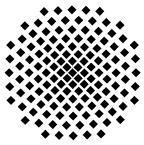


H L R S

Institut für
Höchstleistungsrechnen

FORSCHUNGS- UND ENTWICKLUNGSBERICHT

GLOBAL TASK DATA DEPENDENCIES IN THE PARTITIONED GLOBAL ADDRESS SPACE

Joseph Konstantin Schuchart

ISSN 0941 - 4665 Januar 2021 HLRS-23

Höchstleistungsrechenzentrum
Universität Stuttgart
Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Nobelstrasse 19 - 70569 Stuttgart
Institut für Höchstleistungsrechnen

GLOBAL TASK DATA DEPENDENCIES IN THE PARTITIONED GLOBAL ADDRESS SPACE

von der Fakultät Energie-, Verfahrens- und Biotechnik
der Universität Stuttgart zur Erlangung der Würde
eines Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung
vorgelegt von

Joseph Konstantin Schuchart
aus Stralsund

Hauptberichter:	Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Mitberichter:	Prof. Dr. Barbara Chapman
Tag der Einreichung:	30. April 2020
Tag der mündlichen Prüfung:	05. Oktober 2020

D93

ISSN 0941 - 4665 Januar 2021 HLRS-23

Erklärung über die Eigenständigkeit der Dissertation

Ich versichere, dass ich die vorliegende Arbeit mit dem Titel *Global Task Data Dependencies in the Partitioned Global Address Space* selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe; aus fremden Quellen entnommene Passagen und Gedanken sind als solche kenntlich gemacht.

Declaration of Authorship

I hereby certify that the dissertation entitled *Global Task Data Dependencies in the Partitioned Global Address Space* is entirely my own work except where otherwise indicated. Passages and ideas from other sources have been clearly quoted.

Name: _____

Acknowledgements

I feel great gratitude towards my family for their lasting support on this long journey. This work would not have been possible without you and I am forever grateful for the opportunities you have provided me with.

Many thanks go to all my friends and current and former colleagues for the many interesting, entertaining, and inspiring discussions we have had and hopefully will continue to have. They are too many to name them all.

I would like to express my deep appreciation for my mentors past and present: Prof. Wolfgang E. Nagel and Dr. Andreas Knüpfer of ZIH who many years ago hired me as a young student and introduced me to this exciting field; Daniel Hackenberg, Dr. Robert Schöne, and Thomas Ilsche of ZIH who helped me through times of soul-searching and from whom I have learned invaluable skills; and last but not least Prof. Michael M. Resch and Dr. José Gracia of HLRS for your guidance and for providing me with the freedom and resources to pursue my research interests over the last three years. Thank you all for the great research environments you have created.

A particular acknowledgment is due for all the staff members at HLRS and ZIH whose tireless efforts help keep the lights on and who have more than once provided fast support in critical moments. Thank you!

My gratitude also belongs to the Open MPI project for fostering an open community and providing an amazing MPI implementation.

To Mitzi, for your constant encouragement and support as well as the many miles traveled along the way. You have made this time all the more special.

Contents

Contents 7

Acronyms 11

Abstract 15

Zusammenfassung 17

List of Figures 19

List of Tables 23

List of Listings 25

List of Algorithms 27

1. Introduction 29
- 1.1. Background and Motivation . . . 29
- 1.2. Terms Used . . . 34
- 1.3. The Partitioned Global Address Space (PGAS) . . . 36
- 1.4. Synchronization Mechanisms . . . 38
- 1.5. Task-based Parallelization and Synchronization . . . 40
- 1.6. Target Applications . . . 43
  - 1.6.1. Tiled Linear Algebra Algorithms . . . 43
  - 1.6.2. Multi-Zone Solver used in CFD Applications . . . 44
  - 1.6.3. Unstructured Mesh Application . . . 44
- 1.7. Contributions . . . 44

2. An Interface for Distributed Task Synchronization **47**
2.1. The Basic Interface 47
2.2. The Copyin Dependency 49
2.3. The Taskloop Construct 50
2.4. Example: Tiled Cholesky Decomposition 51

3. Global Task Data Dependencies **55**
3.1. Local Task Graphs 56
3.2. Connecting Local Components 58
3.3. Scheduler Interaction 60
3.4. Synchronization and Execution Constraints 61
3.5. Deadlock Freedom 62

4. Inter-Scheduler Communication **65**
4.1. Requirements 66
4.2. Active Message Queue Designs 67
4.2.1. Send/Recv-based Queue 67
4.2.2. Message Queues Based on MPI-RMA 68
4.3. Evaluation 73
4.3.1. MPI RMA Operation Latencies 73
4.3.2. Average Message Latency 78
4.3.3. Throughput at Scale 82
4.4. Proposed Improvements to the MPI Standard 84
4.4.1. RMA Operation Ordering 84
4.4.2. Fine-grained RMA Communication Contexts 85
4.4.3. Request-based Flush 85
4.4.4. Callback-driven Request Completion 86

5. Implementation **87**
5.1. The DASH PGAS library 87
5.2. Scheduler Implementation 89
5.2.1. Task Discovery Throttling 90
5.2.2. Task-Yield 90
5.2.3. Tasklets 91
5.2.4. Blocked and Detached Tasks 91
5.2.5. Progress Thread 92
5.2.6. NUMA-aware Scheduling 92
5.2.7. Exception Handling 93

5.2.8. Cancellation . . . 93
5.2.9. Copyin Dependency . . . 94
5.2.10. Tool Support . . . 95

6. Results **97**
6.1. Test Setup . . . 97
6.2. A Note on the State of the Software Ecosystem . . . 97
6.3. Task Micro-Benchmarks . . . 98
6.3.1. Task Management Overhead . . . 99
6.3.2. Dependency Management Overhead . . . 100
6.4. Tiled Linear Algebra Algorithms . . . 102
6.4.1. State of the Art Implementations . . . 102
6.4.2. Tiled Cholesky Decomposition . . . 104
6.4.3. Tiled QR Decomposition . . . 111
6.4.4. Discussion . . . 114
6.5. Multi-Zone Block Tri-Diagonal Solver . . . 114
6.5.1. Implementations . . . 117
6.5.2. Evaluation . . . 120
6.5.3. Discussion . . . 126
6.6. Livermore Unstructured Langrangian Shock Hydrodynamics (LULESH) 128
6.6.1. Porting Strategy . . . 129
6.6.2. Issues Encountered . . . 131
6.6.3. Evaluation . . . 133
6.6.4. Discussion . . . 138

7. Related Work **141**
7.1. Process-local Tasking Models . . . 141
7.2. Distributed Tasking Models . . . 142

8. Conclusions and Future Work **147**

Bibliography **151**

A. Test Configurations **167**
A.1. System Overview . . . 167
A.2. Used Software . . . 168
A.2.1. Message Queue Benchmarks . . . 168
A.2.2. Tiled Linear Algebra Experiments . . . 168
A.2.3. NPB BT-MZ Build-time Options . . . 170

A.2.4. LULESH Build-time Options 170

B. QR Factorization implemented using DASH Tasks 171

C. Callback-Driven Request Completion 177
C.1. Use with OmpSs-2 180
C.1.1. Implementation 180
C.1.2. Evaluation with NPB BT-MZ 182
C.2. Use in an Active Message Queue Implementation 183
C.3. Discussion 184

D. LULESH Task Graph 185

E. Curriculum Vitae 189

Acronyms

Notation	Description
AGAS	Active Global Address Space.
AM	Active Message.
AMO	atomic memory operation.
AMR	Adaptive Mesh Refinement.
BLAS	Basic Linear Algebra Subprograms.
BSP	Bulk Synchronous Parallel.
BT	Block Tri-diagonal solver.
BT-MZ	Block Tri-diagonal solver, multi-zone version.
CAF	Co-Array Fortran.
CAS	compare-and-swap.
CFD	Computational Fluid Dynamics.
CFL	Courant-Friedrichs-Lewy constraint.
CPU	Central Processing Unit.
DAG	Directed Acyclic Graph.
DART	DASH RunTime.
DDF	Data Driven Future.
DDT	Data Driven Tasks.
DLR	Deutsches Zentrum für Luft- und Raumfahrt.
DPLASMA	Distributed Parallel Linear Algebra Software for Multicore Architectures.
DTD	Dynamic Task Discovery.

Notation	Description
ECP	Exascale Computing Project.
EGREQ	Extended Generalized Request.
FIFO	First In First Out.
GB	Gigabyte.
GCC	GNU Compiler Collection.
GPGPU	General Purpose Graphics Processing Unit.
GREQ	Generalized Request.
HCMPI	Habanero-C MPI.
HLRS	Höchstleistungsrechenzentrum Stuttgart.
HPC	High-Performance Computing.
HPX	High Performance ParalleX.
LIFO	Last In First Out.
LU	Lower-Upper Symmetric Gauss-Seidel solver.
LU-MZ	Lower-Upper Symmetric Gauss-Seidel solver, multi-zone version.
LULESH	Livermore Unstructured Langrangian Explicit Shock Hydrodynamics.
MB	Megabyte.
MKL	Math Kernel LIbrary.
MPI	Message Passing Interface.
NAS	NASA Aerodynamic Simulation.
NASA	National Aeronautics and Space Administration.
NIC	Network Interface Card.
NPB	NAS Parallel Benchmarks.
NPN	number of processes per node.
NUMA	Non-Uniform Memory Access.
OCR	Open Community Runtime.

Notation	Description
PaRSEC	Parallel Runtime Scheduling and Execution Controller.
PGAS	Partitioned Global Address Space.
PLASMA	Parallel Linear Algebra Software for Multicore Architectures.
PMPI	MPI profiling interface.
POSIX	Portable Operating System Interface.
PTG	Parameterized Task Graph.
pthread	POSIX Thread.
RDMA	Remote Direct Memory Access.
RMA	Remote Memory Access.
SP	Scalar Penta-diagonal solver.
SP-MZ	Scalar Penta-diagonal solver, multi-zone version.
SPMD	Single Program Multiple Data.
STF	Sequential Task Flow.
STL	standard template library.
TAMPI	Task-Aware Message Passing Interface (MPI).
TCMalloc	Thread-Caching Malloc.
TLF	Task Loop Factor.
ULT	user-level thread.
UPC++	Unified Parallel C++.
UPC	Unified Parallel C.
ZIH	Zentrum für Informationsdienste und Hochleistungsrechnen.

Abstract

High-Performance Computing (HPC) has become an important part of scientific discovery in many fields and takes an important role in many engineering processes, harnessing the power of large amounts of computational resources to gain insights into otherwise hidden technological and natural phenomena. The dominating programming model driving today's parallel applications is a two-level approach consisting of message-based communication between processes using MPI and static loop-level thread-parallel execution using OpenMP constructs. However, two programming models have tried to challenge this status quo. First, the Partitioned Global Address Space (PGAS) model is an attempt to elevate shared memory programming to the level of distributed systems and to directly expose modern network hardware features to the application developer. Second, task-based programming aims at providing abstractions that help discover a greater amount of concurrency in parallel applications, which in turn can be used to better exploit the computational resources at hand.

Both models are an attempt to break up the strict synchronization imposed by the traditional models: the PGAS model decouples synchronization and communication while task-based programming models minimize the required synchronization to a set of constraints on the order of the execution of tasks. This work proposes a novel way of orchestrating the execution of tasks at a global scale by using distributed task graph discovery and data dependencies in the global memory space.

The results demonstrate that applications exhibiting concurrency beyond single loop parallelism may use this new model to significantly improve performance and scalability by combining the benefits of task-based programming and one-sided communication in the PGAS model.

Zusammenfassung

Der Bereich des Hochleistungsrechnens (HPC) hat sich zu einem wichtigen Bestandteil heutiger Forschungs- und Entwicklungsarbeit entwickelt. Dabei werden hohe Rechenkapazitäten genutzt um Einsichten in ansonsten weitgehend unsichtbare natürliche und technische Phenomäne zu erhalten. Das dominante Programmiermodell für parallel Anwendungen ist dabei eine Kombination von Parallelität auf zwei Ebenen: die nachrichtenbasierte inter-prozess Kommunikation mit Hilfe von MPI sowie schleifenbasierte Parallelisierung auf der Basis von Threads mit Hilfe von OpenMP.

Zwei neue Programmiermodelle sind angetreten um diesen Status Quo zu ändern. Zum Einen hat der Partitioned Global Address Space (PGAS, *partitionierter globaler Addressraum*) zum Ziel, die Programmierung auf gemeinsamem Speicher in den Bereich der Systeme mit verteiltem Speicher abzubilden und dabei die Fähigkeiten moderner Netzwerke direkt den Entwicklern paralleler Anwendungen zur Verfügung zu stellen. Zum Anderen zielt die Task-basierte Programmierung darauf ab, einen größeren Grad an verfügbarer Parallelität in Anwendungen zu nutzen um die verfügbaren Rechenressourcen besser auszulasten.

Beide Modelle sind ein Versuch die strikte Synchronität der traditionellen Programmiermodelle aufzuheben: im Partitioned Global Address Space (PGAS)-Modell sind Kommunikation und Synchronisierung voneinander getrennt, während bei Task-basierter Programmierung im Idealfall nur genau soviel Synchronisierung stattfindet wie für die korrekte Ausführung einer Anwendung nötig ist. Die vorliegende Arbeit untersucht daher die Verbindung dieser beiden Programmiermodelle in dem Versuch deren Vorteile in einem Modell zu kombinieren, in dem auf globaler Ebene die Ausführungsreihenfolge von Tasks anhand von Datenabhängigkeiten im globalen Addressraum organisiert wird.

Die in dieser Arbeit präsentierten Ergebnisse zeigen dabei, dass für Anwendungen mit hinreichend verfügbarer Parallelität die Kombination aus Task-basierter Programmierung und der Nutzung von PGAS zur Kommunikation eine signifikan-

te Effizienzsteigerung möglich ist. Das entwickelte Programmiermodell macht es Entwicklern dabei einfach existierende Anwendungen zu portieren.

List of Figures

1.1. Synchronization in traditional two-sided MPI communication. 31
1.2. One-sided communication example 32
1.3. Different approaches towards global task synchronization 34
1.4. Overview of the Partitioned Global Address Space paradigm. 37
1.5. Happens-before relations between two operations 39
1.6. Happens-before relations between two operations using unidirectional synchronization. 40

3.1. Example of local task graph discovery with dependencies reaching across process boundaries 56
3.2. Formal description of the local memory task data dependency matching rules. 57
3.3. The global task graph of Figure 3.1 partitioned by phases. 58
3.4. Formal description of the global memory task data dependency matching rules. 60
3.5. The scheduler interaction required to handle remote dependencies. 60
3.6. Local task graphs extended with the received remote data dependencies. 61

4.1. Sequence of operation on a message queue using window locks. . . 69
4.2. Sequence of operation on a lock-free message queue using atomic RMA operations (atomic operations in bold). 71
4.3. Basic RMA operation latencies using different MPI implementation on two different hardware platforms. 75
4.4. Window lock-unlock and send/recv latencies. 77
4.5. Average per-message latency using different active message queue implementations. 78

4.6. State machine used to write a message into a RMA-based queue . . 79
4.7. Sequence of operations involved in writing to two target message queues. 81
4.8. All-to-all message rate of different message queue implementations on Taurus 81
4.9. Message rate of message queue implementations with increasing number of writers to a single target 83

5.1. A tiled matrix with 2D-cyclic distribution in DASH. 88
5.2. The DASH Architecture 89
5.3. NUMA-aware work-stealing algorithm 93

6.1. Overhead of handling tasks and tasklets. 99
6.2. Local and remote dependency latencies on a Cray XC40. 101
6.3. Tiled Cholesky Decomposition of a matrix with 4×4 tiles. 105
6.4. Number of tasks in a Tiled Cholesky Decomposition 107
6.5. Performance of Tiled Cholesky Decomposition on the Cray XC40 in different configurations 108
6.6. Performance of Tiled Cholesky Decomposition on Taurus 109
6.7. Percentage of time spent on operations by the progress thread when executing Tiled Cholesky Decomposition 111
6.8. Performance of Tiled QR decomposition on the Cray XC40 113
6.9. Depiction of the NPB multi-zone grid. 115
6.10. Dependencies between zones and the tasks computing them. 119
6.11. Runtime and speedup of the different BT-MZ variants with increasing number of threads on a single node. 121
6.12. Scaling behavior and speedup of different implementations of BT-MZ using class C on Cray XC40. 123
6.13. Scaling behavior and speedup of different implementations of BT-MZ using class D on Cray XC40 125
6.14. Scaling behavior and speedup of different implementations of BT-MZ using class D on Taurus 127
6.15. Structure of the mesh used by LULESH 129
6.16. Screenshot of Extrae traces of two timesteps of LULESH 132
6.17. Intra-node scaling of LULESH on a Cray XC40. 134
6.18. Multi-node scaling of LULESH on a Cray XC40. 137
6.19. Large-scale LULESH runs on the Cray XC40. 138

C.1. Speedup of BT-MZ class D using OmpSs-2 in combination with TAMPI and request completion callbacks. 182
C.2. Speedup of an experimental message queue implementation using request completion callbacks. 184

D.1. High-level task graph of a single timestep in LULESH. 187

List of Tables

6.1. Overview of the problem sizes available in the NPB BT-MZ benchmark 116

A.1. Systems under test. 168
A.2. MPI software build-time configuration options. 169
A.3. Software packages used in tiled linear algebra experiments 169
A.4. Run-time options used to run tiled Cholesky decomposition benchmarks. 170
A.5. Run-time options used to run tiled QR decomposition benchmarks. . 170

List of Listings

1.1. Examples of a simple algorithm using C++ futures and OpenMP task data dependencies. 41

2.1. The simple algorithm of Listing 1.1 expressed using C++ and data dependencies. 48
2.2. The simple algorithm of Listing 2.1 using distributed task data dependencies 49
2.3. The basic DASH `taskloop` interface. 50
2.4. The DASH task-loop interface with dependency generator. 51
2.5. Tiled Cholesky decomposition implemented using distributed tasks data dependencies. 52

B.1. Allocation of DASH matrix with super-tiles. 171

C.1. Interface of the function `MPI_Request_on_completion` 179
C.2. Usage example of the function `MPIX_Request_on_completion`. 179
C.3. Example for using the proposed callback-driven request completion notification with OmpSs-2. 181

List of Algorithms

6.1. Tiled Cholesky Decomposition 104
6.2. Tiled QR Decomposition 112
6.3. Simplified timestep algorithm of the reference implementation of NPB BT-MZ. 116
6.4. Task-based timestep algorithm of NPB BT-MZ. 118

Chapter 1

Introduction

1.1. Background and Motivation

High-Performance Computing (HPC) has become an important driver of progress across a wide range of scientific and engineering disciplines, from Computational Fluid Dynamics (CFD) and structural engineering, over climate simulation and weather prediction, to social sciences [1]. The end of Dennard's scaling [2], which predicted the increase in frequency of a computer's Central Processing Unit (CPU) proportional to the decrease of transistor dimensions, at the beginning of the 21^{st} century [3] and the upholding of Moore's law [4] led to a steady increase in parallelism in the design of computer systems. As a consequence, modern computer systems offer significant amounts of parallel computing capacity, ranging from growing vector units over multiple processing elements per CPU (multi- and many-core systems, General Purpose Graphics Processing Units (GPGPUs)) to massively parallel systems consisting of thousands of compute nodes, with each node containing multiple CPUs [5, 6].

