasks with edges directed to tasks residing in different processes. Such prioritization scheme ensures that communication between such tasks is initiated earlier, increasing the interval of time between when data transfers are initiated and when the data is needed. As a form of pre-fetching, the prioritization scheme described reduces the likelihood that dependencies prevent a task from executing because of communication latencies, creating more opportunities for overlapping computation with communication.

3.3 Core API

The Core API defines the basic classes required to interact with the RTS. Basic operations include graph construction, data motion, and problem decomposition including mapping to physical resources, and error handling. The fundamental classes of the Core API are listed in Table 3.3.

The classes of the core API include the fundamental abstractions, which are represented by abstract classes, and some concrete implementations. The concrete classes extend the abstract classes to provide users with a basic set of ready-to-use classes.

Among the fundamental classes *Tarragon* is the only concrete class and it implements the interface to the RTS (described in Section 3.2). The remaining classes are abstract and they support Tarragon's programming model and address the phases of the workflow that typically characterize parallel applications: decomposition, mapping, and data motion. In addition, the Core API includes an exception class that Tarragon uses to raise error notifications to the control level.

All the classes listed are explained in detail in the following Subsections, which also present helper classes and concrete classes that are also part of the Core API.

[2]Users define most of the application code within *Task*. In addition to the computation, the resulting application-specific task will therefore participate to decomposition, mapping, and data motion.

Table 3.3: Fundamental classes of the Core API.

Class	Purpose	Description	Type
Tarragon	RTS management	RTS interface class	Concrete
Graph	graph construction and mapping	task graph	Abstract
Task	graph construction[2]	node of the graph	Abstract
Dependency	graph construction and data motion	dependency between tasks	Abstract
Message	data motion	data to be sent via dependency	Abstract
Map	decomposition	naming scheme for tasks	Abstract
Distribution	mapping	distribution of tasks between processes	Abstract
TarragonException	error handling error handling	exception raised by Tarragon	Abstract

3.3.1 Decomposition

In a process called problem decomposition, developers break problems down in order to expose parallelism. A problem decomposition method is either applied to the data, exposing data parallelism, or to the operations of the computation, exposing task parallelism. The Tarragon *Map* supports decomposition into tasks, such that tasks maintain the logical structure of the computation.

A *Map* simplifies task identification by providing a mapping function from points in $\mathbb{N}_0^n$ to task ids, and the inverse. In this way, tasks can be identified by their logical position within the *Map*, or by their unique id. It is also possible to query a *Map* and gather information about the map itself, such as the number of tasks it enumerates, the size of the map, and whether a task is defined by the map. The complete list of methods of *Map* is given in Table 3.4.

Users create ad-hoc naming schemes by defining a subclass of *Map* and overriding its methods: *next*, *prev*, *coordinate_index*, and *index_coordinate*. The methods

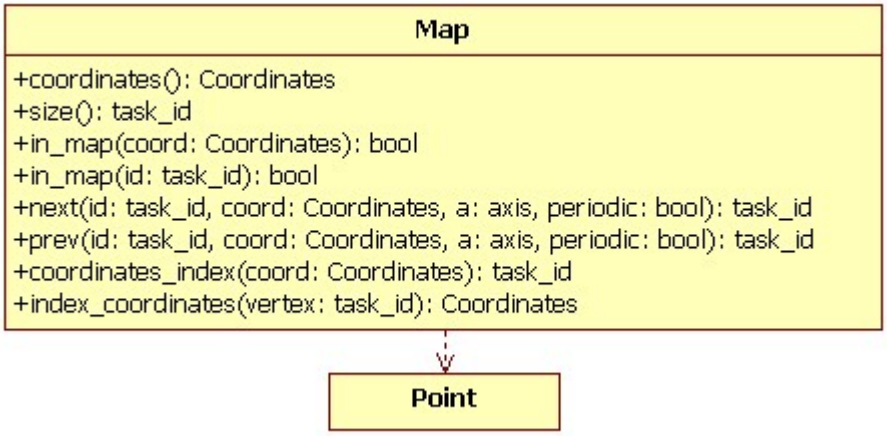


Figure 3.7: Class diagram of Map.

enable the mapping, but also to traverse the coordinate system in the order defined by the coordinates, rather than by id order, via the methods *next*, *prev*. As an illustrative example, the reader can refer to the ring example presented in Chapter 2, which used an *Identity* map. For example, an implementation of *Identity* would define the method *next* as illustrated in Algorithm 5.

Algorithm 5 Identity::next(id,periodic)

```
1: return periodic ? (id+1) mod size() : id+1
```

Algorithm 6 Identity::coordinate_index(coordinate)

```
1: return coordinate[0]
```

According to the signatures given in Figure 3.7, *id* and *periodic* are two arguments of next: the former is the reference task id, the latter indicates the type of boundary. *prev* can be similarly defined. The mapping in *coordinate_index*, which is defined using the *Point* template class, simply converts the value of a 1-dimensional point into a task id, as illustrated in Algorithm 6. Conversely, in *index_coordinate*, the mapping converts a task id to a 1-dimensional point, as illustrated in Algorithm 7.

Table 3.4: Virtual methods of class Map.

Method	Description
size	number of tasks enumerated by the map
coordinates	boundaries of the map
in_map	evaluates the presence of a task id or point
next	next point on the map
prev	previous point on the map
coordinate_index	map a point to a task id
index_coordinate	map a task id to a point

Algorithm 7 Identity::index_coordinate(id)

1: **return** Point(id)

3.3.2 Mapping

The class *Graph* defines a distributed container of tasks and encapsulates a *storage structure*, an *allocation scheme*, and a *mapping of tasks* to processes in the form

$$m : [0..t-1] \rightarrow [0..p-1]$$

in which t and p are the number of tasks and the number of processes respectively. As illustrated in Figure 3.8, the mapping, which is defined by the *Distribution* class, is decoupled from storage management, which is defined by the *Graph* class. A distribution is in fact associated with a *Graph* when the *Graph* is constructed. In this way, it is possible to combine any *Graph* implementation to any *Distribution* implementation.

Table 3.5: Virtual methods of class Distribution.

Method	Description
locate	process where a task resides
range	number of tasks mapped to a process
update	update the distribution

Tarragon provides two concrete distributions: *RegularDistribution* and *Vari-*

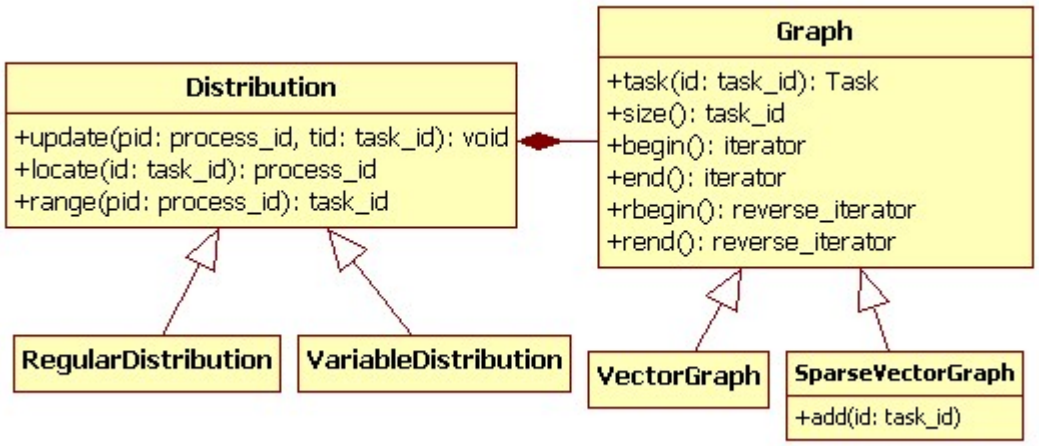


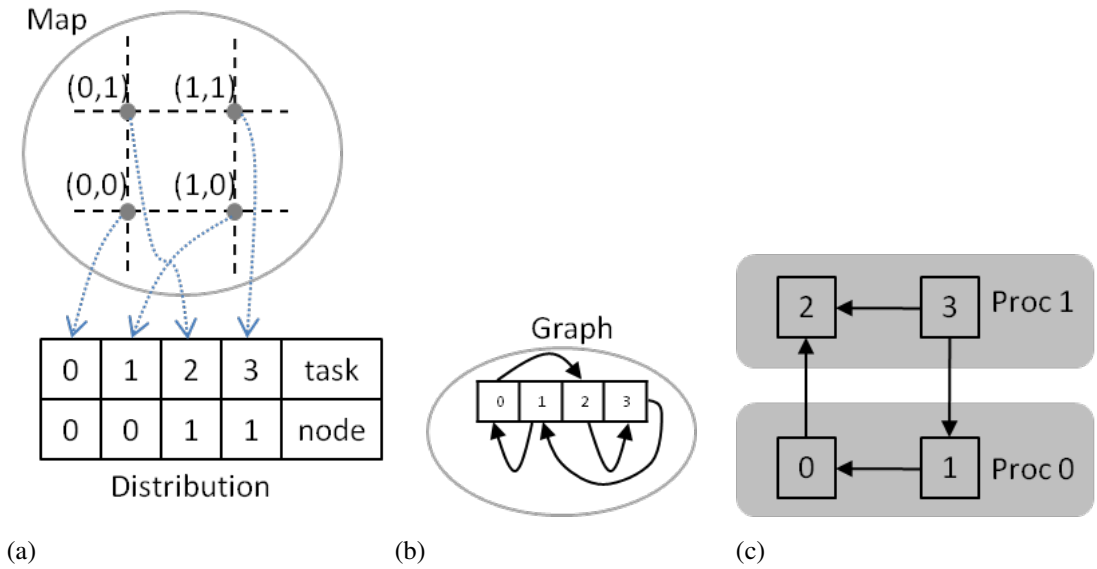
Figure 3.8: Class diagram of Graph.

ableDistribution. With *RegularDistribution* task ids are mapped to processes in consecutive blocks and evenly distributed between processes. For example, assuming t tasks and p processes such that t is a multiple of p, tasks $[0..\frac{t}{p}-1]$ are mapped to process 0. In the case that t is not a multiple of p, an extra task would be assigned to the first $t \bmod p$ processes. Instead, *VariableDistribution* does not attempt to distribute tasks evenly but still uses consecutive blocks of ids. The blocks are defined via *update*: by invoking *update* users set the boundaries of a block of ids, once for each process.

Figure 3.9 illustrates an example where a graph of 4 tasks is distributed across two processes. The tasks are associated to a 2-dimensional domain and their coordinates are defined to match the problem domain. The *Map* used in the example enumerates tasks lower row first, left to right. The tasks of the example are also connected to form a ring clockwise. Finally, the tasks are distributed between processes using *RegularDistribution*.

By default, *Graph* uses *RegularDistribution*. Users can define new and more complex distributions by extending *Distribution* and overriding its methods. The methods of *Distribution* are summarized in Table 3.5.

While *Map* connects the problem domain to Tarragon's task space, *Distribution* defines a mapping of the task space to processes. To a first approximation, *Map* could be used to implement mapping and load balancing by carefully assigning ids to tasks



(a) (b) (c)

Figure 3.9: Example of *Map*, *Distribution*, and *Graph*. Figure 3.9a illustrates the enumeration, represented by dotted lines, and the distribution, represented by a table. Figure 3.9b illustrates the graph and the dependencies between tasks. Figure 3.9c illustrates the resulting deployment on two processes.

and using either *RegularDistribution* or *VariableDistribution*. Often, this may be a necessary solution in regular problems when tasks perform the same amount of work and *RegularDistribution* trivially achieves optimal load balancing. However, when a complex distribution is required to achieve load balance, it may be convenient to define an ad-hoc *Distribution*. In this way, problem decomposition, defined by *Map*, and distribution, defined by *Distribution*, are decoupled and addressed separately.

Graph is also an abstract class and requires implementations that specify different allocation strategies and data structures. For example, Tarragon provides two concrete implementations: *VectorGraph*, that implements a distributed vector container, and *SparseVectorGraph* that implements a distributed vector container that allows for gaps. When instantiated, *VectorGraph* also instantiates all the tasks of the graph, since the number and the distribution of the tasks is known. However, *SparseVectorGraph* is useful in the case that a distribution method is known but it is not known which tasks should be actually instantiated. This could be the case, for example, in sparse linear algebra applications where the number of tasks instantiated depends on the sparsity

Table 3.6: Virtual methods of class Graph.

Method	Description
size	total number of tasks
task	task reference
begin	graph iterator
end	graph iterator end
rbegin	graph reverse iterator
rend	graph reverse iterator end

pattern of the data. To accommodate such situations, *SparseVectorGraph* requires the application to instantiate the tasks and add them to a *SparseVectorGraph* via its *add* method.

Table 3.6 lists the methods of *Graph*. Besides access to tasks by their id, *Graph* supports iterators. There are two types of iterators: *Graph::iterator* and *Graph::riterator*. The former is a forward iterator and iterates through all the local tasks in increasing id order, the latter is a reverse iterator and iterates through all the local tasks in decreasing id order. Iterators are instantiated via *begin* and *rbegin* methods and they iterate over tasks by increment (++). Forward and reverse iterators must stop when they are equivalent to *end* and *rend* respectively.

As an example, Algorithm 8 revises the graph construction in the ring example providing more details than in Chapter 2. Notably, Algorithm 8 uses concrete implementations of class *Map* and *Graph*. In addition, the loop that connects the tasks of the graph is implemented using a forward iterator. Using the iterator is convenient because it visits the tasks local to a process without the need to express locality in user code.

3.3.3 Execution and Data Motion

Execution and data motion are defined mostly within *Task*. With the notable exception of *Message*, all the methods that the RTS uses to activate and to interact with the application are defined in *Task*.

Algorithm 8 Ring Program

```
1:  Tarragon::initialize();
2:  Tarragon rts = Tarragon::tarragon();
3:  Map map = new Identity(n);
4:  Graph graph = new VectorGraph(map);
5:  Iterator i = graph.begin();
6:  while i != graph.end() do
7:      i->connect(map.next(i->id))
8:  end while
9:  rts.initialize_graph(graph)
10: rts.execute_graph(graph)
11: Tarragon::finalize()
```

Task

Users define the core of the computation within a *Task* subclass, specifying firing rules and the operations that a tasks carries out. A subclass of *Task* is therefore characterized by its virtual methods and the application programmer must extend the *Task* class to define a concrete subclass. The user-defined task inherits the virtual methods to be overridden, and such methods are the interface that the RTS uses when managing task execution and data motion. The RTS executes application code by invoking the virtual methods of *Task*. The virtual methods of *Task* are listed in Table 3.7.

Table 3.7: Virtual methods of class Task.

Method	Description	Default
vinit	initialize task	set EXEC state
vinject	inject data into task	delete message
vexecute	execute task	set DONE state
vcreate	create message	create a buffered message
vdestroy	destroy message	delete message
vterminate	notify termination	no action

The state of a task regulates the interaction with the RTS. A task is a state machine whose transitions are triggered by method invocation. The RTS inspects the *_state* attribute, encoding the observable state of the task, and invokes *Task* methods accordingly: *vinit* is invoked when a task is in the *INIT* state and the graph is initialized; *vexecute* is invoked when a task is in the *EXEC* and the graph is executed; and *vinject* is executed when a task is in *WAIT* state and a message must be delivered to the task. Figure 3.10 illustrates the states of *Task*, the transitions between states, and the corresponding methods causing the transitions. As part of their implementation, methods of concrete *Task* instances change the observable state. For example, method *vexecute* should set the *_state* to either *WAIT* or *DONE*, such that after execution, the task is either waiting or completed and ready to be retired. If a task sets its state to *ERROR*, then the computation aborts.

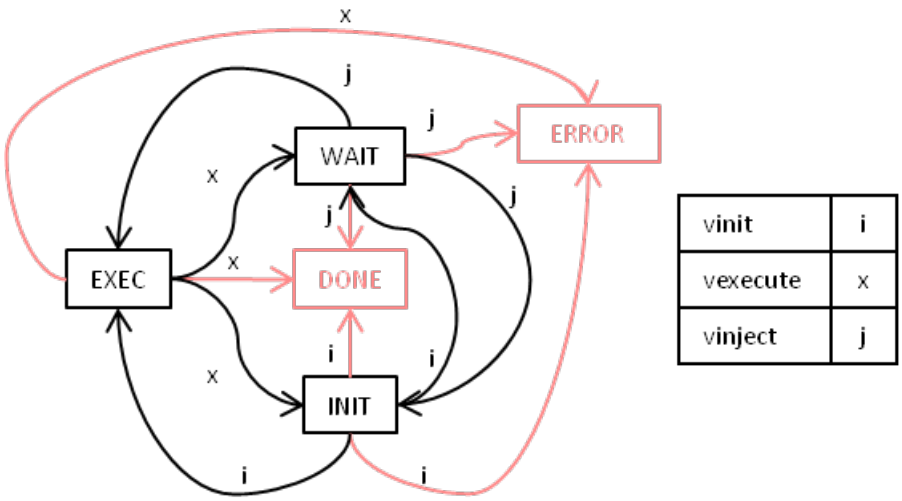


Figure 3.10: States and transitions. The lines denote state transitions between the possible states of a Task. The labels denote the method causing the transition. Notably, ERROR and DONE are reachable from all the other states and are final states.

Task execution, which occurs at the task level of execution, is implemented by *vinit* and *vexecute*. *vinit* is invoked when the graph is initialized. Defining *vinit* is optional and is employed only when there is a logical distinction between operations that are part of a task initialization and operations that are part of the actual computation. In addition, initialization via *vinit* takes place after the tasks have been connected, allowing

for parallel initialization involving communication between tasks.

In the case of the ring example, initialization is a simple operation and yet defining *vinit* contributes to make the code of RingTask easier to understand. Algorithm 9 shows the implementation of *vinit*. Task 0 is the first of the ring and the one that instantiates and sends a message around the ring. Therefore, task 0 sets its state to ready for execution (*EXEC*) and allocates the message. Every other task waits (*WAIT*).

Algorithm 9 RingTask::vinit

```
1: if id==0 then
2:     msg = make_message<BufferedMessage>(size);
3:     →EXEC;
4: else
5:     →WAIT;
6: end if
```

When a task is ready, it is scheduled for execution and eventually assigned to a worker that invokes *vexecute*. The method *vexecute* represents task execution: it encapsulates the kernel of the computation.

The last method related to task execution is *vterminate*. *vterminate* is invoked when there are only waiting tasks and no RTS instance has tasks running. Tarragon detects this state of quiescence and uses *vterminate* to notify each waiting task that graph execution has completed. Tarragon cannot distinguish a deadlock from completion, but the application developer can implement error handling operations in *vterminate*, to detect and recover from a deadlock.

In order to form a graph, tasks are connected by *Dependency* objects. *Dependency* objects are not directly instantiated by the programmer but they are instantiated via the *connect* method. When invoked on a task, *connect* instantiates a *Dependency* representing the dependence between the task and the argument:

$$task.connect(some\_task)$$

.

During execution, tasks exchange information by sending messages via their dependencies. The method *put*, which is a method of *OutDependency* is the only method

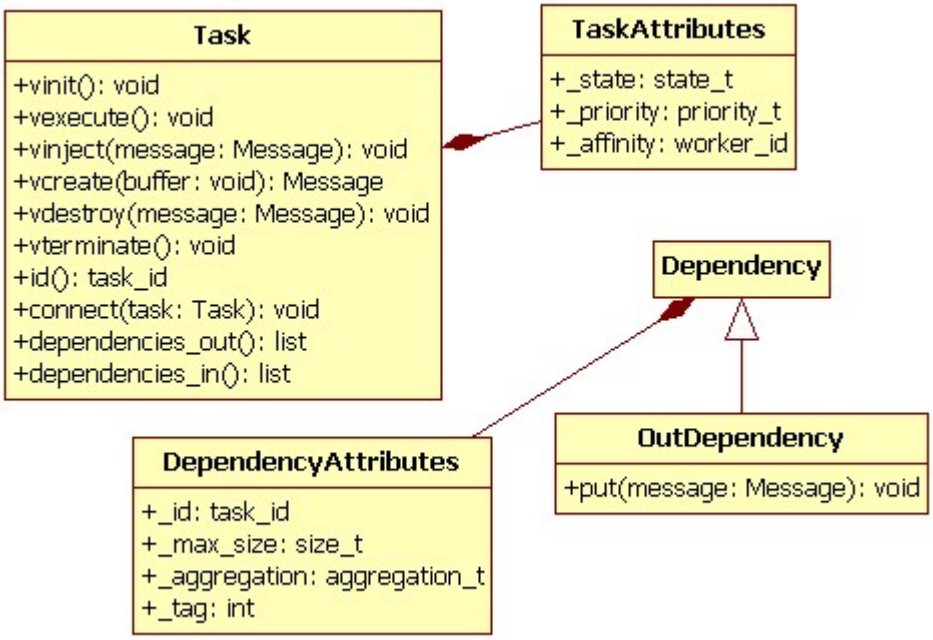


Figure 3.11: Class diagram of Task and Dependency.

that triggers data motion and it is asynchronous. When a message is sent via the *put* method (see Algorithm 3, line 10), it is eventually delivered to the target of the dependency. For each invocation of *put* there is a corresponding invocation of *vinject*, a short message handler that is invoked when a message arrives at a task. *vinject* implements the firing rule of the task. By checking the state of a task, *vinject* determines whether all the dependencies have been satisfied and the task is ready for execution. *vinject* is intended to implement the firing rule, which should be a short handler, and should not carry out extensive computations.

Memory-to-memory copies are expensive and often unnecessary during message transmissions. When receiving messages from other processes, the RTS avoids extra memory copies by using RTS-allocated buffers. Similarly, the RTS also uses application-allocated buffers when sending messages to other processes.

To simplify memory management and enable applications to have complete control over memory allocation, Tarragon relies on two methods of *Task*:

vcreate is invoked whenever a message is received from a task residing in a differ-

ent process. Data is received in serialized form and the RTS invokes *vcreate* to initialize a message.

vdestroy is invoked whenever a message is sent to a task residing in a different process. After a data transfer, Tarragon uses *vdestroy* to notify the sending task that the message previously *put* is no longer required[3].

By interacting with the application through *vcreate* and *vdestroy*, Tarragon delegates memory management to the application. However, the RTS is capable of managing memory as well. As long as a *Task* subclass does not redefine *vcreate* and *vdestroy*, the default implementation applies. When a message is received, the default *vcreate* leaves the message in an RTS-allocated buffer. When a message is sent, the default *vdestroy* determines whether the message is on an RTS-allocated buffer, and in that case it returns the message to the RTS.

The class diagram of *Task* and *Dependency* is illustrated in Figure 3.11. In addition to the classes, the diagram shows the attributes of both classes. Besides state attributes, *Task* has *priority* and *affinity* attributes: *priority* defines the scheduling priority and *affinity* defines the worker affinity. *Dependency* has three attributes: *max_size*, *aggregation*, and *tag*. The *max_size* attribute optionally[4] defines the capacity of the *Dependency*, that is, the largest amount of the data that can be sent along the edge at once. When the graph initialized, the RTS inspects *max_size* to determine the size of its communication buffers.

The *aggregation* attribute enables data aggregation. When aggregation is enabled, the RTS attempts to pack messages together. For example, when sending many small messages, enabling aggregation may reduce the cost of communication because the RTS can perform fewer transfers moving data in larger blocks. The *tag* attribute is used to notify the RTS when edges carry the same data. If the edges are not tagged, the RTS cannot determine when the same message is sent across multiple edges and requiring multiple copies of the same data. However, when edges are tagged, the RTS

[3]Data transfer has been completed although there is no guarantee that the message has been delivered. The semantic of communication is defined by the underlying communication substrate which may not guarantee delivery.

[4]When max_size is not specified, a default size is utilized. Messages that are longer than the default size will cause a communication error.

automatically sends the same message across equally tagged edges avoiding unnecessary memory copies.

Message

Message is a class that defines data to be transferred between tasks. A *Message* has three parts: data (payload), an *Envelope*, and a *Label*. In addition, as illustrated in Figure 3.12, a *Message* defines the *serialize* method. When a message object is not stored as a consecutive sequence of bytes, the RTS uses *serialize* to lay out the message as required.

Message has a data reference in the *Envelope* structure and metadata defined in the *Label*. The type field of the *Label* defines the type of the message, determining how the message will be treated by the RTS. There are three properties encoded in the type: whether a message needs to be serialized or it is already stored contiguously in memory, whether a message is stored in memory allocated by the application or by the RTS, and whether a message can be aggregated to other messages. For convenience, the label carries also a *key* field, which is a value for user-defined metadata. For example, by using *key* to tag messages, a programmer can define dynamic dataflow semantics [Arv90, J. 85]. Applications can also define different classes of messages and use *key* as a type identifier. In this case, when a message is received in serialized form, the type identifier is used to select the initialization procedure.

In the Core API there are two types of messages: *BufferedMessage* and *WrappedMessage*. A *BufferedMessage* is stored sequentially in memory and as such, it does not require serialization. The *WrappedMessage* class defines a message wrapper with a generic pointer to a data buffer. *WrappedMessage* handles serialization automatically via its *serialize* method. The serialize method of *WrappedMessage* first writes all the metadata into a communication buffer, and then it appends on the same buffer data referenced by the generic pointer. A programmer can define additional message classes by extending *Message* or one of its subclasses. The newly defined class can have any structure as long as it overrides the *serialize* method or it is stored sequentially as, for example, *BufferedMessage*. The class hierarchy of the predefined message classes is illustrated in Figure 3.12.

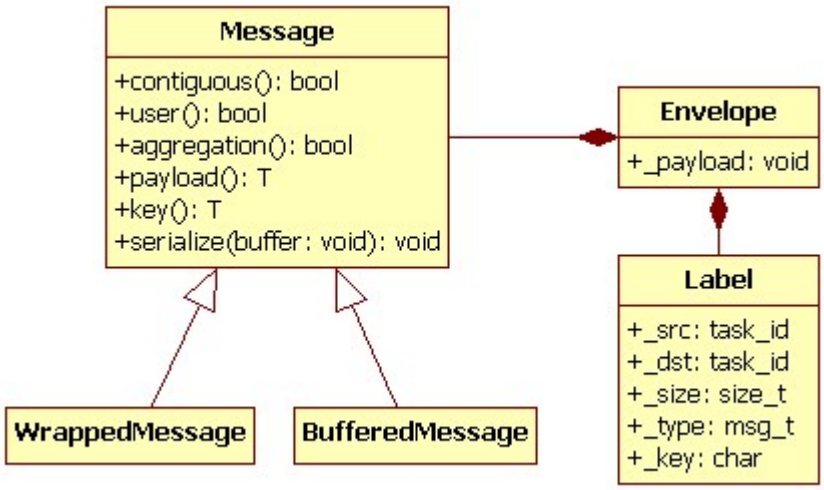


Figure 3.12: Class diagram of Message.

3.3.4 Error Handling

When an error occurs, the RTS raises an exception. Figure 3.13 illustrates the types of exceptions in Tarragon. Tarragon defines four concrete exception classes *CommunicationException*, *ThreadException*, *ExecutionException*, and *AllocationException*. All four are subclasses of *TException*, the base abstract class of the exceptions in Tarragon. *TException* contains an error code and a description of the error that subclasses fill according to the type of error.

CommunicationException is thrown whenever an error occurs in the underlying communication layer.

ThreadException is thrown whenever an error occurs in the underlying threads library (e.g. PThreads).

AllocationException is thrown when memory allocation within the RTS fails.

ExecutionException is thrown when an error occurs in the execution of a task and the task is in error state. In this case, the error code is the id of the task causing the exception.

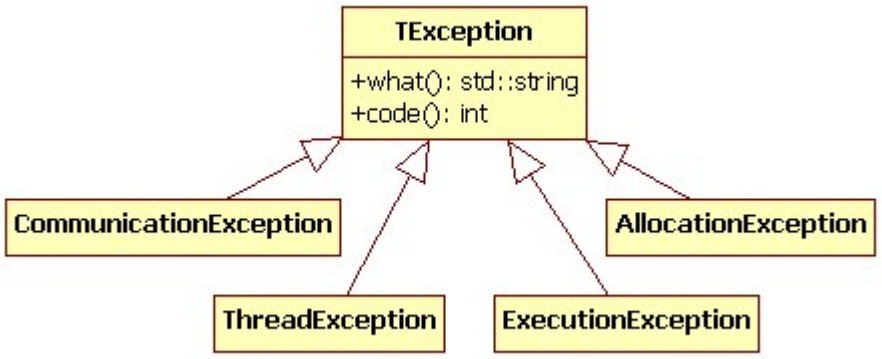


Figure 3.13: Class diagram of the exceptions hierarchy.

As an example, Algorithm 10 illustrates how to catch communication errors during execution and print an error message.

Algorithm 10 Ring Program

```
1: try
2:     rts.execute_graph(graph)
3: catch CommunicationException e
4:     print "Communication error" + e.what()
5: endtry
```

3.4 Extended API

Tarragon is designed to facilitate extensibility, allowing new functionality to be layered on top of the basic abstractions. The goal is to raise the level of abstraction to that of the application and hence simplify the application development. Tarragon imposes virtually no restrictions on task subclasses. This flexibility is inherited by the C++ language and it is not hindered by the design of Tarragon. In fact, *Task* can be simply used as an adaptor class [Gam02], a proxy class [Gam02], or some other kind of interface between Tarragon and the application[5]. Therefore, programmers are free to

[5] Adaptor and proxy are two design patterns that decouple an implementation from its interface. In this case, the task would be a simple interface to the application.

reuse existing code and libraries, even those written in different languages.