Applications attempting to exploit the parallel computing capacity are required to explicitly (or implicitly through higher-level programming abstractions) decompose a problem domain into smaller sub-domains and distribute these sub-domains across the available compute nodes. For engineering applications, this may involve splitting a mesh into smaller meshes or cutting through a regular grid and distributing the resulting subsets of grid points across the available processes. In both cases, some mesh elements or grid points will be situated at the boundary of the local sub-domain with some of their neighboring elements situated on different processes. At regular intervals, i.e., after some computation on the local sub-domain, communication

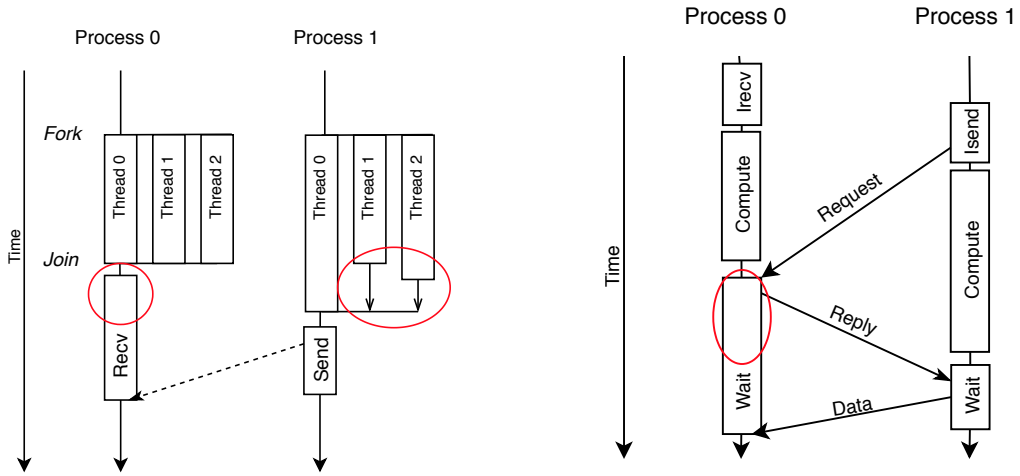
with logical neighbor processes is required to exchange information on elements at the boundary of the sub-domains in order to facilitate the next computational step.

This communication is performed using inter-process communication methods, most commonly by employing the Message Passing Interface (MPI) [7] standard, which is the quasi-standard for communication in high-performance computing. In a recent survey of parallel applications conducted as part of the U.S. Department of Energy's Exascale Computing Project (ECP) [8], a vast majority of applications uses Message Passing Interface (MPI) to communicate in a neighborhood pattern and is able to overlap communication and (some) computation [9].

To better harness the intra-node concurrency of multi- and many-core systems, a two-level hybrid parallelization is commonly employed, e.g., by employing OpenMP's work-sharing constructs to annotate loops whose iterations can be executed concurrently by multiple threads within a single process [10, 11]. It has been observed that this hybrid programming positively impacts the performance of parallel applications [12, 13]. Instead of running one process per CPU core, applications may use one or a few processes per compute node, which leads to larger per-process domains and thus smaller surface-to-volume ratios, i.e., the ratio between the number of elements of the discretized domain that have to be communicated to neighbors (the surface) versus the number of elements owned by a single process (the volume). This leads to significantly fewer (but larger) messages in total [14].

This multi-level programming paradigm leads to multiple synchronization points within an application: between OpenMP threads at the end of work-sharing constructs, between processes communicating through MPI messages, and between all or a subset of processes participating in collective operations, e.g., an MPI reduction operation. Many parallel applications loosely follow the Bulk Synchronous Parallel (BSP) paradigm [15], in which phases of computation are followed by phases of communication with little or no overlap of the two. These synchronization points typically cause some processors to perform operations that do not directly contribute to the result of the computation, e.g., handling communication or waiting for delayed threads to enter a synchronization point at the end of a work-sharing construct. This non-useful work is commonly referred to as *synchronization overhead*.

An example for different synchronization points is presented in Figure 1.1a: both `Thread 1` and `Thread 2` have to wait at the end of the OpenMP parallel region for `Thread 0` to complete, and `Process 0` eventually waits for `Process 1` to issue a `Send` matching the posted `Recv`. These imbalances between threads and processes can be caused by both intrinsic and extrinsic factors [16] such as imperfect domain decomposition and shifting computational load or operating system noise [17].



(a) Synchronization overhead in hybrid MPI+OpenMP applications.

(b) Synchronization overhead in MPI applications when using the rendezvous protocol.

Figure 1.1.: Synchronization in traditional two-sided MPI communication.

One way to partially mitigate the impact of communication overhead is to overlap communication and computation by starting communication operations as early as possible and only waiting for their completion when required. As an example, an application might be structured such that in each timestep the local boundary is computed first and the neighborhood communication is started before the computation on the inner domain is performed. Eventually, the boundary of the neighboring processes are received and integrated into the local domain. However, this scheme relies on communication to progress in the background while all threads of the application perform computation, which is not guaranteed by the MPI standard. Most MPI implementations typically distinguish between small messages, which are sent immediately using what is called the *eager protocol* in MPI. However, with medium and large messages[1], a *rendezvous* protocol is employed that involves a hand-shake between sender and receiver, as depicted in Figure 1.1b [18]. This issue becomes more prevalent with the larger messages typically present in hybrid applications, as described above. The different approaches to mitigate the issue of missing progress in hybrid application all come with their own drawbacks: interrupt-based approaches impose significant overheads caused by the interrupt handling logic [19]; library-internal progress threads used to handle communication [20] may disturb computation and potentially introduce additional overhead (see also Section 4.3.1); or hardware-based solutions that are specific to certain platforms [21, 22].

[1]The eager protocol limit is dependent on the hardware and software stack in use.

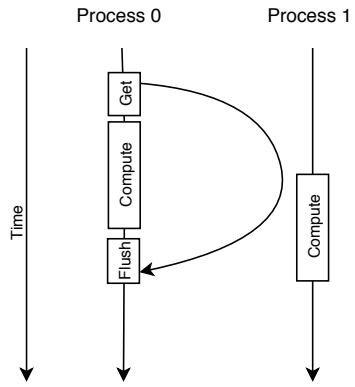


Figure 1.2.: One-sided communication example, similar to Figure 1.1b. Process 0 starts the transfer using `Get` and waits for its completion using `Flush`. Process 1 is not actively involved in the communication operation.

Two approaches towards parallel programming have emerged that try to challenge the existing combination of two-sided communication and OpenMP work-sharing constructs. First, the Partitioned Global Address Space (PGAS) programming model aims at reducing the communication overhead by avoiding the synchronization of message-based communication [23, 24]. In the Partitioned Global Address Space (PGAS) paradigm, communication does not rely on a sender sending data and a receiver receiving it. Instead, communication happens in a one-sided fashion: a process is able to directly access (parts of) the memory of another process, avoiding the time spent on waiting for a message to be sent or received. An example using one-sided communication is depicted in Figure 1.2: `Process 0` starts the transfer directly using an operation typically called `Get` and after finishing computation waits for the completion of the operation using the `Flush` function. `Process 1` is not actively involved in this transfer, i.e., no matching communication routines have to be called explicitly by `Process 1` to participate in the communication.

Second, task-based programming models aim at exposing a maximum degree of concurrency to a runtime system to allow for hiding synchronization overheads and avoiding serial execution. The computational work of an application is thereby divided into work-packages—so-called *tasks*—whose execution is mapped onto the available execution resources, e.g., CPU cores. By oversubscribing the available CPU cores with a larger number of tasks and allowing tasks to be executed out-of-order[2], the scheduler may be able to i) hide occurring latencies by executing a different task until the delaying operation has completed; and ii) maximize the utilization of the available hardware resources by overlapping different parts of the application, effectively reducing serial execution paths of an application and thus potentially

[2] *Read*: executing tasks in an order different from the order in which they were created.

improving scalability [25]. Fine-grained concurrency and synchronization has been identified as one of the challenges on the way to achieving exascale computing[3] [26].

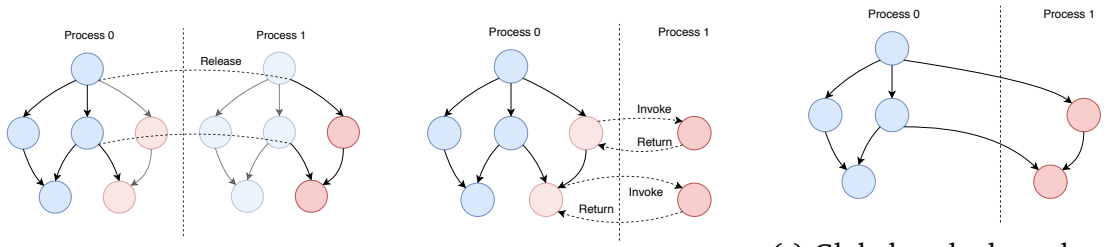
Distributed task-based programming models have been an active research area with different approaches that have been proposed and used in practice [27–31]. A detailed discussion of related work is provided in Chapter 7. The main contribution of this work is a novel approach for synchronizing the execution of tasks that are operating in the partitioned global address space. Existing approaches for distributed task synchronization either rely on dynamic synchronization mechanisms or require users to express applications in a way such that the information on all tasks to be synchronized is available to the respective scheduler instances involved. This is either achieved by expressing the algorithm from a *locality-agnostic global view* (as depicted in Figure 1.3a) or by shipping tasks to other execution units and waiting for their completion (depicted in Figure 1.3b). While global view programming models are suitable for high-level programming abstractions and for the development of new applications, it usually requires a complete rewrite of existing MPI applications. Synchronization through task (or function) shipping, on the other hand, is limited to tasks discovered by the same scheduler instance.

In contrast to these approaches, the approach presented in this work allows users to express algorithms with a *local view* (locality-aware) following the still-dominating Single Program Multiple Data (SPMD) model [32] by focusing on the discovery of the process-local task graph as depicted in Figure 1.3c. This programming model is familiar to developers of MPI applications, which may help in porting applications and facilitating adoption in general.

Moreover, the distributed and partitioned discovery of the task graph promises improved scalability due to the near-constant size of the local task graph in weak scaling scenarios, i.e., with an increasing number of processes and constant per-process problem size, by avoiding the scaling effects seen for the global task discovery model or the coordination required in remote task invocation scenarios.

In the remainder of this chapter, the definitions of used terms will be given before the concepts of PGAS and task-based concurrency are introduced and commonly used synchronization mechanisms are discussed. The main contributions of this work are then summarized at the end of this chapter.

[3] The term *exascale* commonly refers to the simultaneous execution of 10^{18} floating point operations per second.



(a) Global (locality-agnostic) task graph discovery.

(b) Task shipping to remote process.

(c) Global task data dependencies with local discovery.

Figure 1.3.: Different approaches towards global task synchronization, as seen by the user. Tasks colored in blue shall be executed at `Process 0` while tasks colored red shall be executed at `Process 1`. Transparent tasks have been discovered by the process but are executed at another process.

1.2. Terms Used

Throughout this work, the following terms will be used to differentiate between different concepts. The term *concurrent* will be used to describe actions that *may happen* at the same time, e.g., two independent operations *may* be executed at the same time. The term *parallel*, on the other hand, will be used to describe actions that *actually happen* at the same time, e.g., two processes executing on two different nodes run in parallel. The term *concurrency* describes the maximally possible degree of *parallelism*, i.e., the set of actions that could be executed at the same time in the absence of resource restrictions.

A *compute node* (or simply referred to as *node* throughout this work) is an entity that is comprised of at least local memory, one or more CPUs, and an interconnect used to communicate with other nodes in a distributed computing system. A *CPU* may consist of one or more *CPU cores* (or *cores*), which typically contain the necessary hardware to independently execute streams of instructions that transform a set of input data into a set of output data. Typically, each core also hosts a set of fast memory *caches* that hold data that was transferred from memory or will eventually be written back to memory. Caches are meant to exploit spatial and temporal locality and typically exhibit an inverse relation between capacity and access bandwidth and latency. In addition to the cache memory private to a core, some CPU architectures also provide high-level caches that are shared among several cores of the CPU.

On many high-performance systems, a node may be comprised of multiple CPUs, which will be referred to as *sockets* (named after the mounting of a CPU on the board). Each socket manages a portion of the memory on the node. Accesses from a core on one socket to memory owned by another socket may incur high latencies

due to the overhead of using the socket interconnect [33]. This is typically referred to as Non-Uniform Memory Access (NUMA), i.e., not all cores on a node have equal access to all portions of the node's memory.

A *process* is an entity that encapsulates a private local memory space, i.e., memory that by default is only accessible from within that process. A process may expose parts of its local memory space to other processes, either through *shared memory* to allow access to other processes on the same node, or by contributing it to a *global memory space*, in which memory is available to processes executing on different nodes. Inside a process, several *threads of execution* (or *threads*) may execute concurrently with shared access to the memory in the local memory space and shared node-local and global memory that the process has access to. A thread typically executes in a local execution context, i.e., local variables and a private stack to call functions, but has access to global variables shared by all threads in the process. The concurrent execution may be real parallel execution on distinct CPU cores or seemingly parallel through time-slicing performed by the operating system or a user-level runtime library. Thus, threads (in the meaning of the term used in this work) are *preemptable*, i.e., the operating system scheduler may force the thread to yield the core and schedule another thread or process to execute on it.

In contrast, *tasks* are typically non-preemptable and a cooperative scheduler relies on the task to eventually complete its execution. A task is a self-contained work-package that transforms a set of input data into a set of output data. A task typically consists of an *action* to be executed, i.e., a function call, as well as a set of inputs and outputs. The task's action is executed either directly on the stack of the executing thread or in the context of a *user-level thread (ULT)*, which then contains the execution state throughout the course of execution of the action [34].

The term *scheduler* refers to an entity that manages a set of restricted resources and maps competing consumers onto them. In modern high-performance computing systems, schedulers can be found on many levels: job schedulers assign sets of nodes to application runs, operating system schedulers assign time-slices on the CPU cores to processes and threads potentially competing for them, and task schedulers orchestrate the execution of tasks by a set of threads running on CPU cores. The focus of this work in on the level of task schedulers, which manage the execution of tasks by threads within a process. However, the role of the operating system scheduler cannot be neglected and will be discussed where necessary.

The term *runtime system* (or *runtime*) refers to an entity that is part of the software stack and typically resides between the application and the operating system or hardware, coordinating operations and resources requested by the application. Task schedulers and MPI implementations are two examples of runtime systems.

All of this management happens usually at *run-time*, the time of execution of the application.

The term *locality-aware* refers to the ability to actively control the specific set of resources used to execute a process, thread, or task. Locality-awareness can happen on many levels, including application level and the runtime level. The opposite is referred to as *locality-agnostic*, a state in which it is not possible or desirable to select the execution resources explicitly but instead leave that decision to other entities such as the runtime system or the operating system.

Global-view programming models require algorithms to be expressed unconditionally with regards to locality. More specifically, algorithms are generally expressed in a way that *all* participating processes could potentially be aware of all aspects of the algorithm. A subset of these are *locality-agnostic* programming models, in which the mapping of the different parts of the application is left entirely to the compiler or runtime system. *Local-view* programming models, on the other hand, leave this mapping to the application developer, following the traditional SPMD programming model. MPI is a classic example of a local-view programming model.

Values for *speedup* in this work are determined using the following formula: for two implementations A and B with the respective run-times T_A and T_B, the speedup S of A *over* B is determined as $S = \frac{T_B - T_A}{T_B}$.

1.3. The Partitioned Global Address Space (PGAS)

The PGAS programming model provides an abstraction of a shared address space spanning multiple processes. Each process contributes parts of its local memory space to a global memory space that other processes have direct access to. This is depicted in Figure 1.4: the variable `V` is physically located in the memory of `Process 1` but logically resides in the global memory space. Thus, all processes are able to access it by reading and writing data from and to the shared variable, e.g., `Process 0` reads the value of the variable `V` into its private variable p_0 and `Process 2` may write into `V` from its private variable p_2. These operations in the global memory space are typically called `Get` for reading and `Put` for writing.

The PGAS model promises to simplify the development of distributed parallel applications by reducing the programming burden of two-sided send/recv operations used in traditional MPI applications. It also allows direct use of low-level network features such as Remote Direct Memory Access (RDMA) by the user, which enables more efficient communication by circumventing the CPU and eliminating involvement of the remote process in the data transfer [35].

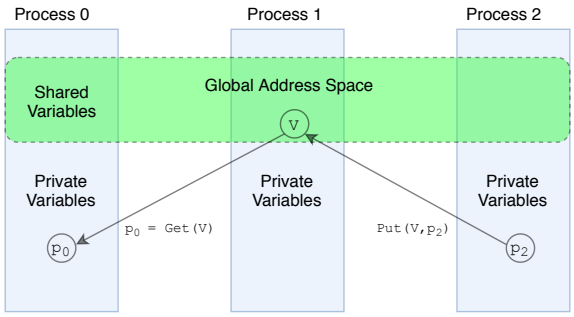


Figure 1.4.: Overview of the Partitioned Global Address Space paradigm.

The MPI standard has adopted a low-level PGAS abstraction through the Remote Memory Access (RMA) extensions introduced with MPI version 2 and later refined in MPI versions 3 and 3.1. In the context of the Exascale Computing Project (ECP) study cited previously, 20 % of the applications responded that they expected to use MPI-Remote Memory Access (RMA) for Exascale [9]. Especially the low latency overhead of accesses make one-sided programming models an interesting alternative to traditional (two-sided) MPI communication [36].

The DASH PGAS abstraction provides the technical basis for this work [37]. Among its main features are distributed data structures whose layout can easily be controlled through patterns, global pointers that enable byte-wise addressing in the global address space, and random global memory access. DASH provides a locality-aware programming model that allows for both global and local memory access, the latter being necessary to leverage the performance of the local memory subsystem by using load/store instructions instead of going through multiple layers of abstraction to access local elements. Underneath its C++ interface, DASH employs MPI-RMA as a communication backend [38]. While the techniques described in this work are implemented based on the DASH abstraction, they are general in nature and can be applied to all PGAS abstractions that provide global memory addressing.

In the PGAS model, transfer and synchronization are decoupled, i.e., any process can access elements in the global memory space at any time without the involvement of the target process. While this allows for more efficient data transfer, it places the burden of the synchronization aspect on the user. After all, correct synchronization is an essential aspect of parallel programming as conflicting concurrent memory accesses may lead to undefined results. In the example provided in Figure 1.4, the order of the `Get` and `Put` issued by the two processes is highly consequential for the result in p_0.

While synchronization is inherent in two-sided communication operations, it is entirely absent in one-sided communication. Thus, appropriate synchronization

mechanisms are an essential aspect of PGAS abstractions to achieve performance at scale. A discussion of possible synchronization mechanisms will be provided in Section 1.4. A central aspect of this work is the design of a mechanism to facilitate the synchronization of tasks operating in the global memory space.

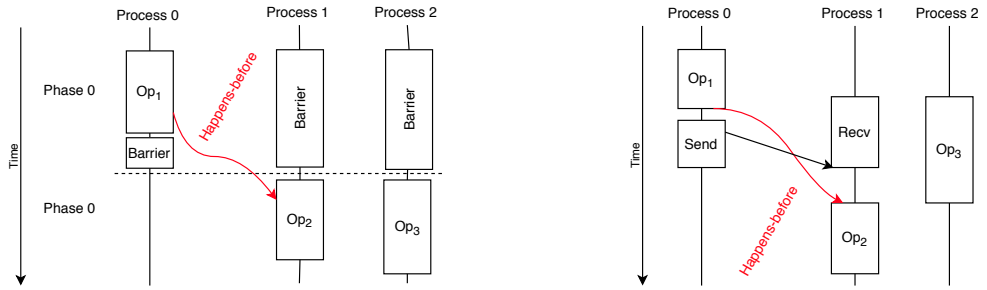
1.4. Synchronization Mechanisms

Correct synchronization requires the establishment of a *happens-before* relation between conflicting accesses. For two operations ω_1 at time t_1 and ω_2 at time t_2 accessing the same memory location M with an intended ordering of $t_1 < t_2$ the effect of ω_1 on M (if any) must be visible for t_2 and no effect of ω_2 on M shall be visible in ω_1.

With regard to conflicting read and write operations, i.e., operations accessing the same memory location, so-called data hazards impose a strict ordering among operations: *write-after-write*, *read-after-write*, and *write-after-read*. The constraints on the ordering of such operations are imposed by the application, e.g., by the sequential order of operations in the application's source code. Compilers may typically reorder operations as long as these ordering constraints are not violated, which otherwise would lead to erroneous results.

In a parallel application, no implicit ordering of operations can typically be derived from the sequence of operations executed by different execution units as compilers and runtime systems are missing information on the concurrent nature of the execution. Thus, synchronization devices are required to ensure the correct order of operations. The simplest form of synchronization is provided by a *barrier* operation across a set of execution units, which ensures the ordering of all previous operations with respect to all subsequent operations on all execution units involved. This behavior is achieved by requiring all participating execution units to communicate and agree on a coherent state, the completion of the barrier. For a barrier at time t_B, all operations ω_i with $t_i < t_B$ will have completed before any operation ω_j with $t_j > t_B$ $(i, j, B \in \mathbb{N})$. In essence, barrier synchronization establishes execution phases $\Phi \in \mathbb{N}$ across all participating execution units, i.e., all operations between two barriers at t_{B-1} and t_B are in phase Φ_{B-1} while all subsequent operations are in phases $\geq \Phi_B$.[4] An example is provided in Figure 1.5a: the barrier is used to ensure the ordering of Op_1 and Op_2. Due to the barrier's collective nature, the operation Op_3 is included in the ordering even though that may not be necessary.

[4]The discussion in this section neglects possibly detrimental effects of out-of-order architectures, compiler reordering of instruction, and weak memory consistency. These implementation details are acknowledged but beyond the scope of this discussion.



(a) Happens-before relation ensured using a collective barrier.

(b) Happens-before relation ensured using point-to-point communication.

Figure 1.5.: Happens-before relations between two operations (Op_1 and Op_2) ensured using different synchronization mechanisms. The operations Op_3 is independent from the two operations.

A more fine-grained synchronization is possible by exchanging signals between individual execution units, e.g., through the use of mutual exclusion devices (*locks*, *mutex*, and *semaphores*) or explicit unidirectional signaling. A mutex is commonly used to ensure mutual exclusion of execution units entering a critical code path, i.e., at most one execution unit may hold the mutex while all others have to wait for that unit to release the mutex. Although mutexes are commonly used for inter-thread synchronization, implementations for inter-process synchronization have been proposed as well [39]. However, previous work has shown that locks in the context of PGAS may inhibit scalability [40], potentially due to the higher costs of memory accesses incurred by Remote Direct Memory Access (RDMA) compared to local memory accesses [41].

For inter-process synchronization, the use of messages constitutes an easy-to-use synchronization device involving two executing units, as depicted in Figure 1.5b. In that case, `Process 0` issues a `Send` operation that matches a `Recv` operation at `Process 1`. Naturally, any operation on `Process 0` issued before the send will be executed before any operation following the completion of the `Recv` at `Process 1`.

Regarding the communication aspect inherent in send/recv-based communication, the synchronization semantics entailed in a message are two-fold: i) the obvious signaling of availability of data in the receive buffer upon completion of the `Recv`; and ii) the less obvious signaling of the availability of buffer space at the beginning of the `Recv` call. Thus, the common data hazards between two operations are all implicitly handled through the message semantics. However, the semantics of two-sided messages conflict with the one-sided communication model at the core of the PGAS model. Moreover, messages introduce a communication channel (the data transfer aspect) parallel to the global address space.

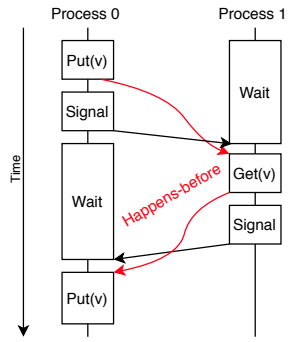


Figure 1.6.: Happens-before relations between two operations using unidirectional synchronization.

Some PGAS implementations provide synchronization mechanisms that are more fine-grained than barriers, providing unidirectional notification mechanisms that can be used to wait for a remote execution unit to trigger a state change in local memory [42–44]. Similar proposals have been made for MPI-RMA [45, 46]. These mechanisms have proven useful for relatively simple synchronization schemes and vaguely resemble the synchronization semantics of two-sided communication primitives, albeit in a limited form as they only cover the signaling in one direction, i.e., to resolve any one of the three data hazards at a time. Thus, building complex synchronization patterns in PGAS applications requires extensive use of such unidirectional signaling mechanisms to ensure correct order of global memory operations, e.g., to signal the availability of data at the target process and later signal to the next writer that the data has been processed and can be overwritten, as depicted in Figure 1.6.

By encapsulating operations in the global address space in tasks and explicitly specifying constraints on the order of execution of these tasks, a scheduler is able to dynamically create an execution plan by incorporating the provided dependencies, similar to a compiler scheduling machine instructions of sequential applications.

1.5. Task-based Parallelization and Synchronization

As mentioned earlier, the concept of tasks provides the possibility to divide an application's work into work packages with well-defined inputs and outputs. These work-packages are thus not required to be independent. In fact, the ability of the scheduler to hide and avoid synchronization overhead stems from the fact that the input data of one task may be produced by one or several other tasks and that the results may again be consumed by one or several other tasks. These tasks are *dependent* on each other.

```cpp
int x = input();
int res;
auto fut = std::async(
          [=](){ return x*x; });

auto fut2 = fut.then(
          [](int y){ return y/2; });
res = fut2.get();
```

(a) Explicit: futures and continuations.

```cpp
int x = input();
int res;
#pragma omp task depend(in:x)
                 depend(out:res)
{ res = x*x; }
#pragma omp task depend(inout:res)
{ res /= 2; }
#pragma omp taskwait
```

(b) Implicit: OpenMP task data dependencies.

Listing 1.1.: Examples of a simple algorithm using C++ futures and OpenMP task data dependencies.

Such dependencies may either be detected automatically by a compiler in the case of specialized languages [47] or be specified explicitly by the application or library developer. However, the adoption of new languages outside of their niche has been found to be marginal in the overwhelming number of cases [48]. Thus, this work focuses on a library-based approaches with explicit definition of dependencies.

The synchronization of tasks can be expressed in three ways, either by *explicitly* specifying the *data-flow* between tasks, by *implicitly* defining the (expected) data-flow through *dependencies* on variables, or *ad-hoc* using dynamic synchronization mechanisms. An example of the explicit synchronization is the concept of *futures and continuations* in C++. As depicted in Figure 1.1a, the programmer explicitly specifies the relative order in which tasks are to be executed by using channels (*futures* and their counterpart *promises*) and continuations to facilitate the synchronization of and communication between tasks [49]. The synchronization of tasks is only possible if handles identifying the tasks to synchronize with are available at run-time, e.g., because these tasks have previously been created by the same thread or the handles have been stored in local or global memory. Popular implementations employing explicit task synchronization are High Performance ParalleX (HPX) and Unified Parallel C++ (UPC++) [28, 50].

In the implicit model, the specification of the synchronization is done *descriptively*, i.e., by specifying (expected) inputs and outputs of the task. An example of such a description is provided in Figure 1.1b using OpenMP task data dependencies. The data movement may be coupled to these specifications, i.e., the underlying runtime system may use the specification to perform communication to bring the data closer to the task requiring it as input, e.g., by moving data from a remote process into the local memory of the process executing the task [51]. In shared memory systems, however, this movement is typically not necessary as the output of one task may be directly accessed by the task(s) consuming it. The scheduler is responsible

for correctly handling the three relevant data hazards discussed in Section 1.4 to prevent conflicting accesses to shared memory by different tasks executing in parallel. Thus, it may be beneficial to copy input data even in shared memory to avoid blocking tasks due to WAR dependencies, a technique called *renaming*, which is also employed by modern processors and compilers [52, 53]. The most prominent implementation of the implicit model can be found in OpenMP [54, 55] and OmpSs [56], both focusing on shared memory parallelism.

The constraints on the order of execution of tasks imposed by both the expression of data-flow and dependencies between tasks can be represented in the form of Directed Acyclic Graphs (DAGs), which encode the expected *happens-before* relations between tasks, as displayed in Figure 1.3. These DAGs can be used by runtime systems to make advanced scheduling decisions and by analysis tools for visualization and correctness-checking [57].

With the *ad-hoc* model, the synchronization of tasks is performed using dynamic synchronization devices as described in Section 1.4 during the execution of the tasks. The execution of a task is suspended while it is waiting for a signal from another task. In this model, the dependencies are not exposed explicitly to the underlying runtime system, giving the scheduler only a limited view on the structure of the application. This model may be beneficial in cases where tight interactions between tasks are required, mimicking the functionality of co-routines [58].

The random memory access capabilities provided by PGAS are similar to random memory accesses in shared memory, albeit with potentially higher latencies. At the same time, the inherent parallel nature of the PGAS model requires a suitable task-based synchronization model to incur as little synchronization and serialization as possible. In that regard, the explicit synchronization scheme requires the communication of handles (e.g., futures) identifying tasks created in parallel, which in turn introduces additional synchronization and communication, or the strictly serialized creation of tasks that are to be synchronized as in global-view models.

The unidirectional signaling at the core of the ad-hoc approach to task synchronization may only incur a small communication overhead. However, in order to implement the more complex synchronization patterns necessary to handle the above mentioned data hazards, the user is required to implement multi-directional signaling schemes manually, a process that is tedious and potentially error-prone.

This work will thus focus on the use of implicit synchronization through the specification of dependencies between tasks on data in the global address space and the necessary interactions between the scheduler instances on different processes.

1.6. Target Applications

Task-based parallelization is most effective in applications that allow for different aspects or parts of the algorithm to be performed concurrently. This overlap can be achieved through both data partitioning and functional partitioning, or a combination of both. With data partitioning, the same operations are applied concurrently to different parts of the input data, e.g., by parallelizing the execution of a loop body in the absence of dependencies between the iterations. This is the parallelization strategy at the heart of traditional hybrid MPI+OpenMP applications. Functional partitioning, on the other hand, enables the execution of different operations on distinct data, e.g., by concurrently calculating different aspects of a physical model within one timestep or by concurrently applying different operations to the tiles of a matrix.

Task-based parallelization is especially useful in a combination of the two. In addition to the typical data partitioning in parallel application, functional partitioning of different parts of the application combined with the expression of the data-flow between them exposes a maximum degree of concurrency to the runtime system and allows for these parts to be executed as soon as all required input data is available.

In the course of this work, two classes of applications will be of interest: tiled linear algebra algorithms and applications exhibiting neighborhood communication.

1.6.1. Tiled Linear Algebra Algorithms

The factorization of large systems of equations expressed as matrices is an important basic operation in many engineering and scientific applications.

A family of tiled matrix algorithms has been developed specifically in response to the rising number of processing elements found in today's computer systems [59]. Instead of relying on the parallelism inside Basic Linear Algebra Subprograms (BLAS) routines, tiled algorithms are structured such that concurrency is expressed at the algorithmic level. Therefore, a matrix A with $N \times M$ elements is divided into tiles of size $NT \times MT$ with operations being applied to the individual tiles. At their core, many of these algorithms are structured such that in each iteration the tile on the matrix diagonal is updated, followed by an update of a row or column and the update of the trailing part of the matrix.

By encapsulating the operations on the tiles in tasks and specifying the data-flow between them, a large amount of concurrency can be exposed, leading to minimized idle times in the threads executing the tasks. In many cases, the communication

pattern is an irregular many-to-many exchange of tiles, which is evolving throughout the course of the execution of the algorithm. The required coordination of the execution of tasks at a global scale is challenging due to the sheer amount of tasks and the irregular communication pattern. These properties make this class of algorithms an interesting target for a one-sided task-based programming model. In Section 6.4, the implementations of the tiled versions of the Cholesky decomposition and the QR decomposition will be discussed.

1.6.2. Multi-Zone Solver used in CFD Applications

The Chimera method has been employed in different production CFD packages such as NASA's Overset or TAU code used at the Deutsches Zentrum für Luft- und Raumfahrt (DLR), the German aerospace center [60–62]. This method has been successfully used for the modeling of complex geometries by creating different meshes for parts of the target object. These meshes (or zones) are solved individually and coupled only through the boundaries between them.

The NAS Parallel Benchmarks (NPB) are a collection of benchmarks designed to emulate different aspects of Computational Fluid Dynamics (CFD) applications. In Section 6.5, one of the benchmarks, a block-tridiagonal solver, is ported to different task-based programming models in an attempt to exploit the coarse-grain concurrency inherent in the use of loosely coupled meshes.

1.6.3. Unstructured Mesh Application

The Livermore Unstructured Langrangian Explicit Shock Hydrodynamics (LULESH) code simulates a Sedov blast in a homogeneous material using an unstructured mesh [63]. It has been one of the proxy applications in the CORAL[5] project and a widely used sample benchmark. A port to the task-based programming model proposed in this work is discussed in Section 6.6.

1.7. Contributions

The main contribution of this work is a novel approach to the global coordination of the execution of tasks operating in the Partitioned Global Address Space without requiring either central coordination or a global view of the application. The resulting local-view programming model follows the SPMD model, which allows

[5] An overview over the CORAL benchmarks is available at `https://asc.llnl.gov/CORAL-benchmarks/`.

easy porting of existing parallel MPI applications with their domain decomposition, data exchange, and hybrid parallelization. This new model does not require a substantial redesign of the algorithms but instead allows for a gradual adaptation of existing codes. At the same time, it enables the benefits of task-based programming by helping developers expose the available concurrency to the runtime system, potentially leading to reduced idle times and thus improved efficiencies in parallel execution.

The basic interface for this system is introduced—together with an example—in Chapter 2. The formal model that serves as the basis for global task coordination is discussed in Chapter 3. In Chapter 4, the challenge of efficient inter-scheduler communication is discussed and a novel approach towards exchanging active messages is proposed. Moreover, a set of additions to the MPI standard are proposed that are meant to improve both performance and usability of the RMA interface. The implementation within the DASH PGAS library is briefly discussed in Chapter 5 followed by results of different benchmarks and proxy-applications in Chapter 6. A review of related literature is conducted in Chapter 7.

Chapter 2

An Interface for Distributed Task Synchronization

Parts of the research presented in Chapters 2, 3 and 5 have previously been published in the following peer-reviewed paper: J. Schuchart, J. Gracia. "Global Task Data-Dependencies in PGAS Applications." The author of this document claims main responsibility for the design and implementation of the distributed task dependency system presented here [64].