One way of extending the library is to enhance the support for constructing graphs. In particular, to define specialized tasks, maps, and helper functions for connecting tasks according to specific dependence structures. The *Extended API* of Tarragon comprises added functionality for constructing and optimizing commonly used graphs. While certainly useful for a number of problems, the Extended API presented in this dissertation serves primarily as an illustrative example of how it is possible to extend the classes in Tarragon and to build domain-specific libraries. Such *Domain-Specific Extensions* (DSE) to the library provide ready-to-use classes for solving problems within a specific domain.

3.4.1 Graph Analysis

In order to support additional functionality and algorithms, the objects of a graph are *element* according to the *visitor* design pattern [Gam02]. The visitor design pattern provides a way to define operations on data structures that separates the implementation of the operations from the implementation of the data structure. In this way, it is possible to extend the functionality of the graph by creating new visitors, without affecting the interface of existing objects, such as *Task* and *Graph*, on which the library depend. In addition, the visitor design pattern makes it possible to separate the implementation of the visitors from the implementation of user-defined tasks. As a result, visitors can be defined for generic task structures and then applied to graphs of a user defined task.

To realize the pattern, the *Extended API* defines interfaces to the data structure classes and the operation classes. By agreeing on the interfaces, it is possible to have visitors and elements interact through call back functions. The data structure elements implement an interface with an *accept* method that the visitor uses to notify its intent to visit the element. When notified, the element invokes the *visit* method of the visitor that is appropriate for its type. Figure 3.14 illustrates the implementation of the visitor design pattern in Tarragon and the classes that implement the *Element* interface. Specifically *Graph*, *Task*, and *Dependency* implement the *Element* interface, which defines the *accept(Visitor)* method. Visitors must implement the *Visitor* interface including the overloaded visit method (*Visitor* takes any one of *Graph*, *Task*, or *Dependency*as argu-

ment).

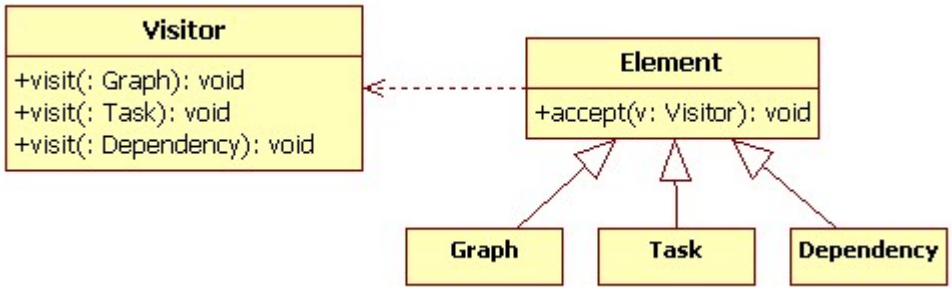


Figure 3.14: Class diagram of the visitor design pattern in Tarragon.

One use of visitors is to connect the tasks. A *Visitor* can start from *Graph* and then iterate through all the tasks connecting them according to the desired topology. In addition, once the graph is built and the tasks connected, visitors can traverse the graph and set task priorities, affinities, and other relevant attributes, or collect information on the graph.

Tarragon also takes advantage of the visitor pattern. For example, when initializing the graph, the RTS visits the graph to determine the size of any required communication buffer. The visitor traverses the graph, and examines all the dependencies of each tasks. At the end, the visitor can determine the size of the largest message and allocate buffers that sufficiently large.

The visitor is defined by the three visit methods in Algorithm 11, Algorithm 12, and Algorithm 13. A *Visitor* starts from an instance of *Graph*, and recursively calls *accept* on each task of the graph; on *Task*, *Visitor* recursively calls *accept* on the dependencies of the graph; on *Dependency*, *Visitor* simply updates the value of the *max_size* variable. When the visit completes, *max_size* will store the max of the *max_size* attributes.

Algorithm 11 SizeVisitor::visit(graph)

```
1: max_size=0;
2: for all task∈graph do
3:     task.accept(this)
4: end for
```

Algorithm 12 SizeVisitor::visit(task)

```
1: for all dependency∈task do
2:     dependency.accept(this)
3: end for
```

Algorithm 13 SizeVisitor::visit(dependency)

```
1: update_max_size(dependency.max_size)
```

3.4.2 Performance Tuning

Tuning application performance involves evaluating design decisions and parameter choices that affect performance, and discovering how they can be changed to improve performance. One of the most important parameters is the *granularity* of the decomposition, which is defined as the amount of computation carried out by each task relatively to the amount of data transferred in each communication event. Granularity is a critical parameter because it is strongly related to both the available parallelism and the incurred overhead: fine-grained computations typically expose more parallelism because there are more small units of computation that can be executed concurrently, but they incur at a higher overhead. For example, the cost of fine-grain communication tends to be dominated by latency rather than by bandwidth. The most profitable trade-off defines the optimal granularity. In SPMD models, a first gross decomposition is imposed by the programming model because data must be split between the processes. Tarragon relaxes this restriction. The number of tasks does not have to exactly match the number of physical processor cores; programmers can focus on a logical decomposition of the work while trying to achieve the best performing granularity.

Carefully mapping tasks to processes is also a crucial aspect of parallelization. A good mapping minimizes communication among tasks residing in different processes [Sha81]. Several other criteria may apply, but the common principle is always the same: try to maximize locality. In Tarragon, it is possible to formally define mapping strategies via a *Distribution* subclass. It is also possible to visit the graph and analyze its connectivity to infer a better mapping. Similarly, after executing a graph, the application could improve the mapping based on retrospective information collected during a previous execution and implement off-line load balancing.

There are also opportunities for preserving locality within each process; for example, by scheduling back to back tasks related by data dependencies [Suv99]. The RTS tries to preserve locality in this way and also by giving precedence to tasks that are executed recently. Also, when multiple ready queues are used, it assigns affinities to tasks that express affinity for a specific worker. In addition, a programmer can define tasks priorities and affinities to better match application-specific patterns and improve locality further.

Finally, while Tarragon automatically overlaps communication with computation, it is possible to change the schedule, using priorities, to improve latency hiding. For example, the Extended API defines the *OverlapVisitor* that prioritizes execution of tasks with dependencies on tasks residing in different processes. The idea is to initiate data transfer as soon as possible [Cos05] in order to maximize overlap opportunities. The implementation of the *Visitor* method in *OverlapVisitor* is given in Algorithm 14.

Algorithm 14 OverlapVisitor::visit(task)

```
1: for all dependency∈task do
2:     priority += graph.is_local(dependency.id())
3: end for
```

3.4.3 Domain Specific Extensions

A Domain Specific Extension (DSE) to the library is a set of classes and functions that raise the abstraction to the application level by defining types and operations specific to the application domain.

The goal of a DSE is to improve productivity. By defining high level abstractions, a DSE presents a familiar interface to the application developer while hiding lower-level implementation details. Such layered software architecture promotes software reuse: lower layers define more general and widely used abstractions while specialization increases at the higher levels. In a similar fashion, a developer can create performance optimization libraries targeting certain classes of applications.

Maps and Spaces

Cartesian spaces are often used in scientific computing when the physical problem space is conceived and discretized as a subset of Z^3. Also, matrices can be represented conveniently as a rectangular domain of points in two dimensions.

To support higher level abstractions for rectangular domains, the *Extended API* of Tarragon defines an abstract class, called *Space*. A *Space* represent data and, the association of a *Space* with a *Map* induces a decomposition of *Space* into subspaces, in which the subspaces are associated to the tasks of the *Map*.

Figure 3.15 illustrates the example of a 2-dimensional *Space* represented by a 6×6 mesh. The mesh is associated with a 2×2 *Map* resulting in a decomposition into subspaces and a mapping between subspaces and tasks; each task is associated with a 3×3 section of the mesh.

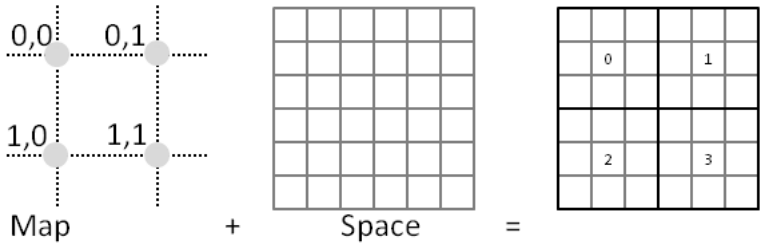


Figure 3.15: Association of Space and Map.

The *Extended API* of Tarragon defines several subclasses of *Map* and *Space* to support cartesian and rectilinear geometries. The use of these classes is illustrated in the applications presented in this dissertation. A complete reference is given in Appendix B.

Connectors

The *Extended API* provides visitors to automatically connect neighboring tasks, a scenario which is common in many communication patterns. Given a graph, a *connector* visitor traverses the graph connecting tasks according to a certain pattern. For example, a connector that connects each task to its successor creates a uni-directional ring on a 1-dimensional map. The *OneNeighborForwardConnector* visitor is such a connector. By using *OneNeighborForwardConnector*, the loop that connects the tasks in Algorithm 8 (lines 5-8) may be replaced by the following statements:

$$OneNeighborForwardConnector\ c;\ graph.accept(c);$$

A complete list of the connectors provided by the *Extended API* is given in Appendix B.

3.5 Performance Evaluation

Tarragon creates the graph abstraction by adding a software layer between application and low-level system libraries. In particular, data motion between virtualized tasks is built on top of a lower level communication substrate (in the current implementation the communication substrate is an MPI library). As a result, applications experience increased communication delays due to software overheads.

The ring example is used as a benchmark to measure communication latency overhead and bandwidth losses in Tarragon compared with an implementation of the ring in MPI, on which Tarragon relies to carry out inter-process communication. The ring benchmark creates a set of tasks (processes in the MPI implementation) forming a communication ring and circulate data around the ring. By timing a number of complete loops it is possible to determine the achieved bandwidth and latency. The experiment is then repeated with different message sizes to collect results across a range of message length scales. In particular, experiments to obtain a latency measure use small messages, for which latency is dominant, and experiments to obtain the peak bandwidth use a range of large messages. Algorithm 15 illustrates the MPI implementation of the ring program.

Algorithm 15 MPI Ring Program

```
1:  while trips do
2:    trips=trips-1
3:    if myrank then
4:      MPI_Recv(buffer,size,prevrank)
5:      MPI_Send(buffer,size,nextrank)
6:    else
7:      MPI_IRecv(buffer,size,prevrank)
8:      MPI_Send(buffer,size,nextrank)
9:      MPI_Wait()
10:   end if
11: end while
```

Overheads are evaluated both within a node and between nodes. In the former case, tasks (processes in the MPI version) are instantiated on the same node, whereas in

the latter case only one task is instantiated on a node. In addition, tests with Tarragon are repeated using a multi-threaded RTS and a single-threaded RTS. Tarragon usually runs multi-threaded, that is with one service thread and several worker threads. Thus, it occupies all the available cores on a shared-memory node with a RTS instance. However, it may be convenient in certain cases to execute a RTS on each core, that is with only one thread executing both services and tasks. For example, in applications combining existing MPI software with Tarragon software, such mappings may be imposed by the MPI portion of the software. The configurations described, for both MPI and Tarragon, are illustrated in Figure 3.16.

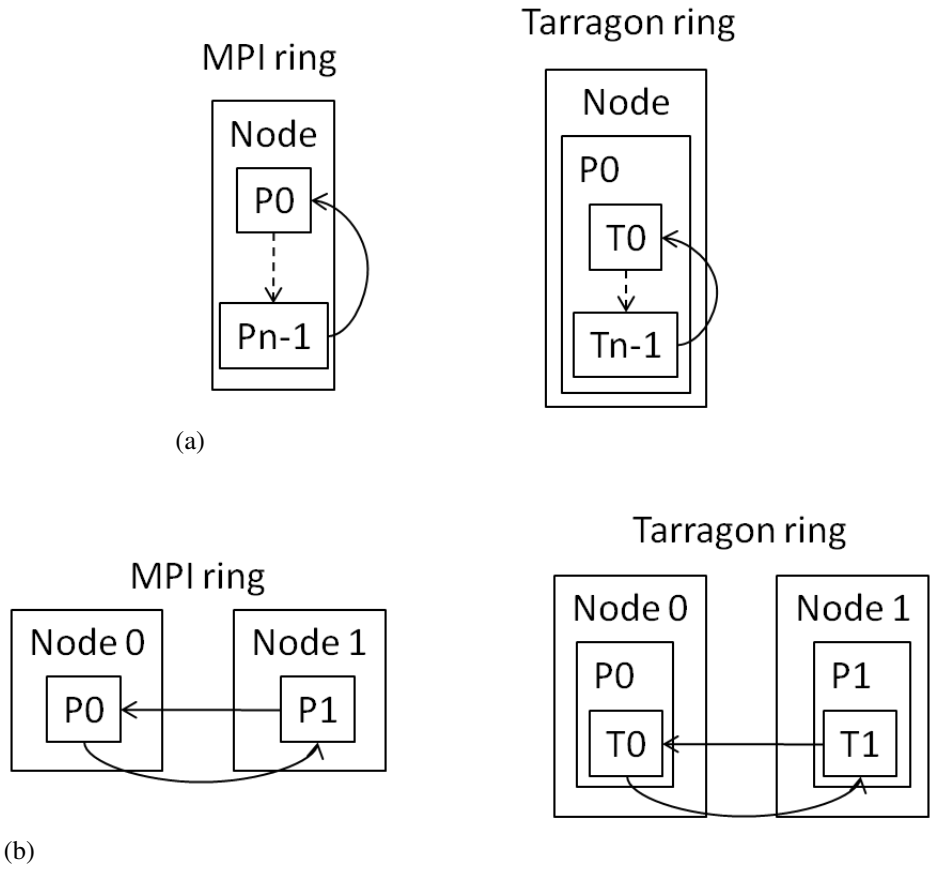


Figure 3.16: Ring configurations. In the single-node configuration, MPI processes (tasks in the Tarragon implementation) all execute on the same node, as illustrated in Figure 3.16a. In the multi-node configurations, there is one MPI process (task in the Tarragon implementation) per node. Figure 3.16b illustrates the two-node configurations.

Tests are performed on Abe, the Intel cluster at the National Center for Supercomputing Applications [Nata], and on Kraken, the Cray XT5 at the National Institute for Computational Sciences [Natb]. Details of both machines are given in Appendix A.

3.5.1 Latency

Latency is defined as the time elapsed from the beginning of a data transfer, on the sending side, until data arrival, on the receiving side[6]. With the ring benchmark, messages of length ranging from 1B to 16KB are transferred between tasks, and the wallclock time is measured for a hundred complete loops to average latency measurements and to spread the timing overhead.

Intra-node latency is measured with one task per core, or a process per core in MPI, within a node. In the MPI implementation, message transfer is performed via the communication stack of the MPI library implementation and involves copying the message in memory, between separate address spaces. In Tarragon, message transfer is performed within a single address space.

On a node of Abe, the measured latency of the MPI implementation is approximately 1.4 microseconds. In Tarragon, whether the RTS is single-threaded or multi-threaded greatly affects the outcome. In single-threaded mode, communication is very efficient and its latency is 0.4 microseconds, less than a third of the latency in the MPI version. The performance advantage in this case is given by the fact that the same thread executed all the tasks, a working set that fits in cache. In multi-threaded mode, the added cost of synchronization between threads and of data transfer between caches results in a 4.4 microseconds latency.

Figure 3.17a illustrates the transfer time as a function of message length on Abe. As the message length is increased, in multi-threaded mode the transfer time remains approximately constant, dominated by the overheads. In single-threaded mode, transfer time grows when data no longer fits in cache, but for messages up to 16KB long, transfer latencies are lower in single-threaded mode than in the MPI implementation. In the MPI implementation, transfer time increases gradually, according to the length of the

[6] Although latency is generally defined from the beginning of a data transfer until the first piece of data is received, in this context there is no distinction because data is delivered to the user space in toto.

message, and it is larger than with Tarragon in multi-threaded mode for messages that are 4KB or longer; the lower cost of copying memory within the same address space gives a performance advantage to the Tarragon implementation even in multi-threaded mode.

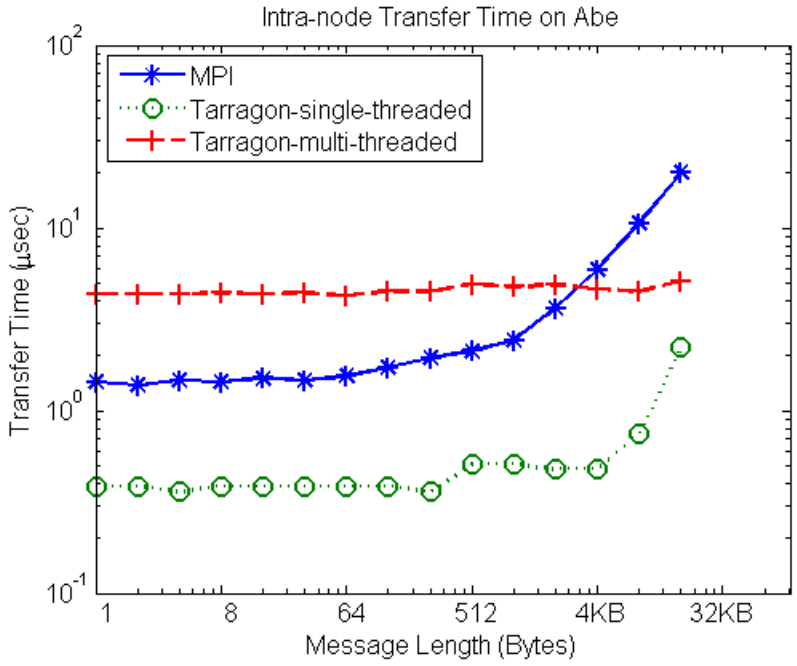
On Kraken, intra-node latency measured with the MPI implementation is approximately 0.6 microseconds. In the Tarragon implementation, in single-threaded mode the latency is 0.2 microseconds, whereas in multi-threaded mode is 4.5 microseconds. The effect of caching gives to the single-threaded a performance advantage, hence the low latency, whereas overheads penalize the multi-threaded mode. In the MPI implementation, transfer time grows proportionally to the message length, and at 8KB, transfer time is greater than in the Tarragon implementation running in multi-threaded mode. Figure 3.17b illustrates transfer time as a function of message length on Kraken.

Inter-node latency is measured by instantiating a task per node, or process per node in MPI, ensuring that every hop on the ring corresponds to a data transfer between nodes. Transfer time is measured using 2, 4, and 8 nodes.

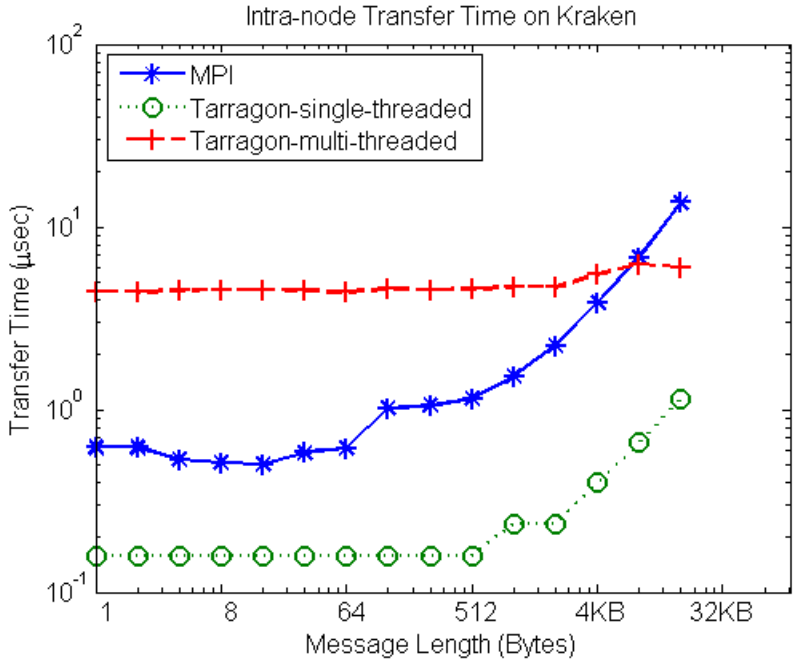
On Abe, the latency measured with the MPI implementation ranges from 5.6 microseconds to 8.3 microseconds, increasing slightly as more nodes are used. Latency in the Tarragon implementation in single-threaded mode, matches the latency in the MPI implementation. However, latency measured with the multi-threaded version is twice than with the MPI version due to overheads incurred because of threading; such overheads are relatively lower than within a node because between nodes transfer time is greater, and are only observable for messages that are 4KB or shorter. Figure 3.18a illustrates transfer times on 8 nodes of Abe.

On Kraken, overheads in Tarragon are higher than on Abe. On Kraken, the latency measured with the MPI version is 6.2 microseconds, whereas with Tarragon it is over 2 times higher in single-threaded mode, and almost 3 times higher in multi-threaded mode. However, the gap between transfer times narrows rapidly as message length is increased, and it becomes negligible for messages that are 8KB or longer.

A summary of latency and overheads is given in Table 3.8.

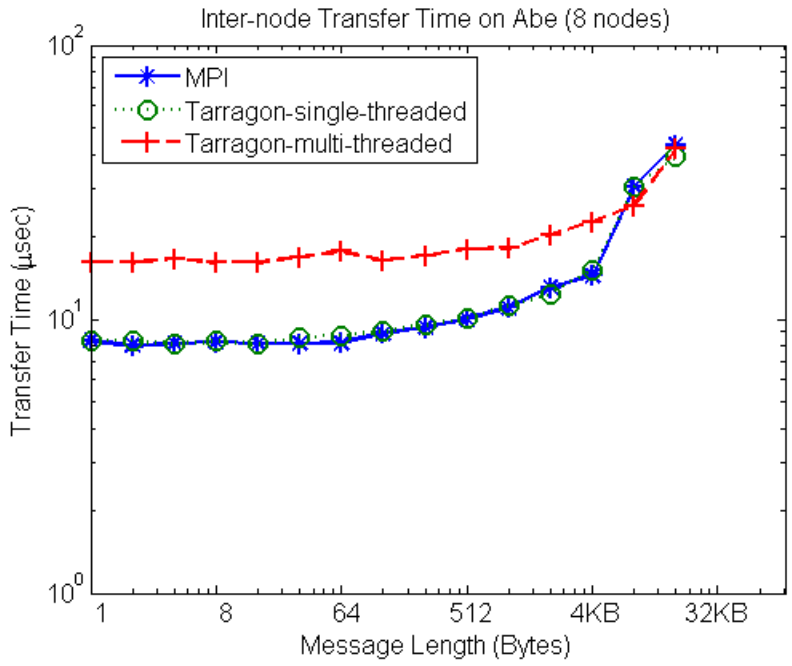


(a)

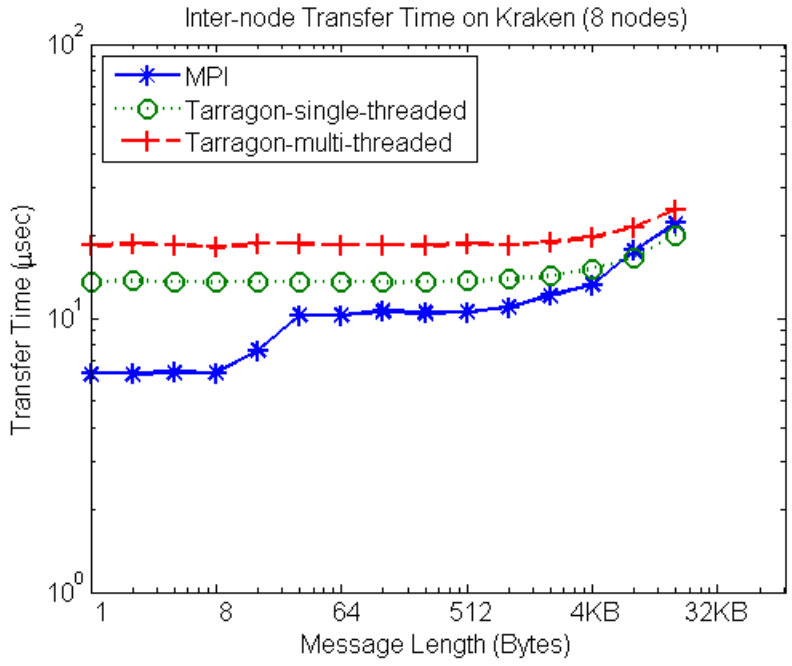


(b)

Figure 3.17: Intra-node point-to-point transfer time on Abe (Figure 3.17a) and on Kraken (Figure 3.17b).



(a)



(b)

Figure 3.18: Inter-node point-to-point transfer time on Abe (Figure 3.18a) and on Kraken (Figure 3.18b). Transfer time measured with a ring spanning 8 nodes.

Table 3.8: Transfer time and overhead on Abe and on Kraken. Overhead is reported relative to MPI measurements (e.g. 1.0x means no overhead). Single-node results are measured running a process/task per core.

Machine	Nodes	Latency (μsec)			Tarragon Overhead	
		MPI	Tarragon ST	Tarragon MT	ST	MT
Abe	1	1.4	0.4	4.4	0.3x	3.1x
Abe	2	5.6	5.6	10.5	1.0x	1.9x
Abe	4	7.7	7.7	14.2	1.0x	1.8x
Abe	8	8.3	8.3	16.1	1.0x	1.9x
Kraken	1	0.6	0.2	4.5	0.3x	7.5x
Kraken	2	6.2	13.5	17.4	2.2x	2.8x
Kraken	4	6.2	13.5	18.0	2.2x	2.9x
Kraken	8	6.3	13.5	18.2	2.1x	2.9x

3.5.2 Bandwidth

Bandwidth is defined as the data transfer rate in a communication channel; in the ring benchmark, the transfer rate between tasks and processes. Bandwidth is measured on messages of length ranging from 64KB to 64MB, ensuring that transfer time reflects bandwidth rather than latency.

Intra-node bandwidth is measured by instantiating a task per core, or a process per core in MPI, within a node. Intra-node bandwidth is bounded by memory bandwidth since the transfer takes place in memory and does not involve the interconnect. In the MPI implementation, messages are copied between processes, whereas in Tarragon, messages are copied within the same address space. As a result, the Tarragon implementations achieve higher bandwidth than the MPI implementation. On Abe, the MPI implementation achieves an 1.1GB/s peak bandwidth, whereas the Tarragon implementations achieve an 1.8GB/s peak bandwidth.

Also on Kraken, the Tarragon implementations achieve higher bandwidth than the MPI implementation. The Tarragon implementations achieve a 4.5GB/s peak band-

width whereas the MPI implementation achieves a 2.3GB/s peak bandwidth.

The peak bandwidth considered for the Tarragon versions is measured for the longest messages, because short messages fit in cache and the bandwidth achieved on messages up to 1MB long is even higher. However, for the sake of comparison, peak bandwidth on messages that are longer than 1MB provides a better indication of the achievable bandwidth. Intra-node bandwidth is illustrated in Figure 3.19. On both Abe and Kraken, the Tarragon implementations achieve higher bandwidth with small messages than with large messages, and the bandwidth decreases narrowing the gap with the MPI implementation.

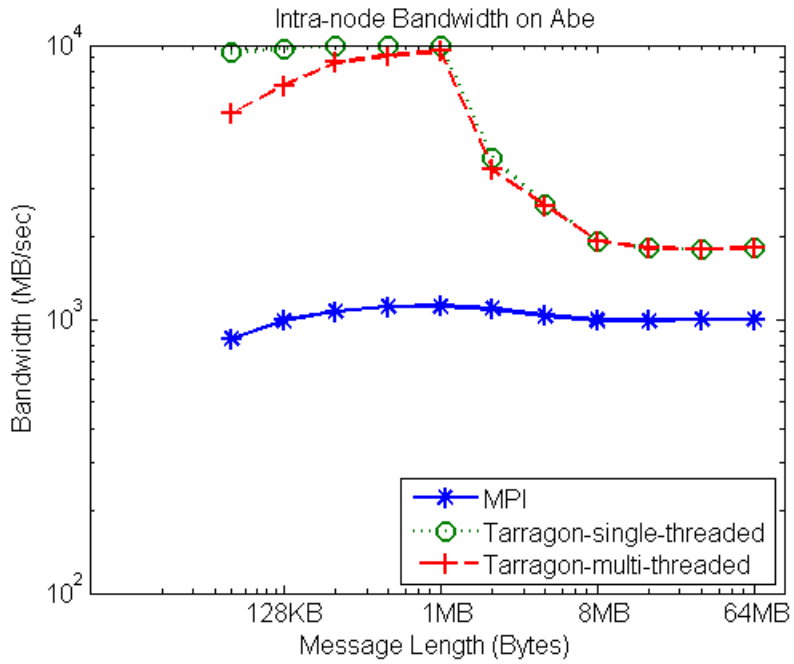
As for inter-node latency, inter-node bandwidth is measured instantiating a task per node, or a process per node in MPI, using 2, 4, and 8 nodes.

On Abe, the peak bandwidth measured with the MPI implementation ranged from 880 MB/s to 895 MB/s. No significant difference is observed in peak bandwidth when using a different number of nodes. Despite the overheads in Tarragon, there is no substantial difference in peak bandwidth. In fact, overheads only affect latency, and for long messages the impact on performance is negligible. Figure 3.20a illustrates bandwidth measured on 8 nodes of Abe, for increasing message length.