The main contributions of this work are the methodology and the design of a distributed system for the synchronization of tasks executing in the global address space. These tasks are executed within the isolation of a private context (call-stack with local variables) by threads inside a process. Tasks are able to communicate with other tasks running in the same process (through *local memory*) or with tasks executing inside other processes (through *global memory*). In contrast to traditional MPI, communication thus happens implicitly through shared variables. The order in which the tasks are executed determines the direction in which data flows between tasks, i.e., a task τ_1 writing a variable V communicates with a task τ_2 reading the same variable afterwards without the explicit notion of a message, as depicted in Figure 1.4. Similar to traditional MPI, the solution proposed in this work follows the SPMD model and leaves the control of where a task should be executed to the user.

2.1. The Basic Interface

The following interface is proposed for the synchronization of tasks executing in the global address space. Similar to C++, asynchronous activities are created using

```
1 int x = input();
2 int res;
3 async([&](){ res  = x*x; }, in(x), out(res));
4 async([&](){ res /= 2; }, inout(res));
5 complete();
```

Listing 2.1.: The simple algorithm of Listing 1.1 expressed using C++ and data dependencies.

the `async` function, which is being passed a functor in the form of a C++ lambda[6] object (similar to the C++ example provided in Listing 1.1a). However, in contrast to standard C++, the call to `async` has no return value. Instead, additional arguments passed to the `async` describe the expected input and output variables of the task in the form of dependencies. The example of Listing 1.1 is provided in Listing 2.1 using this interface. Similar to the OpenMP model, each task carries its input and output dependencies, expressed through the `in`, `out`, and `inout` arguments to `async`.

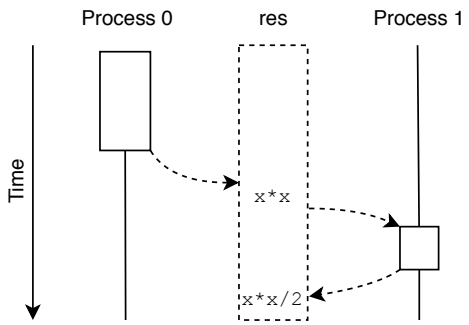
In an extension to this example, Listing 2.2 provides an example of these two tasks executing on two different processes. The example uses an object of a (hypothetical) class `Shared`, which provides a single variable in the global address space (Line 1). The process owning the variable writes the result of the first operation to it (Lines 2–6). The second process later performs the second operation by reading and writing to it (Lines 8–12), actions that map to `get` and `put` operations in the underlying communication layer. The outcome is the same as in the previous examples.

Each process creates its task independently so that the two tasks are created in parallel: the creation of the tasks is conditioned on whether the shared object `res` object resides in the local memory space of the current process (determined through the property `is_local`). An important detail to note is the call to `async_fence` on Line 7: this fence is required to establish an ordering among the two tasks, i.e., to signal to the scheduler that any previously discovered tasks should execute before any tasks discovered later if these tasks have matching dependencies. In other words, the fence dictates the ordering in which the dependencies should be evaluated. The details of this operation will be discussed in Section 3.2. It suffices to mention here that this call does not entail any communication but is merely a synchronized signal to the scheduler instances on both processes.

The `async` interface with its definition of dependencies is the basic building block for the creation of tasks and the definition of dependencies between them. It is

[6] In essence, the C++ lambda construct is an anonymous class with an overloaded `operator()`, which will execute the statements provided in its body. Variables referenced inside the lambda body that are not declared inside the body can be either captured by value or by reference. The compiler will generate the code necessary for the capturing. A detailed explanation of C++ lambdas can be found in [65].

```
1  Shared<int> res{};
2  if (res.is_local()) {
3    int x = input();
4    async([&, x](){ res.set(x*x); },
5          in(x), out(res));
6  }
7  async_fence();
8  if (!s.is_local()) {
9  {
10   async([&](){ res.set(res.get() / 2); },
11         inout(res));
12 }
13 complete();
```



Listing 2.2.: The simple algorithm of Listing 2.1 using distributed task data dependencies running on two processes. The depiction on the right shows the execution order and the value of `res` over time.

important that to note that there are no restrictions on the operations a task may execute, including reading and writing from and to local and global memory.

2.2. The Copyin Dependency

A central premise of the interface proposed above is that synchronization and communication are decoupled. However, there are cases where the global memory location referenced in an input dependency represents a contiguous memory region that will be copied into a temporary buffer inside the task once that task has started its execution. For these cases, it might be desirable to leave the transfer to the runtime system before the task commences execution. In order to facilitate this, the interface provides the `copyin` dependency that takes a consecutive global memory range and optionally a target buffer into which the global memory range will be copied. If no target buffer is provided, a buffer will be allocated internally and passed as argument to the task.

The dependency itself acts as an input dependency and will match with the last previous output dependency on the same memory location. However, later output dependencies on the same global memory location will be released as soon as the transfer has completed and will not wait for the actual task to complete execution. The task will start its execution only after the transfer has completed.

A variant of the `copyin` dependency is the `copyin_r` dependency, which only copies the specified range if it is *remote* (or non-local), which helps the application avoid unnecessary local data movement. If the range is local, the pointer to the local range will be passed to the task.

In addition to reducing the burden on the developer, this dependency also allows an optimization in which multiple tasks in the same phase carry the same `copyin` dependency. The scheduler will avoid multiple transfers of the same memory region and instead copy the data once before resolving all similar `copyin` dependencies. An example using this dependency will be provided in Section 2.4.

The interface does not include high-level data management (e.g., renaming of variables or tiles, movement of ownership) that is commonly found in higher-level linear algebra abstractions such as the Parallel Runtime Scheduling and Execution Controller (PaRSEC) or Chameleon [29, 66, 67]. Instead, a conscious decision has been made to design a general-purpose abstraction for distributed task synchronization in the global memory space, leaving full control to the user while providing data movement abstractions for convenience, without limiting the generality of the interface.

2.3. The Taskloop Construct

Another useful construct included in this proposed interface is the `taskloop` construct, which partitions the provided iteration space into chunks assigned to individual tasks. Similar constructs exists in OpenMP 5.0 [10] and parallel languages such as Chapel [68]. However, the `taskloop` proposed here also offers support for dependencies, allowing chunks from different `taskloop` statements to be chained.

An example on the use of the `taskloop` construct is shown in Listing 2.3. The arguments to the function are the iteration space over which tasks are to be created $\left([\texttt{A.lbegin()}, \texttt{A.lend()})\right)$[7], an optional specification of the size of each chunk or the number of chunks in total and the action to perform on each chunk. In the example, the elements in the array `A` are scaled by a factor of two with each task being assigned at most 64 elements.

```
1 Array<double> A(N);
2 taskloop(A.lbegin(), A.lend(), dash::chunk_size(64),
3   [&](auto begin, auto end){
4     for (auto iter = begin; iter != end; ++iter) {
5       *iter *= 2;
6     }
7 );
```

Listing 2.3.: The basic DASH `taskloop` interface.

[7] The half-open interval $[a, b)$ includes all elements starting with a up to but not including b. Following C++ conventions, the iterator returned by a call to `end()` points one element past the last element in the container.

```
Array<double> A(N);
taskloop(A.lbegin(), A.lend(), dash::chunk_size(64),
  [&](auto begin, auto end){
    for (auto iter = begin; iter != end; ++iter) {
      *iter *= 2;
    }
  },
  [&](auto begin, auto end, auto deps)
  {
    // sentinel dependency on the first element
    deps = out(*begin);
  }
);

taskloop(A.lbegin(), A.lend(), dash::chunk_size(64),
  [&](auto begin, auto end){
    for (auto iter = begin; iter != end; ++iter) {
      consume(*iter);
    }
  },
  [&](auto begin, auto end, auto deps)
  {
    // sentinel dependency on the first element
    deps = in(*begin);
  }
);
```

Listing 2.4.: The DASH task-loop interface with dependency generator.

As an optional additional parameter, a second lambda expression can be passed that is used to dynamically generate the dependencies for each task, as depicted in Listing 2.4. Similar to the first example, the first `taskloop` scales the array by a factor of two but in addition defines output dependencies on the first element of each chunk, which serve as a sentinel representing the whole chunk. The iterations of the second `taskloop` consume the elements of the scaled array and thus carry an input dependency on the same sentinel elements. These dependencies effectively chain the computation on the chunks of the two `taskloop` statements. Using a depth-first execution strategy, the scheduler is able to execute the same chunk of multiple `taskloop` statements consecutively, which helps to increase data locality and thus cache re-use.

2.4. Example: Tiled Cholesky Decomposition

An implementation of the tiled Cholesky decomposition that will be evaluated in detail in Section 6.4.2 is presented in Listing 2.5. The type `Matrix` represents a distributed tiled two-dimensional array with configurable data distribution, similar to the data structures provided by the DASH PGAS abstraction to be discussed in Section 5.1.

```
void potrf(Matrix<double> &A) {
  for (int k = 0; k < NTILES; ++k) {
    if (A.block(k,k).is_local()) {
      async([&](){
          potrf(A.block(k,k));
        },
        out(A.block(k,k))
      );
    }

    async_fence(); // <- advance to next phase
    for (int i = k+1; i < NTILES; ++i) {
      if (A.block(k,i).is_local()) {
        async([&](const double* block_kk){
            trsm(block_kk, A.block(k,i));
          },
          copyin_r(A.block(k,k)),
          out(A.block(k,i))
        );
      }
    }

    async_fence(); // <- advance to next phase
    for (int i = k+1; i < NTILES; ++i) {
      for (int j = k+1; j < i; ++j) {
        if (A.block(j,i).is_local()) {
          async([&](const double* block_ki, const double* block_kj){
              gemm(block_ki, block_kj, A.block(j,i));
            },
            copyin_r(A.block(k,i)),
            copyin_r(A.block(k,j)),
            out(A.block(j,i))
          );
        }
      }

      if (A.block(i,i).is_local()) {
        async([&](const double* block_ki){
            syrk(block_ki, A.block(i,i));
          },
          copyin_r(A.block(k,i)),
          out(A.block(i,i))
        );
      }
    }
    async_fence();  // <- advance to next phase
  }
  complete(); // <- wait for all tasks to execute
}
```

Listing 2.5.: Tiled Cholesky decomposition implemented using distributed tasks data dependencies.

In this implementation, the locality of the output block determines the location at which the task producing the block will be executed. The locality of blocks is determined using the `is_local()` property of the block (Lines 3, 13, 26, and 37). The implementation makes use of `copyin_r` dependencies (e.g., Line 13) described in Section 2.2 to let the runtime transfer tiles before the execution of the task begins and to avoid unnecessary copy operations if multiple tasks require the same tile as input. These `copyin_r` dependencies match with the output dependencies of previous tasks.

The correct matching of these dependencies is facilitated by the use of `async_fence` statements (Lines 11, 23, and 46), which are required to ensure unambiguous matching between dependencies. Without these fences, the scheduler would be unable to distinguish whether the copyin dependency of the TRSM operation refers to the output dependency of a previous POTRF or SYRK task. The runtime system will detect these cases and treat any ambiguities as errors. Compared to missing (barrier) synchronization in regular PGAS applications this provides immediate feedback on incorrect synchronization to the developer instead of silently corrupting data.

The loops in the code are executed by the main thread in each process. All calls to `async` create tasks that will eventually be scheduled for execution by the task scheduler, either by a set of worker threads or the main thread itself. After all threads have been discovered, the main threads call `complete` to wait for their completion.

Although the call to `async_fence` does not necessarily incur any communication, it may serve as a *matching point* at which dependencies are matched and the local task graph is extended by dependencies reaching across process boundaries. The underlying mechanism will be described in the following chapter.

Chapter 3

Global Task Data Dependencies

The dependencies between tasks form a Directed Acyclic Graph (DAG) with the set of vertices V representing the set of tasks and the edges $E \subseteq \{(v_1, v_2) \mid (v_1, v_2) \in V^2 \wedge v_1 \neq v_2 \wedge (v_2, v_1) \notin E\}$ connecting the vertices [69]. A task is represented in the DAG by exactly one vertex. In the case of nested tasks (tasks created inside a parent task), the resulting inner DAG is independent of the outer graph.

Dependencies between tasks are represented by the edges in the graph. An edge (v_f, v_g) between two vertices representing the two tasks τ_f and τ_g, respectively, implies that there is a happens-before relation between the two tasks. In the following, this relation is expressed as $\tau_f \prec \tau_g$, meaning that τ_f has to complete its execution before τ_g may start executing.

In a directed and acyclic graph, there may be no paths $(v_i, v_{i+1}, v_{i+2}, \ldots, v_i) \in E^{+}$[8] through the graph. An application's task graph typically consist of one or more connected components, potentially requiring the traversal of multiple components to reach all vertices.

In the tasking model proposed in this work, processes discover the local task graphs only, using the definition of data dependencies provided by the user. Thus, in a first step individual connected components are discovered at each process using the *local* task dependencies. In a second step, these components of the graph have to be connected using the specified *remote* task dependencies, i.e., dependencies reaching across process boundaries.

In the example given in Figure 3.1, the process in the center (green) first discovers the local task graph (bold edges) before connecting with the components in the neighboring processes (dashed edges).

[8] E^{+} represents the transitive closure of E.

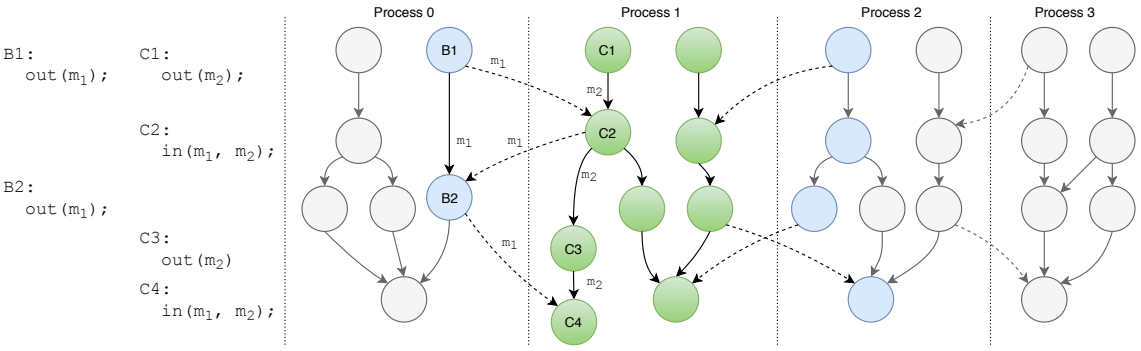


Figure 3.1.: Example of local task graph discovery with dependencies reaching across process boundaries. `Process 1` only discovers its local (green) tasks without direct knowledge of dependent or dependee tasks (blue). The code listing on the left contains the dependencies used to determine the edges between the tasks in `Process 0` and `Process 1`.

The rules used to populate the local task graph based on the dependency definitions provided by the user will be outlined in the following sections. Section 3.1 formalizes the rules governing the discovery of a task graph in which tasks are discovered sequentially, e.g., the local components of the global task graph. Following that, Section 3.2 extends this model with the information and rules necessary to unambiguously connect these local components.

3.1. Local Task Graphs

In the tasking model introduced in this work, a task commonly consists of an *action* a that will be performed when the task is executed, a number of dependencies $n \geq 0$ with each dependency $d_i \in \mathcal{D}, 0 \leq i < n$, and the information on which parent task p has discovered the task. The special parent task p_0 represents an implicit root task that is executed by the main thread of the application. The following part first introduces a formal model used for the purpose of establishing a partial ordering of tasks created sequentially, similar to the way tasks are synchronized in OpenMP through dependencies.

In a local task dependency model, each dependency d_i is a tuple (k_i, m_i) that consists of the type of dependency $k_i \in \{\text{in}, \text{out}\}$ and a referenced memory location $m_i \in \mathcal{M}_L$. The dependency can thus be either an input or output dependency[9] on a memory location in the local memory space $\mathcal{M}_L$, which is the set of all local memory addresses.

[9]For simplicity, the dependency type inout is treated as an output dependency.

$$\tau_p \in \mathcal{T} \cup \{p_0\},\ \mathcal{T}_p \subseteq \mathcal{T},\ E_p \subseteq \mathcal{T}_p^2 \tag{3.1}$$

$$\tau_i, \tau_f, \tau_g, \tau_h \in \mathcal{T}_p,\ i, f, g, h \in \mathbb{N},\ 0 \leq i, f, g, h < |\mathcal{T}_p| \tag{3.2}$$

$$\tau_i := (a_i, D_i, \tau_p) \tag{3.3}$$

$$d_i := (k_i, m_i) \in D_i,\ k_i \in \{in, out\},\ m_i \in \mathcal{M}_{\mathcal{L}} \tag{3.4}$$

$$(\tau_f, \tau_g) \in \Big\{ E_p \Big| \exists d_f\, d_g : m_f = m_g \wedge f < g \wedge \tag{3.5a}$$

$$\big(k_f = out \vee k_g = out\big) \wedge \tag{3.5b}$$

$$\big(\nexists d_h : m_g = m_h \wedge k_h = out \wedge f < h < g\big)\Big\} \tag{3.5c}$$

Figure 3.2.: Formal description of the local memory task data dependency matching rules. $\mathcal{T}_p$ represents the set of child-tasks of task τ_p; E_p is the set of edges connecting these children in the task graph of τ_p; a_i the action, D_i the set of dependencies, and p the parent of task τ_i; $\mathcal{M}_{\mathcal{L}}$ represents the memory locations in the local memory space; p_0 is the global root task.

For a set of tasks $\mathcal{T}_p$ that are child-tasks of a task τ_p, the creation of the dependency graph from the dependency definitions is governed by the rules outlined in Figure 3.2. Rules 3.1–3.4 define the structure of tasks and dependencies, as discussed above. Rule 3.5 defines the matching rule used to insert edges into a task graph: an edge (τ_f, τ_g) may only exist if there are dependencies in both tasks that reference the same memory location and task τ_f has been discovered before task τ_g ($f < g$) in the sequential task discovery process (3.5a). Furthermore, either (or both) of the dependencies must be an output dependency (3.5b), and there may not exist a task τ_h that has been discovered after τ_f but before τ_g with an output dependency on the same memory location (3.5c). The last condition is used to avoid transitive dependencies to occur explicitly in the task graph, i.e., if $\tau_f \prec \tau_h$ and $\tau_h \prec \tau_g$ then the dependency $\tau_f \prec \tau_g$ is transitive and not marked by an edge in the graph.