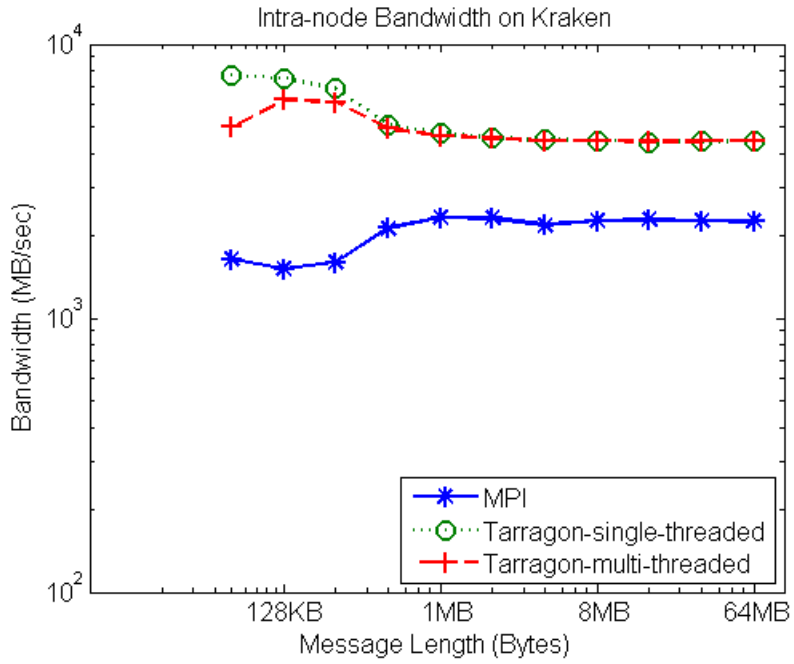
On Kraken, the peak bandwidth measured with the MPI implementation is approximately 1.6GB/s, which is matched by the Tarragon implementation. Also on Kraken there is no significant difference in peak bandwidth when using a different number of nodes. Figure 3.20b illustrates bandwidth measured on 8 nodes of Kraken, for increasing message length.

Table 3.9: Peak bandwidth and overhead on Abe and on Kraken. Overhead is reported relative to MPI measurements (e.g. 1.0x means no overhead). Single-node results are measured running a process/task per core. Peak bandwidth on 1 node, for the Tarragon implementation, is measured on messages that are 8MB or longer.

Machine	Nodes	Bandwidth ($\frac{MB}{sec}$)			Tarragon Overhead	
		MPI	Tarragon ST	Tarragon MT	ST	MT
Abe	1	1118	1820	1820	0.6x	0.6x
Abe	2	895	918	920	1.0x	1.0x
Abe	4	894	906	907	1.0x	1.0x
Abe	8	880	881	877	1.0x	1.0x
Kraken	1	2321	4450	4420	0.5x	0.5x
Kraken	2	1662	1610	1670	1.0x	1.0x
Kraken	4	1659	1610	1620	1.0x	1.0x
Kraken	8	1640	1580	1570	1.0x	1.0x

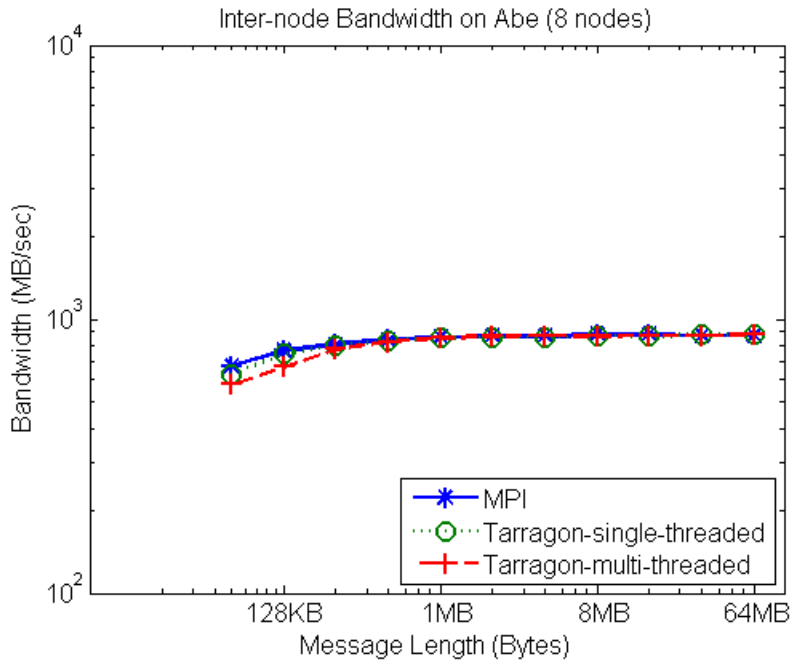


(a)

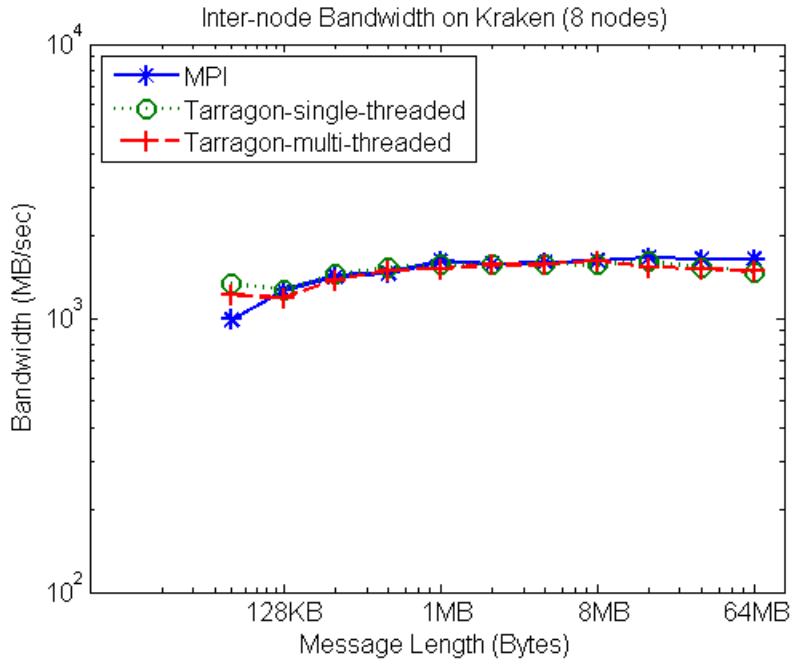


(b)

Figure 3.19: Intra-node point-to-point bandwidth on Abe (Figure 3.19a) and on Kraken (Figure 3.19b).



(a)



(b)

Figure 3.20: Point-to-point bandwidth on Abe (Figure 3.20a) and on Kraken (Figure 3.20b). Bandwidth measured with a ring spanning 8 nodes.

3.5.3 Discussion

Communication in Tarragon builds on top of communication substrate, which in the current implementation is MPI. Consequently, Tarragon introduces overheads that may affect performance. On the other hand, Tarragon tasks can live within the same address space and therefore communicate more efficiently than MPI processes within a node.

On both platforms, Tarragon has an advantage in intra-node communication. With Tarragon, tasks communicate sharing data within the same address space and, in some cases, benefit from cached data. However, MPI processes always involve copying across address spaces. As a result, when running in single-threaded mode, the intra-node latency in the Tarragon implementation is approximately a third of the latency in the MPI implementation, while it is larger when running in multi-threaded mode, due to thread synchronization. As message size increases, in both modes the Tarragon implementation outperforms the MPI implementation and achieves roughly twice as much bandwidth than the MPI implementation on the longest messages.

Overheads affect latencies also in inter-node communication. On Abe, in single-threaded mode overheads are not significant. However, in multi-threaded mode on Abe, and in both single-threaded and multi-threaded mode on Kraken, overheads result in approximately a 2 to 3 fold increase in latency. However, even for messages that are just tens of kilobytes long, the gap is almost negligible, and both implementations achieve the same peak bandwidth.

References

[Arv90] Arvind, K. and Nikhil, Rishiyur S. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39:300–318, March 1990.

[Bad00] Baden, S.B. and Fink, S.J. A programming methodology for dual-tier multicomputers. *Software Engineering, IEEE Transactions on*, 26(3):212 –226, March 2000.

[Cos05] Costin Iancu and Parry Husbands and Paul Hargrove. HUNTing the Overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society.

[Fow03] Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[Gam02] Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. *Design Patterns*, pages 331–344. Addison-Wesley Reading, MA, 2002.

[J. 85] J. R Gurd and C. C Kirkham and I. Watson. The Manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[Lim97] Lim, B.-H. and Heidelberger, P. and Pattnaik, P. and Snir, M. Message proxies for efficient, protected communication on SMP clusters. In *High-Performance Computer Architecture, 1997., Third International Symposium on*, pages 116 –127, Feb 1997.

[Mes94] Message Passing Interface Forum. MPI:A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4):165–414, 1994.

[Nata] National Center for Supercomputing Applications. Intel 64 Cluster Abe.

[Natb] National Institute for Computational Science. Kraken Cray XT5.

[Nic96] Nichols, Bradford and Buttlar, Dick and Farrell, Jacqueline Proulx. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[R. 96] R. D. Blumofe and C. F. Joerg and B. C. Kuszmaul and C. E. Leiserson and K. H Randall and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[Sha81] Shahid Bokhari. On the mapping problem. *IEEE Trans. Computers*, 30(3):207–214, 1981.

[Suv99] Suvas Vajracharya and Steve Karmesin and Peter Beckman and James Crotinger and Allen Malony and Sameer Shende and Rod Oldehoeft and Stephen Smith. SMARTS: exploiting temporal locality and parallelism through vertical execution. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 302–310, New York, NY, USA, 1999. ACM Press.

Chapter 4

Structured Grids

4.1 The Motif

Structured grids are used in scientific codes to represent a discretization of space using regularly spaced points. When solving Partial Differential Equations (PDE), structured grids can be used in both finite-volume and finite-element codes.

This dissertation will focus on finite-difference methods. In finite-difference codes the grid points are associated with field variables that are numerically differentiated, by means of a finite-difference method, to approximate functions and derivatives as they appear in an equation of interest.

Structured-grid and finite-difference computations are also used in the basic steps of sophisticated computational methods such as multigrid [Bra] and adaptive mesh refinement [Ber89], in which grids with different resolutions are used on the same domain. Without any loss of generality, this dissertation will consider single level meshes.

Structured-grid computations exhibit a high degree of locality and, because of their regularity, the data are stored in contiguous memory and accessed by indices. However, depending on the equation, structured-grid codes exhibit different ratios of floating point operations to memory accesses, and this ratio determines whether a kernel is compute bound or memory bound.

The communication pattern is relatively simple and is usually some form of nearest-neighbor communication pattern enabling boundary values exchange. Boundary values are exchanged after every update, a behavior that is an ideal candidate for Bulk

synchronous Parallel (BSP) formulations. During the computation phase, each process updates the local values; then, during the communication phase, processes exchange boundary values.

Overlapping communication in structured grid application has been subject of prior research, although it has been approached in most cases with solutions based on split-phase coding [Sco98, Bad00, Fin98].

4.2 A Jacobi Iterative Solver

The finite-difference model computation considered in this dissertation is an iterative solver for Poisson's equation in a three-dimensional domain with *Dirichlet* boundary conditions. Poisson's equation, which is shown in Equation 4.1, is a PDE that expresses a potential function, here denoted by u, in terms of a known source function, here denoted by v, on an open region $\Omega \in \mathbb{R}^3$. The value of the potential on the boundary is given by a known function f.

$$\begin{cases} \Delta u = v \; in \; \Omega \\ u = f \; on \; \partial\Omega \end{cases} \tag{4.1}$$

The solver considered implements Jacobi's method. Space is discretized and represented by a Cartesian grid, in which consecutive points are uniformly spaced with a distance h. The Laplacian (Δ) is approximated using a centered 7-point finite-difference stencil. To solve Equation 4.1, the stencil in Equation 4.2 is applied to each point of the grid.

$$u'_{i,j,k} = \frac{(u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} - h^2 v_{i,j,k})}{6}. \tag{4.2}$$

Two copies of the grid are stored in memory to support out-of-place updates, as illustrated by Figure 4.1a. Figure 4.1a shows an example of a $4 \times 4 \times 4$ grid with boundary points. The two grids represent values across two iterations. Each iteration computes a new set of values. The new values, which in this example are written on the grid on the right, are computed reading the values from the previous iteration, which

in this example are read from the grid on the left. The central point $v_{i,j,k}$ is read from a third grid that discretizes the source function v. Finally, before a new iteration takes place, the roles of the two grid copies is reversed.

In typical Single Program Multiple Data (SPMD) formulations, the grids are partitioned and each partition mapped uniquely to a process. With this partitioning scheme, in addition to the physical boundary values, additional grid points are necessary for the calculations on the internal boundaries. Such additional points, usually referred to as *ghost cells*, are copies of points that belong to neighboring processes. The values of ghost cells are refreshed between iterations.

Figure 4.1b illustrates the decomposition of a $4 \times 4 \times 4$ grid ($6 \times 6 \times 6$ including the boundaries). On a $2 \times 2 \times 2$ process geometry, each process owns a block of 8 points ($2 \times 2 \times 2$), plus boundaries and ghost cells. In practice, meshes would be much larger and the ghost cells would account for a much smaller fraction of the mesh. The ghost cells lie on the internal faces. Figure 4.1c shows an example of a block with boundaries and ghost cells; ghost cells are highlighted.

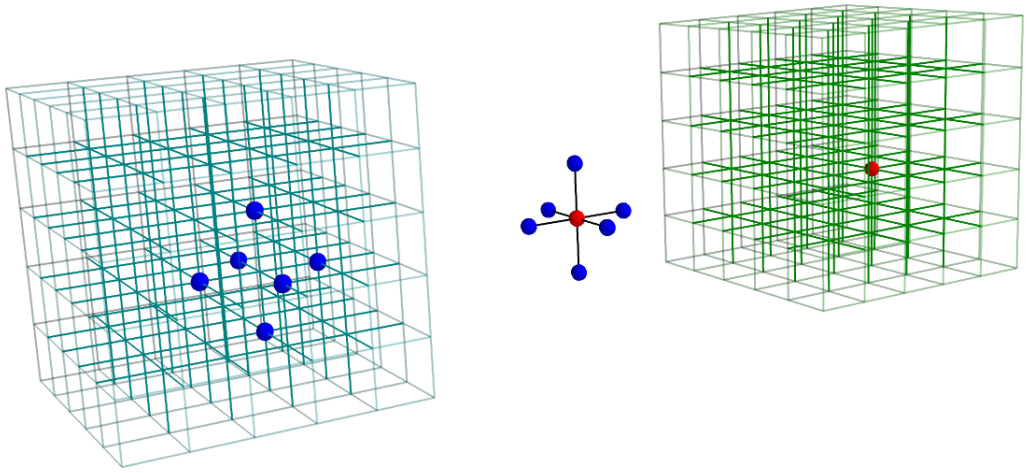
4.2.1 Reference Implementations

In this dissertation, experiments use two MPI implementations as reference: a *Synchronous* variant and an *Asynchronous* variant.

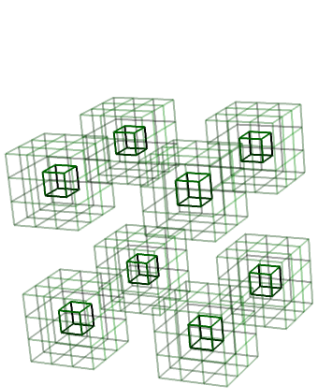
In the Synchronous variant, each MPI process owns and is responsible for updating a partition of the grid; between updates, ghost cells are exchanged synchronously. The kernel is implemented with a set of nested loops, as represented in Algorithm 16. The first loop (lines 3-9) is the iteration loop, that is, how many relaxation steps are executed[1]. Communication takes place at the beginning of each iteration (line 2). Then, the three nested loops iterate over each point of the $M \times N \times P$ local partition excluding ghost cells and boundaries. Finally, the two grid copies are logically swapped.

The Asynchronous variant is a split-phase reformulation of the Synchronous variant. Computation is divided into two phases: one to compute points in the inside of the local grid, and one to compute points on the surface, which are the points whose new values depend on the ghost cells. The Asynchronous variant appears in Algorithm

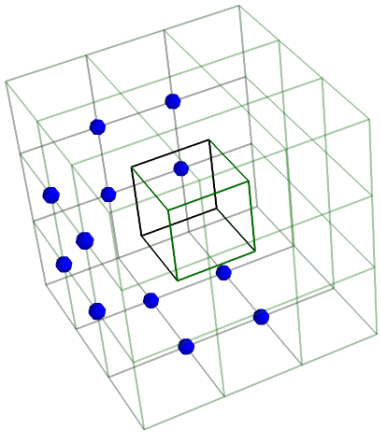
[1] Usually an additional condition ensures that, when the desired accuracy is reached, the loop exits.



(a)



(b)



(c)

Figure 4.1: Mesh, stencil, and decomposition. Figure 4.1a illustrates two meshes holding values for two iterations: current (left) and next (right). Applying the stencil (between meshes) to the values the previous iteration produces the updated value for the next iteration, which is the value at the center of the stencil. Figure 4.1b illustrates a regular 3-dimensional decomposition resulting in 8 blocks. The block at the top right corner is illustrated in Figure 4.1c. The internal highlighted structure represents the points of the original mesh, while the highlighted points on the surface represent points of the ghost cells that are exchanged with the neighbors.

Algorithm 16 Synchronous

```
1:  for all r ∈ [1..R] do
2:      exchange ghost cells
3:      for all i ∈ [1..M] do
4:          for all j ∈ [1..N] do
5:              for all k ∈ [1..P] do
6:                  u'(i, j, k) = stencil(u(i, j, k), l(i, j, k))
7:              end for
8:          end for
9:      end for
10:     swap(u, u')
11: end for
```

17. Communication is initiated before the computation (line 2), but there is no immediate wait; rather, the relaxation of the inner points is done first (the three nested loops iterating over $[2..M-1] \times [2..N-1] \times [2..P-1]$ are summarized in lines 3-5). Then, processes wait for communication to complete (line 6). Finally, once the ghost cells have been received, the points on the surface of the local grid are updated. Updating the surface takes 6 additional groups of loops, one per face, with two nesting levels (lines 7-17).

4.2.2 Tarragon Implementations

The MPI reference implementations are compared to two Tarragon variants: *BGraph*, and *Graph*. In both Graph and BGraph, tasks are equivalent to the processes in the MPI variants, except that Tarragon permits the number of tasks to exceed the number of processor cores. In addition, tasks do not wait on communication, and execute asynchronously.

Algorithm 18 shows the steps to define decomposition and mapping, to allocate the graph, and to connect the tasks in the Graph variant. The Graph variant uses a regular three-dimensional decomposition to map tasks to each shared memory node. This distribution is optimal because it distributes tasks evenly to processes, and because

Algorithm 17 Asynchronous

1: **for all** $r \in [1..R]$ **do**
2: *initiate ghost cells exchange*
3: **for all** $(i,j,k) \in [2..M-1] \times [2..N-1] \times [2..P-1]$ **do**
4: $u'(i,j,k) = stencil(u(i,j,k), l(i,j,k))$
5: **end for**
6: *complete ghost cells exchange*
7: **for all** $(i,j,k) \in \{1\} \times [1..N-1] \times [2..P-1]$ **do**
8: $u'(i,j,k) = stencil(u(i,j,k), l(i,j,k))$
9: **end for**
10: **for all** $(i,j,k) \in \{M\} \times [1..N-1] \times [2..P-1]$ **do**
11: $u'(i,j,k) = stencil(u(i,j,k), l(i,j,k))$
12: **end for**
13: **for all** $(i,j,k) \in [2..M-1] \times \{1\} \times [2..P-1]$ **do**
14: $u'(i,j,k) = stencil(u(i,j,k), l(i,j,k))$
15: **end for**
16: **for all** $(i,j,k) \in [2..M-1] \times \{N\} \times [2..P-1]$ **do**
17: $u'(i,j,k) = stencil(u(i,j,k), l(i,j,k))$
18: **end for**
19: **for all** $(i,j,k) \in [2..M-1] \times [2..N-2] \times \{1\}$ **do**
20: $u'(i,j,k) = stencil(u(i,j,k), l(i,j,k))$
21: **end for**
22: **for all** $(i,j,k) \in [2..M-1] \times [2..N-2] \times \{P\}$ **do**
23: $u'(i,j,k) = stencil(u(i,j,k), l(i,j,k))$
24: **end for**
25: $swap(u, u')$
26: **end for**

Algorithm 18 Jacobi

1: $map = BlockingRegularGridMap(procs, tasks)$
2: $connector = NearestNeighborConnector\ connector(map)$
3: $graph = new\ VectorGraph < 3, Jacobi > (map)$
4: $graph.accept(connector)$

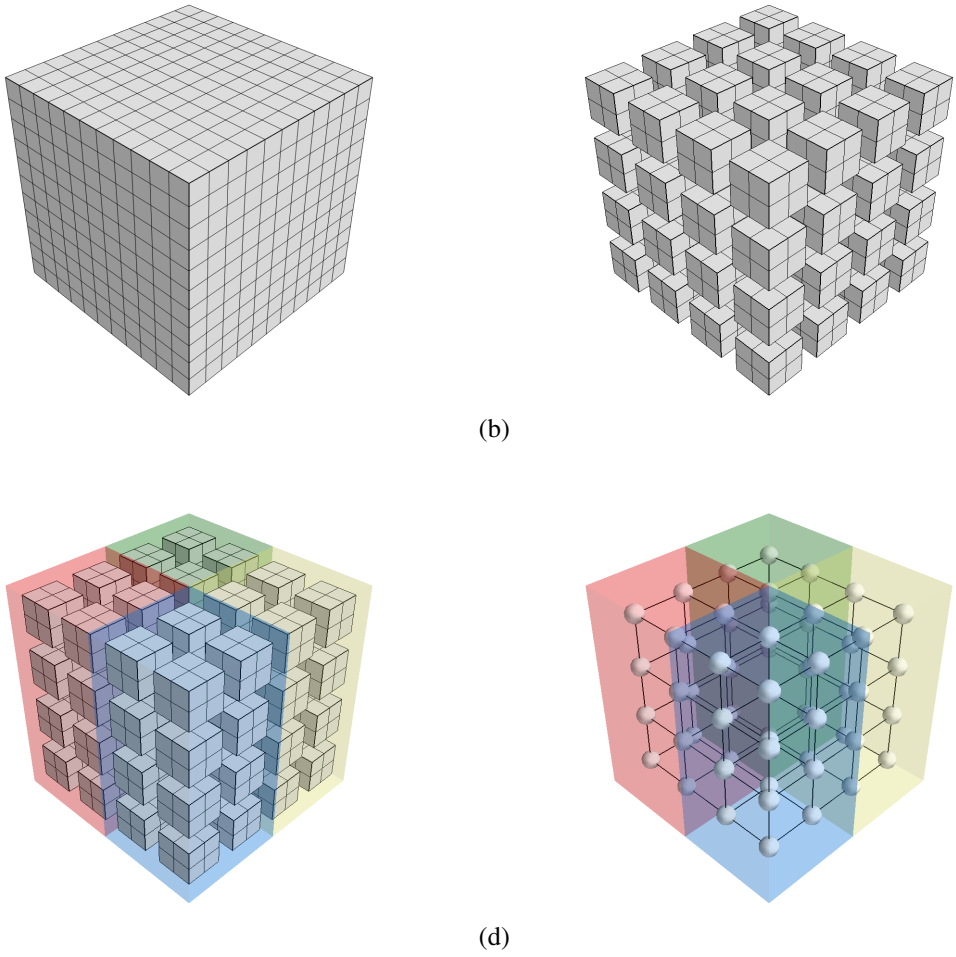
it minimizes off-node communication.

The mapping is defined by instantiating a *BlockingRegularGridMap* (line 1), which is a class of the Extended API of Tarragon (see Section 3.4 and Section 4.4). The instance of *BlockingRegularGridMap* defines a rectangular set of points which is regularly decomposed into tasks, and a mapping of the tasks to the given set of processes. Figure 4.2 illustrates an example in which a $12 \times 12 \times 12$ grid is decomposed into 64 $3 \times 3 \times 3$ blocks; each block is then assigned to a $2 \times 2 \times 1$ mesh of processes.

In the Graph variant, tasks are connected by an instance of the *NearestNeighborConnector* class (line 2), which is one of the connectors defined in the Extended API of Tarragon. When visiting a graph (line 4), the *NearestNeighborConnector* connects neighboring tasks in both directions, enabling neighboring tasks to exchange ghost cells, as illustrated in Figure 4.2d.

Stencil computations are good candidates for cache blocking because they are characterized by high spatial locality but low temporal locality. In such cases, it may be possible to rearrange the order of the operations to maximize temporal locality, hence improving performance. Cache blocking is such an optimization. Cache-blocking treats blocks of values that fit in cache as a unit, and all the operations on the block are executed before the computation proceeds to another block. In this way, temporal locality is maximized by re-using the data of a block, as many times as possible, while the block is in cache.

While stencil computations are good candidates for cache blocking, optimizations are not always necessary when comparing the communication cost of different implementations: as long as the sequential performance is the same, optimizations should not affect the outcome of the comparison. However, While cache blocking can be an effective optimization for stencil methods, the same effect can be achieved through over-decomposition. Over-decomposition conveys the same locality effect of cache block-



(a) (b)

(c) (d)

Figure 4.2: Mesh decomposition in Tarragon. The complete 12×12 mesh is illustrated in Figure 4.2a. The mesh is partitioned into 64 blocks, as illustrated in Figure 4.2b. The blocks are then mapped to 4 processes. Each process owns 16 blocks, organized in $2 \times 2 \times 4$ structures. In Figure 4.2c the structures are at the top, bottom, left, and right of the mesh. Figure 4.2d illustrates the corresponding task graph. Each block is associated to a task, and neighboring tasks are connected.

ing, because tasks are treated like units of computation. While extensive optimization and tuning of stencil computations are out of the scope of this thesis [2], the effect of cache blocking must be taken into account when comparing the Tarragon variants to the MPI variants. For this reason, BGraph and the MPI variants implement cache blocking, whereas Graph relies solely on over-decomposition as a means of achieving the effect of cache-blocking. For the sake of clarity, the additional loops required to implement cache blocking have been omitted in Algorithm 16 and Algorithm 17.

BGraph also uses a different partitioning scheme than Graph does. Graph achieves better locality through over-decomposition, creating a large set of tasks. Instead, BGraph employs cache-blocking and it has no incentive to create a large number of tasks. Rather, since using fewer tasks may reduce communication and scheduling overheads, BGraph uses a rectilinear partition to create thinner tasks on the boundaries to ensure that even when a small number of tasks are created there is enough computation to overlap with communication. Figure 4.3 shows an example of rectilinear partition.

Algorithm 19 shows the steps to define decomposition and mapping, to allocate the task graph, and to connect the tasks: to automatically create a rectilinear partition, BGraph allocates an instance of *BlockingRectilinearGridMap*, a class defined in the Extended API of Tarragon. By creating an instance of *BlockingRectilinearGridMap* (line 1), BGraph obtains a *Map* with the desired number of tasks, as directed via the argument *tasks*, and which defines the desired decomposition, as directed via the arguments *procs* and *steps*. In particular, *steps* defines the width of the partitions.

Algorithm 19 Jacobi

1: $map = BlockingRectilinearGridMap(procs, tasks, steps)$
2: $connector = NearestNeighborConnector\ connector(map)$
3: $graph = new\ VectorGraph < 3, Jacobi > (map)$
4: $graph.accept(connector)$

Both Graph and BGraph use a customized subclass of *Task*, called *Jacobi*. The class overrides the methods *vinject* and *vexecute*, which are illustrated in Algorithm 20

[2] An overview of stencil computations optimization and tuning is given by Datta et. al. [Dat08].

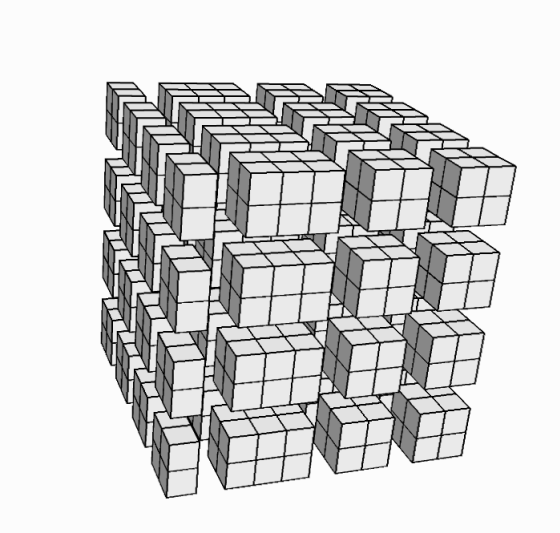


Figure 4.3: Rectilinear decomposition scheme. Blocks with off-node neighbors are smaller than with regular decomposition; in the figure, these are the blocks on the left. The adjacent blocks that do not border on neighbors are bigger than with regular decomposition.

and Algorithm 21, respectively. The method *vinject* handles incoming data and defines the firing rule. In Jacobi, when all the ghost cells arrive, the receiving task becomes ready for execution (line 7). It can also happen that two messages arrive from the same direction before messages from other directions have arrived. In fact, because of the asynchronous nature the algorithm, it is possible that after an initial exchange of ghost cells a task does not execute while its neighbors do, and that it also receives the following iteration's ghost cells. For this reason, an extra buffer is provided for pending messages (lines 2 and 3). However, not more than one level is necessary, because reciprocal dependencies prevent a task from proceeding two or more iterations ahead of its neighbors.

vexecute encapsulates the code for the relaxation of the grid partition associated with the task. There is no iterations loop as in traditional implementations. Instead, a task executes as many times as required by controlling the transitions between its states: *WAIT*, *EXEC*, and *DONE*. At the end of each invocation of *vexecute*, before releasing control, the task state is set based on the number of executions. If the desired number of

Algorithm 20 vinject(message)

```
1: direction = message.direction()
2: if ghosts[direction] then
3:     ghosts_hold[direction] = message
4: else
5:     ghosts[direction] = message
6:     if ++received = dependencies then
7:         →EXEC
8:     end if
9: end if
```

iterations is reached, the task sets its state to *DONE*; otherwise, it sets its state to *WAIT*, stalling execution until ghost cells arrive, or to *EXEC* if ghost cells have been received already.