This set of matching rules allows for the unambiguous creation of a DAG of tasks from the set of dependency definitions encountered during the discovery of the local task graph. This model should be applicable to similar task-based programming models employing data dependencies between sequentially discovered tasks, including OpenMP, PaRSEC DTD, and StarPU.

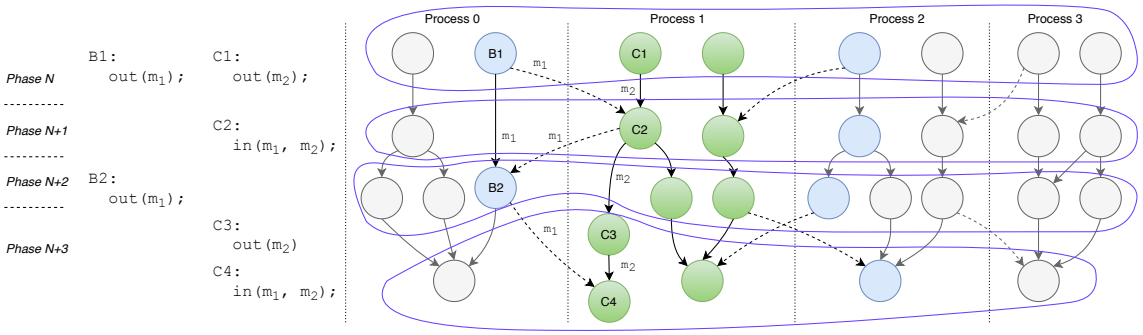


Figure 3.3.: The global task graph of Figure 3.1 partitioned by phases.

3.2. Connecting Local Components

In order to connect the local task graphs, edges have to be found between otherwise independently populated components. Since the components have been populated in parallel, there is no ordering between any two tasks belonging to different components. It is thus impossible to reliably insert edges connecting the components based on the information available so far. For example, in Figure 3.1 the information on the dependencies on m_1 of the tasks B_1, B_2, C_2, and C_4 are not sufficient to establish whether there exists an edge between B_1 and C_2 or between B_1 and C_4 as the definition of the dependencies might suggest both to be possible.

It should be noted that since processes may create different numbers of tasks the discovered sequence of tasks within the processes cannot be used as an indicator for the relative ordering between tasks in different components.

As the additionally required information cannot be derived from the existing information, it can only be provided by the user. Therefore, the concept of *task phases* is introduced in this work. Where the discovery of local components of the graph described above form a *vertical* partitioning of the global task graph, the concept of phases creates a *horizontal* partitioning of the graph. In contrast to the local partitions, phase partitions can cross process boundaries and contain all vertices in the graph that were discovered in the same phase. Such a partitioning of the task graph in Figure 3.1 using phases is depicted in Figure 3.3.

Figure 3.4 lists an extended set of matching rules that include the phase information. Based on the definition provided in Figure 3.2, Rule 3.3, the definition of a task is extended to include a phase $\Phi \in \mathbb{N}$ and the scheduler instance $\sigma \in \Sigma$ in whose local partition it resides (Rule 3.8, Σ is the set of scheduler instances). While in the local case described above, the sequence of discovery of tasks was sufficient to establish the relation $\tau_f \prec \tau_g$ between two tasks τ_f and τ_g with matching data dependencies through the index numbers f and g, it is not sufficient for tasks

residing in different local partitions as there can be two tasks with the same index number, e.g., B_2 and C_2 in Figure 3.1. At the same time, the model should permit multiple dependent tasks per phase in one local partition. Thus, the phase Φ cannot be the only ordering criteria in this model.

Instead, an *index-phase tuple* $\omega_i = (i, \Phi_i, \sigma_i)$ is introduced to establish an ordering (Rule 3.10). A partial ordering relation $\leq$ between two index-phase tuples ω_a and ω_b is defined as

$$(a, \Phi_a, \sigma_a) \leq (b, \Phi_b, \sigma_b) = \begin{cases} \text{true}, & \text{if } \Phi_a < \Phi_b \\ \text{true}, & \text{if } \sigma_a = \sigma_b \wedge \Phi_a = \Phi_b \wedge a \leq b \\ \text{undefined}, & \text{otherwise} \end{cases}$$

The relation $\omega_a \leq \omega_b$ thus holds if either task τ_b has been discovered in a later phase than τ_a or—if both were discovered in the same phase—τ_b has been discovered after τ_a by the same scheduler. The relation is undefined for tuples representing tasks in the same phase discovered by different scheduler instances.

The relation $\leq$ on ω_i forms a partial order over the set of tasks with matching dependencies: it is reflexive ($\omega_a \leq \omega_a$), antisymmetric ($\omega_a \leq \omega_b$ and $\omega_b \leq \omega_a$ implies $\omega_a = \omega_b$), and transitive ($\omega_a \leq \omega_b$ and $\omega_b \leq \omega_c$ implies $\omega_a \leq \omega_c$).

As noted in Rule 3.9, dependencies can now reference any global memory location ($\mathcal{M}_G$) in addition to the local memory locations. The root task p_0 is an implicit task that is the same at each process.

Similar to the local matching rules discussed in Section 3.2, an edge between two tasks in the task graph is inserted if a happens-before relation between the two tasks exists. This is true for tasks that have matching dependencies, whose $\leq$-relation holds on their index-phase tuples, and for which the dependency relation is not transitive (Rules 3.11a–3.11c). Dependencies across process boundaries always also cross phase-partition boundaries, as depicted in Figure 3.3. Thus it is now possible for the scheduler to unambiguously connect the local partition(s) of the global task graph with the edges connecting tasks in different phase-partitions, e.g., $B_1 \prec C_2$, $C_2 \prec B_2$, and $B_2 \prec C4$ in Figure 3.3.

$$\tau_p \in \mathcal{T} \cup \{p_0\},\ \mathcal{T}_p \subseteq \mathcal{T},\ E_p \subseteq \mathcal{T}_p^2 \tag{3.6}$$
$$\tau_i, \tau_f, \tau_g, \tau_h \in \mathcal{T}_p,\ i,f,g,h \in \mathbb{N},\ 0 \leq i,f,g,h < |\mathcal{T}_p| \tag{3.7}$$
$$\tau_i := (a_i, D_i, \tau_p, \Phi_i, \sigma_i),\ \Phi_i \in \mathbb{N},\ \sigma_i \in \Sigma \tag{3.8}$$
$$d_i := (k_i, m_i) \in D_i,\ k_i \in \{in, out\},\ m_i \in \mathcal{M}_\mathcal{L} \cup \mathcal{M}_\mathcal{G} \tag{3.9}$$
$$\omega_i := (i, \Phi_i, \sigma_i) \tag{3.10}$$

$$(\tau_f, \tau_g) \in \Big\{ E_p \Big| \exists d_f\, d_g : m_f = m_g \wedge \omega_f \leq \omega_g \ \wedge f \neq g \ \wedge \tag{3.11a}$$
$$(k_f = out \vee k_g = out) \wedge \tag{3.11b}$$
$$\big(\nexists d_h : m_g = m_h \wedge k_h = out \wedge \omega_f \leq \omega_h \leq \omega_g \big) \Big\} \tag{3.11c}$$

Figure 3.4.: Formal description of the global memory task data dependency matching rules. $\mathcal{T}_p$ represents the set of child-tasks of task τ_p; a_i the action, D_i the set of dependencies, and τ_p the parent of task τ_i; $\mathcal{M}_\mathcal{L}$ and $\mathcal{M}_\mathcal{G}$ represent memory locations in the local and global memory spaces; p_0 is the global root task; Σ is the set of schedulers; and Φ describes a task phase.

3.3. Scheduler Interaction

As mentioned above, the scheduler instances have to exchange information on the discovered dependencies with references to remote global memory locations. Figure 3.5 depicts the sequence of inter-scheduler communication necessary. As an example, assuming that the memory location m_1 is owned by `Process 0` in Figure 3.1, `Process 1` would send the information about the input dependencies on m_1 of tasks C_2 and C_4 to the scheduler in `Process 0`. This is marked as ① in Figure 3.5.

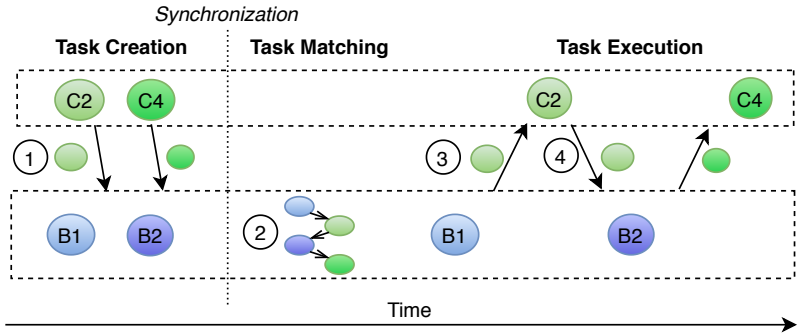


Figure 3.5.: The scheduler interaction required to handle remote dependencies.

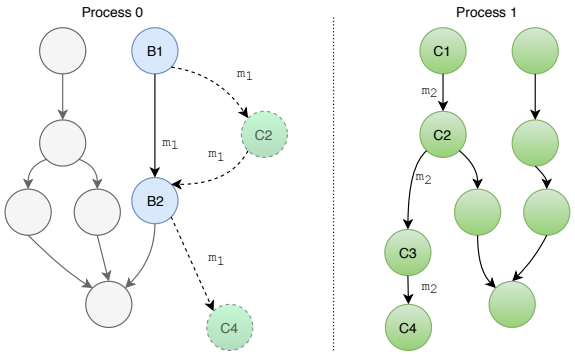


Figure 3.6.: Local task graphs extended with the received remote data dependencies.

After all remote dependencies have been communicated, the scheduler instances extend their local task graphs with the dependency information they have received. In order to ensure that all dependency information has been exchanged, a collective synchronization has to be performed across all processes. The following extension of the local task graph is marked with ②. This extension step (for `Process 0` and `Process 1`) is also depicted in Figure 3.6: Only `Process 0` extends its local task graph because it has received remote dependency information from `Process 1`. `Process 1` has no information on any of the tasks discovered on `Process 0`.

As soon as the local task graph extension has completed, tasks may commence execution, provided all necessary dependencies have been satisfied. After the completion of a task, all of its local and remote dependencies are released. For this purpose, the edges in the task graph leaving the current task's node are traversed to visit all successor nodes and decrement the number of open dependencies. In case the edge points to a remote task, the scheduler sends a dependency release message to the scheduler owning the respective task (③ in Figure 3.5).

Once the remote task has finished its execution, the scheduler will send back the information that the task has completed ④, signaling that any potential write-after-read dependency (in the case of remote input dependencies) or write-after-write and read-after-write dependency (in case of remote output dependencies) can be released. Eventually, all dependencies will be released this way and all tasks in the task graph will have been executed.

3.4. Synchronization and Execution Constraints

As discussed in Section 3.2, the relation $\leq$ between index-phase tuples of two tasks τ_a and τ_b is not defined for two tasks discovered by two different scheduler instances in the same phase. As a consequence, the specification of conflicting

dependencies for two tasks on the same global memory location in the same phase is erroneous as the schedulers cannot determine the ordering between the two tasks. Since the result of the execution of a task-based application should be deterministic the scheduler cannot randomly guess the order. Similarly, the two tasks cannot be executed in parallel as the resulting conflicting writes to global memory would introduce non-determinism.

This constraint does not preclude schedulers from discovering tasks with conflicting dependencies on the same local or global memory location within the same phase on the same process. The $\leq$ relation on index-phase tuples is well-defined in that case such that the ordering is determined by the index of the task within the phase. In the presence of multiple output dependencies on the same memory location within one phase, the last dependency is the *dominant* dependency in that phase and thus remote dependencies will be matched against this dependency. This is again similar to the behavior of barrier synchronization, where the last written value before the barrier will be consumed by a read from the same memory location after the barrier.

In general, no task may be executed before the matching step has completed. Without this constraint, it would be impossible to reliably handle dependencies as not all remote dependencies have been incorporated into the local task graph. As an exception to this rule, tasks in the first phase may start executing immediately. Given the prohibition of conflicting remote dependencies in the same phase outlined above, this exception is safe as no write-after-read or write-after-write dependencies crossing process boundaries may occur within the first phase, that is, a write in one task cannot overwrite data that is consumed by another task in the first phase.

3.5. Deadlock Freedom

It can easily be shown that the dependency graph created through data dependencies does not contain cycles that could lead to a deadlock.

Two cases have to be distinguished: i) dependencies between tasks discovered on the same process; and ii) dependencies spanning across process boundaries. For the first case, it is sufficient to note that a task τ_b can only depend on a previously discovered tasks, i.e., the dependency $\tau_a \prec \tau_b$ can only occur if $a < b$. This property is guaranteed by the sequential discovery of tasks within a single process in combination with the fact that dependency matching is strictly backward-looking.

For the second case, the tasks are discovered in parallel and thus dependency matching cannot be done strictly backward-looking. Instead, the $\leq$ relation on

index-phase tuples introduced above can be used to show that the extension of the local task graph does not introduce cycles. Given an existing DAG that is the result of local task discovery, it is trivial to see that the $\leq$ relation on the tasks' index-phase tuples holds between tasks in that graph. For tasks discovered in the same phase by the same scheduler, the $\leq$ relation on the tasks' index-phase tuples is defined in terms of the index. Otherwise it is defined in terms of the phase.

Considering two tasks τ_a discovered by scheduler σ_a and τ_b discovered by scheduler σ_b with $\sigma_a \neq \sigma_b$. A cycle between these tasks would exist if dependencies exist such that $\tau_a \prec \ldots \prec \tau_b$ and $\tau_b \prec \ldots \prec \tau_a$, either as direct or transitive dependencies. The first dependency chain can be constructed only if τ_a was discovered in an earlier phase than τ_b, i.e., $\phi_a < \phi_b$ and thus $\omega_a \leq \omega_b$. The second dependency chain would only be possible if $\phi_b < \phi_a$ and thus $\omega_a \leq \omega_b$. However, given the asymmetry property of the $\leq$ relation on index-phase tuples, $\omega_a \leq \omega_b$ and $\omega_b \leq \omega_a$ imply that $\omega_a = \omega_b$. This would directly contradict the requirement that $\sigma_a \neq \sigma_b$. □

Chapter 4

Inter-Scheduler Communication

Parts of the research presented in this chapter have been published in the following peer-reviewed paper: J. Schuchart, G. Bosilca, A. Boutellier. "Using MPI RMA for Active Messages." In: Proceedings of the 2019 Supercomputing Conference, Workshop series. 2019. The author of this work claims responsibility for the design and implementation of the message queues and benchmarks.

The interaction between the different scheduler instances as described in the previous chapter requires low-latency high-throughput communication that scales well both with the number of messages sent and the number of instances involved. Each message triggers an action at the receiving side, e.g., releasing a task's dependency, and thus can be modeled as active messages (Active Messages (AMs)), a concept that has first been introduced as part of the experimental programming language Split-C [70]. At its core, an AM contains a header with a user-level function that is executed upon reception with the body of the message as the argument. While in their original form AMs have been implemented using interrupt handlers triggered by the network interface to asynchronously deliver a message, many software-based implementations today rely on active participation of the CPU at the receiving side to execute the handler, i.e., by polling for new messages.

The most prominent example is GASnet [71], which provides a portable implementation for AMs on different platforms, including an implementation based on MPI-1 functionality [72]. Both GASnet and MPI implementations commonly rely on AMs for the implementation of RMA primitives on platforms without RDMA capabilities [73]. However, where possible both GASnet and MPI implementation move to employ RDMA [74, 75]. Other implementations based on MPI have been proposed [76] and efforts to standardize an AM interface have been made in the

past [77]. Support for AMs has also been proposed for inclusion into the MPI [78] and OpenSHMEM standards [79]. At least the efforts in the context of MPI seem to have been suspended in the meantime. Vendor-specific active message implementations have been proposed as well [80, 81]. However, the goal of this work is to provide vendor-neutral implementations based on the capabilities of MPI.

The remainder of this chapter will derive the requirements for AMs to be used in this work before discussing and evaluating different implementations.

4.1. Requirements

Based on Figure 3.5, three types of messages can be inferred: i) to communicate discovered remote dependencies (type *A*, ① in Figure 3.5); ii) to release these dependencies (type *B*, ③); and to signal task completion (type *C*, ④). Type *A* messages are critical for the discovery and matching of the local task graph and occur in bulk during the discovery process. A high throughput is essential for this type of messages as these are critical for a fast discovery process.

Moreover, the notion of communication rounds is essential to ensure that all discovered remote dependencies have been communicated before starting the task matching process. The delivery of type *A* messages is not time-critical and could even be delayed until the end of a communication round. Message coalescing is thus a viable option to increase throughput of type *A* messages.

Types *B* and *C*, on the other hand, occur more infrequently but their low-latency delivery is imperative to avoid starvation of threads by transitioning tasks into the *runnable* state as soon as possible so that they are available for threads to execute. The delivery of these messages should happen mostly instantaneous, although a certain degree of coalescing might be desirable in order to increase throughput and thus lower the overall latency. Hence, a purely round-based system such as the one proposed in [76] would not fulfill the latency requirements. However, the amount of coalescing is a trade-off between single-message latency and overall throughput.

The message queue to be designed should not be limited to the three message types described above. Other messages, e.g., transfer of task ownership, should be possible as well. All messages may have different sizes and while it would be possible to set a fixed transfer size as the maximum size of all possible messages, that approach is inefficient.