Algorithm 21 vexecute

```
1: for all (i, j, k) ∈ [1..M − 1] × [1..N − 1] × [1..P − 1] do
2:     u′(i, j, k) = stencil(u(i, j, k), l(i, j, k))
3: end for
4: put messages
5: swap(u, u′)
6: if ++iteration = R then
7:     →DONE
8: else
9:     reset dependencies
10:    if received ≠ dependencies then
11:        →WAIT
12:    end if
13: end if
```

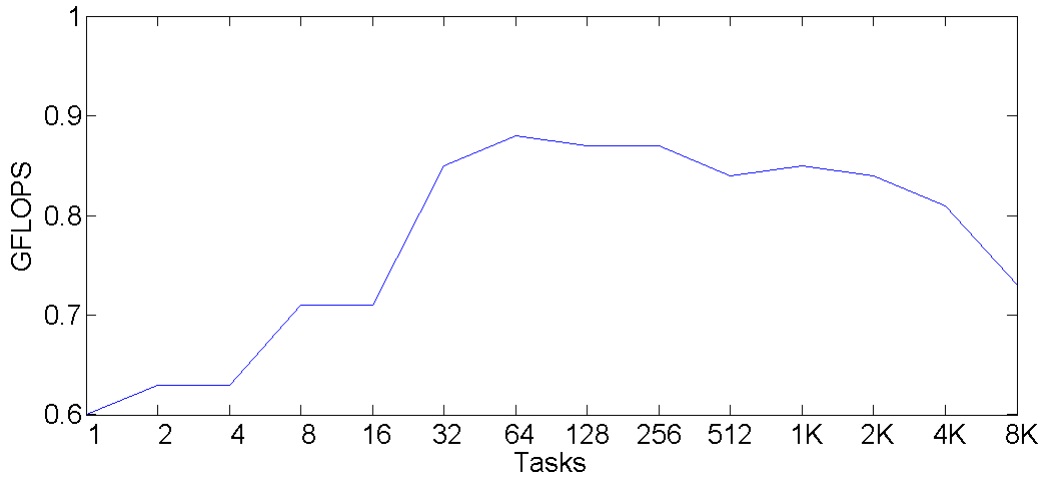


Figure 4.4: Cache-blocking effect with over-decomposition on Abe. The performance achieved by Graph, running on Abe, is reported as a function of the number of tasks instantiated.

4.3 Performance Evaluation

The four variants are evaluated on two platforms: Abe and Kraken. Abe is an Intel powered cluster, Kraken is a Cray XT5. Detailed specifications of the two platforms are given in Appendix A.

4.3.1 Results on Abe

The first experiment measures the effect of cache blocking on Abe. Figure 4.4 shows the effect of increasing the number of tasks when executing Graph on 1 core. On a 512^3 grid, changing the number of tasks visibly affects performance and, up to 64 tasks, performance increases because of better locality due to the cache blocking effect. At 64 tasks, Graph achieves its peak performance of 0.87 GFLOPS. With more than 64 tasks, performance degrades. The loss in performance is due to scheduling overheads: the cost of scheduling and activating a task becomes large relative to the time the task spends computing.

The effect of cache blocking also suggests that the application is memory bound. In fact, the stencil implemented is characterized by a relatively small number of arith-

metic operations per data point loaded from memory. The systems considered in this thesis are hierarchically organized platforms in which each node has multiple cores sharing the available memory bandwidth. As a result, a fraction of the available cores can saturate the available bandwidth, and using all the available cores might not be beneficial, or could even be detrimental. The next experiment measures the incremental performance contribution of each core.

Table 4.1: Single node performance on Abe. The table shows the performance, measured in GFLOPS, of Synchronous and Graph when increasing the number of cores.

Cores	1	2	4	8
Synchronous	0.84	1.40	1.42	1.53
Graph	0.87	1.40	1.53	1.60

Table 4.1 shows the performance of the Synchronous and the Graph variants when the number of cores is increased from 1 to 8. The achieved GFLOPS rate increases as more cores are added but the contribution of each added core decreases. Memory bandwidth is the limiting factor and even on two cores, one per socket, can saturate a large fraction of the available bandwidth. Therefore, using more cores only marginally improves performance since most of the available bandwidth is already fully utilized.

In the following experiments on Abe, both the Synchronous and Asynchronous variant use all the available cores, whereas Graph and BGraph use 7 cores per node, one for the Tarragon service thread, and 6 for the worker threads.

In weak scaling, the user increases the workload in proportion to the number of processes, for example, to increase mesh resolution. The weak scaling study that follows compares the performance of the four variants on Abe. As shown in Table 4.2, the problem size ranges from 512^3 to 3072^3, running on 8 to 2048 cores. In every configuration there are approximately 16M points per core and the computation completes 50 iterations in approximately 30 seconds.

The Asynchronous variant suffers a performance loss, in comparison to the Synchronous variant, as a result of the poor locality of split-phase coding [Bad01, Pie]. The cost of updating the surface points outweighs the gain of the overlap of computation with communication. Graph and BGraph perform better than the Synchronous

Table 4.2: Weak scaling on Abe. The Table shows the performance, measured in GFLOPS, of the four variants: Synchronous (S), Asynchronous (A), BGraph (BG), and Graph (G). The Table also shows the percentage of time spent communicating in Synchronous (Comm), the speedup of Graph over Synchronous (S/G), and the percentage of the communication time that is hidden in Graph (Hidden).

Problem Size	Cores	S	Comm	A	BG	G	S/G	Hidden
512^3	8	1.5	7%	1.4	1.6	1.6	1.06	74%
640^3	16	3.1	4%	2.9	3.2	3.2	1.04	93%
800^3	32	6.0	7%	5.7	6.2	6.3	1.05	74%
1000^3	64	11.6	7%	11.3	12.4	12.5	1.08	98%
1200^3	128	23.6	8%	22.4	24.0	24.9	1.06	64%
1680^3	256	47.3	6%	45.7	49.7	49.8	1.05	83%
2000^3	512	93.4	7%	90.2	99.2	99.3	1.06	87%
2432^3	1024	185.7	9%	175.0	197.8	197.6	1.06	68%
3072^3	2048	372.5	8%	364.7	390.7	394.6	1.06	70%

variant. The speedup enjoyed is the result of the reduced communication time that, in Synchronous lies on the critical path, whereas in Graph and BGraph it is hidden. The comparison between the cost of communication, which is measured by difference running the Synchronous variant omitting calls to communication primitives, and the speedup achieved by Graph indicates that Graph hides at least 64% of the communication cost, and more than 70% of the communication cost in most of the cases. BGraph and Graph achieve similar performance. Although BGraph can create fewer tasks and possibly reduce scheduling overheads, compared to Graph there is no substantial difference in performance.

Strong scaling is considered next. In strong scaling the workload remains fixed as the number of processes increases. With strong scaling, the surface-to-volume ratio increases: the number of partitions increases but the workload does not. Consequently, the volume of the partitions decreases faster than the surface does and, as a result, efficiency degrades because the amount of work done per data transferred is reduced.

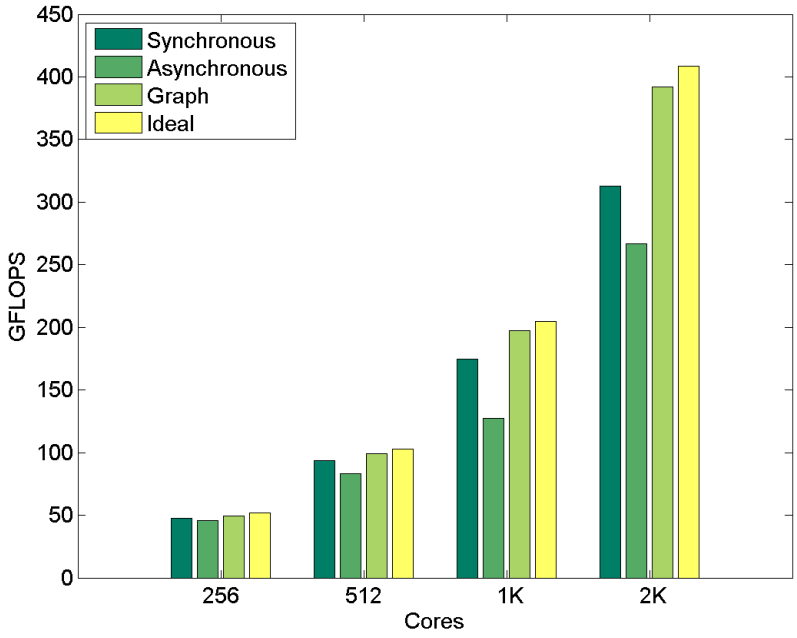


Figure 4.5: Strong scaling on Abe. Comparison between the performance of Synchronous, Asynchronous, Graph, and ideal performance, which is the performance that Synchronous achieves omitting communication. The ideal values represents an upper bound on performance. Graph gets to within 5% of the upper bound.

Figure 4.5 compares Synchronous, Asynchronous, and Graph on a 1600^3 problem, on 256, 512, 1024, and 2048 cores. Figure 4.5 also shows the ideal performance, which is measured by running Synchronous but omitting communication. For Synchronous, the gap between ideal performance and actual performance increases with the number of cores, whereas the performance of Graph closely follows the ideal curve. In fact, as shown in Table 4.3, the speedup of Graph over Synchronous increases, following the trend of the relative cost of communication. On 2048 cores, Graph achieves a 1.25 speedup over Synchronous reducing the wait time on communication by 86%. As before, Asynchronous achieves worse performance than Synchronous.

Table 4.3: Strong scaling on Abe. Performance, measured in GFLOPS, of the four variants: Synchronous (S), Asynchronous (A), and Graph (G), with the percentage of total running time spent communicating in Synchronous (Comm), the speedup of Graph over Synchronous (S/G), and the percentage of the communication time that is hidden in Graph (Hidden).

Cores	S	Comm	A	G	S/G	Hidden
256	47.3	9%	45.6	49.7	1.05	57%
512	93.8	9%	83.2	99.4	1.06	63%
1024	174.8	15%	127.4	197.2	1.12	78%
2048	312.8	23%	266.8	391.9	1.25	86%

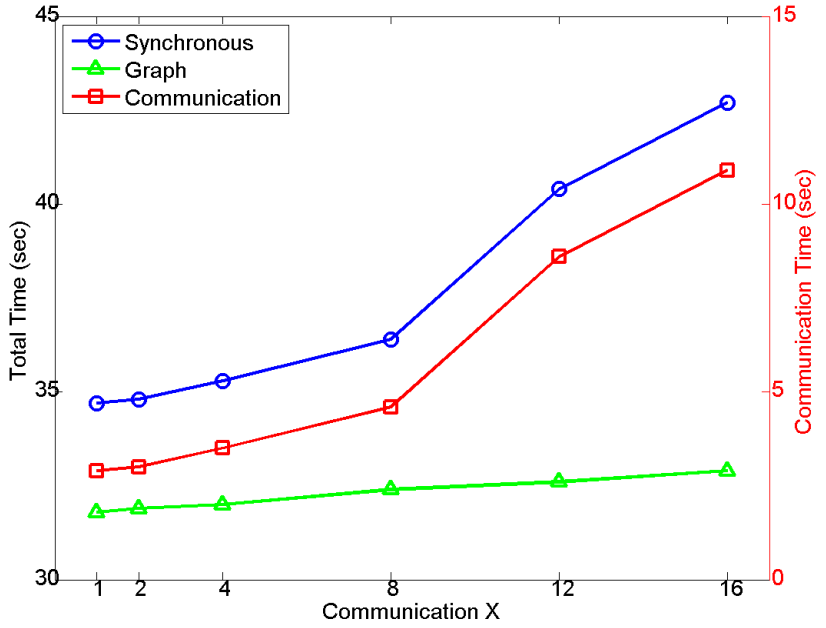
The last experiment on Abe explores Tarragon's ability to tolerate increasing communication latencies. In this experiment, communication delays are artificially increased by padding the end of the message buffer with extra data. Figure 4.6a compares Synchronous and Graph when solving a 1000^3 problem on 64 cores, as a multiplicative factor is applied to the amount of data exchanged. Figure 4.6a also shows the communication cost.

The running time of Synchronous exhibits the same growth as the communication cost, whereas the running time of the Tarragon version increases gracefully, and at a much lower rate. Figure 4.6b illustrates the correlation between the percentage of time that Synchronous spends in communication and the speedup achieved by Graph over synchronous. The two curves are characterized by the same slopes indicating a strong

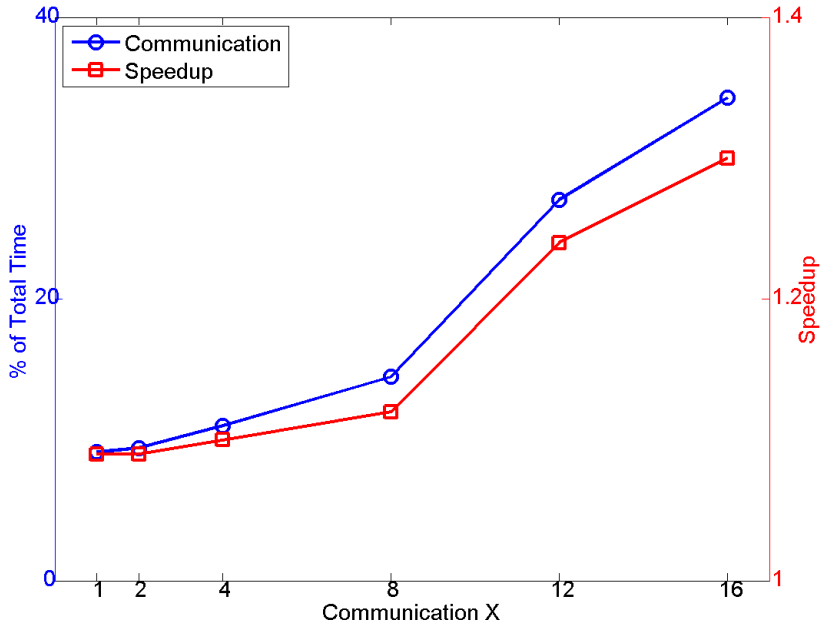
correlation between communication cost and speedup. In fact, while Graph successfully overlaps communication with computation, the larger the contribution of communication to the total running time, the larger the speedup achieved. As reported in Table 4.4, communication time increases, contributing from 9% to 34% of the total running time, and Graph, which hides from 90% to 100% of the communication time, achieves a speedup over Synchronous that increases from 1.09 to 1.3.

Table 4.4: Communication cost scaling on Abe. The Table shows the running time, measured in seconds, of the two variants: Synchronous (S) and Graph (G). The Table also shows the increase in size in the communication buffer (Comm X), the percentage of total running time that is spent communicating (Comm), the speedup of Graph over Synchronous (S/G), and the percentage of the communication time that is hidden in Graph (Hidden).

Comm X	S	Comm	G	S/G	Hidden
1	34.7	9%	31.8	1.09	100%
2	34.8	9%	31.9	1.09	97%
4	35.3	11%	32.0	1.10	94%
8	36.4	14%	32.4	1.12	87%
12	40.4	27%	32.6	1.24	91%
16	42.7	34%	32.9	1.30	90%



(a)



(b)

Figure 4.6: Effect of communication increase on performance on Abe. Figure 4.6a illustrates wallclock time relative to communication cost. Figure 4.6b illustrates the speedup of Graph over Synchronous, in relation to communication time.

4.3.2 Results on Kraken

The first experiment measures the effect of cache blocking. Figure 4.7 shows the effect of increasing the number of tasks when executing Graph on 1 core. On a 600^3 grid, while the number of tasks affects performance, the peak performance of 0.82 GFLOPS is achieved on a wide range of configurations. The peak is first reached at just 32 tasks, but the achieved performance stays within 1.2% of the peak from 8 to 1800 tasks; then, the performance degrades due to overheads: the cost of scheduling and activating a task becomes very larger relative to the time spent computing.

Kraken is a Non-Uniform Memory Access [3] (NUMA) architecture and, in addition to evaluating how core occupancy saturates the available memory bandwidth, it is also important to evaluate how data location in memory affects performance. Specifically, the variants that use Tarragon are multi-threaded and they may suffer a performance loss due to memory traffic across sockets if memory is not carefully allocated. Tarragon does not expose the physical properties of the local memory hierarchy, and traffic between sockets cannot avoided when one process occupies all the cores. However, by instantiating one process per socket, each process accesses only the closest memory.

The next experiment evaluates occupancy configurations in Synchronous and Graph. Table 4.5 shows the performance of the Synchronous variant and of the Graph variant, with the latter running with one process per node (Graph-1), and with one process per socket (Graph-2). While Graph-1 seems to be limited to scale only up to 8 cores, Graph-2 scales further and achieves its peak at 10 cores. On the NUMA architecture, using one process per socket restricts memory access to the closest memory, efficiently utilizing the available memory bandwidth. In the experiments that follow, Graph is always deployed with one process per socket. In the Synchronous variant, each process is assigned to a core and memory accesses are localized to the closest memory. The Synchronous variant achieves its peak performance on twelve cores.

[3] Non-Uniform Memory Access (NUMA) architectures are characterized by memory access time that depends on the relative distance between memory and the processor.

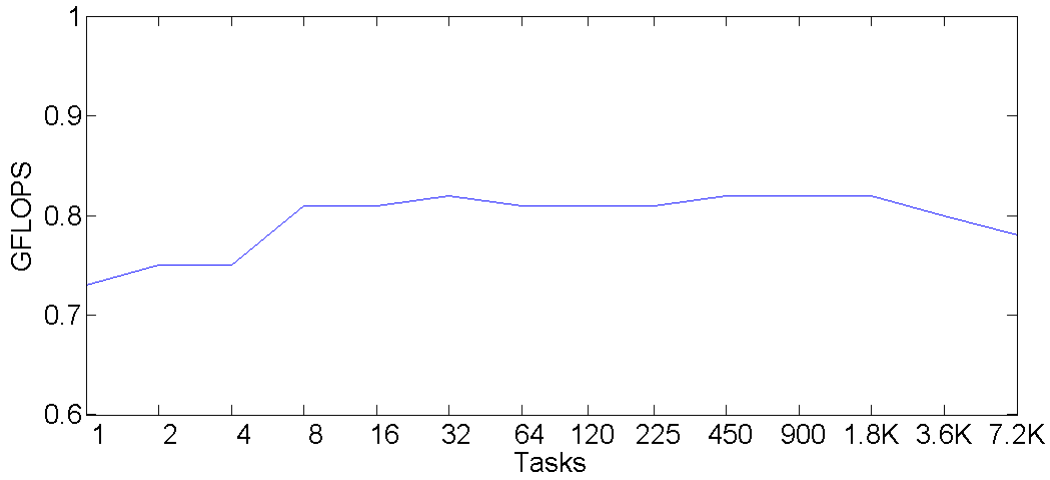


Figure 4.7: Cache-blocking effect with over-decomposition on Kraken. The performance achieved by Graph, running on Kraken, is reported as a function of the number of tasks instantiated.

Table 4.5: Single node performance on Kraken. The table shows the performance, measured in GFLOPS, of Synchronous and Graph as a function of the number of cores. Graph results are reported for both the 1-process (Graph-1) and 2-process configuration (Graph-2).

Cores	1	2	4	6	8	10	12
Synchronous	1.16	3.06	4.16	4.59	4.48	4.43	4.77
Graph-1	1.17	1.87	2.20	2.38	2.48	2.50	2.50
Graph-2	1.17	2.61	3.85	4.56	4.88	4.92	4.86

Synchronous, Asynchronous, and Graph are next compared under weak scaling. The problem size ranges from 600^3 to 3840^3, running on 12 to 3072 cores. In every configuration there are approximately 18M points per core and the computation completes 50 iterations in approximately 30 seconds.

Table 4.6 shows the performance of the three variants on Kraken. The Asynchronous variant is the slowest because of its poor locality. On 24 cores, the cost of communication is much higher than the average 6%, probably because of an inefficient data alignment which increases the cost of data packing and unpacking. However, such

high communication cost may not be present in the Graph variant, which uses a different decomposition, and the speedup of Graph over Synchronous is 8% on 24 cores, the highest observed. Graph is always outperforming Synchronous due to the overlap of communication with computation; in fact, Graph hides at least half of the communication cost in all the cases, and more than 70% of the communication cost in the majority of the cases.

Table 4.6: Weak scaling on Kraken. The Table shows the performance, measured in GFLOPS, of the three variants: Synchronous (S), Asynchronous (A), and Graph (G). The Table also shows the percentage of time spent communicating in Synchronous (Comm), the speedup of Graph over Synchronous (S/G), and the percentage of the communication time that is hidden in Graph (Hidden).

Problem Size	Cores	S	Comm	A	G	S/G	Hidden
600^3	12	4.8	3%	4.5	4.9	1.03	100%
768^3	24	8.8	11%	8.5	9.5	1.08	73%
960^3	48	17.1	5%	16.6	17.8	1.04	75%
1200^3	96	34.4	5%	31.7	35.9	1.04	75%
1500^3	192	66.1	7%	64.2	69.2	1.05	64%
1920^3	384	138.3	8%	137.1	147.3	1.07	76%
2400^3	768	265.3	7%	261.6	276.2	1.04	58%
3024^3	1536	539.2	5%	514.1	556.2	1.03	61%
3840^3	3072	1101.7	5%	1056.8	1146.9	1.04	61%

The last experiment on Kraken evaluates the two MPI variants and Graph in strong scaling. Figure 4.8 compares Synchronous, Asynchronous, and Graph on a 1536^3 problem, on 384, 768, 1536, and 3072 cores. Figure 4.8 also shows the ideal performance, which is measured by running Synchronous but omitting communication. The Asynchronous variant is the slowest because of its poor locality, although its performance is in this case closer to the performance of the Synchronous variant than in weak scaling; while the overall efficiency of the computation decreases, the performance penalty due to poor locality is relatively smaller.

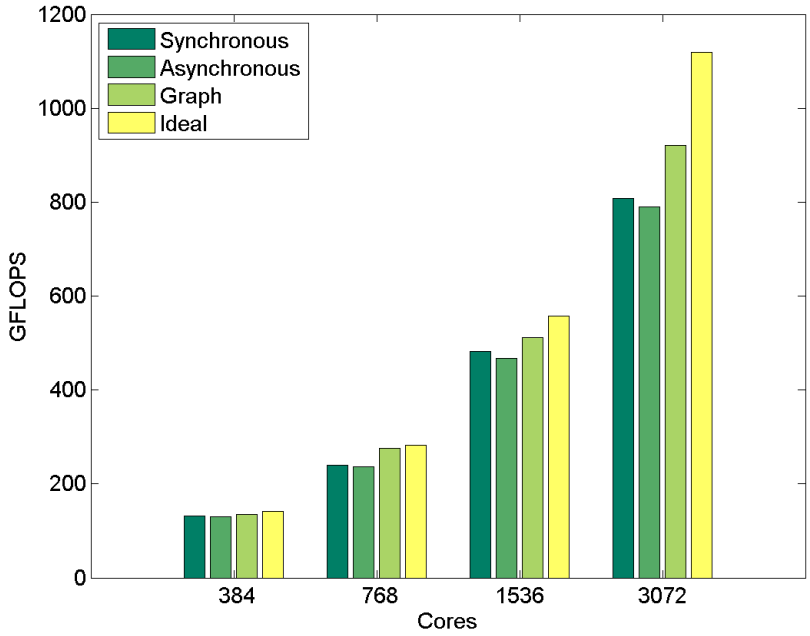


Figure 4.8: Strong scaling on Kraken. Comparison between the performance of Synchronous, Asynchronous, Graph, and ideal performance, which is the performance that Synchronous achieves omitting communication. The ideal values represents an upper bound on performance. Graph gets to within 10% of the upper bound on up to 1536 cores, and it gets to 82% of the upper bound on 3072 where Synchronous gets to 72% of the upper bound.

For Synchronous, the gap between ideal performance and actual performance increases with the number of cores. Also for Graph the gap increases in absolute terms, although Graph is able to hide 38% or more of the communication. As shown in Table 4.7, the speedup of Graph over Synchronous varies from 1.02 to 1.15, but there is no clear trend like there was on Abe. Although Graph achieves an average 1.09 speedup, performance appears to become more dependent on locality effects induced by the fine-grained partitioning.

Table 4.7: Strong scaling on Kraken. Performance, measured in GFLOPS, of the four variants: Synchronous (S), Asynchronous (A), and Graph (G), with the percentage of total running time spent communicating in Synchronous (Comm), the speedup of Graph over Synchronous (S/G), and the percentage of the communication time that is hidden in Graph (Hidden).

Cores	S	Comm	A	G	S/G	Hidden
384	131.8	6%	129.6	135.1	1.02	38%
768	239.5	15%	237.3	276.3	1.15	87%
1536	482.9	13%	467.8	512.3	1.06	44%
3072	808.5	28%	790.5	920.6	1.14	44%

4.3.3 Discussion

On both testbeds, results show that a split-phase implementation of an overlapping MPI implementation degrades performance due to its suboptimal locality. In memory bound applications, split-phase coding may greatly reduce performance if locality is not taken into account. However, since tasks in Tarragon are treated like units of computation, locality is preserved.

The Tarragon variants achieve overlap and outperform the Synchronous variant by hiding 50% or more of the total communication cost, in weak scaling, and 38% or more in strong scaling. In addition, the Tarragon variants successfully hide communication even when the relative cost of communication increases, indicating the ability to

achieve overlap when communication cost is very high (e.g. on platforms with accelerators).

4.4 A Finite-Difference Library Extension

This section illustrates the design of a Domain-Specific Extension (DSE) for finite-difference computations on Cartesian grids, inspired by the implementation of the Jacobi solver. The extension encapsulates all the encodings that characterize the observed computation and communication patterns.

According to the patterns, computations will have the following requirements: memory allocated to represent the points of the grid, including boundaries and ghost cells; nearest neighbor communication, in order to exchange ghost cells; and a main relaxation loop. In this context, codes differ in the type of cells represented, how the grid is initialized, and the equations that govern the system.

To satisfy these requirements, the extension provides template classes and functions that users can easily combine and customize. Creating the graph and the underlying grid of values should be very simple. In order to meet these goals, the extension proposed provides the three template classes and the template function illustrated in Figure 4.9.

Algorithm 22 compute

1: **if** $++iteration \leq R$ **then**
2: **for all** $(i,j,k) \in [1..M] \times [1..N] \times [1..P]$ **do**
3: $u'(i,j,k) = stencil(u(i,j,k), l(i,j,k))$
4: **end for**
5: $swap(u,u')$
6: **else**
7: $\rightarrow$DONE
8: **end if**

Most of the implementation is encapsulated in the *GridTask* class. *GridTask* extends *Task* and provides a pointer to the storage associated to the task (*_grid*), the

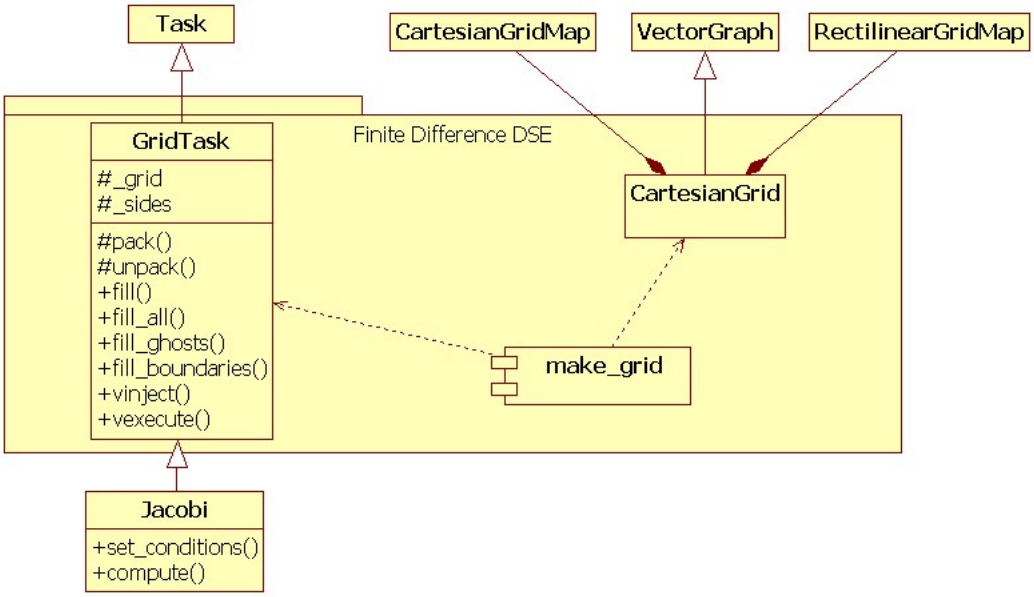


Figure 4.9: Class diagram of the Finite-Difference extension. The extension defines two classes, *GridTask* and *CartesianGrid*, and a template function, *make_grid*. *CartesianGrid* is based on classes of the Extended API of Tarragon: *CartesianGridMap*, *RectilinearGridMap*, and *VectorGraph*. The template function *make_grid* helps in the instantiation of a graph of tasks which are instances of a subclass of *GridTask*. In the diagram, the # indicates protected access to a member, a + indicates public access to a member.

size of the partition (*_sides*), and some methods, including data packing and unpacking methods. In addition, the class provides helper methods to initialize the grid.