4.2. Active Message Queue Designs

This section explores the designs of different message queue implementations. The goal is to design a message queue that can provide both high throughput and low latency message delivery. All queues have been implemented using MPI and their designs are geared towards the capabilities provided by the MPI-3.1 standard.

Three basic operations have to be provided by a queue implementation. First, a `trysend` procedure that should make an attempt to start the delivery of a message but is not required to complete the delivery. This routine may fail in case of insufficient resources, leaving it to the calling entity to retry at a later point in time. A second operation, `process`, is provided to read and execute received messages. A variation of this procedure, `process_round`, completes a processing round by ensuring that all messages have been delivered and processed on all participating processes before returning. While `process` allows polling for new messages and may return immediately, `process_round` is a collective operation across all processes that is required to complete all communication operations active at the time of the call.

On top of these primitives, it is possible to implement message coalescing by combining multiple AMs into a single message. Thus, the term *message* in the following refers to a buffer that is transferred and may include an arbitrary number of AMs that are then processed individually by the receiver.

All designs are multiple-producer-single-consumer queues: multiple writers can send messages to a process but a process will only read its local queue.

4.2.1. Send/Recv-based Queue

A simple message queue based on MPI can be implemented using non-blocking send and receive operations, i.e., `MPI_Isend` and `MPI_Irecv`. Each participating process posts a number of receive operations with unspecified sender using `MPI_SOURCE_ANY`. A call to `process` tests for the completion of any of these receive operations and processes the received message(s) before re-initiating the receive operation.

The implementation of `trysend` starts a send operation, as described above. In order to prevent resource exhaustion, it is desirable to limit the number of active non-blocking operations by testing previously issued send operations for completion and reusing the buffers of completed operations. It should be noted that the communication pattern exhibited by the inter-scheduler communication differs from the pattern seen in typical HPC applications in that the number of messages to be sent and received is not know in advance and thus requires a mapping of M messages to N message buffers, typically with $M \gg N$.

Fortunately, the MPI standard allows the reception of messages that are smaller than the size provided for the receive operation [7, §3.2.4]. This allows for the definition of an upper bound for the message size while enabling smaller messages to be transferred, e.g., in the case of a lack of additional available messages.

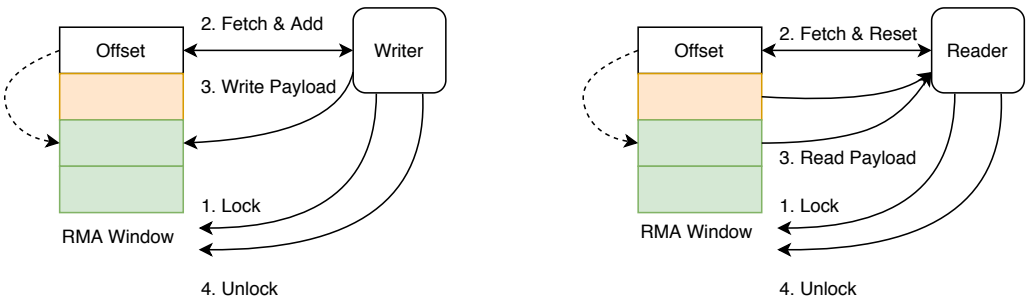
The completion of a communication round requires a certain amount of bookkeeping on the number of sent and received messages: the completion of a standard send operation does not imply the actual reception of the message. Instead, the message may be buffered at any level of the underlying communication stack. Thus, each process keeps track of the number of messages received from and sent to each other process. The end of a round requires the communication of the number of sent messages to each target, ensuring that all sent messages have been received and processed. MPI provides the `all-to-all` collective communication operation, which can be used in its *immediate* form: upon entering the `process_round` routine the collective operation is joined by issuing a call to `MPI_Ialltoall` before processing incoming messages until all participating processes have joined the `all-to-all` operations and the number of expected messages has been processed.

4.2.2. Message Queues Based on MPI-RMA

Starting with MPI-2.0, the MPI standard has included an abstraction for one-sided communication called MPI-RMA. Local memory is exposed to other processes in so-called *windows* through which memory can be accessed by all processes in the group that created the window. RMA operations are initiated by an *origin* process and directed at a *target* process.

This support has been extended in MPI-3.0 with the addition of *passive-target* synchronization, which allows processes to lock and unlock a window, either using shared or exclusive locks, and complete outstanding RMA operations without requiring the active participation of the target process. An origin process can wait for all locally issued outstanding operations to complete by waiting for a flush operation to complete. MPI offers both one-sided `put` and `get` operation as well as atomic operations. The latter allow processes to concurrently modify data within a window safely without requiring external synchronization, e.g., by atomically incrementing a counter or conditionally exchanging a value.

The current standard also imposes restrictions on the mix of operations that may be employed. By default, only one operation type may be issued on the same memory location. This restriction can be slightly relaxed to also allow for a concurrent `no-op`, allowing for atomic reads in addition to modification. While proposals have been



(a) Writing into a queue.

(b) Reading from a queue.

Figure 4.1.: Sequence of operation on a message queue using window locks.

made to allow for further relaxation [82], there does not seem to be an effort to include these changes in the upcoming MPI-4.0 standard.

The following sections explore several different designs for implementing active message queues using MPI-RMA windows.

4.2.2.1. Using Window Locks

MPI-RMA window locks can be used in a similar fashion as reader/writer locks in Portable Operating System Interface (POSIX) [83]: taking an exclusive lock is similar to a writer lock while the shared lock is equivalent to a reader lock. However, the naming of POSIX locks can be misleading: a thread taking a reader lock signals that concurrent accesses by other threads owning a reader lock are safe while a thread taking a writer lock signals that exclusive access is required. Hence, the naming scheme employed by MPI is better suited.

Figure 4.1 displays the sequence of operations needed for writing to and reading from a message queue implemented using RMA windows. In this design, a queue consists of one RMA window that can be locked and unlocked by all participating threads. The beginning of the queue contains a header that stores the current tail position, i.e., a pointer past the end of the last entry. A process wishing to deposit a message into that queue acquires a *shared lock*, atomically adds the size of the message to the offset, and writes the message at the previous offset before releasing the shared lock (Figure 4.1a). In total, writing to a queue requires a sequence of four operations (`lock`, `fetch-and-op`, `put`, and `unlock`). Assuming that window locks are implemented using atomic memory operations (AMOs), the `unlock` can be overlapped with the `put` operation, leading to a latency of three operations.

If the remaining space in the queue is not sufficient for the message to be deposited, the writer rolls back the update of the offset and releases the lock. In that case,

`trysend` returns an error indicating that the message has not been sent and the caller may try again later. This allows the caller to perform useful work while waiting for the queue to become empty again, e.g., by processing the local queue or attempting to send messages to other processes.

Similarly, the local process's attempt to process incoming messages involves acquiring an *exclusive lock*, reading and resetting the offset, and reading the payload before releasing the lock and again allowing writers to deposit messages.

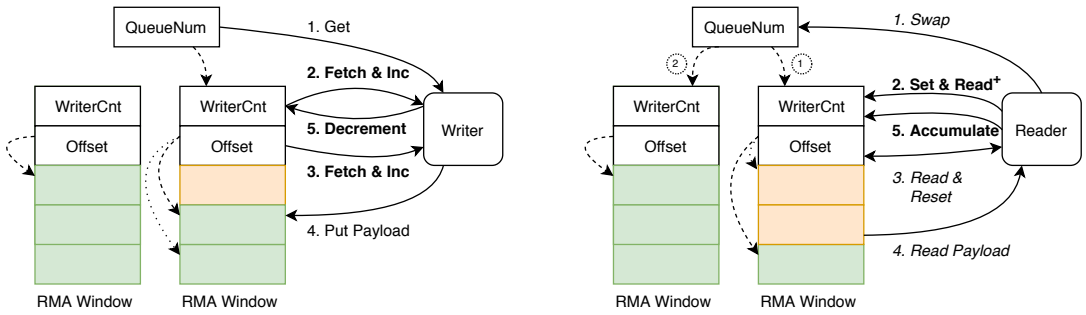
It should be noted that the MPI standard does not provide any guarantees regarding fairness: a reader trying to acquire an exclusive lock might starve if writers continuously acquire shared locks, essentially prohibiting progress because the queue might never be processed even when full. The problem of non-scalable locks has been widely discussed and scalable RMA locks have been presented in the past [84, 39]. Currently, the only viable solution available is to implement a back-off strategy for the writers to allow a reader to eventually acquire an exclusive lock and clear the queue.

Thread-safety Considerations When using the proper initialization using `MPI_THREAD_MULTIPLE`, the MPI standard provides the guarantee that operations issued by multiple threads in the same process must appear to have been executed in some order [7, §12.4.1], i.e., that concurrent calls into the MPI library are thread-safe. However, a process is only allowed to hold one lock at a time at any target process in a window [7, §11.5.3]. Thus, multiple threads in the same origin process may not lock a window concurrently at the same target, requiring coordination and potentially mutual exclusion between multiple writer threads.

4.2.2.2. Using Atomic Memory Operations

In order to avoid the mutual exclusion of writers and the reader and to allow concurrent threads to write to the same queue, this section devises a queue design that avoids using window locks by entirely relying on AMOs provided by MPI. This scheme is depicted in Figure 4.2. Here, a message queue consists of two memory areas (henceforth called queues) that are accessible by all processes. Each area has a separate header containing its `Offset`. In addition, a `WriterCnt` is used to track the number of active writers. A single value `QueueNum` is used to signal the currently active queue.

A process attempting to deposit a message (Figure 4.2a) first queries the `QueueNum` variable to determine the currently active queue. In a second step, the process atomically increments the `WriterCnt` on the active queue before atomically updating



(a) Writing into a queue. (b) Reading from a queue.

Figure 4.2.: Sequence of operation on a lock-free message queue using atomic RMA operations (atomic operations in bold).

the `Offset` variable. After writing the message to the queue, the process deregisters from the queue by decrementing the `WriterCnt`. Thus, depositing a message requires at least five operations: $1 \times$ `get` $+ 2 \times$ `fetch-and-op` $+ 1 \times$ `put` $+ 1 \times$ `accumulate`.

The reader, in turn, first updates the `QueueNum` variable to activate the currently inactive queue. The old queue is then marked as inactive by subtracting a large value `WRITE_SIGNAL` from the `WriterCnt` variable and waiting for all active writers to complete (marked as `Set & Read`$^+$ in Figure 4.2b). After reading the `Offset` from the old queue the reader can start processing data. Before activating this queue in the next processing round, the `WRITE_SIGNAL` has to be added back to `WriterCnt` to zero out the signal.

Writing to a queue may fail for two reasons: i) the queue is full; or ii) a reader has started processing the queue after the writer queried the `QueueNum` but before successful registration. In the first case, the modifications to `Offset` should be rolled back and the `WriterCnt` be decremented before signaling to the caller that the call to `trysend` failed. In the second case, it is sufficient to deregister and restart the process immediately as the newly active queue is already available for writing.

This queue relies on the previously mentioned relaxed constraint on the RMA operation mix, called `same-op-no-op`. Henceforth, this queue design will be referred to as `sopnop`.

Operation Ordering Considerations It is important to note that the sequence of operations needed for depositing a message has to be performed in strictly sequential order. In particular, this requires a flush after each operation, leading to full round-trip latencies in the network. As an example, the writer has to i) complete registration before incrementing the `Offset` variable and ii) complete writing before deregistration. While the first case requires *completion* of the first operation and

checking of the state of `WriterCnt` before modifying `Offset`, the second case only requires *ordering*. However, MPI in its current form does not offer any ordering mechanisms for non-overlapping or non-atomic operations except for `flush`, i.e., waiting for the operation to complete at the target. A proposal addressing this issue will be discussed in Section 4.4.1.

A Note on Multi-threading This queue design does not require any coordination or mutual exclusion of threads within the same origin process trying to deposit messages at the same target process. All RMA operations will be executed in some order and no deadlock may occur. However, each `flush` operation is required to flush *all* outstanding operations to the same target, i.e., wait for the completion of all in-flight operations issued by any thread. This may cause threads to wait unnecessarily for operations issued by other threads, thus potentially increasing latencies.

4.2.2.3. Lock-free Alternative Designs

It should be noted that neither of the RMA-based queue designs described above are truly lock-free. In the AMO-based queue described in Section 4.2.2.2, the `WriterCnt` effectively serves as a reader-writer lock. Thus, both algorithms are not suitable for application that require either real-time guarantees or fault-tolerance as one writer may block the reader (and thus eventually all other writers) from progressing.

An interesting alternative are lock-free data structures, for which a large body of work has been established in the past [85–88], where some partially rely on hardware support such as transactional memory [89] or software-based consistency [90]. Of particular interest are lock-free First In First Out (FIFO) and circular buffer implementations [91].

However, many (if not all) of these algorithms rely on *conditional-swap* (or compare-and-swap (CAS)) operations to ensure memory consistency while avoiding mutual exclusion. As will be discussed in Section 4.3.1, these conditional-swap operations are orders of magnitude more expensive in a distributed setting than in shared memory[10]. Especially in highly contentious settings such as the message queue used for frequent inter-scheduler communication, the cost of repeating CAS operations until a successful attempt can become prohibitively high.

For completeness, however, one such algorithm should be sketched out here in the form of a Last In First Out (LIFO) queue implemented using CAS operations.

[10]The number of clock cycles for an atomic CAS operation (`LOCK CMPXCHG`) has been determined as 19 cycles, or 5.4 ns, on an Intel i7-4770K (Haswell, 3.5 GHz) processor in [92, p. 211].

The statically allocated window memory is divided into slots that are inserted into a LIFO of free slots, called F. A writer attempting to deposit a message takes a free slot from F, deposits the message into the slot and pushes the slot onto a second LIFO of ready messages R, from which a reader will eventually read it and return it to F.

Using AMOs, this algorithm requires a sequence of $2 \times$ `get` to query the first and second element of F and at least one CAS to replace the head pointer of F with the second element. The message can now be `put` into the acquired free slot. In order to push the slot into R, another `get` and at least one CAS operation are necessary. Overall, the sequence of operations is $\left([2 \times \texttt{get} + \texttt{put} + \texttt{cas}]^{+} + \texttt{get} + [\texttt{put} + \texttt{cas}]^{+}\right)$. In the case of no conflicts, this algorithm already requires seven RMA operations, more than any of the previously devised algorithms. The parts marked with $^{+}$ have to be repeated upon conflicts during the i) acquisition of a free slot and ii) the insertion into the LIFO, respectively. It remains as future work to further determine whether and at what cost lock-free (and wait-free) algorithms can be implemented using AMOs for runtime systems and applications that require fault tolerance. Overall, the fault tolerance of the message queue would only be one aspect in the design of the cooperating scheduler instances. Moreover, fault tolerance of MPI and MPI implementations [93, 94] as well as task runtime systems [95] has been an ongoing research effort in itself.

4.3. Evaluation

In the following section, the general performance of the proposed queue designs will be evaluated. In order to understand the results of the RMA-based queues, an analysis of the basic RMA operation latencies on two different high-performance networks is presented in Section 4.3.1 before investigating the latency (Section 4.3.2) and throughput at scale (Section 4.3.3) of the different implementations.

The experiments have been conducted on two different systems (a Cray XC40 and Taurus, a Bull Linux cluster). A detailed description of these systems can be found in Appendix A.

4.3.1. MPI RMA Operation Latencies

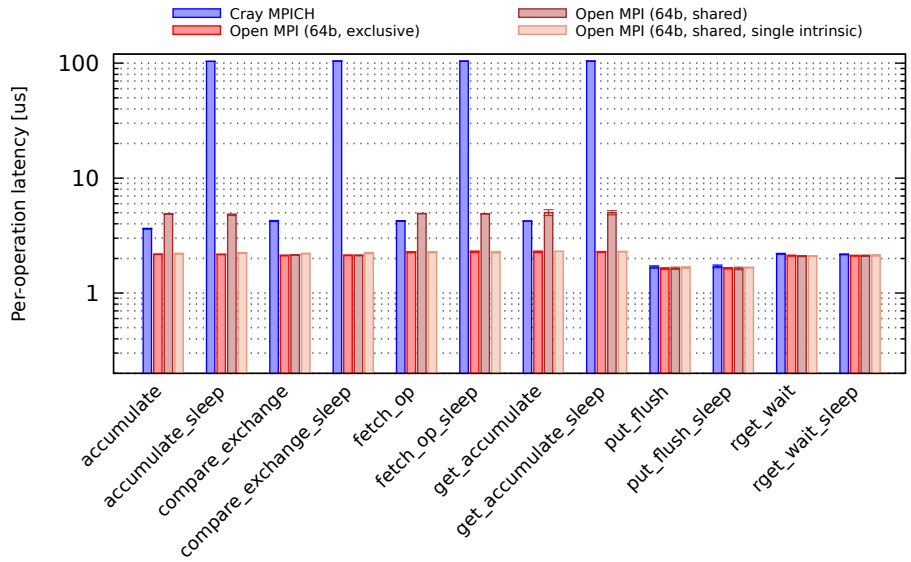
A detailed understanding of the system behavior is indispensable for the understanding of the performance of the message queue implementations. Two different aspects are important for this: i) the latency of a single operations; and ii) whether this latency is dependent on the behavior of the target process. The MPI standard

leaves implementations great freedom in how RMA operations are implemented. As mentioned in the beginning of this chapter, some MPI implementations may rely on a two-sided protocol to implement MPI-RMA operations. On systems that do not support Remote Direct Memory Access (RDMA), this maybe the only viable implementation. However, as will be shown, not all MPI implementation leverage the capabilities of modern high-performance networks. For the message queues based on MPI-RMA this means that the individual steps involved in delivering a message to a target will not progress unless a thread at the target triggers the implementation's progress engine. This can be either a software progress thread or any of the application threads calling into the MPI library.

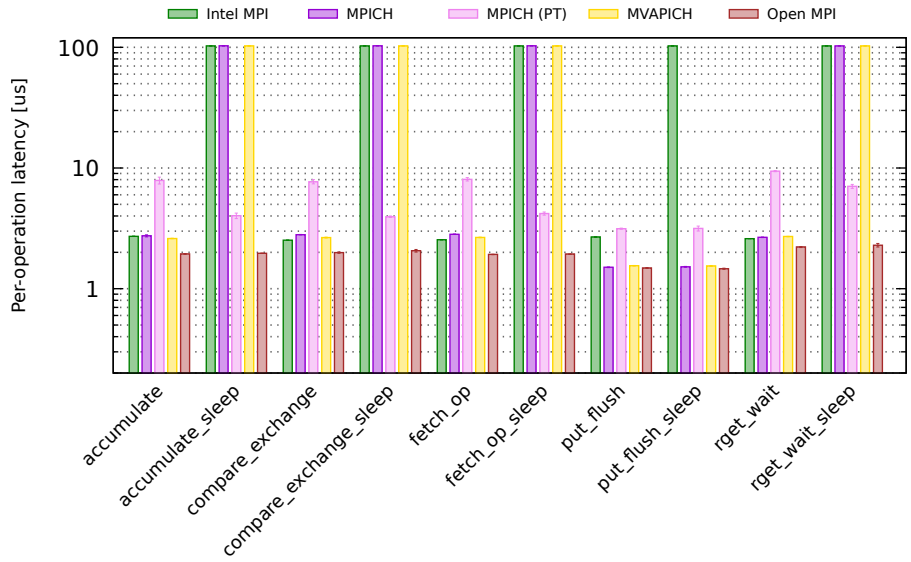
In order to determine both the progress behavior and the latency, a set of micro-benchmarks has been developed similar to the OSU benchmark suite [96]. The benchmarks perform each operation in two different configurations: i) with the target actively waiting inside an MPI call (`MPI_Barrier`); and ii) with the target busy outside of MPI for a given amount of time t_w. For a fixed number of repetitions N of a given operation, the average latency of that operation t_o reported in the first case is simply the time measured t_m for all repetitions divided by the number of repetitions: $t_o = \frac{t_m}{N}$. The expected range of the average operation is in the order of microseconds [97].

For the second variant of the benchmark, two cases have to be distinguished: if an RMA operation is handled by the network hardware at the target and does not involve the target CPU, the average latency should be similar to the latency observed in the first benchmark version. However, if the operation does involve the remote CPU and thus is not processed while the target is busy for t_w seconds outside of MPI, the average latency will be $t'_o = \frac{t_w}{N} + t_o$. In order to assess the progress characteristics of an MPI implementation, it is important that $\frac{t_w}{N} \gg t_o$. In the following discussion, the values $t_w = 10\,\text{s}$ and $N = 100\,000$ were chosen, resulting in $\frac{t_w}{N} = 100\,\mu\text{s}$.

Figure 4.3 presents the average latency of individual RMA operations as measured on the two test systems described in Appendix A. The measurements suffixed with `_sleep` represent the case in which the remote side is busy for the time t_w outside of MPI. The MPI window was locked using a shared lock and the operations were performed on a single 64-bit integer value. On the Cray system (Figure 4.3a), non-atomic operations such as `put` (1.5 μs) and `get` (2.1 μs) show latencies that are independent of the MPI implementation in use. The difference between the two is that a `put` operation is not required to wait for a response other than the signal that the operation has completed whereas `get` operations imply a full roundtrip in the network. These values are independent of whether the target process is active in MPI.



(a) Cray XC40



(b) Taurus

Figure 4.3.: Basic RMA operation latencies using different MPI implementation on two different hardware platforms.

However, with AMOs the situation is different. Most strikingly, Cray MPICH does not provide progress for operations if the target process is busy outside of MPI (blue bars reaching 100 μs in Figure 4.3a). On the other hand, the behavior of the target process does not influence the latencies observed with Open MPI. In the default configuration (`Open MPI (64b, shared)`), the latency for AMOs is slightly above 5 μs, significantly higher than for a `get` operation. The reason for this increase is not to be found in higher latencies for AMOs imposed by the network hardware itself, however. Instead, these latencies stem from an optimization in Open MPI that favors the performance of larger numbers of elements passed to a call to `MPI_Accumulate` over the latency of single-element operations[11] and imposes two additional atomic network operations for each accumulate operation. A special `info` key `acc_single_intrinsic` may be used in Open MPI to change the protocol used for accumulate operations and force Open MPI to use network AMOs instead. While this is not part of the standard, measures to define a common interface are under active discussion within the MPI forum [82].[12]

With this `info` key set (`Open MPI (64b, shared, single intrinsic)`), the latencies of accumulate operations match the latency of a `get`, i.e., they require only a single round-trip in the network. Hence, this option will be enabled for the remainder of the discussion of performance where Open MPI is involved in this work.

A similar picture emerges on *Taurus*: similarly to the Cray system, MPICH and its derivatives (Intel MPI, MVAPICH) provide latencies for `put` that are comparable to Open MPI on both systems. Interestingly, even for `put` the Intel MPI implementation appears to lack progress in case of an inactive target. Similarly, the MPICH-based implementations do not provide progress for `get` and accumulate operations either on that system while Open MPI is able to offload all operations to the network.

While the MPICH-based implementations do not provide progress in hardware, some of them offer progress using a *progress thread* that is used to offload communication operations to a dedicated CPU core. The effects of enabling such software-based progress can be seen in the measurements for `MPICH (PT)` in Figure 4.3b. There are two interesting observations to make: i) while operations are progressing even if the target is inactive, their latencies are generally higher than those observed with

[11] Open MPI user mailing list: `https://users.open-mpi.narkive.com/d1SiDG5J/ompi-users-latencies-of-atomic-operations-on-high-performance-networks`. Last accessed May 4, 2019. The optimization allows Open MPI to acquire a special lock, fetch all values and perform the accumulation before writing back the results and releasing the special lock.

[12] MPI-RMA working group discussion on "New info hints to enable network hardware atomics in RMA atomics": `https://github.com/mpiwg-rma/rma-issues/issues/8`, last accessed October 12, 2019.

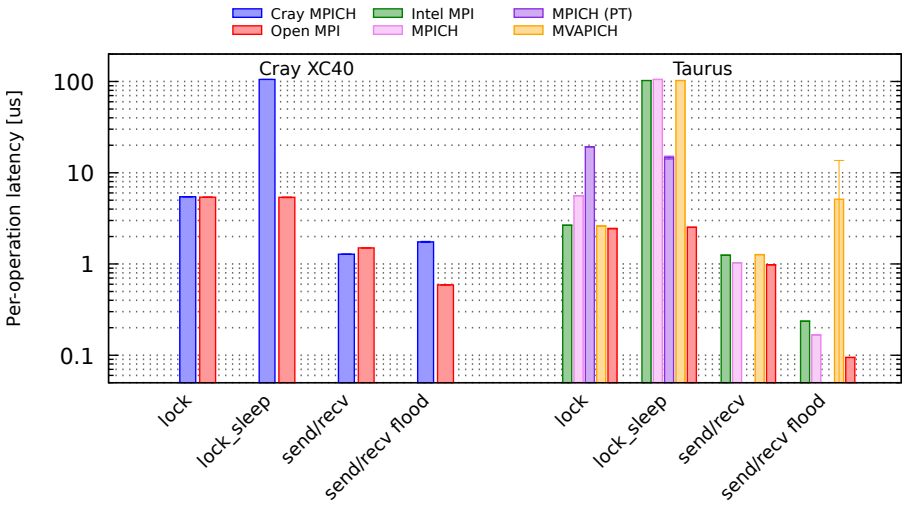


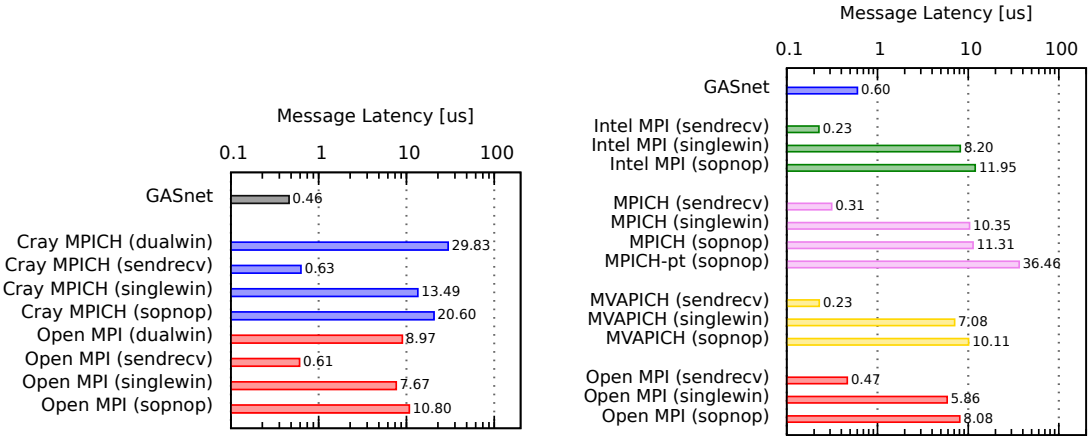
Figure 4.4.: Window lock-unlock and send/recv latencies.

hardware support (by factors between two and four); and ii) latencies are generally lower when the target is not active in MPI, hinting at contention between the main thread and the progress thread.

Miscellaneous Measurements In addition to the basic RMA operation latencies, the latencies of `send/recv`-based data transfer and of locking a window are also of interest for the discussion of the performance of the active message queue implementation. Figure 4.4 displays measurements for these aspects on both systems. The latencies of a sequence of window `lock` and `unlock` operations on the Cray system hint at the use of two sequential atomic operations within Open MPI.[13] The Cray MPICH implementation, again, does not progress window locks when the target is inactive, which is similar to the MPICH-based implementations on Taurus.

The latency of a single direction `send/recv` in a bi-directional communication (*ping pong*) is displayed as `send/recv` in Figure 4.4. On both systems the latency is found to be between 1 μs and 1.5 μs. For the discussion of the `sendrecv` message queue, however, the behavior under pressure is more relevant, i.e., a sender flooding the receiver without waiting for an acknowledgment. Here, the average per-message latency is generally lower than in the bi-directional case, except for Cray MPICH and MVAPICH, with the latter exhibiting a high degree of volatility.

[13] *Note*: while the MPI standard allows implementations to defer the acquisition of a lock up until the matching `unlock` operation, none of the implementations exploit the fact that, without RMA operations in between, such an empty access period can be safely optimized out.



(a) Cray XC40 (b) Taurus

Figure 4.5.: Average per-message latency using different active message queue implementations.

4.3.2. Average Message Latency

The first interesting metric in evaluating the performance of a message queue is the average latency of single messages exchanged between two processes. In these measurements, one process sends a fixed number empty messages to a target process that is continuously processing incoming messages. The results displayed in Figure 4.5 represent the average latency for a single message without a reply. Across all systems and implementations, the `sendrecv` queue exhibits the lowest latency with less than 1 μs, which is expected given the measurement presented in Section 4.3.1. On the Cray system Figure 4.5a, the RMA-based queues achieve 7.76 μs and 10.8 μs when using Open MPI, respectively. The `singlewin` queue yields lower latencies because the MPI implementation is able to combine the release of the window lock with the transfer of the message as long as the transfer completes at the target before the release of the lock. In contrast to that, the `sopnop` queue has to wait for the completion of the transfer before the deregistration as a writer. Unfortunately, MPI does not provide any means of ordering RMA operations other than by waiting for remote completion by issuing a `flush` in between two operations. The RMA-based queues yield significantly higher latencies with Cray MPICH, presumably because the writer stalls while the target processes messages due to the missing progress. Given the average latencies presented in Section 4.3.1, the 10.8 μs are the result of $1 \times 2.1\,\mu s$ (`get`)$+2 \times 2.2\,\mu s$ (`fetch-op`) $+1 \times 1.7\,\mu s$ (`put`) $+1 \times 2.2\,\mu s$ (`accumulate`) $+c$ with $c = 0.4\,\mu s$ a residual that may stem from sporadically encountering a full or processing queue.

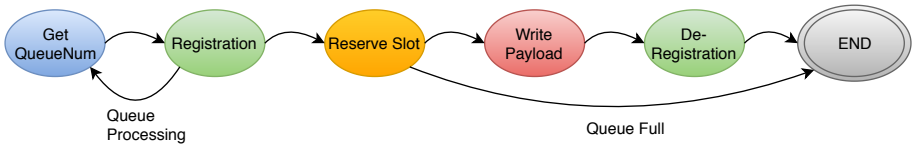


Figure 4.6.: State machine used to write a message into a RMA-based queue using the protocol discussed in Section 4.2.2.2.

On Taurus, a similar picture emerges: the RMA-based message queues perform best under Open MPI and worst under MPICH with progress thread enabled. The lower latency of the `sopnop` queue can be explained with an optimization that allows the final `accumulate` used for deregistration to be overlapped with the next operation, i.e., avoiding the latency of the final flush.[14]

For comparison, the measurements also include the latencies of active messages sent using GASnet. On both systems, the latency of GASnet is similar to that of the `sendrecv` queue.

4.3.2.1. Pipelining

Given the complex sequence of operations required to write a single message with the RMA-based queues and the significant latencies of individual RMA operations, it is desirable to pipeline the writing of messages to different target processes in order to reduce the overall latency of the a set of messages. To achieve this, a sequence of operations for one target has to be encapsulated in a structure that carries the state of the writing process and whose execution can be interleaved with the steps of other write operations to partially hide the latencies.

While it would be possible to encapsulate each write operation to a different target inside a task and schedule these tasks for execution by the worker threads using the task scheduler described earlier, this approach would have several drawbacks: yielding a task to hide communication latency introduces uncertainty about when the task will be rescheduled to continue executing the next step in the sequence. Moreover, the computational work within these tasks is quite limited and mostly consists of coordinating communication operations such as triggering the next operation based on the result of the previous operation.

MPI offers an abstraction for user-defined operations in the form of Generalized Requests (GREQs). Unfortunately, these GREQs are limited in their applicability. The original intent of this abstraction was to create a way for applications to track the progress of a separate execution thread without progressing that operation

[14]While this optimization was present on the Cray system as well, the Open MPI implementation does not seem to allow for overlapping of RMA operations on that system.

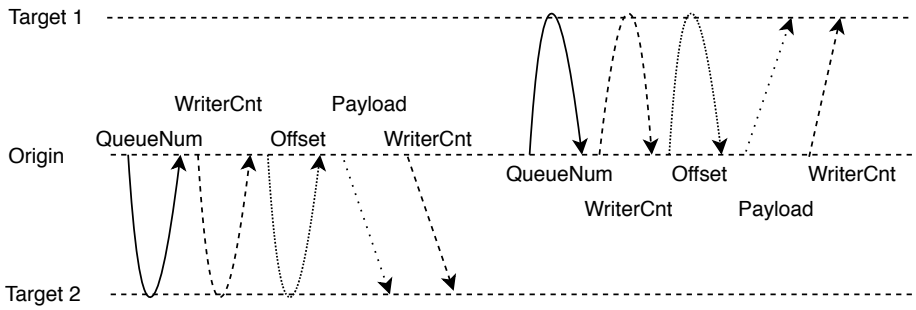
itself. Thus, the current standard only provides GREQs with a `wait` function that blocks until the operation completes and returns the status of that operation. An extension to this interface has been proposed as Extended Generalized Requests (EGREQs) in the context of asynchronous I/O processing [98] and are supported by the two major MPI implementation MPICH and Open MPI. These EGREQs have been used in the past in the processing of complex communication operations such as non-blocking collective operations [99]. In contrast to the standardized GREQs, an EGREQs includes a function used to poll the state of the operation and ensure progress, which allows for progressing the sequence of steps involved in the operation represented by the EGREQs. The MPI implementation thus acts as a simple (sequential) scheduler for stackless tasks [100].

The implementation encapsulates a simple state machine (as displayed in Figure 4.6) that is used to store the state of the operation between calls to the `poll` function of the EGREQs. The resulting (ideal) execution timeline is displayed in Figure 4.7. It is clear that the pipelined execution in Figure 4.7b has the potential to significantly reduce the time needed to write two messages to two different targets compared to the sequential approach depicted in Figure 4.7a. As highlighted in Figure 4.7b, the processing of the individual operations does not happen in lock-step but instead allows operations to proceed to the next step independently, e.g., in the case of delayed communication with one target or the need for retrying if a queue is currently being processed.

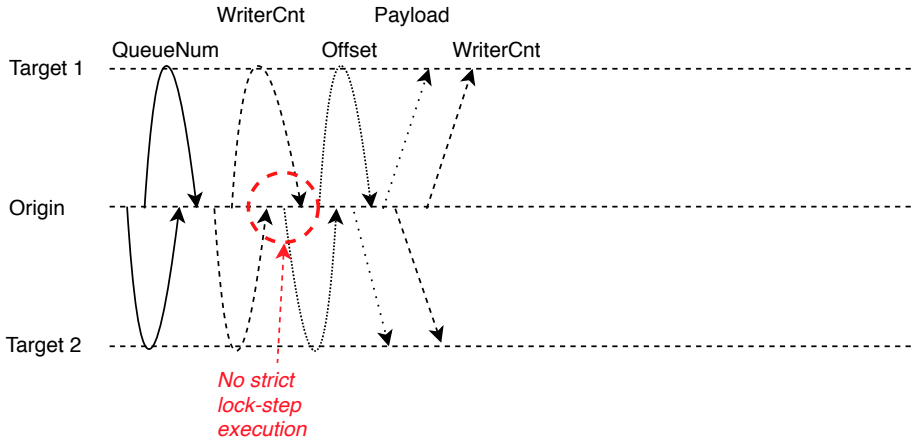
An issue that directly arises from this is that of remote completion of `put` operation in MPI. As discussed in Section 4.2.2.2, a strict ordering between the writing of the message and the deregistration as a writer is required to ensure consistent data for the reader. While MPI offers `put` operations with requests, the subsequent completion of the request only ensures local completion, requiring a `flush` to ensure completion of the operation in the target's memory. Thus, at least one step in the pipeline has a latency of a full round-trip in the network, as detailed in Section 4.3.1. Two possible changes to the MPI standard to mitigate this issue will be proposed in Section 4.4.