GridTask fills in most of the details of the implementation of *pack* and *unpack*, *vinit* and *vinject*, and defers the application-specific implementation to the virtual methods *set_conditions* and *compute*. When using this extension, users are only required to define the initial conditions in the *set_conditions* method, and to encode the stencil operator in the *compute* method. Algorithm 22 shows a DSE implementation of the *compute* method for a Poisson solver that is equivalent to Graph. Using the same notation as in Algorithm 21, Algorithm 22 shows the DSE implementation of *compute*, which simply encapsulates the relaxation loop of the stencil operator and does not contain any dependency checks.

The rest of the extension deals with constructing the task graph. Class *CartesianGrid* extends *VectorGraph* in order to instantiate tasks of type *GridTask*. In addition, it has a reference to the associated map, which is either a *CartesianGridMap* or a *RectilinearGridMap*. Finally, the *make_grid* template function conveniently creates the graph and the associated map, and connects the tasks:

```
Graph* jgraph = make_grid<dimensions,fields,CartesianGridMap>
                (tasks_decomposition, bounding_box);
```

Table 4.8 compares the complexity of the code of Synchronous, Asynchronous, Graph, and the DSE based variant. The table presents the number of source lines and the Cyclomatic Complexity [McC96] of the functions that implement the stencil code. Cyclomatic Complexity is a simple metric that represents the complexity of a program based on the number of conditions it contains, and it is often used to indicate the number of tests required for statement coverage testing. In this context, Cyclomatic Complexity is used to compare the complexity of the variants providing an indication of the complexity in using Tarragon, rather than MPI or MPI with split-phase coding, and the benefits of DSE libraries.

Asynchronous and Graph require more lines of code then Synchronous. However, they do so for two different reasons: the Asynchronous code is larger because the loops are split to separate the computation of the values on the surface; the Graph code is

larger because data packing and unpacking is implemented explicitly. In contrast, Synchronous takes advantage of MPI's derived data types, persistent communication, and multiple initiation/completion primitives, which favor a compact formulation. However, in DSEGraph, communication details are encapsulated in the library extension and the code is much shorter than in any other variant.

The comparison using Cyclomatic Complexity gives a different perspective. Asynchronous is more complex than Synchronous due to the additional loops introduced by split-phase coding. However, the complexity of Graph is lower than the complexity of Synchronous, because Graph does not implement cache blocking, and it is much lower than the complexity of Asynchronous. As expected, since most of the complexity is encapsulated by the library, DSEGraph is the simplest implementation in terms of Cyclomatic Complexity.

Table 4.8: Comparison of complexity. For the comparison, the codes were stripped of I/O operations, and include statements, and other portions of the code that would be removed in a production version. The table compares the Synchronous, Asynchronous, Graph, and DSEGraph variants. Two measures are compared: source lines of code (SLOC) and Cyclomatic Complexity (CC).

Metric	Synchronous	Asynchronous	Graph	DSEGraph
SLOC	198	235	247	32
CC	20	29	16	5

The Finite-Difference Extension proposed and the comparison presented offer an example of how the gory details of common operations can be encapsulated in ad-hoc library extensions. Doing so promotes software reuse and considerably reduces the complexity of the software. For example, converting DSEGraph to a solver that uses a 19-point stencil requires a different implementation of the method *stencil*, which is invoked in Algorithm 22 (line 3), but the codes are otherwise identical.

The Finite-Difference Extension presented serves as an example that illustrates the development process for a DSE library. While Building domain-specific libraries for block structured computations has been subject of prior research [Sco98, Man95, Agr95, Col], the design and implementation of a comprehensive extension is out of the scope

of this thesis.

4.5 Acknowledgments

This chapter, in part, is currently being prepared for submission for publication. Pietro Cicotti; Scott B. Baden. The dissertation author is the primary investigator and author of this material.

References

[Agr95] Agrawal, G. and Sussman, A. and Saltz, J. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *Parallel and Distributed Systems, IEEE Transactions on*, 6(7):747 –754, July 1995.

[Bad00] Baden, S.B. and Fink, S.J. A programming methodology for dual-tier multicomputers. *Software Engineering, IEEE Transactions on*, 26(3):212 –226, March 2000.

[Bad01] Baden, Scott and Shalit, Daniel. Performance Tradeoffs in Multi-tier Formulation of a Finite Difference Method. In Alexandrov, Vassil and Dongarra, Jack and Juliano, Benjoe and Renner, RenÃl'nd Tan, C., editor, *Computational Science ICCS 2001*, volume 2073 of *Lecture Notes in Computer Science*, pages 785–794. Springer Berlin / Heidelberg, 2001.

[Ber89] Berger, MJ and Colella, P. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[Bra] Brandt, A. Multi-level adaptive technique (MLAT) for fast numerical solution to boundary value problems. In *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, pages 82–89. Springer.

[Col] Colella, P. and Graves, D. and Ligocki, T. and Martin, D. and Modiano, D. and Serafini, D. and Straalen B. V. Chombo software package for AMR applications.

[Dat08] Datta, Kaushik and Murphy, Mark and Volkov, Vasily and Williams, Samuel and Carter, Jonathan and Oliker, Leonid and Patterson, David and Shalf, John and Yelick, Katherine. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[Fin98] Fink, S.J. *A programming methodology for dual-tier multicomputers*. PhD thesis, University of California, San Diego, 1998.

[Man95] Manish Parashar and James C. Browne. An infrastructure for parallel adaptive mesh refinement techniques. Technical report, University of Texas, Austin, 1995.

[McC96] McCabe, T. Cyclomatic Complexity and the Year 2000. *Software, IEEE*, 13(3):115 –117, May 1996.

[Pie] Pietro Cicotti and Scott B. Baden. Tarragon: a Programming Model for Latency Hiding Scientific Applications. *In Preparation*.

[Sco98] Scott B. Baden and Stephen J. Fink. Communication overlap in multi-tier parallel algorithms. In *Proc. of SC '98*, Orlando, Florida, November 1998.

Chapter 5

Sparse Linear Algebra

5.1 The Motif

Sparse linear algebra problems often arise in scientific computing. The task at hand is to solve a system of linear equations of the form $Ax = b$, where x and b are dense vectors and A is a sparse matrix, i.e. most of its entries are zero. Implementations of sparse linear algebra operations take advantage of sparsity in two ways: they avoid operations involving zeros, and store only nonzeros in a compressed format. When the sparsity pattern is irregular, it is impossible to define a scheme for storing nonzeros without also storing their position within the matrix; therefore, nonzeros are accessed through some level of indirection. As a result, additional memory traffic is generated and the memory access pattern depends not only on the algorithm, but also on the sparsity pattern. It follows that, when data is characterized by irregular sparsity patterns, sparse linear algebra kernels exhibit poor locality and irregular communication patterns.

5.2 Sparse LU Factorization

The sparse linear algebra operation considered in this chapter is LU factorization. LU factorization is employed in direct solvers of systems of linear equations. In such solvers, given a linear system of the form $Ax = b$, first A is factorized, via some variant of Gaussian Elimination, into two matrices which are a lower and an upper tri-

angular matrix (L and U); then, the resulting triangular systems, $Ly = b$ and $Ux = y$, are solved by means of forward and backward substitution, respectively. The result of the latter solve gives the solution to the system $Ax = b$.

$$A^k = \left(\begin{array}{ccccccc} U_{1,1} & \cdots & \cdots & \cdots & \cdots & \cdots & U_{1,n} \\ L_{2,1} & \ddots & \cdots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & U_{k-1,k-1} & \cdots & \cdots & \cdots & U_{k-1,n} \\ \vdots & \cdots & L_{k,k-1} & U_{k,k} & \cdots & \cdots & U_{k,n} \\ \vdots & \cdots & L_{k+1,k-1} & L_{k+1,k} & \multicolumn{3}{c}{\multirow{3}{*}{$A^k_{(k+1:n,k+1:n)}$}} \\ \vdots & \cdots & \vdots & \vdots & & & \\ L_{n,1} & \cdots & L_{n,k-1} & L_{n,k} & & & \end{array}\right)$$

Figure 5.1: Doolittle's in-place right-looking LU factorization, step k. A^k is obtained by scaling $A^{k-1}_{(k+1:n,k)}$, as in Equation 5.1, and then updating the trailing submatrix $A^{k-1}_{(k+1:n,k+1:n)}$, as in Equation 5.3. Entries outside $A^{k-1}_{(k:n,k:n)}$ persist in A_k.

In LU factorization, the lower triangular matrix L is a matrix of transformations and U is the upper triangular matrix that results from the elimination process. In Doolittle's algorithm, L and U are constructed column by column, from left to right (right-looking factorization) [Gol96].

Figure 5.1 illustrates a step of the factorization with in-place updates to transform A into $L+U-I$. Starting from the $n \times n$ matrix $A^0 = A$, each step produces a column of L and a row of U. At step k, $L_{k+1:n,k}$ is obtained by *scaling* values of $A^{k-1}_{(k+1:n,k)}$ by $a^{k-1}_{k,k}$, as illustrated in Equation 5.1.

$$L_{(k+1:n,k)} = \frac{A_{(k+1:n,k)}}{a^{k-1}_{k,k}} \tag{5.1}$$

Since $l_{k,k} = 1$ for each k, therefore it follows from Equation 5.2 that $U_{(k,k:n)} = A^{k-1}_{(k,k:n)}$.

$$U_{(k:n,k+1)} = \frac{A^{k-1}(k:n,k)}{l_{k,k}} \tag{5.2}$$

Finally, the trailing submatrix $A^{k-1}_{(k+1:n,k+1:n)}$ is updated with a *rank-1* update operation of the form $M = M + \alpha xy^T$, in which xy^T is the outer product; Equation 5.3 illustrates the rank-1 update in the factorization step.

$$A^{k}_{(k+1:n,k+1:n)} = A^{k-1}_{(k+1:n,k+1:n)} - L_{(k+1:n,k)}U_{(k,k+1:n)} \tag{5.3}$$

Algorithm 23 illustrates the steps of the LU factorization described. The outer loop iterates over the columns of the matrix (lines 2-7); each column of the lower triangular matrix is scaled (lines 3-5); finally, the trailing submatrix is updated with a rank-1 update (line 6).

Algorithm 23 LU Factorization

1: LU=A
2: **for all** $k \in [1..n]$ **do**
3: **for all** $r \in [k+1..n]$ **do**
4: $LU_{r,k} = LU_{r,k}/LU_{k,k}$
5: **end for**
6: rank-1_update($LU_{(k+1:n,k)}, LU_{(k,k+1:n)}, LU_{(k+1:n,k+1:n)}, -1$)
7: **end for**

To take advantage of cache locality and of highly-tuned single-core dense linear algebra libraries, factorization algorithms are implemented using blocking. With blocking, the algorithm is expressed in terms of submatrices of contiguous rows and columns rather than individual rows and columns, and operations between submatrices are executed by invoking dense linear algebra routines, such as those provided in the Basic Linear Algebra Subprograms library [Law79, R. 00]. In parallel formulations, blocking also results in more efficient communication because it increases messages length while decreasing their number than for un-blocked algorithms. Consequently, the available bandwidth is used more efficiently and the total communication cost is reduced.

Figure 5.2 illustrates a step of blocked factorization. Since operations involve blocks, the number of steps matches the number of block columns, which are the blocks of contiguous columns of L. In addition, the basic operations of the algorithm are expressed in terms of submatrices. Algorithm 24 illustrates LU factorization with blocking. With respect to the un-blocked algorithm, the loop iterates through N, the number

$$A^k = \begin{array}{|c|c|c|c|c|}
\hline
D_1^k & \multicolumn{4}{c|}{R_1^k} \\
\hline
 & \ddots & \multicolumn{3}{c|}{\vdots} \\
\hline
 & & D_{k-1}^k & \multicolumn{2}{c|}{R_{k-1}^k} \\
\hline
 & & & D_k^k & R_k^k \\
\cline{4-5}
C_1^k & \cdots & C_{k-1}^k & C_k^k & A_k^k \\
\hline
\end{array}$$

Figure 5.2: Blocked in-place right-looking LU factorization, step k. A^k is obtained by factorizing block column $(D^(k-1)_k, C^(k-1)_k)$ (Algorithm 23), then updating block row $R^(k-1)_k$ (triangular solve), and finally updating the trailing submatrix $A^(k-1)_k$ (Equation 5.4). Entries outside D_k^k, C_k^k, R_k^k, and A_k^k persist in A^k.

of block columns; the scaling of a column is replaced by a factorization routine that applies the un-blocked algorithm to the block column (line 2); the scaling in Equation 5.2, which was omitted in Algorithm 23 because $l_{k,k} = 1$, is replaced by a lower triangular solve (line 3); finally, the rank-1 update is replaced by a matrix multiplication routine that updates the trailing submatrix as illustrated in Equation 5.4 (line 4).

$$A_k^k = A_k^{k-1} - C_k^k R_k^k \tag{5.4}$$

Algorithm 24 Blocked LU Factorization

1: **for all** $k \in [1..N]$ **do**
2: LUfactorization(D_k, C_k)
3: triangular_solve(D_k, R_k)
4: matrix_multiplication($C_k, R_k, A_k, -1$)
5: **end for**

In sparse LU factorization, block rows and block columns are further divided into blocks, and only blocks that contain nonzeros are stored. Blocked matrices are

stored in a compressed format, which stores only blocks containing nonzeros, together with information to indicate their location.

5.2.1 Reference Implementation

In this dissertation, an LU factorization code implemented with Tarragon is evaluated and compared to an MPI reference implementation. The reference MPI implementation is provided by the SuperLU_DIST software package [Li,05, Xia03]. The solver in SuperLU_DIST is a distributed memory sparse direct solver for general systems of equations[1]. The solver is characterized by a symbolic factorization phase in support of a static pivoting strategy [Li,98]; during the symbolic factorization the matrix is transformed to ensure stability and to preserve sparsity. Tarragon is applied to the numerical factorization which takes place after the symbolic factorization.

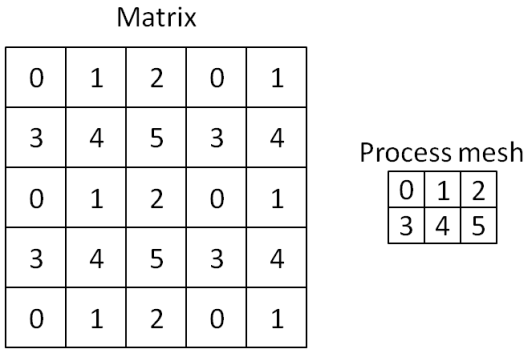


Figure 5.3: 2-dimensional cyclic mapping. A matrix partitioned into blocks is mapped to a 2×3 process mesh. The number inside each block of the matrix indicates the process mapping.

SuperLU_DIST stores L and U in a distributed compressed format. After the symbolic factorization, which defines the blocking structure of the matrices, blocks are mapped to a 2-dimensional process mesh in a cyclic fashion: block (i, j) is mapped to process $(i \, mod \, r, j \, mod \, c)$ where r and c are the dimensions of the process mesh. Figure 5.3 illustrates a mapping example.

[1] Systems of equations characterized by a positive-definite matrix can be solved more efficiently using Cholesky factorization.

Each process stores its blocks in block columns containing blocks of L, and block rows containing blocks of U. The two data structures have a different format. Blocks of L, as well as the rows within a block, are stored out of order due to the permutations applied during the symbolic factorization. The rows span the whole block column width, and the nonzeros of a block column are therefore stored as a dense matrix. However, nonzeros are stored in column major order and the nonzeros within a row are not stored contiguously. Blocks of U are stored in order. However, within a block, columns may have a different length, and the nonzeros are therefore stored contiguously in memory, but do not form a dense matrix. When operating on a block of U, nonzeros need to be first stored as a dense matrix. Both L and U data structures include an index structure that provides blocks and subscript information. Figure 5.4 illustrates the data structures of L and U.

Algorithm 26 shows the steps of the factorization in SuperLU_DIST. The factorization routine receives the input matrix A' (A' is derived by A in the symbolic factorization step). A' is stored in the data structure that, at the end of the factorization, will contain the resulting L and U matrices. During the factorization process, A' is updated in-place and eventually transformed into $L+U-I$. The factorization is divided into steps, one step per block along the main diagonal of the matrix, and the main loop iterates over the steps of the factorization routine (lines 1-11). In each step, a block column of L is factorized (line 2-4); this factorization, illustrated in Algorithm 25, involves a scaling operation and rank-1 updates as previously described; then, a block row of U is obtained via a sequence of triangular solves (line 6); finally, the trailing submatrix is updated (matrix multiply updates, line 10).

Algorithm 25 block_column_factorization(s)

```
1: if row(s)=myrow then
2:     factorize block column s
3:     send diagonal block
4: else
5:     wait diagonal block
6:     factorize block column s
7: end if
```

Algorithm 26 LU factorization

```
1:  for all s∈[1..block columns] do
2:    if col(s)=mycol then
3:      block_column_factorization(s)
4:      send block column s
5:    else
6:      wait block column s
7:    end if
8:    if row(s)=myrow then
9:      update and send block row s
10:   else
11:     wait block row s
12:   end if
13:   update trailing submatrix
14: end for
```

Algorithm 25 and Algorithm 26 also shows interprocess communication, and the conditions that determine which processes contribute to the operations in a step. The block column factorization starts with the process that owns the block on the diagonal. This process factorizes the local block column, then sends the pivots and the upper triangular part of the diagonal block to the processes along its process column (lines 2-3). Each other process that owns part of the block column waits for the diagonal block, and then factorizes the local part of the block column. When the factorization is complete, the block column is sent to the other processes along rows of the process mesh (line of).

After receiving the block column, processes that own the block row perform the update and then send the block row to the other processes along columns of the process mesh (line 6). Lastly, all the processes participate in updating the trailing submatrix (line 10). In the following step, processes may execute different phases of the pipeline, depending on which process owns the block on the diagonal. Because there is no explicit synchronization between steps, execution is pipelined across different steps.

Figure 5.5 illustrates a step of the factorization and the accompanying communi-

cation pattern. In the example, after the factorization of the first block column, process 0 sends the pivots to process 3, enabling the factorization of the remote blocks of the block column. This phase of the step is illustrated in Figure 5.5a. Then, process 0 sends the local blocks of the block column to other processes on the same process row, as illustrated in Figure 5.5b. The receiving processes can then update the block row via triangular solves. Finally, processes with blocks of the block column, and processes with blocks of the block row send the local data, enabling the trailing matrix update on every process, as illustrated in Figure 5.5c.

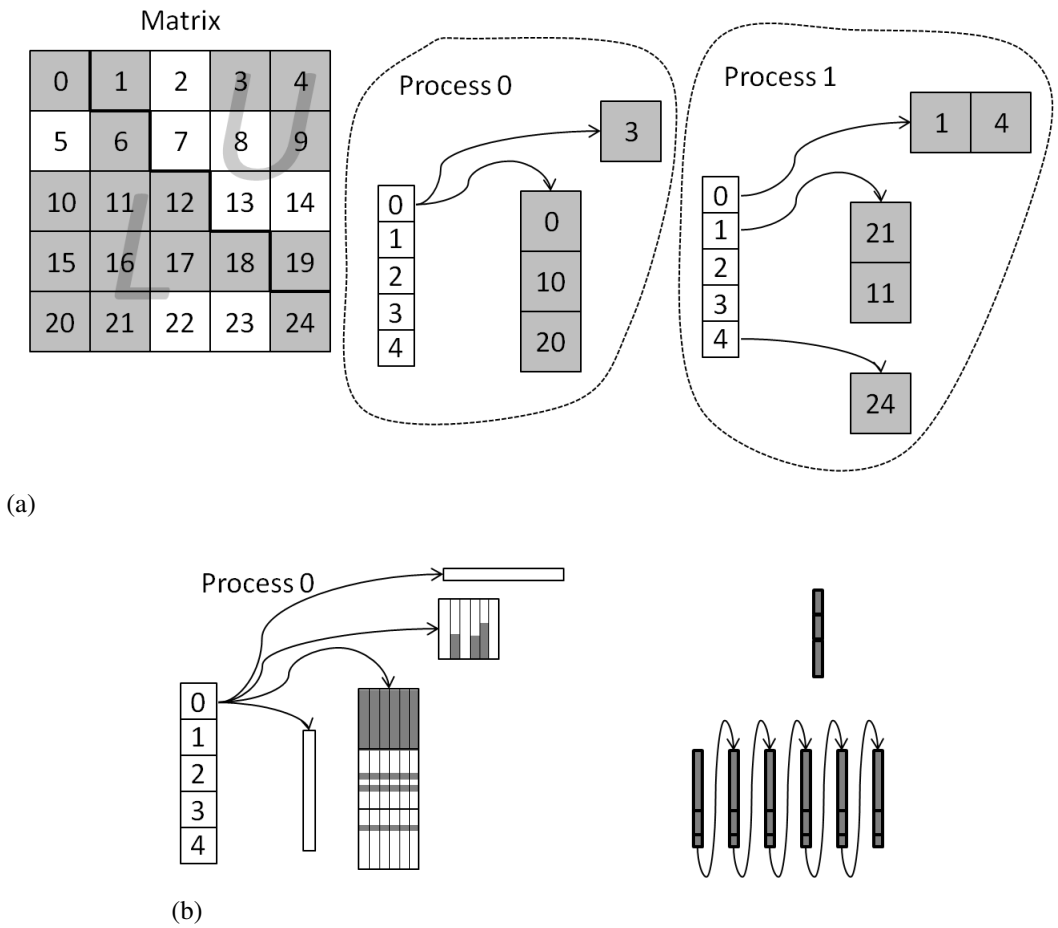


Figure 5.4: SuperLU_DIST decomposition, mapping, and data structures. Figure 5.4a illustrates a matrix that is decomposed into blocks. Grey blocks contain nonzeros; typical matrices are much sparser but this example uses a fairly dense matrix for the sake of simplicity. Blocks are stored as columns, which include blocks of L and the blocks on the diagonal, and as rows, which include blocks of U. For each block column, processes have a pair of pointers to the local block row and block column. Figure 5.4b illustrates the storage format of blocks rows and block columns. Block rows are stored as an index files containing row subscripts, and a values array in which nonzeros are stored in column major format; as illustrated on the right of the blocks, column of nonzeros span all the blocks of the block column. Block columns are stored as an index files containing the row subscript of the first element of each column, and a values array in which nonzeros are stored in column major format.

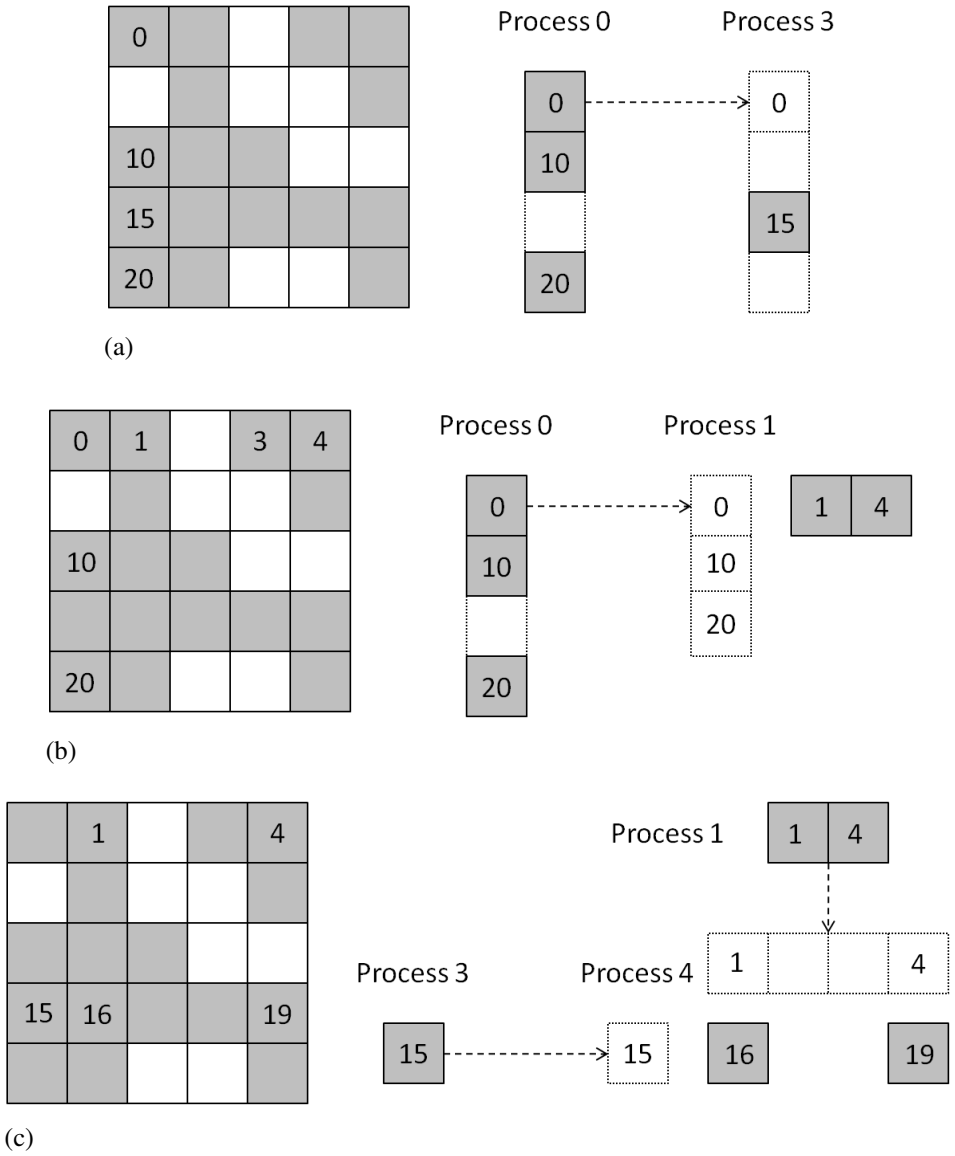


Figure 5.5: Factorization step. Figure 5.5a, Figure 5.5b, and Figure 5.5c illustrate three snapshots in a factorization step. Figure 5.5a illustrates the communication in factorizing the block column; after the factorization of the diagonal block, process 0 sends the pivots to process 3, which is the only other process with blocks of the block column. Figure 5.5b illustrates the update of the block row, which is enabled when process 0 sends the block column to process 1, which it the only other process with blocks of the block row. Figure 5.5c illustrates the update of the trailing submatrix; processes with blocks of the block column and processes with blocks of the block row send the local data enabling every process to participate to the trailing matrix update.

Algorithm 27 LU factorization with overlap

```
1:  block_column_factorization(1)
2:  initiate communication of block column 1
3:  for all s∈[1..block columns] do
4:     complete pending block column communication
5:     if row(s)=myrow then
6:        update and send block row s
7:     else
8:        wait block row
9:     end if
10:    if col(s+1)=mycol then
11:       update block column s+1
12:       block_column_factorization(s+1)
13:       initiate communication of block column s+1
14:       update remainder of trailing submatrix
15:    else
16:       update trailing submatrix
17:    end if
18: end for
```

SuperLU_DIST employs software pipelining to overlap communication with computation. The resulting one-step look-ahead is implemented by split-phase coding [Xia03]. The look-ahead step, which was omitted in Algorithm 26 for the sake of simplicity, is now illustrated in Algorithm 27. The first block column is factorized and communication initiated before the main loop (lines 1-2). Then, in each iteration of the loop, the trailing matrix update is split into two phases. In the first phase it updates the block column which should be factorized in the following step (line 11); the update of the trailing submatrix is then suspended and the updated block column is factorized (line 12), and communication initiated (line 13). In the second phase, the update of the trailing submatrix is resumed and completed (line 14). In this way, the communication of the block column is initiated before completing the update and overlapped with the second phase of the update.

The factorization of the block on the diagonal is asynchronous with respect to the block column factorization: after factorization, the process owning that block sends the pivots and then continues computing. It does not wait for communication to complete until it performs another block column factorization.

5.2.2 Tarragon Implementation

The Tarragon implementation of factorization is embedded in SuperLU_DIST and replaces the original factorization based on MPI, which is illustrated in Algorithm 26. As illustrated in Algorithm 28, after an *LUMap* and an *LUGraph* are instantiated (lines 3 and 4), tasks are created within a modified symbolic factorization (line 4). Then, the call to the numerical factorization routine is replaced by graph initialization and execution. Finally, the resulting triangular systems are solved as in the original solver (line 8).

Algorithm 28 compute

```
1: Tarragon::initialize()
2: Tarragon rts = Tarragon::tarragon()
3: Map map = new LUMap()
4: Graph graph = new LUGraph(map)
5: symbolic_factorization(A,graph)
6: rts.initialize_graph(graph)
7: rts.execute_graph(graph)
8: Tarragon::finalize()
9: solve_backward_forward(A)
```

In SuperLU_DIST, Blocks are mapped onto a process mesh, and in Tarragon, the tasks must reflect this mapping. In the Tarragon implementation, the number of tasks depends both on the number of steps and the number of processes. *LUMap* defines a mapping from a 3-dimensional space (process id, step, and type of task) to task ids of the form

$$m : [0..p-1] \times [0..sn-1] \times [0..3] \rightarrow [0..t-1]$$

where p is the number of processes, sn is the number of block columns, and t is the total

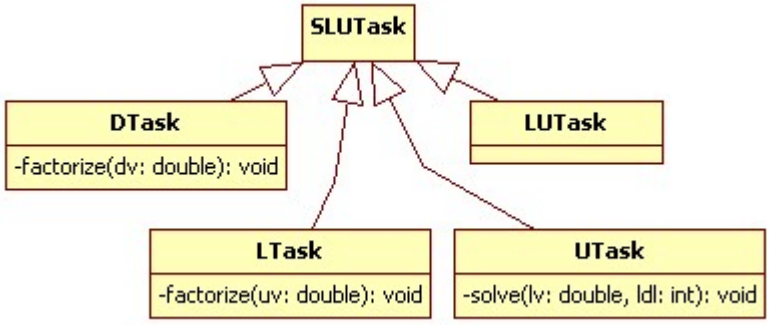


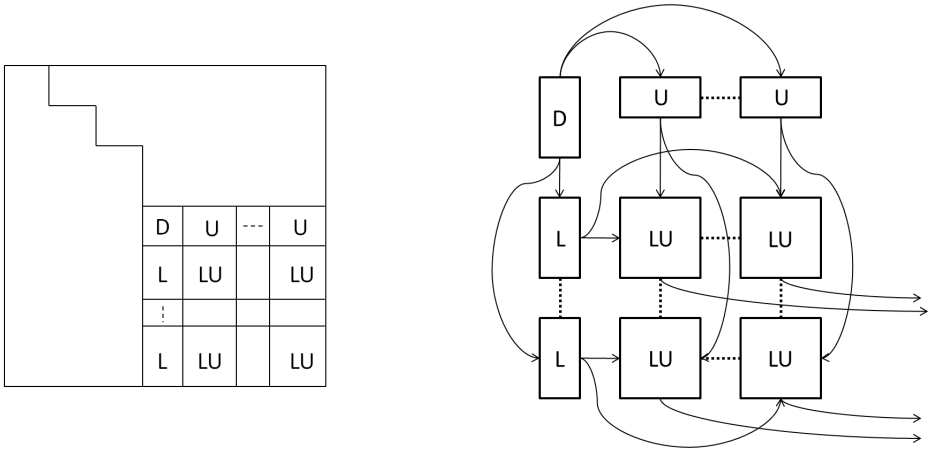
Figure 5.6: Class diagram of the tasks in LU factorization. *SLU* Task is the base class and it is extended by *DTask*, *SLU*, *SLU*, and *SLU*. The subclasses define the different types of tasks corresponding to the operations within a factorization step.

number of tasks. LUMap assigns contiguous ids to triplets with the same value in the first position; consequently, by virtue of the *RegularDistribution*, which maps blocks of consecutive ids to processes, such triplets are mapped to the same process ensuring that a subspace $p' \times [0..sn-1] \times [0..3]$ is mapped to each process.

Not all the tasks are instantiated. The symbolic factorization is modified such that while the data structures of L and U are prepared, only the tasks whose associated blocks have nonzeros are instantiated and added to the graph. At the same time, the graph is formed and the dependencies are defined. Dependencies intended for tasks that are not instantiated are either *directed* to other tasks, such as in the case in which there are transitive dependencies, or not defined at all.

Figure 5.6 shows the class diagram of the five types of tasks defined: *SLUTask*, *DTask*, *LTask*, *UTask*, and *LUTask*. SLUTask is the basic task used to define *LUGraph*; in fact, LUGraph is a *SparseVectorGraph* of SLUTask objects, that is, a graph built on a sparse set of tasks (*SparseVectorGraph* is described in Section 3.3.2). DTask is in charge of starting a new factorization step; DTask updates and factorizes a block column and sends the block on the diagonal to LTasks and UTasks. LTask updates and completes the block column factorization on processes other than the one that owns the block on the diagonal. UTask performs triangular solves to update the corresponding block row. Finally, DTasks, LTasks, and UTasks send data to LUTask, enabling the trailing matrix

update, as well as to the tasks that belong to the following step, to enable updates and to continue the process. Once the update is completed, LUTask signals the next local task that the step is completed. Figure 5.7 illustrates the tasks involved in a factorization step and their dependencies.



(a) (b)

Figure 5.7: LU factorization task-dependency graph. Figure 5.7a illustrates the blocks within the matrix associated with the tasks. The step is illustrated in Figure 5.7b. LTasks (L) depend on DTask (D) for the block column factorization, UTasks (U) depend on DTask for the block row update, and LUTasks (LU) depend on the other tasks for the trailing matrix update. In addition, LUTasks are connected to tasks in the following steps to signal when the step is completed.

The Tarragon implementation effects a compromise between creating a large number of small tasks and maintaining the coarse granularity of the original formulation. Creating a large number of small tasks would be ideal for exposing a high degree of parallelism. However, preserving interoperability with other SuperLU_DIST routines introduces limitations on the design choices in the Tarragon implementation. For example, due to permutations performed in the symbolic factorization phase, rows of L are stored out of order, and the order differs within each block column. In addition, blocks of U are stored as columns of different size. As a result, factorization involves searches and data *formatting* operations before dense linear algebra routines can be used. Such operations would be duplicated in a fine grained task-parallel implementation causing

high computational overheads. The approach adopted in the Tarragon factorization algorithm is conservative in that it is task-parallel, but the tasks defined match the granularity of the phases of the MPI implementation.

There are also limitations on the way factorization can be executed in the Tarragon implementation. The SuperLU_DIST design assumes an underlying SPMD execution model (MPI) and it uses a block-cyclic mapping of blocks to processes. With Tarragon, when a single multi-threaded run-time system is deployed on each node, a block-cyclic distribution maps consecutive blocks on different nodes causing a significant increase in inter-node communication. For this reason, the Tarragon implementation runs more efficiently if it follows the same process mapping of SuperLU_DIST, though this prevents the run-time system from transferring data via shared-memory and the available parallelism is limited. In addition, while the data structures used in SuperLU_DIST are stored in buffers ready for communication with MPI, in Tarragon such data structures must be integrated with a message header requiring extra memory copies.

5.3 Performance Evaluation

The test suite used for the experiments is composed of twelve matrices from real world science and engineering problems. Eight matrices are taken from the University of Florida Sparse Matrix Collection [Duf89, Tim94], two from a fusion energy study [cem], one from an accelerator structural design problem [com], and one is a dense matrix. The matrices were selected with different sizes, number of nonzeros, sparsity, and sparsity pattern to ensure that performance is evaluated under different conditions. In particular, the dense matrix generated for this study is used to create a balanced workload distribution in the attempt to isolate communication delays due to data transfer from delays due to load imbalance. This test case is useful because with an uneven workload distribution, waiting times due to load imbalance are also accounted for as communication delays, but cannot be overlapped with computation. The characteristics of each matrix are summarized in Table 5.1.

To accommodate memory requirements in the symbolic factorization, in the experiments that follow the three largest matrices (the dense matrix, dds15, and matrix181)

Table 5.1: Benchmark matrices characterization: order of the matrix (N), number of nonzeros (nnz(A)), number of nonzeros in the factorized matrix (nnz(L+U)), sparsity (nnz(A)/N), defined as the number of nonzeros per row, structural symmetry (Sym), defined as nnz(S & S')/nnz(S) where S is the sparsity pattern of A, and discipline of the problem (Discipline).

Matrix	N	nnz(A)	nnz(L+U)	$\frac{nnz(A)}{N}$	Sym	Discipline
bbmat	38744	1771722	36074161	46	53%	CFD
dense	8000	64000000	64000000	8000	100%	generated
dds15	834575	13100653	875305401	16	N/A	structural
g7jac200	59310	717620	37369148	12	3%	economic
inv-extrusion	30410	1793881	30245222	59	97%	CFD
matrix31	17298	2683044	12323340	155	N/A	fusion
matrix181	589698	95179212	898865100	161	N/A	fusion
mixing-tank	29957	1990919	44562362	67	99%	CFD
nasasrb	54870	2677324	21128018	49	100%	structural
stomach	213360	3021648	140580464	14	85%	2D
torso1	116158	8516500	27742019	71	42%	3D
twotone	120750	1206265	11360029	10	24%	circuit

are factorized occupying only half of the available cores on a node. This limitation affects both implementations and in the experiments that follow, both implementations are executed occupying the same number of cores.

Understanding performance differences between the two implementations is very difficult. While both implement the same algorithm, the order of the operations may differ. In fact, in SuperLU_DIST, two communication phases are asynchronous, but block row communication is synchronous; in contrast, communication is always asynchronous in Tarragon. In addition, because of the inherent load imbalance, the communication cost perceived by the application includes delays due to load imbalance, which cannot be overlapped, and that expose communication overheads in Tarragon.

5.3.1 Results on Abe

The first experiment on Abe illustrates the impact on performance of memory copies that, in Tarragon, take place during communication. When a task sends the same data to different tasks, it must send a message to each destination. As a result, the same data is copied once per destination task.

To avoid such inefficiency, Tarragon enables *tagging*: by annotating the edges of the graph with tags, an application can notify the run-time system of which edges will be carrying the same data (see Section 3.3). Then, during the execution of the graph, every time that a message is sent through a tagged edge, the message is also sent automatically over identically tagged edges, avoiding the extra memory copies. For example, in Figure 5.6, the edges from the D task to the L tasks carry the same data and are therefore be tagged. Similarly, the edges from the D task to the U tasks are tagged, as are the edges from the L tasks and from the U tasks to the LU tasks.

Table 5.2 shows the peak performance that the Tarragon implementation achieves in LU factorization with and without tagging. Overall, tagging improves performance, and it leads to an average 1.02 speedup. However, the improvement is observed only on certain matrices. In fact, the effectiveness of tagging depends on the sparsity pattern, which determines the number of equal messages that are sent by each task. Other factors also affect the outcome. For example, even when sending many messages at once, not all the messages are communicated concurrently, and avoiding copies and posting

communication requests faster may not result in much faster communication.

Table 5.2: Comparison of peak performance on Abe. The table compares the peak performance that the Tarragon implementation achieves with and without tagging.

Matrix	No Tagging		Tagging		Speedup
	GFLOP/s	Cores	GFLOP/s	Cores	
bbmat	13.9	16	14.4	16	1.03
g7jac200	14.3	32	14.3	32	1.00
inv-extrusion-1	8.3	16	8.6	16	1.04
matrix31	12.4	16	12.7	16	1.03
mixing-tank	17.2	16	17.7	16	1.03
nasasrb	8.6	16	8.7	16	1.01
stomach	9.3	8	9.5	16	1.02
torso1	13.3	16	13.4	16	1.01
twotone	2.9	16	2.9	16	1.00
dense	235.4	512	235.4	512	1.00
dds15	19.5	64	19.5	64	1.00
matrix181	76.3	512	82.1	256	1.08

The next experiment compares the Tarragon implementation, with tagging enabled, to the SuperLU_DIST implementation. The results are shown in two tables: Table 5.3 shows the results of the set of small matrices, solved on 1 to 32 cores, Table 5.4 shows the results of the set of large matrices, solved on 16 to 512 cores.

For most of the small matrices, the peak performance (timing in boldface) is achieved on a small number of cores due to load imbalance and the little available parallelism. However, in many cases the Tarragon implementation achieves its peak on a larger number of cores. However, the Tarragon implementation is less efficient in some cases in which, on an equal number of nodes, it is slower than the SuperLU_DIST implementation. Overall, on the small matrices, the Tarragon implementation achieves a 1.09 average speedup over the SuperLU_DIST implementation.

On two of the large matrices, the Tarragon implementation outperforms the

SuperLU_DIST implementation. However, on matrix *matrix181*, the Tarragon implementation does not improve its performance when scaling from 256 cores to 512 cores, whereas the SuperLU_DIST implementation, which is significantly slower on 256 cores, enjoys a performance improvement scaling to 512 cores. As a result, the SuperLU_DIST implementation is faster than the Tarragon implementation. Overall, on the large matrices, the Tarragon implementation achieves a 1.05 average speedup over the SuperLU_DIST implementation.

To summarize, Table 5.5 shows the peak performance achieved by both implementations on each matrix. The highest performance is achieved on the dense matrix owing to even load distribution and to the presence of large dense blocks; two ideal conditions which enable efficient computations and good scaling. In the factorization of the dense matrix, the Tarragon implementation is faster than the SuperLU_DIST implementation and achieves a 1.16 speedup, indicating a better overlap. On most of the matrices, the Tarragon implementation outperforms the SuperLU_DIST implementation. Only in two occurrences is Tarragon outperformed by the SuperLU_DIST implementation, in which cases load imbalance may be exposing overheads in Tarragon. For example, though tagging avoids duplicate messages, data in Tarragon is serialized by the run-time system whereas SuperLU_DIST sends the data in the format they are is stored. As a result, when load imbalance causes processes to idle because of a dependency, there are no opportunities for overlap and the communication delay directly affect the running time. Overall, the Tarragon version achieves better performance than SuperLU_DIST and achieves an average 1.08 speedup on Abe.

Table 5.3: Comparison of running time on Abe. The table compares the LU factorization in SuperLU_DIST to the Tarragon implementation. The comparison, which is on the set of small matrices, is based on the wallclock time. Timings are given in seconds. Boldface timings indicate the best performance achieved on a given matrix.

Implementation	Matrix	Processors			
		1	8	16	32
SuperLU_DIST	bbmat	13.75	2.45	**2.03**	2.35
Tarragon	bbmat	13.35	2.57	**1.78**	2.09
SuperLU_DIST	g7jac200	29.46	5.37	**3.55**	3.85
TAR	g7jac200	28.61	5.90	3.38	**3.03**
SuperLU_DIST	inv-extrusion-1	9.03	1.54	**1.25**	1.51
Tarragon	inv-extrusion-1	9.07	1.61	**1.12**	1.34
SuperLU_DIST	matrix31	1.53	**0.45**	0.47	0.51
Tarragon	matrix31	1.49	0.46	**0.40**	0.47
SuperLU_DIST	mixing-tank	7.10	1.40	**1.19**	1.24
Tarragon	mixing-tank	7.22	1.54	**1.02**	1.16
SuperLU_DIST	nasasrb	2.86	**0.81**	1.03	1.37
Tarragon	nasasrb	2.88	0.85	**0.80**	1.09
SuperLU_DIST	stomach	28.32	**5.51**	6.11	9.91
Tarragon	stomach	28.56	6.03	**4.93**	7.60
SuperLU_DIST	torso1	6.79	**1.61**	1.70	2.40
Tarragon	torso1	6.83	1.89	**1.69**	1.99
SuperLU_DIST	twotone	18.89	3.56	**3.12**	4.52
Tarragon	twotone	18.18	3.70	**2.98**	3.47

Table 5.4: Comparison of running time on Abe. The table compares the LU factorization in SuperLU_DIST to the Tarragon implementation. The comparison, which is on the set of large matrices, is based on the wallclock time. Timings are given in seconds. Boldface timings indicate the best performance achieved on a given matrix.

Implementation	Matrix	Processors					
		16	32	64	128	256	512
SuperLU_DIST	dds15	40.75	31.40	**28.66**	28.98		
Tarragon	dds15	43.47	34.63	**27.72**	34.19		
SuperLU_DIST	matrix181	160.93	79.33	51.31	31.75	27.41	**22.41**
Tarragon	matrix181	216.90	103.70	53.10	35.70	**23.23**	24.23
SuperLU_DIST	dense	12.89	11.74	5.04	3.01	1.75	**1.68**
Tarragon	dense	16.95	8.56	4.91	2.77	1.77	**1.45**

Table 5.5: Comparison of peak performance on Abe. The table compares the peak performance that the two implementations achieve in the LU factorization.

Matrix	SuperLU		Tarragon		Speedup
	GFLOP/s	Cores	GFLOP/s	Cores	
bbmat	12.6	16	14.4	16	1.14
g7jac200	12.2	16	14.3	32	1.17
inv-extrusion-1	7.7	16	8.6	16	1.10
matrix31	11.3	8	12.7	16	1.13
mixing-tank	15.2	16	17.7	16	1.17
nasasrb	8.6	8	8.7	16	1.01
stomach	8.5	8	9.5	16	1.12
torso1	14.1	8	13.4	16	0.95
twotone	2.8	16	2.9	16	1.05
dense	203.2	512	235.4	512	1.16
dds15	19.0	64	19.5	64	1.03
matrix181	85.9	512	82.1	256	0.96

5.3.2 Results on Kraken

The first experiment on Kraken examines the impact on performance of memory copies involved with communication. Table 5.6 shows the peak performance that the Tarragon implementation achieves in LU factorization, with and without tagging. On Kraken, tagging improves performance on all matrices and it leads to an average speedup of 1.06, indicating that communication delays on Kraken account for a higher fraction of the running-time than on Abe (on Abe, the average speedup achieved by tagging is 1.02).

Table 5.6: Comparison of peak performance on Kraken. The table compares the peak performance that the Tarragon implementation achieves with and without tagging.

Matrix	No Tagging		Tagging		Speedup
	GFLOP/s	Cores	GFLOP/s	Cores	
bbmat	16.9	12	17.3	12	1.02
g7jac200	15.6	12	15.8	12	1.01
inv-extrusion-1	11.1	12	11.4	12	1.03
matrix31	23.1	24	24.3	24	1.05
mixing-tank	12.7	24	13.8	24	1.09
nasasrb	14.6	12	14.9	12	1.02
stomach	14.3	12	14.4	12	1.01
torso1	19.9	12	22.7	12	1.14
twotone	4.6	12	4.7	12	1.02
dense	220.2	120	237.0	120	1.08
dds15	88.4	48	101.9	48	1.15
matrix181	20.8	120	22.6	120	1.09

The next experiment compares the Tarragon implementation, with tagging enabled, to the SuperLU_DIST implementation. The results are shown in two tables: Table 5.7 shows the results of the set of small matrices, solved on 1 to 36 cores, Table 5.8 shows the results of the set of large matrices, solved on 16 to 120 cores.

For most of the small matrices, the peak performance (timing in boldface) is

achieved on 12 cores, that is, on just a single node of Kraken. Communication delays have a significant impact on performance and limit the scalability of the application. Even on Abe, peak performance on the small matrices is achieved with at most 16 cores. Kraken has 12 cores per node and using an additional node, therefore introducing off-node communication, does not lead to a performance improvement. Overall, on the small matrices, the Tarragon implementation achieves a 1.02 average speedup over the SuperLU_DIST implementation.

On the large matrices, the peak performance of the two implementations is within a 1% difference on *matrix181* and the dense matrix, but on matrix *dds15*, the Tarragon implementation achieves a 1.08 speedup over the SuperLU_DIST implementation. In that case, the Tarragon implementation enjoys a performance improvement when using 60 cores instead of 48, leading to its peak performance, whereas the SuperLU_DIST implementation achieves its best on 48 cores. In this case, the ability to scale to a larger number of cores leads to the performance advantage observed. Overall, on the large matrices, the Tarragon implementation achieves a 1.02 average speedup over the SuperLU_DIST implementation.

To summarize, Table 5.9 shows the peak performance achieved by both implementations on each matrix. On most matrices, the Tarragon implementation outperforms the SuperLU_DIST implementation by a small margin and, overall, it achieves a 1.02 speedup. Only in two cases the SuperLU_DIST implementation is slightly faster, with a less than 2% difference in peak performance, in which case the Tarragon implementation may be penalized by its overheads.

Table 5.7: Comparison of running time on Kraken. The table compares the LU factorization in SuperLU_DIST to the Tarragon implementation. The comparison, which is on the set of small matrices, is based on the wallclock time. Timings are given in seconds. Boldface timings indicate the best performance achieved on a given matrix.

Implementation	Matrix	Processors			
		1	12	24	36
SuperLU_DIST	bbmat	11.76	**1.47**	1.54	1.78
Tarragon	bbmat	10.66	**1.41**	1.47	1.61
SuperLU_DIST	g7jac200	23.44	**2.80**	3.15	3.31
Tarragon	g7jac200	21.31	**2.74**	2.87	3.00
SuperLU_DIST	inv-extrusion-1	7.29	**0.91**	0.97	1.25
Tarragon	inv-extrusion-1	6.45	**0.84**	0.89	1.05
SuperLU_DIST	matrix31	1.26	0.23	**0.21**	0.24
Tarragon	matrix31	1.17	0.23	**0.21**	0.23
SuperLU_DIST	mixing-tank	5.88	0.74	**0.68**	0.79
Tarragon	mixing-tank	5.48	0.74	**0.68**	0.74
SuperLU_DIST	nasasrb	2.33	**0.46**	0.79	1.01
Tarragon	nasasrb	2.17	**0.47**	0.63	0.70
SuperLU_DIST	stomach	23.76	**3.27**	6.77	9.54
Tarragon	stomach	20.89	**3.25**	6.19	6.80
SuperLU_DIST	torso1	6.51	**1.05**	1.90	2.18
Tarragon	torso1	6.27	**1.00**	1.84	2.12
SuperLU_DIST	twotone	16.29	**1.92**	4.22	4.59
Tarragon	twotone	14.47	**1.89**	3.61	4.99

Table 5.8: Comparison of running time on Kraken. The table compares the LU factorization in SuperLU_DIST to the Tarragon implementation. The comparison, which is on the set of large matrices, is based on the wallclock time. Timings are given in seconds. Boldface timings indicate the best performance achieved on a given matrix.

Implementation	Matrix	Processors				
		24	36	48	60	120
SuperLU_DIST	dds15	29.04	24.71	**23.97**	26.95	
Tarragon	dds15	35.29	29.13	24.71	**22.42**	
SuperLU_DIST	matrix181	45.56	33.10	26.69	24.70	**17.47**
Tarragon	matrix181	44.05	35.07	26.47	23.27	**17.41**
SuperLU_DIST	dense	6.25	4.25	3.21	2.66	**1.43**
Tarragon	dense	6.11	4.10	3.03	2.53	**1.44**

Table 5.9: Comparison of peak performance on Kraken. The table compares the peak performance that the two implementations achieve in the LU factorization.

Matrix	SuperLU		Tarragon		Speedup
	GFLOP/s	Cores	GFLOP/s	Cores	
bbmat	16.6	12	17.3	12	1.04
g7jac200	15.5	12	15.8	12	1.02
inv-extrusion-1	10.5	12	11.4	12	1.08
matrix31	24.3	24	24.3	24	1.00
mixing-tank	13.8	24	13.8	24	1.00
nasasrb	15.2	12	14.9	12	0.98
stomach	14.3	12	14.4	12	1.01
torso1	21.6	12	22.7	12	1.05
twotone	4.6	12	4.7	12	1.03
dense	238.7	120	237.0	120	0.99
dds15	95.3	48	101.9	60	1.07
matrix181	22.6	120	22.6	120	1.00

5.3.3 Discussion

The Tarragon implementation takes advantage of tagging as a way to avoid unnecessary memory copies when communicating the same data to different tasks. In MPI, such optimization is not required because the MPI implementation allows the same buffer to be sent to different destinations concurrently. Tarragon does not recognize tasks that send identical messages to different tasks, and therefore it creates copies of the same message. There are two situations when this occurs. First, the destination field of the message differs. Second, if the message is not serialized, it is serialized once for each destination and stored in a different buffer. Tagging, which is based on graph metadata, avoids such inefficiencies and demonstrates how high level abstractions support performance tuning.

It is difficult to compare the SuperLU_DIST implementation and the Tarragon implementation, due to irregularity that characterizes the problem. But the results, overall, suggest that the Tarragon implementation achieves overlap. In particular, on Abe, in the factorization of the dense matrix, Tarragon achieves a 1.16 speedup over the SuperLU_DIST implementation. However, the same performance is not achieved on Kraken on the dense matrix, where the Tarragon factorization is 1% slower. In the Tarragon implementation, the ability to achieve overlap may be hindered because on Kraken, communication is less expensive and Tarragon experiences higher overheads than on Abe (see Section 3.5). In fact, even in the factorization of the dense matrix, messages are only tens to few hundreds of kilobytes long.

In many cases, there is no significant difference in performance between the two implementations, although on average, the Tarragon implementation is slightly faster than the SuperLU_DIST implementation. While both implementations often achieve their peak on the same number of cores, in some cases an implementation can enjoy a speedup over the other by using more cores. In these cases, differences in the implementations, such as different scheduling and overheads, determine which implementation performs better. Nevertheless, the performance of the Tarragon implementation is never below 5% of the performance of the SuperLU_DIST implementation, and it is better in most of the cases, proving its ability to achieve high performance and overlap.

While Tarragon achieves overlap and meets the performance of the

SuperLU_DIST implementation, which is also overlapping communication with computation, the potential of applying Tarragon to sparse LU factorization is not completely expressed. In particular, the data decomposition and mapping imposed on Tarragon limits its ability to execute one process per shared memory node and to execute fine-grained tasks. A finer decomposition would create more opportunities for scheduling tasks adapting to communication delays and achieving better overlap. Future research should explore ways to enable the conditions that are most favorable to Tarragon, starting from the symbolic factorization.

5.4 Acknowledgments

This chapter, in part, is currently being prepared for submission for publication. Pietro Cicotti; Scott B. Baden. The dissertation author is the primary investigator and author of this material.

References

[cem] Center for Extended MHD Modeling (CEMM). `http://w3.pppl.gov/cemm`.

[com] Community Petascale Project for Accelerator Science and Simulation (COMPASS). `https://compass.fnal.gov/`.

[Duf89] Duff, I. S. and Grimes, Roger G. and Lewis, John G. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1):1–14, 1989.

[Gol96] Golub, Gene H. and Van Loan, Charles F. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[Law79] Lawson, C. L. and Hanson, R. J. and Kincaid, D. R. and Krogh, F. T. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5:308–323, September 1979.

[Li,98] Li, Xiaoye S. and Demmel, James W. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–17, Washington, DC, USA, 1998. IEEE Computer Society.

[Li,05] Li, Xiaoye S. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, 2005.

[R. 00] R. Clint Whaley, Antoine Petitet and Jack Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. Technical Report UT-CS-00-449, September 2000.

[Tim94] Timothy A. Davis. university of Florida Sparse Matrix Collection. 1994.

[Xia03] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

Chapter 6

Dynamic programming

6.1 The Motif

Dynamic programming is a method for solving a problem by breaking down the problem into simpler overlapping subproblems. Dynamic programming is widely applicable, especially to optimization problems in which an optimal solution can be built on optimal solutions of subproblems.

A common implementation technique relies on a multi-dimensional table representing all the subproblem instances and their solution. The table is filled incrementally, according to the underlying dependence structure, and the solution is reached when the table is completely filled.

The behavior of a dynamic programming algorithm depends on the problem to be solved. However, since the underlying recurrence relation describing the dependence structure used to combine the subproblem solution is repeatedly applied, until the final solution is reached, the computational pattern is repeated over time. In addition, the underlying communication pattern exhibits the dependence structure that characterizes the recurrence relation, and it is also repeated over time. Therefore, a dynamic programming algorithm often exhibits some regularity in its computation and communication patterns.

Data reuse and locality can vary greatly. The recurrence relation and the amount of overlap in subproblems determine how often data is reused, and the type of locality characterizes the computation.

6.2 Needleman-Wunsch

In computational molecular biology, sequence comparison (e.g. DNA) is a fundamental primitive operation and is at the basis of more complex manipulations and analysis [Alu05]. For example, the comparison between two sequences can provide a measure of how similar the sequences are and suggest some kind of correlation. Tools that solve these types of problem are used in DNA sequence manipulations, such as fragment assembly, and analysis, such as database searches based on similarity.

One way to define similarity is to quantify the minimum number of transformations needed to make two sequences equal. Other approaches are concerned not only with similarity, but rather, with an *alignment* that maximizes the number of matching characters that occupy the same position in the sequence. Usually this type of alignments allows using special characters to accommodate differences in sequences and to maximize the number of matches.

Needleman and Wunsch were the first to design an algorithm that finds a provably optimal alignment [Nee70]. Their algorithm is based on the observation that an optimal alignment can be built by extending optimal alignments of prefixes of the sequences. The observation led to a dynamic-programming algorithm that incrementally builds the final solution. Subsequently, other dynamic-programming algorithms were proposed to solve different formulations of the problem. For example, the Smith-Waterman algorithm for *local alignments* [Smi81] is perhaps the most well known.

The model problem considered in this section is to find an optimal global alignment between two DNA sequences defined on the alphabet $\Sigma = \{A, C, G, T\}$. The implementation is a variation of Needleman-Wunsch, but the results presented apply to other variants, such as Smith-Waterman, that are formulated using the same dynamic-programming approach.

The Needleman-Wunsch algorithm is based on the computation of an *edit matrix* for the given input sequences, which are usually referred to as *reference* and *query*. The edit matrix represents the optimal alignments between prefixes of the input sequences. Rows and columns of the edit matrix are associated with the characters of the query and the reference, respectively, and each entry of the edit matrix is the *score* of the alignment between the corresponding prefixes of the sequences. The score measures the quality of

the alignment. At the end of the computation, the bottom-right corner holds the score of the complete alignment. As an example, Figure 6.1a illustrates the edit matrix that results from the sequences *ACCT* and *ACTG*.

Scores are assigned according to a recurrence relation and a *scoring function* $\Sigma \times \Sigma :\rightarrow \mathbb{Z}$. The recurrence relation defines the dependencies between entries and how the scoring function is applied when prefixes are extended. The scoring function assigns scores to matches and mismatches, and assigns a penalty to the insertion of a *gap* (_), representing a simple genetic mutation that caused insertion or deletion of a base.

This dissertation uses the recurrence relation with linear gap penalties shown in Equation 6.1. *GAP* is a given constant for a single gap penalty (-1), and *sim* is the scoring function. The scoring function assigns a negative unit score for mismatches, and a unit score for matches. Though the resulting alignment is affected by the choice of the scoring function and the gap penalty, the number of operations and performance do not. Choosing an optimal scoring function is out of the scope of this dissertation and will not be discussed.

$$A[i,j] = max \begin{cases} A[i-1,j-1] + sim(r(j),q(i)) \\ A[i,j-1] + GAP \\ A[i-1,j] + GAP \end{cases} \tag{6.1}$$

Needleman-Wunsch has two phases: the first phase computes the *edit matrix*; the second phase traverses the edit matrix along the path that corresponds to the optimal alignment. The computational complexity of the first phase is $O(mn)$, where m and n are the length of the two sequences, and the cost of computing an entry of the matrix is constant. The assumption holds for the class of scoring functions considered in this dissertation. The computational complexity of the second phase is $O(m+n)$ because the length of the path that traverses the matrix from the bottom-right corner to the top-left corner is bounded by the lengths of the two sequences; each step is either a move up or a move to the left, and has constant cost. It follows that the first phase accounts for the majority of the computation time. For this reason, only the first phase is considered.

The recurrence formula in Equation 6.1 establishes data dependencies such that each entry of the matrix depends on three previously filled entries. In particular, each

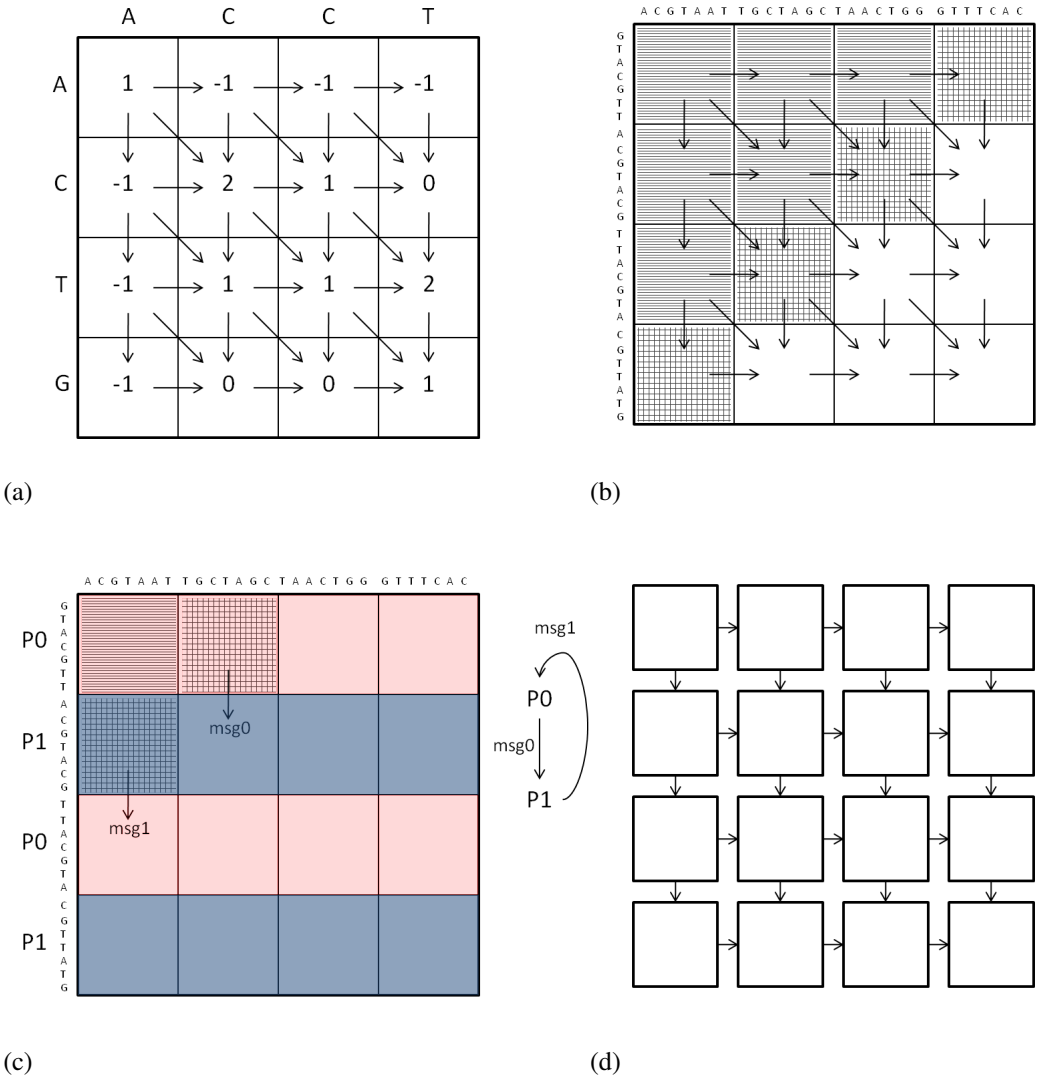


Figure 6.1: Needleman-Wunsch alignment. Figure 6.1a shows the edit matrix with arrows representing the dependencies between entries. Figure 6.1b shows the wavefront execution on a matrix decomposed into blocks, the arrows represent the dependencies between blocks. Striped blocks are filled; tiled blocks are ready for execution, that is the wavefront; empty blocks cannot be computed yet because of their dependencies. Figure 6.1c shows the wavefront after the completion of the second diagonal. The blocks are mapped to two processes forming a communication ring. The ring is represented on the right of the matrix. The diagonal dependency is implied and satisfied by communication along the columns. Figure 6.1d shows the corresponding task graph in Tarragon.

entry (i,j) depends on three entries in the previous row and previous column, namely entries (i-1,j), (i-1,j-1), (i,j-1), as shown in Figure 6.1a. The dependence structure makes the algorithm an ideal candidate for 2-dimensional wavefront parallelization [Lam74, Gen].

In wavefront algorithms, as dependencies are satisfied, entries are filled in parallel, along the anti-diagonal, creating a wavefront that advances through the table. Wavefront algorithms can also be blocked. With blocking, the dependencies are inherited by the blocks so that a block depends on three previously filled blocks: as with matrix entries, blocks on the top left, top, and left, respectively. Figure 6.1b illustrates the dependence structure. Starting from the top-left corner, as soon as the first submatrix is filled it satisfies the dependencies on the right, down, and down-right. Two blocks can then be computed concurrently, then three, and so on. Figure 6.1b illustrates the progress of the wavefront on a matrix decomposed into 16 submatrices (4×4).

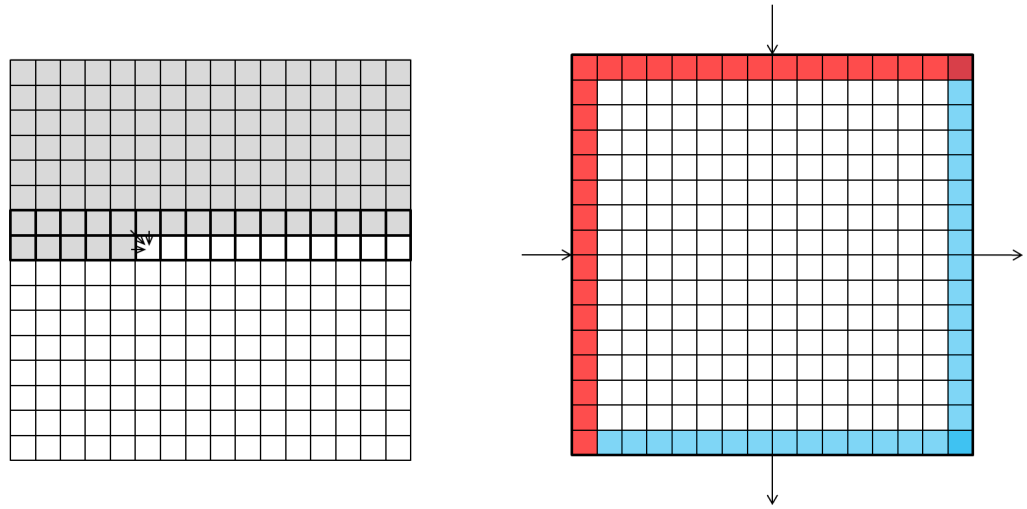
6.2.1 Reference Implementations

In this dissertation, two MPI implementations are used as reference: a *Synchronous* variant and an *Asynchronous* variant.

The Synchronous variant is written in MPI. In Synchronous, blocks are assigned to processes using a 1-dimensional *cyclic* mapping along the columns, such that row j is assigned to process $j \bmod p$, where p is the total number of processes. Each process fills the blocks of the matrix that it owns, row by row, left to right.

Each block is filled using a *linear-space* computation [Hir75]. Since each entry of the matrix depends on just three neighboring entries, a new row is computed accessing values on the preceding row of the matrix, and there is no need to store the entire block (Figure 6.2a). This principle is applied locally to the computation in each block. However, the traceback phase requires the content of the blocks. To preserve the ability to reconstruct the whole block, the left and top sides are stored together with the right column and the bottom row (Figure 6.2b). The right column and the bottom row are then sent to the neighboring blocks.

Although this solution is not linear in space, it reduces the amount of memory required, considerably improving the scalability of the algorithm. Globally, the algorithm



(a) (b)

Figure 6.2: Hirschberg linear-space computation. Figure 6.2a shows the computation of a block where only two rows are stored in memory. At any stage, each row can be computed as long as the previous row is stored, as indicated by the dependencies. Figure 6.2b shows the block and columns required to compute the block and the communication pattern. Left and top entries are received by neighboring blocks, bottom and right entries are sent to neighboring blocks.

requires $O(\frac{m\times n}{r\times c}\times(r+c))$ space, where r and c are the number of rows and columns in a block, because two rows and two columns are stored to represent each block, and there are $\frac{m\times n}{r\times c}$ blocks. However, storing the entire blocks requires $O(m\times n)$ space, that is, the total amount of space required to store the entire matrix.

r and c define not only the space used, but also the granularity of the computation. In addition, the space saving technique renders the computation of a block CPU bound as it generates virtually no memory traffic. In the experiments that follow, r and c were varied to tune performance. Memory size was not a limiting factor in selecting the optimal values of r and c.

Communication takes place after each process completes a block. Processes send the bottom of the computed block, forming a communication ring, as illustrated in

Figure 6.1c. The ring is implemented using blocking communication primitives and the exchange is done in two phases: first even-rank processes send and odd-rank processes receive, then even-rank processes receive and odd-rank processes send.

The Asynchronous variant implements the communication phase differently than the Synchronous variant. In Asynchronous, processes maintain a communication schedule, which defines in which order communication will take place, and post non-blocking receives in advance. In this way, communication requests a posted in advance enabling overlap, and avoiding extra memory copies within MPI. Then, as the computation progresses, processes follow the communication schedule retiring completed receives and posting new ones. In this way, each process tracks a window of pending receives using a circular queue. Similarly, sends are posted as soon as possible but retired only later, as the window of pending sends slides forward.

6.2.2 Tarragon Implementations

The MPI reference implementations are compared to two Tarragon variants: *Block*, and *Panel*.

In the Block variant, the graph is obtained by the blocked representation of the matrix and the corresponding dependencies, with the exception that there is no diagonal edge. The diagonal dependency is implied by the vertical edge. Figure 6.1d illustrates the resulting graph. Each submatrix is associated with a task. Tasks execute after receiving a message from the task on the left, and the task from the top. The two messages provide data on the leftmost column of the block and the first row, and therefore satisfy the dependencies enabling the computation in the associated block. When executing, the task fills the associated block of the edit matrix. Finally, the task terminates by sending the last row and the rightmost column to the task below and to the task on the right.

In the Panel variant a whole row of blocks is associated with a task. Blocks are still the unit of computation; because each block in a row can execute only after the preceding block, the amount of parallelism is not reduced with respect to Block. However, the size of the graph is reduced, and so is the memory utilized to store the graph. In addition, scheduling overheads are also reduced, and the run time system (RTS) can therefore be more responsive in handling communication events, an issue

that appeared to affect the performance in Block.

6.3 Performance Evaluation

The four variants are compared in the computation of the edit matrix under weak scaling. The number of entries of the edit matrix per core is kept constant as the number of cores increases.

The four variants are compared on two platforms: Abe and Kraken. Abe is an Intel powered cluster, Kraken is a Cray XT5. Detailed specifications of the two platforms are given in Appendix A.

6.3.1 Results on Abe

The four variants are compared in the computation of the edit matrix under experiments are repeated weakly scaling the length of the sequences, such that the number of entries of the edit matrix per core is constant while the number of cores increases.

The first experiment on Abe evaluates the performance on a single node comparing the four variants on 2, and 8 cores, using weak scaling. On 8 cores, the compared sequences are twice as long as in the 2 core test. Since the computational complexity is quadratic, both tests are expected to finish in the same amount of time. In addition, since the communication cost on a node is very low, the results give an empirical indication of whether the application is CPU bound or memory bound, a characteristic that affects the optimal configuration of the service threads in Tarragon.

The results are given in Table 6.1. The running times of the Synchronous and the Asynchronous variant are very similar on both 2 and 8 cores, indicating that the application is CPU bound.

Since the application is CPU bound in Tarragon, dedicating a CPU core to a service thread results in a performance penalty. Therefore, the Tarragon variants are tested in both multi-threaded mode, that is, with one process per node, and single-threaded mode, that is with one process per core. In multi-threaded mode each core is occupied by a worker thread, and one core is shared by a service thread and a worker thread. In

Table 6.1: Running times on Abe. The table presents single-node running times in comparing two sequences whose length is reported in the first column. In addition to the running times of the Synchronous and the Asynchronous variant, the table lists the running times of 4 configurations: the Block variant running single-threaded (TB1), and multi-threaded (TBM), and the Panel variant running single-threaded (TP1), and multi-threaded (TPM). Times are given in seconds.

Length	Cores	Synchronous	Asynchronous	TB1	TBM	TP1	TPM
2^{16}	2	11.5	11.4	10.1	10.9	10.4	10.4
2^{17}	8	10.9	10.8	10.0	19.2	10.3	11.4

single-threaded mode, there is only one thread per core that alternates service execution with task execution.

On 2 cores, all the variants achieve approximately the same performance. However, on 8 cores, Block running multi-threaded is slower than the other variants. In Block, the run-time system is frequently scheduling tasks and synchronizing with worker threads due to the large number of tasks. The resulting overheads heavily affect the performance of the Block variant. In contrast, Panel has coarser grained tasks and overheads do not appear to affect performance.

When running in single-threaded mode, Block is the fastest on 8 cores. Since there is no thread synchronization, Block achieves 8% and 9% speedup compared to the Synchronous and the Asynchronous variants, respectively.

Another aspect of the application that affects performance is the order of execution of the blocks. If the rows of blocks are considered as stages of a pipeline, blocks that are on the same row are processed in the same stage, sequentially, from left to right; the computation pipeline extends vertically.

In the MPI implementations the pipeline is implicitly defined as part of the hard-coded schedule, in which the top rows of blocks are completed first. The schedule is optimal for locality because it ensures that blocks are executed from left to right, starting from the top rows, and the ordering matches the order in which data are stored in memory.

By default, tasks in Tarragon are scheduled dynamically and without any prede-

Table 6.2: Running times on Abe with and without prioritized execution. The table lists the speedup achieved by prioritizing the tasks in the Block variant running single-threaded (TB1), the Panel variant running single-threaded (TP1), and the Panel variant running multi-threaded.

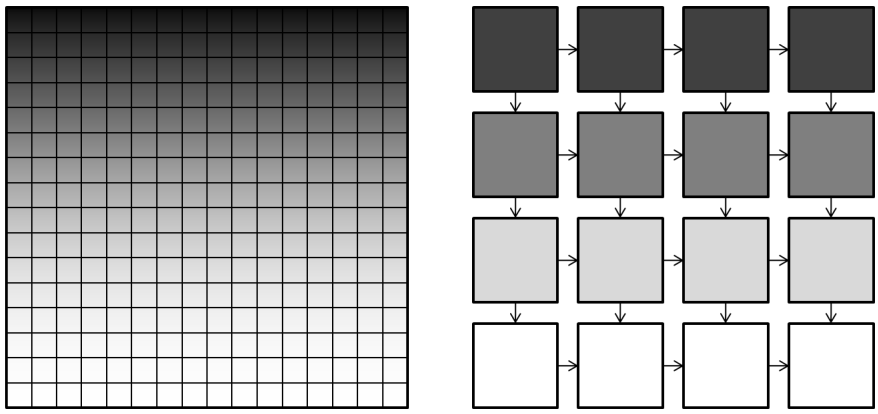
Length	Cores	TB1	TP1	TPM
2^{17}	8	1.00	1.00	1.00
2^{18}	32	1.00	1.00	1.03
2^{19}	128	1.00	1.00	1.06
2^{20}	512	1.07	1.00	1.26

fined order. However, if the order of execution does not match the order in which data are stored in memory, some spatial locality is lost. In addition, when running in multi-threaded mode, concurrent execution of tasks with edges directed to remote tasks may cause an inefficient burst of communication, increasing communication and reducing the overlap.

To establish an efficient scheduling policy, task priority attributes are introduced as task graph metadata giving higher priority to the tasks that are closer to the top of the matrix, as illustrated in Figure 6.3. In this way, tasks with low priority can execute when all the high priority tasks are blocked, in response to communication delays; otherwise, the computation proceeds according to the optimal schedule.

Table 6.2 shows the difference in performance between the Tarragon variants, with and without prioritized execution. In some cases, prioritized execution is crucial to realize high performance. There are two reasons. The first reason is that the order induced by priorities exhibits better locality. The second reason is that the order induced by priorities promotes a more efficient communication schedule. This is especially the case for the Panel variant, when it executes in multi-threaded mode; when more than one thread generates messages directed to another process, concurrency in communication may limit the transfer rate. However, concurrency is avoided when following the order induced by the priorities.

When executing in single-threaded mode, the Panel variant executes tasks following the optimal order of execution and prioritized execution has virtually no effect



(a) (b)

Figure 6.3: Priority assignment. Figure 6.3 shows priority assignment to blocks of a matrix in the MPI codes: darker blocks have higher priorities. Similarly, Figure 6.3b shows how priorities are assigned to tasks in a 4x4 task graph.

on performance.

The next experiment compares the performance of the Block variant, which is the best performing variant that uses Tarragon, to the performance of the MPI variants. Table 6.3 shows that in scaling up to 512 cores, the Block variant is faster than both MPI variants. Both the Block variant and the Asynchronous variant achieve a significant speedup over the Synchronous variant, indicating that both reduce the communication cost significantly by overlapping communication with computation. However, the Block variant is also faster than the Asynchronous variant because it is able to adapt the schedule to tolerate communication delays, achieving a 1.08 speedup over Asynchronous.

6.3.2 Results on Kraken

To evaluate the performance on a single Kraken node, the first experiment compares performance on 3 and 12 cores. The results are given in Table 6.4. As expected, the execution time of the Synchronous variant does not change, which is ideal because the problem is scaled weakly. Also on Kraken the application is CPU bound.

On 3 cores, the Tarragon variants run in times comparable to Synchronous and

Table 6.3: Running times on the Abe. The table presents running times of the Synchronous variant, the Asynchronous variant, and the Block variant running single-threaded (TB1). The table also reports the speedup of the Block variant over the Synchronous variant (TB1/S), and over the Asynchronous variant (TB1/A). Times are given in seconds.

Length	Cores	Synchronous	Asynchronous	TB1	TB1/S	TB1/A
2^{17}	8	10.9	10.9	10.0	1.09	1.09
2^{18}	32	12.1	11.3	10.2	1.19	1.11
2^{19}	128	12.6	11.6	10.6	1.19	1.09
2^{20}	512	14.2	13.2	12.2	1.16	1.08

Asynchronous. However, as previously observed, Block running multi-threaded is slower than the other variants because of its high overheads. There is little improvement in Block running in multi-threaded mode because of the modest communication cost. In fact, intra-node communication is more efficient on Kraken than on Abe (see Section 3.5).

Table 6.4: Running times on Kraken. The table presents single-node running times in comparing two sequences whose length is reported in the first column. In addition to the running times of the Synchronous and the Asynchronous variant, the table lists the running times of 4 configurations: the Block variant running single-threaded (TB1), and multi-threaded (TBM), and the Panel variant running single-threaded (TP1), and multi-threaded (TPM). Times are given in seconds.

Length	Cores	Synchronous	Asynchronous	TB1	TBM	TP1	TPM
2^{17}	3	15.9	15.7	15.7	15.8	16.0	15.8
2^{18}	12	15.9	15.9	15.7	19.1	16.3	17.3

The second experiment assesses the effect of prioritized execution in the Tarragon variants. Table 6.5 shows the difference in performance between the Tarragon variants, with and without prioritized execution. As previously observed on Abe, the Panel variant executes tasks following the optimal order of execution when executing

single-threaded, hence its performance is not affected by priorities. Both Block, executing single-threaded, and Panel, executing multi-threaded, experience a significant speedup. However, the speedup in Panel is much lower than on Abe because of the lower communication cost. The speedup on Panel running in multi-threaded mode is lower on 768 cores than on 192. In that case, since the number of tasks available to run at any time decreases, and because Panel has by construction a smaller number of tasks than Block, there are fewer opportunities for scheduling tasks in an order that is different from the order induced by the priorities.

Table 6.5: Running times on Kraken with and without prioritized execution. The table lists the the speedup achieved by prioritizing the tasks in the Block variant running single-threaded (TB1), the Panel variant running single-threaded (TP1), and the Panel variant running multi-threaded.

Length	Cores	TB1	TP1	TPM
2^{18}	12	1.00	1.00	1.00
2^{19}	48	1.00	1.00	1.02
2^{20}	192	1.01	1.00	1.08
2^{21}	768	1.07	1.00	1.06

The next experiment compares the performance of the Block variant, which is the best performing variant that uses Tarragon, to the performance of the MPI variants. Table 6.6 shows the results scaling up to 768 cores. The Block variant is faster than both MPI variants. Both the Block variant and the Asynchronous variant achieve better performance than the Synchronous variant, but the improvement is much smaller than on Abe due to the lower communication costs on Kraken. Because of its ability to schedule tasks dynamically, the Block variant is also slightly faster than the Asynchronous variant.

6.3.3 Discussion

On both testbeds, results show that the Asynchronous variant is faster than the synchronous variant. The improvement is due to the ability to overlap communication

Table 6.6: Running times on the Kraken. The table presents running times of the Synchronous variant, the Asynchronous variant, and the Block variant running single-threaded (TB1). The table also reports the speedup of the Block variant over the Synchronous variant (TB1/S), and over the Asynchronous variant (TB1/A). Times are given in seconds.

Length	Cores	Synchronous	Asynchronous	TB1	TB1/S	TB1/A
2^{18}	12	15.9	15.9	15.7	1.01	1.01
2^{19}	48	16.7	16.6	16.2	1.03	1.02
2^{20}	192	17.1	16.9	16.5	1.04	1.02
2^{21}	768	19.0	18.5	18.4	1.03	1.01

with computation. The Block variant is also faster than the Synchronous variant indicating that overlap is automatically achieved with Tarragon. In addition, the Block variant also outperforms the Asynchronous variant because of its ability to dynamically adapt to communication delays, producing an optimal schedule that is otherwise unpredictable.

To achieve its peak performance, the Block variant has to ensure that an optimal scheduling policy is established. To do so, the graph is annotated with priorities. Prioritized execution improves performance on both platforms, proving that graph annotations enable optimizations and portable performance.

The speedup achieved by the Block variant is lower on Kraken than on Abe. This is a consequence of the fact that the communication cost is lower on Kraken than on Abe (see Section 3.5). In addition, a lower communication cost implies smaller delays that, on Abe, Tarragon was hiding by adapting the schedule.

6.4 Acknowledgments

This chapter, in part, is currently being prepared for submission for publication. Pietro Cicotti; Scott B. Baden. The dissertation author is the primary investigator and author of this material.

References

[Alu05] Aluru, S., editor. *Handbook of Computational Molecular Biology*, chapter 1. Computer and Information Science. Chapman & Hall/CRC, 2005.

[Gen] Gengler, Marc. An introduction to parallel dynamic programming. In Ferreira, Afonso and Pardalos, Panos, editor, *Solving Combinatorial Optimization Problems in Parallel*, volume 1054 of *Lecture Notes in Computer Science*, pages 87–114. Springer Berlin / Heidelberg.

[Hir75] Hirschberg, D. S. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[Lam74] Lamport, Leslie. The parallel execution of DO loops. *Commun. ACM*, 17:83–93, February 1974.

[Nee70] Needleman, Saul B. and Wunsch, Christian D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[Smi81] Smith, T. F. and Waterman, M. S. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

Chapter 7

Conclusion

7.1 Research Summary

This dissertation presented Tarragon, a programming model that supports latency-hiding applications in scientific computing. Tarragon introduces a novel task graph abstraction that enables the programmer to express parallelism and data dependencies, and that shields the programmer from the complexity of communication, threading, and scheduling details. The task graph and its attributes create a separation of structural and correctness concerns from performance concerns. In addition, by virtue of its data-driven execution model, Tarragon automatically overlaps communication with computation.

Tarragon is a generic programming model for scientific computations and it is applicable to problems that exhibit very different computation and data motion patterns. The Finite-Difference solver presented in Chapter 4 is characterized by regular decomposition and communication pattern, coarse granularity, and high locality. The sparse linear system solver presented in Chapter 5 is characterized by highly irregular decomposition and communication pattern, fine granularity, and little locality but some degree of reuse. The sequence alignment tool presented in Chapter 6 is characterized by regular decomposition and communication pattern, fine granularity, and no reuse.

Tarragon supports the definition of library extensions for specific classes of problems. Such extensions provide programmers with a set of abstractions that reduce the programming effort, as demonstrated in the example in Section 4.4, and improve pro-

ductivity facilitating rapid development.

The results in the application studies demonstrate Tarragon's ability to achieve high performance and to realize overlap. In all of the applications, the Tarragon implementation meets or exceeds the performance of the corresponding overlapping MPI reference most of the times. In addition, even when split-phase coding affects the memory access pattern and negates the performance improvement due to overlap, Tarragon supports overlapping algorithms that preserve locality. For example, results with the Finite-Difference solver show that the MPI overlapping version suffers a performance loss compared to the non-overlapping MPI version, whereas the Tarragon version exceeds the performance of the non-overlapping MPI version owing to overlap and good locality.

The results of the sequence alignment application demonstrate that the Tarragon version achieves overlap and that its performance is maximized by an appropriate scheduling policy. While Tarragon always achieves overlap, default scheduling policies might cause execution to diverge from the critical path. However, with graph analysis and prioritization users can identify and establish the most appropriate scheduling policies. These results demonstrate that graph-metadata can be used to tune performance without affecting correctness of execution.

The results in LU factorization code indicate that certain conditions, in the case examined imposed by existing code, may limit the potential of Tarragon. While on Abe the Tarragon version clearly outperforms the SuperLU_DIST implementation, on Kraken, where communication is more efficient and overheads in Tarragon are higher, the Tarragon implementation improves performance only marginally. Tarragon's ability to achieve overlap, and hide also communication overheads, in the LU factorization code is hindered by the limited parallelism available. In fact, Tarragon has to adapt to the decomposition, mapping, and data structures of SuperLU_DIST. As a result, on Kraken, the Tarragon version exhibits an overall modest improvement compared to the performance of the SuperLU_DIST implementation.

In conclusion, this dissertation demonstrates that data-driven execution coupled with metadata abstractions support latency tolerance. In addition, Tarragon's programming model supports performance optimization techniques that are decoupled from the

algorithmic formulation and the control flow of the application code; while enabling performance tuning, the resulting separation of concerns between performance and correctness promotes performance portability.

7.2 Limitations

The current implementation of Tarragon introduces communication overheads that may limit performance in fine-grained applications. To alleviate the impact of these overheads, Tarragon supports message aggregation. However, message aggregation did not appear to be beneficial in the applications examined.

Tarragon's interoperability with legacy code may be limited. For example, when the MPI code imposes its Single Program Multiple Data execution model forcing Tarragon to run single-threaded. In addition, as it was the case in the LU factorization, if the MPI code dictates data structures and a data distribution that are inherently inefficient for highly parallel task formulations, a Tarragon application must adapt by defining tasks with coarser granularity than otherwise desirable. Both single-threaded execution and coarse granularity may reduce the available parallelism and hence hinder the ability of the run-time system to achieve overlap.

Finally, a limitation of this dissertation is represented by the scope of the application studies; an exhaustive study should address all the motifs [Col04, Asa06]. However, the applications considered exhibit diverse computation and communication characteristics common to many scientific applications, the target of Tarragon. In addition, the ability to achieve high performance with data-driven models has been demonstrated on other motifs, such as dense linear algebra [Jak09], unstructured grids [Bha00], and N-body methods [Jet08].

7.3 Future Research Directions

The explicit task graph representation used in Tarragon lends itself to analysis-driven performance optimizations. For example, graph analysis can suggest a task redistribution that minimizes inter-node communication and hence the overall communi-

cation cost. Using a richer set of metadata, perhaps collected partly by the application and partly by the run-time system, offers promise for even more complex optimizations. Load balancing is such an example; if the graph is annotated by the run-time system with *measured* load information, or by the application with *expected* load information, graph analysis can determine the optimal workload distribution.

Tarragon can be extended to support clusters of accelerators (i.e. GPUs), an area in which communication latencies are a major performance bottleneck [Nha]. The idea of supporting hybrid code that executes on clusters of multicore nodes with accelerators appears to be promising in a context in which metadata can support dynamically tuned scheduling. For example, by carefully assigning affinities tasks could elect to execute on either a processor core or an accelerator. Finding the optimal mapping of tasks to a set of heterogeneous processing elements while overlapping computation with communication is an open problem. Future research should focus on graph analysis and metadata annotation as a way for defining scheduling policies to optimize execution on clusters of accelerators.

Tarragon mostly delegates productivity concerns to library extensions. However, the use of libraries in refactoring and porting existing code bases to emerging architectures still requires some degree of redevelopment. A promising approach in refactoring existing code is to use automatic translation tools [Qui02]. To this end, automatic translation to apply transformations and optimize MPI code is an active area of research, including an effort for translating MPI code to Tarragon code.

References

[Asa06] Asanovic, Krste and Bodik, Ras and Catanzaro, Bryan Christopher and Gebis, Joseph James and Husbands, Parry and Keutzer, Kurt and Patterson, David A. and Plishker, William Lester and Shalf, John and Williams, Samuel Webb and Yelick, Katherine A. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[Bha00] Laxmikant V. Bhandarkar, Milind A. and Kalé. A Parallel Framework for Explicit FEM. In *Proceedings of the 7th International Conference on High Performance Computing*, HiPC '00, pages 385–394, London, UK, 2000. Springer-Verlag.

[Col04] Colella, Phil. Designing Software Requirements for Scientific Computing, 2004.

[Jak09] Jakub Kurzak and Jack Dongarra. Fully Dynamic Scheduler for Numerical Computing on Multicore Processors. Technical Report UT-CS-09-643, University of Tennessee, Knoxville, June 2009.

[Jet08] Jetley, P. and Gioachin, F. and Mendes, C. and Kale, L.V. and Quinn, T. Massively parallel cosmological simulations with ChaNGa. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –12, april 2008.

[Nha] Nhat Than Nguyen and Pietro Cicotti and Didem Unat and Scott B. Baden. Latency hiding in Multi-GPU Stencil Computations. *In Preparation*.

[Qui02] Quinlan, Daniel J. and Miller, Brian and Philip, Bobby and Schordan, Markus. Treating a User-Defined Parallel Library as a Domain-Specific Language. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 324–, Washington, DC, USA, 2002. IEEE Computer Society.

Appendix A

Testbed

A.1 Abe

Abe is the Dell Intel 64 Cluster at the National Center for Supercomputing Applications (NCSA). Abe is part of the Teragrid infrastructure [Cat07], and it is intended for highly scalable parallel applications. The system counts 1200 Dell PowerEdge 1995 nodes, hosting two Intel Xeons (Clovertown E5345) processors each, for a total of 8 cores per node, and 9600 cores for the whole system. Of the nodes, half are configured with 8GB of memory, and half are configured with 16GB of memory. Nodes are connected via InfiniBand (Mellanox Technologies MT25208 InfiniHost) and share 100TB of storage in a Lustre parallel file system.

Each node hosts two Intel Xeon processors running at 2.33GHz. The processor has four cores, each with 32KB L1 caches (data and instructions); the processor also has two 4MB L2 caches, which are each shared by two cores. The aggregate bandwidth available on a node is 21.33GB/s, which is provided by four DDR2-667 channels (two channels and 10.66GB/s per socket).

A.2 Kraken

Kraken is the Cray XT5 at the National Institute for Computational Science (NICS). Kraken is part of the Teragrid infrastructure [Cat07], and it is intended to pro-

vide a capability resource for computationally challenging problems. The system counts 9408 nodes, each hosting two AMD Opteron (Istanbul) processors for a total of 12 cores per node, and 112,896 cores for the whole system. Each node has 16GB or memory. Nodes are connected via Cray SeaStar (3D Torus) interconnect and share 2400TB of storage in a Lustre parallel file system.

Each node hosts two AMD Istanbul processors running at 2.6GHz. The processor has six cores, each with a 128KB L1 cache (data and instructions) and a 512KB L2 cache. The six cores also share a 6MB L3 cache. The aggregate bandwidth available on a node is 25.6GB/s, which is provided by four DDR2-800 channels (two channels and 12.8GB/s per socket).

References

[Cat07] Catlett, Charlie, et al. *TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications*, volume 16 of *Advances in Parallel Computing*. IOS Press, Amsterdam, 2007.

Appendix B

API Reference

B.1 Core API

The core API defines the basic classes required to define and construct the graphs, and to interact with the run-time system.

B.1.1 Task and Dependency

Task and *Dependency* are the building blocks of a graph. *Task* defines the unit of computation and is a base class that must be extended by the application developers. *Dependency*s connect *Task* objects, represent task dependencies, and enable inter-task communication.

Both *Task* and *Dependency* have metadata attributes that the run-time system can inspect. The attributes of *Task* are used to maintain information that both the task itself and the run-time system access and can modify. The attributes of *Task* Are:

_state , the scheduling state of the task,

_priority , the scheduling priority of the task,

_affinity , the worker affinity,

_last is the last worker that executed the task.

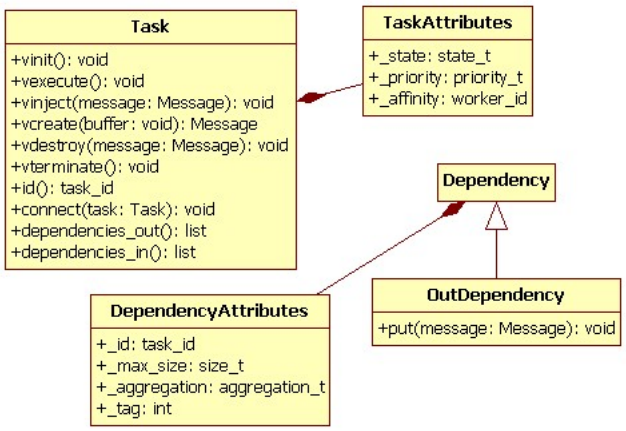


Figure B.1: Class diagram of Task and Dependency

The attributes of *Dependency* are initialized when the graph is constructed and may not be changed during execution. The attributes of *Dependency* are:

_id , is the id of the task that the *Dependency* connects to,

_max_size , is the cpacity of the *Dependency*, that is the largest amount of the data that can be sent along the edge at once,

_aggregation , enables data aggregation.

_tag , enables tagging.

Tasks definition

Tasks are the unit of computation and the application programmers are expected to extend the *Task* class to define customized tasks. The sub-class (class derived from *Task*) can access two fields of *Task*: the *_id* field, which is set to the identifier of the task, and the *_graph* field which is a reference to the graph the task belongs to. The sub-class inherits several virtual methods that may be overriden to define the behavior of the new type of task. The virtual methods of *Task* are:

vinit is invoked when the graph is initialized. Defining *vinit* is optional and should be done only when there is a logical distinction between operations that are part of the

initialization of the tasks and operations that are part of the actual computation. In addition, with respect to the object initialization, *vinit* is executed when the graph is already constructed making it possible to define a globally coordinated initialization of the tasks involving communication between tasks.

vexecute is invoked when the task is ready to be executed and it represents the actual task execution. Most of the application code is exected to be encapsulated in *vexecute*. When the task is ready is scheduled for execution is eventually assigned to a worker which invokes *vexecute*.

vinject(message) is a short message handler that is invoked when a message reaches a task. The method is used to deliver the mssage referenced by the argument *message*. In addition, the *vinject* method implements the semantics of the dependencies. By checking the state of a task, *vinject* determines whether the dependencies have been satisfied and the task is ready for execution. *vinject* is not intended to carry out extensive computations.

vcreate(buffer) is invoked whenever a message is received from a remote task. Data is received in serialized form and the run-time system uses *vcreate* to interpret the data and instantiate a *Message* object.

vdestroy(message) is invoked whenever a message is sent to a remote task. Since data is transferred asynchronously, Tarragon uses *vdestroy* to notify the source that the data transfer has been completed locally; there is no guarantee that the message has been delivered.

vterminate in invoked when there are waiting tasks but no process has tasks running. Tarragon detects quiescence and uses *vterminate* to notify each waiting task that the graph execution can be completed. On tasks that transition to the state indicating completion (*DONE*) *vterminate* is not invoked.

Tasks Connection

Once defined, in order to form a graph tasks are instantiated and then connected by the *Dependency* objects. *Dependency* objects are not directly instantiated by the

programmer but they are instantiated through methods of *Task*:

connect(task) is invoked on a task and takes another task as argument. The argument can be a task id or a reference to a task object and it is the destination of the resulting *Dependency*.

disconnect(task) is invoked when the shape of the graph is changed. *disconnect* is only useful if a graph, after being used, is modified and then reused. In general this not the case and the graph is deallocated as a whole. In that case all the *Dependency* objects are deallocated automatically.

Tasks Communication

During execution, tasks communicate by sending messages along the edges. There is only an asynchronous operation to send data:

put(message) is a method of *Dependency* and it used to *put* data on an edge. Such data will be delivered eventually to the tak the *Dependency* is directed to.

B.1.2 Message

Message is the class that defines data that can be transferred between tasks. *Message* represent data and defines only methods to be used to access data and its metadata. The only functionality method is *Serialize*, which is invoked by the run-time system to layout the message as a contiguous sequence of bytes.

Data and metadata are defined or referenced by the *Envelop* of the message. The envelop has a *Label*, that is metadata fields, and a reference to the payload, which is the data. A message has three types of metadata: message type, transfer information, and user-defined meta-data. The type of the message determines how it will be treated by the run-time system and there are three fields in the type: whether a message needs to be serialized or it is already stored contiguosly in memory, whether a message is stored on memory allocated by the application or by the run-time system, and whether a message can be aggregated to other messages. Transfer information are source and destination task id, and the size of the message. Finally, for convenience, the label carries a *key*, which is a value for user-defined metadata.

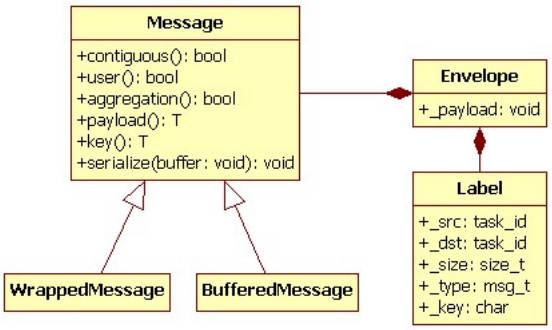


Figure B.2: Class diagram of Message

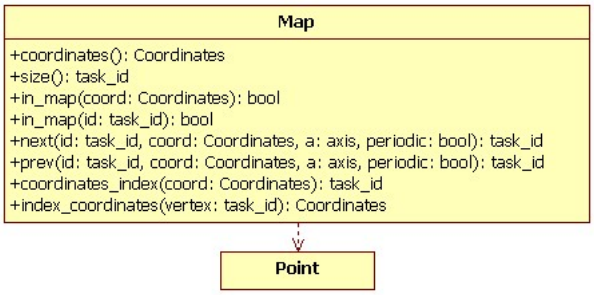


Figure B.3: Class diagram of Map

Tarragon APIs include two predefined types of messages: the *BufferedMessage* class, and the *WrappedMessage* class. A *BufferedMessage* is stored sequentially in memory and as such does not need any explicit serialization whereas the *WrappedMessage* class simply defines a message wrapper with a generic pointer pointing to a data buffer. *WrappedMessage* handles serialization automatically via its it serialize method. The serialize method of *WrappedMessage* first writes all the metadata onto a communication buffer then, to the same buffer, it appends the data pointed to by the reference.

B.1.3 Point, Region, and Map

A *Map* defines a naming scheme for identifying tasks. A *Map* defines a set of n-tuple $M \in \mathbb{Z}^n$ representing the tasks. A *Map* also defines an enumeration $M :\rightarrow \mathbb{N}$ that associates the n-tuples to tasks ids.

It is possible to query the map to gather information about the map itself, as well as to map to associate coordinates and task ids. For this purpose, *Map* defines several methods. Some of them overladed methods (defined for different arguments) and take a *task* argument that can either be a task id, in which case the coordinates associated to task are used for the query, or the coordinates of a task. A brief descrition the methods of *Map* follows:

size returns the total number of points in the map,

coordinates returns a point, whose coordinates are the number of tasks in each dimension,

in_map(task) returns a boolean value: true if the argument given refers to a point that falls inside the map, false otherwise,

next(task,a,periodic) returns the id of the task which, relatively to *task*, is next in the dimension *a*. If *periodic* is true, periodic boundaries are assumed and in this case, the first task in one dimensino is the next with respect to the last in the same dimension,

prev(task,a,periodic) returns the id of the task which, relatively to *task*, is previously in the dimension *a*. If *periodic* is true, periodic boundaries are assumed and in this case, the last task in one dimension is the previous with respect to the first in the same dimension,

coordinates_index(coordinates) performs the mapping between *coordinates* and the id of the associated tasks,

index_coordinates(id) performs the mapping between a task id and the coordinates of the asssociated task.

Point and *Region* are two supporting classes that are used to define maps. *Point* is a multidimentional template class and is used to define the coordinates on a map. A *Region* is a rectangular region identified by two points, the *lower bound* and the *upper bound*.

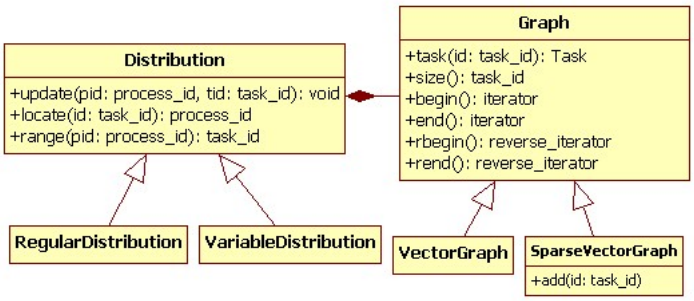


Figure B.4: Class diagram of Graph

B.1.4 Graph

The class *Graph* defines a distributed container of tasks (Figure B.4) and encapsulates different storage and allocation schemes to control task allocation and distribution. It has five methods:

range the number of local tasks

size the total number of tasks

task(id) returns a reference to a task for the given id, or null if the task is not a local task,

is_local(id) returns true is the id identifiies a local task, false otherwise,

locate(id) returns the process id of the process where the task identified by *id* is located,

For convenience Graph can also be iterated through. There are two types of iterators, *Graph::iterator* and *Graph::riterator*. A *Graph::iterator* can be used to iterate through the local tasks of the graph in increasing id order, a *Graph::riterator* can be used to iterate through the local tasks of the graph in decreasing id order. Iterators are instantiated by the *begin* and *rbegin* methods, can be moved over the tasks by increment (++), and can be compared to *end* and *rend* to stop the iteration process.

Graph allocates tasks according to a given mapping of tasks to processes. The mapping is formalized as a *Distribution* object. *Distribution* defines an interface and it is possible to create new mappings. The interface is composed by the following methods:

locate(id) return the process where the tasks identified by id is defined,

range returns the number of local tasks,

sort signals of an opportunity for internal optiomization before use,

begin returns the lowest task id among local tasks,

end returns the highest task id among local tasks,

update(pid,id) updates the distribution, the highest id on processs *pid* is *id*.

Tarragon provides two predefined distrubions: *RegularDistribution* and *SparseDistribution*. By default graph use the *RegularDistribution*, in this case the tasks are equally distributed between processes, in increasing order, such that each process is assigned the same number of tasks (the difference can be at most of one task). The *SparseDistribution* supports sparse sets of tasks, that is set of tasks where only crertain tasks are instantied.

B.1.5 Visitor

In oder to support additional functionalities and algorithms, the objects of a graph are defined as element classes according to the *visitor* design pattern [Gam02]. In the specific case of Tarragon, *Graph*, *Task*, and *Dependency* implement the *Element* interface, which defines the *accept(Visitor)* method. Visitors must implement the *Visitor* interface which includes the visit method overloaded to accept as argument any of *Graph*, *Task*, or *Dependency*. Figure B.5 shows the class diagram including the classes that implement the *Element* interface.

B.1.6 Exceptions

Tarragon raises exception when errors occur within the run-time system. Tarragon defines four exception classes *CommunicationException*, *ThreadException*, *ExecutionException*, and *AllocationException* (Figure B.6). All four are subclasses of *TException* and differ only in what type of error each subclass characterizes. The description of the exception classes follows.

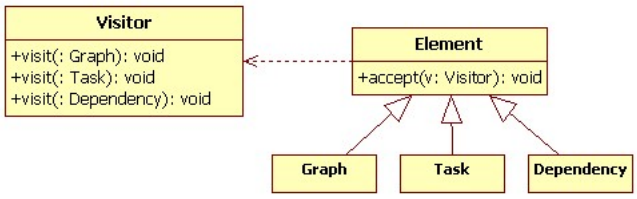


Figure B.5: Class diagram of the visitor design pattern in Tarragon.

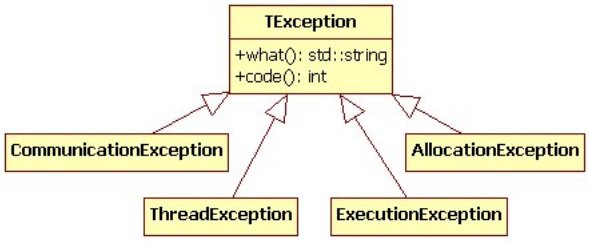


Figure B.6: Class diagram of the exceptions hierarchy

TException is the base exception class of the exceptions thrown in Tarragon. A *TException* is essentially a wrapper exception containing an error code and a description of the error.

CommunicationException are thrown whenever an error occurs in the underlying communication layer.

ThreadException are thrown whenever an error occurs in the underlying threads library.

AllocationException are thrown when memory allocation within the run-time system fails.

ExecutionException is thrown when an error occurs in the execution of a task and the task enters the error state. In this case, the error code is the id of the task causing the exception.

B.2 Extended API

B.2.1 Region and Spaces

The *Extended API* of Tarragon defines the *Space* class to help representing problem domains. Users can define a *Space* subclass that represent a problem domain and data, and then combine a *Map* with the *Space* subclass to induce a domain decomposition, and an association between subdomains and tasks.

The class hierarchy, which is presented in Figure B.8, illustrates how the definition of a *Space* is based on the concept of *Region*: a rectangular domain identified by its *lower bound* and *upper bound*.

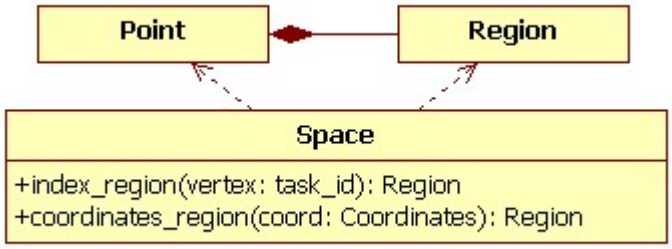


Figure B.7: Class diagram of *Space*. *Space* is based on *Point* and *Region*, which is defined by two *Point*s.

The *Extended API* of Tarragon provides the users with a set of subclasses of *Map* and subclasses of *Space* to support domains that can be represented by a discretized rectangular region. Figure B.8 illustrates the class hierarchy of these classes.

CartesianSpace and *RectilinearSpace* represent the discretization of a rectangular region. *CartesianSpace* assumes a regular decomposition with the same number of points in each dimension, whereas *RectilinearSpace* assumes a rectilinear decomposition and requires the user to specify a vector of distances defining the size of the blocks. Descending from *Map*, the Extended API defines *CartesianMap*, a Cartesian coordinates system. The combination of the spaces and the map results in two more classes: *CartesianGridMap* and *RectilinearGridMap*.

A similar class hierarchy is defined using *BlockingCartesianMap*, leading to the

definition of two more classes: *BlockingCartesianGridMap* and *BlockingRectilinearGridMap*. *BlockingCartesianMap* and *CartesianMap* differ in how they enumerate the tasks. *CartesianMap* enumerates tasks in row major order, counting from the first dimension to the last. *BlockingCartesianMap* defines blocks of tasks and uses the same order hierarchically to enumerate tasks within blocks first, and then to order blocks. Figure B.9 illustrates the difference between the two mappings on a 2-dimensional grid.

All the *leaf* classes of the hierarchies presented, *CartesianGridMap*, *RectilinearGridMap*, *BlockingCartesianGridMap*, and *BlockingRectilinearGridMap*, unify the interfaces of *Map* and of *Space*. The unified interface supports the mapping between task ids, coordinates, and spaces.

Connectors

The *Extended API* provides visitors to connect neighboring tasks. The class hierarchy, which is illustrated in Figure B.10, shows the set of connectors defined. The hierarchy is rooted in *Connector*, which is an abstract visitor defining helper methods that are reused by the other connectors. Each subclass of *Connector* has a specialized purpose, and the result is a hierarchy of visitors in which each can be used to connect neighboring tasks according to a certain pattern:

NearestNeighborConnector connects all the neighboring tasks in both directions,

OneNeighborConnector connects all the neighboring tasks in one dimension only,

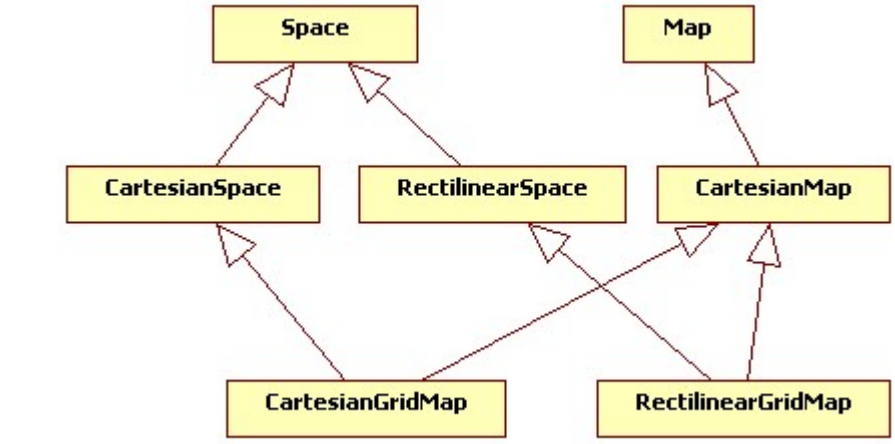
OneNeighborHalfConnector connects all the neighboring tasks in one dimension and one direction only,

OneNeighborForwardConnector connects all the neighboring tasks in one dimension, from lower rank to higher rank,

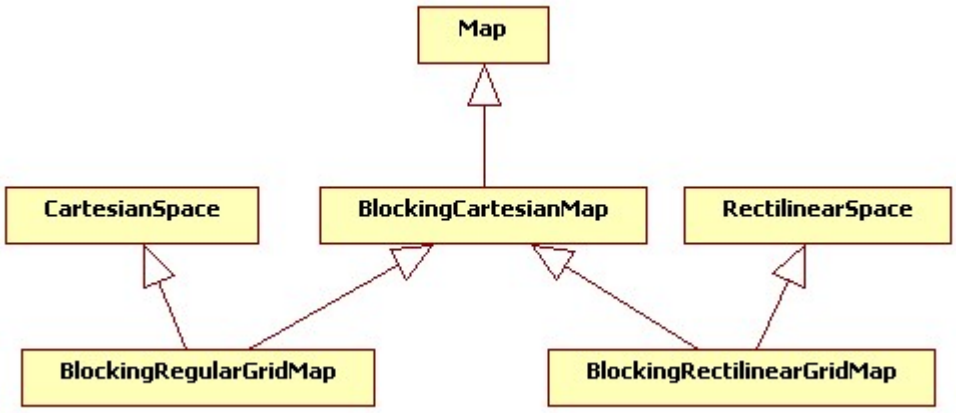
OneNeighborBackwardConnector connects all the neighboring tasks in one dimension, from higher rank to lower rank.

Using any connector is simple. A user can instantiate the connector and use it to visit a graph, as in the following statements:

$$OneNeighborForwardConnector\ c;\ graph.accept(c);$$

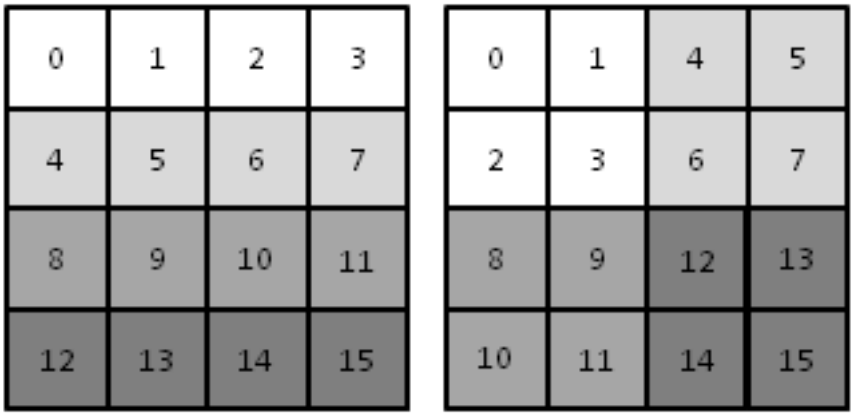


(a)



(b)

Figure B.8: Class diagram of maps, spaces, and grids in the extended API. The classes defined, which combine the interface of *Space* and *Map* and their functionality, represent task defined of discretized rectangular regions.



(a) (b)

Figure B.9: Grid and task mapping. A 2-dimensional grid is decomposed into 16 blocks, each assigned to a task as indicated by the enclosed number. Figure B.9a illustrates the mapping using a CartesianGridMap, that in this case results in a 1-dimensional blocking scheme with rectangular panels of tasks assigned to processes. Figure B.9b illustrates the mapping when using a BlockingRegularGridMap, that in this case results in a 2-dimensional blocking scheme with squared blocks of tasks assigned to processes.

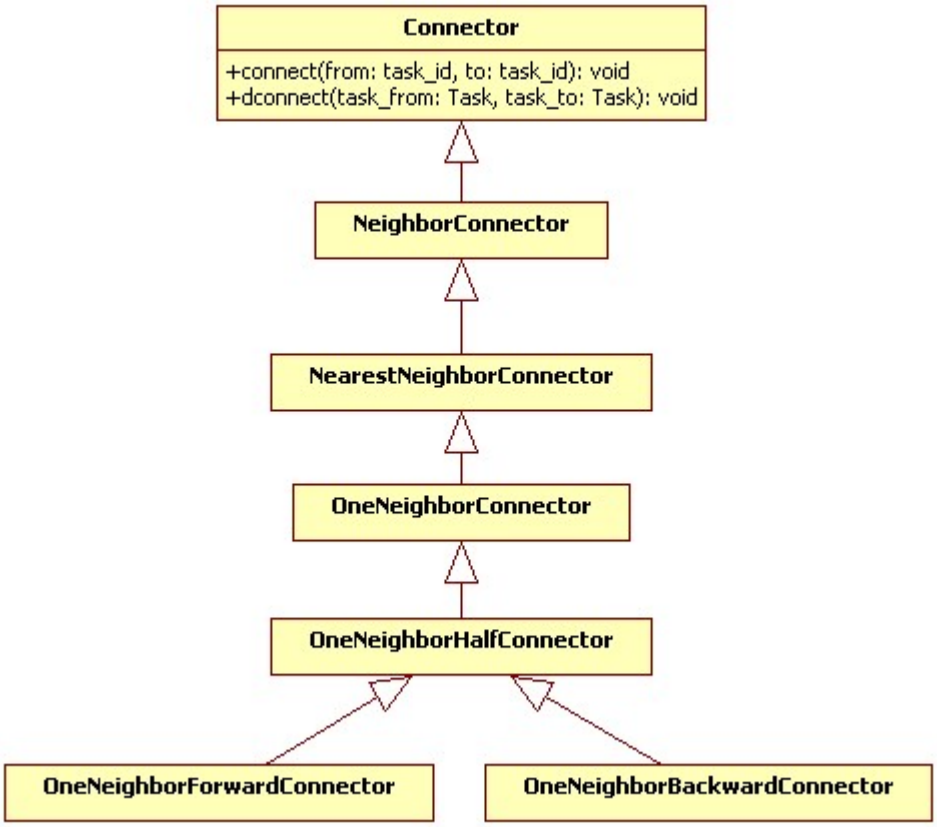


Figure B.10: Class diagram of connectors in the extended API. In the hierarchy rooted in *Connector*, each descendant implements a communication pattern.