4.3.2.2. All-to-all Message Rate

In order to test the improvements in latency (and its inverse, message rate) when using the pipelined `sopnop` message queue implementation, a benchmark is used in which every process sends messages to every other process. The benchmark caches one message to each process and flushes all cached messages at once. This pattern is similar to the coalescing of messages in the inter-scheduler communication.



(a) Sequential execution.



(b) Pipelined execution.

Figure 4.7.: Sequence of operations involved in writing to two target message queues.

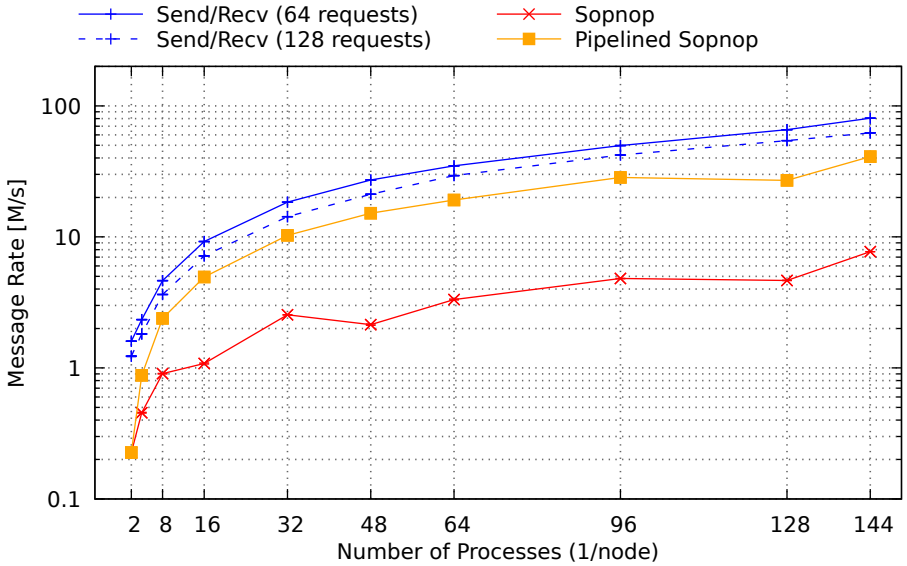


Figure 4.8.: All-to-all message rate of different message queue implementations on Taurus using Open MPI. Error bars represent the standard deviation of at least 50 repetitions.

Figure 4.8 shows the message rate of the `sendrecv` queue (with 64 and 128 send and receive requests each), the `sopnop` queue, and the pipelined `sopnop` queue on Taurus.[15] As expected, the difference between the `sendrecv` queue and the `sopnop` queues is most pronounced at two nodes, where the message rate is essentially the inverse of the latency presented in Section 4.3.2. However, with an increasing number of processes, the gap closes between the pipelined implementation and the `sendrecv` queue where the former yields around 50% of the message rate of the latter. It is interesting to note that even at scale, the `sendrecv` queue yields consistently higher throughput with 64 pre-posted requests than with 128 requests.

While the pipelined `sopnop` queue still does not yield performance that is fully en par with the `sendrecv` queues, further optimizations in its design may be possible based on the suggestions to be discussed in Section 4.4. It also remains to be seen to what degree this design can benefit from more up-to-date interconnects.[16]

4.3.3. Throughput at Scale

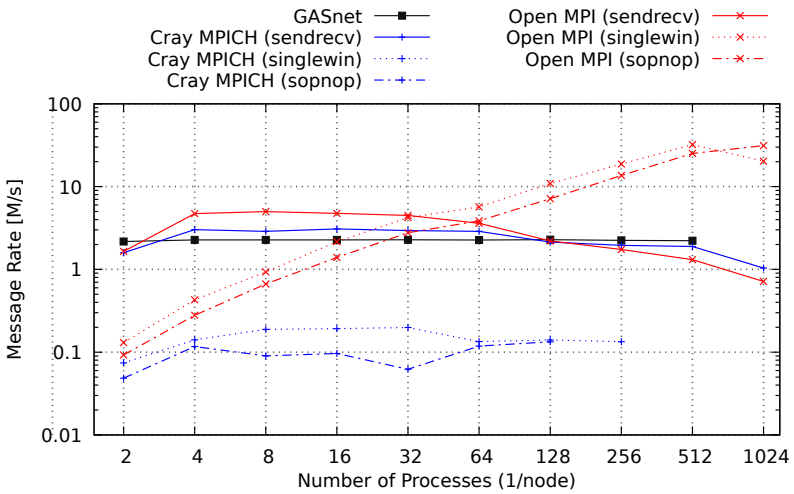
Another important performance aspect of active message queue implementations is the rate at which a single target can receive and process incoming messages. Especially in the context of distributed task schedulers, this is relevant since seemingly random communication patterns may create temporary contention at a single target.

Figure 4.9 shows the message rates achieved by $N-1$ writers sending empty messages to a single target that is continuously processing them. On the Cray system (Figure 4.9a), three groups of behavior can be observed. First, the `sendrecv` queues and GASnet exhibit a stable message rate between two and six million messages per second, with the MPI queues dropping off towards the end. Second, the RMA-based queues using Open MPI yield a continuously increasing message rate between 0.1 M messages with a single writer up to 30 M messages per second at 1023 writers, achieving almost an order of magnitude higher throughput than the send/recv-based queues. The approximately 90 M AMOs per seconds required for a sustained rate of 30 M messages per second with this design are close to the maximum of 120 M AMOs per second advertised by Cray [97].

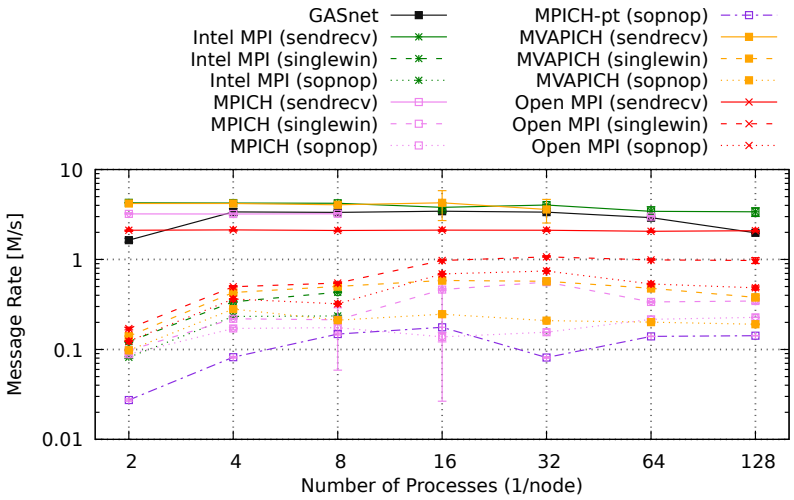
Finally, the RMA-based queues using Cray MPICH yield poor performance and hardly scale with the number of writers, a strong sign that software-based processing of RMA operations is not suitable for the use-case at hand.

[15] As of the time of this writing, the Open MPI implementation for Cray uGNI blocks *all* AMOs before returning, rendering any attempts of overlapping several communication operations futile.

[16] The Connect-IB interconnect available in the Taurus system has been released in 2013 [101].



(a) Cray XC40



(b) Taurus

Figure 4.9.: Message rate of message queue implementations with increasing number of writers to a single target. Error bars represent the standard deviation of at least ten repetitions.

On Taurus, the results are less favorable for the RMA-based queues where the `singlewin` queue with Open MPI yields the highest message rate of 1 M messages per second, trailed by the `sopnop` queue using Open MPI and the RMA queues using the other MPI implementations. The lowest message rate can be achieved using MPICH with the progress thread enabled, underscoring again that software-based progress is not suitable. The `sendrecv` queue and GASnet yield similar message rates as on the Cray system.

It is interesting to observe that the message rate of the `singlewin` queue using Open MPI reaches the plateau of 1 M at 15 writers, indicating some form of congestion. The 3 million AMOs per second needed for 1 M messages per second appear to be an upper bound of what the Connect-IB Network Interface Card (NIC) is able to deliver.[17]

4.4. Proposed Improvements to the MPI Standard

The following section discusses a set of extensions for the MPI standard that are aimed at addressing perceived short-comings of the RMA interface. These proposals are a direct outcome of the work on the RMA-based message queues discussed above.

4.4.1. RMA Operation Ordering

The MPI standard currently offers two ways for ordering RMA operations from the same origin to the same target: i) atomic operations to the same target memory location are executed in program order by default; and ii) waiting at the origin for completion of all previously issued operations at the target using a `flush` before issuing new operations. While the former has limited applicability, the latter incurs significant overheads by forcing the application to wait for operations to complete that otherwise may not require immediate completion. As was shown in Section 4.3.1, operations commonly require several microseconds even on modern high-performance networks. In some cases, two operations are not data-dependent but require ordering due to control flow constraints, e.g., they do not access the same memory location but nevertheless require ordering to ensure correctness. For example, the deregistration as a writer after writing the payload of a message as discussed in Section 4.2.2.2.

At the same time, many high-performance networks such Cray Aries [97] and Mellanox InfiniBand, network abstraction libraries such as UCX [102], and low-level PGAS abstractions such as OpenSHMEM [43] provide means to insert ordering requests into a stream of RMA operations, commonly called *fences*. The MPI standard should thus adopt a similar mechanism for ensuring ordering among RMA operations without waiting for their completion. A first extension[18] has thus been proposed to

[17] Unfortunately, Mellanox does not release public information on the RDMA capabilities of the InfiniBand NICs. The 3 M AMOs per second have been confirmed in separate measurements not shown here.

[18] MPI Forum issue #135: `https://github.com/mpi-forum/mpi-issues/issues/135`, last accessed December 8, 2019.

the MPI forum: the function `MPI_Win_order` would expose the relevant capabilities of the underlying network abstraction to the user. This extension to the MPI standard allows users to express their intent with regards to ordering semantics.

On platforms that do not support native RMA operation ordering, a simple implementation could fall-back to waiting for completion, leading to the same behavior as previously possible through flushes. However, it has been shown that even on systems that do not natively support fence capabilities the implementation may leverage other means to ensure ordering [103].

This topic is under active discussion within the MPI community and the author of this work strongly supports any efforts that drive the adoption of such a feature.

4.4.2. Fine-grained RMA Communication Contexts

As described earlier, thread-parallel writes to the same target suffer from inter-thread synchronization because flushes always happen on the level of the process. Thus, a thread issuing a flush may have to wait for operations from other threads as well. While past optimizations have improved MPI's internal usage of resources for thread-parallel RMA [75, 104], parallel threads issuing RMA operations may still not be independent due to the synchronization requirements.

As a possible mitigation, OpenSHMEM has adopted the concept of contexts [105], an abstraction of resources required for communication. It has been shown that this abstraction, while requiring additional coordination at the application level, may significantly improve thread-parallel access to the network [106]. While the integration of contexts into the MPI standard would be disruptive[19], the MPI community should consider the adoption of such a mechanism to improve thread-parallel RMA performance.

4.4.3. Request-based Flush

As described in Section 4.3.2.1, the completion of a `put` operation at the target is only guaranteed upon completing a `flush`, i.e., a blocking function call that returns only after all previously issued operations have completed. In a task-based execution context, the latency of waiting for a `put` to complete could easily be hidden by scheduling another task for execution in the meantime. However, MPI currently does not offer any *non-blocking* `flush` operations.

At the same time, the network abstraction layer UCX has introduced non-blocking flush operations [102]. Moreover, the low-level interfaces uGNI (Cray) and lib-

[19] The OpenSHMEM standard, for example, has duplicated all function signatures to include variants with and without context support.

fabrics support asynchronous notifications through counters or completion queues, which are polled to receive notification about completed operations [107, 108]. In part, these techniques are already used to implement request-based RMA operations such as `MPI_Raccumulate` and could thus be extended to support a request-based non-blocking flush, provided through the new functions `MPI_Rflush` and `MPI_Rflush_all`. The resulting request would complete as soon as all previously issued RMA operations have completed. Especially for larger transfers sizes—for which the latency is bound by the bandwidth of the network and thus typically exceeds the range of a few microseconds—and in the case of larger numbers of in-flight operations, request-based flushes may prove beneficial in hiding the resulting latency by overlapping computation with these active transfers. However, the evaluation of this interface is beyond the scope of this work and will be part of future work activities.

4.4.4. Callback-driven Request Completion

As part of the work on the pipelined `sopnop` message queue, an interface for callback-driven request completion notifications in MPI has been developed. The details of this interface are described in Appendix C together with an evaluation comparing its use against the message queue implementation using EGREQs and the Task-Aware MPI (TAMPI) variant of the application-level benchmark that will be discussed in Section 6.5.

In combination with the operation-ordering interface discussed in Section 4.4.1 and the request-based flushes proposed in Section 4.4.3, this asynchronous notification scheme provides a high degree of flexibility and allows tasks to insert a sequence of RMA operations before waiting for their completion by yielding the thread and resuming once all operations have completed. Overall, the adoption of the techniques outlined here would improve the usability of MPI-RMA in the context of task-based runtime systems and may lead to higher acceptance among users.

Chapter 5
Implementation

5.1. The DASH PGAS library

As introduced in Section 1.3, the DASH PGAS abstraction provides distributed data structures in the global address space. While attempts at including dynamic data structures have been made in the past, the main focus of the project is on fixed-size data structures constructed at run-time, including single elements (`dash::Shared`), arrays (`dash::Array`), and multi-dimensional arrays (dash::NArray). The distribution of the data in these containers can be controlled through user-provided patterns. While users may develop their own patterns, DASH provides a set of pre-defined patterns leading to *blocked*[20] and *tiled*[21] distributions. The details on the global and local iteration order and memory layouts are discussed in detail in [37].

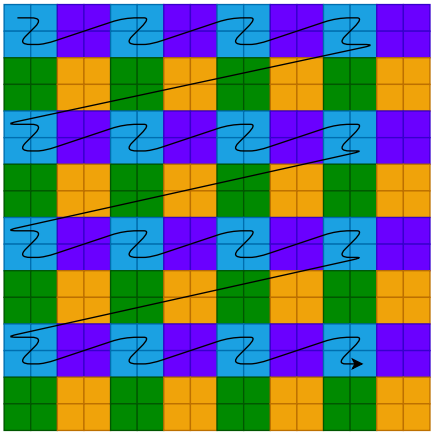
An example of a specification of a two-dimensional matrix can be found in Figure 5.1. The type `TiledMatrix` is constructed (Line 9) by specifying the value of the elements (Line 2), the number of dimensions, the type of the index used, and the pattern, which is a two-dimensional tiled pattern (Line 5). The distribution of the processes is done using a `TeamSpec`, which is constructed (Line 12) and balanced (Line 13) to achieve a two-dimensional distribution. The distributed matrix object `A` is constructed in Line 18 followed by an iteration over all local elements (Lines 21–24), as depicted by the black arrow in Figure 5.1b.

[20] *Blocked* distributions can be specified in arbitrary dimensions and causes a dynamically configurable number of elements to be consecutive in that dimension. The local iteration order remains unaffected by the block-size, i.e., elements in the lowest (or *fastest*) dimension are consecutive in memory regardless of the blocking in this or any other dimension.

[21] *Tiled* distribution affects all dimensions and changes the local iteration order such that each tile is stored consecutive in local memory.

```
// double-precision floating point
using value_t = double;
// select 2D tiled distribution
using PatternT =
        typename dash::TilePattern<2>;
// type of the 2D tiled matrix to use
using TiledMatrix =
     dash::NArray<value_t, 2,
       dash::default_index_t, PatternT>;


// specify 2D distribution of processes
dash::TeamSpec<2> ts(dash::size(), 1);
ts.balance_extents();

// Allocate the NxM matrix with tiles
// of size NBxMB
TiledMatrix A(N, M, dash::TILE(NB),
                    dash::TILE(MB), ts);


// local iteration over all elements
for (value_t* it = A.lbegin();
     it != A.lend(); ++it) {
...
}
```

(a) DASH example code. The parameters `N`, `M`, `NB`, and `MB` can be set at run-time.



(b) Distribution of a 16×16 matrix with 2×2 tiles on a 2×2 process grid. Processes are marked by colors. The local iteration order is depicted by a black arrow, showing that tiles are consecutive in memory.

Figure 5.1.: A tiled matrix with 2D-cyclic distribution in DASH.

An expression such as `dash::pointerof(A[0][0])` yields a *global pointer* to the first element in the matrix, which can be used to read from and write to this element or to iterate over a range of the global memory space in the same order as the local iterator. In contrast to this, a *global iterator* pointing to the first element can be acquired using `A.begin()` and used to iterate (and access elements) in the global iteration order, which in the case of a tiled matrix is following the individual tiles.

Accesses using both global pointer and global iterators map directly to calls into the underlying DASH RunTime (DART) and the communication backend such as MPI, as shows in the architecture depicted in Figure 5.2. In order to facilitate bulk transfers, DASH provides `dash::copy` that allows copying of arbitrary ranges from local memory to global memory, and vice versa. Moreover, dash provides a set of algorithms that are similar to their counterparts in the C++ standard template library (STL), which include `dash::reduce`, `dash::transform`, `dash::find`, and `dash::fill`.

As described in Chapter 2, this work extends the DASH interface by `dash::async` and `dash::taskloop`. The tasks instantiated by calls to these functions are handled by scheduler instances implemented in DASH RunTime (DART), as described below.

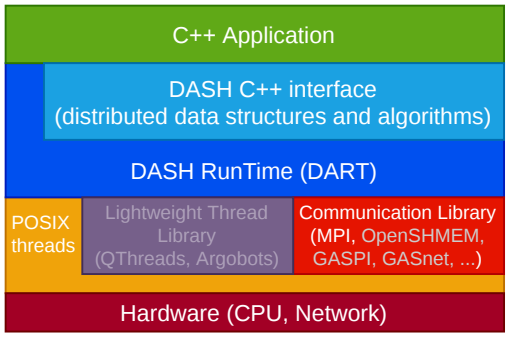


Figure 5.2.: The DASH Architecture: The application uses DASH data structures, which use DART functionality. The functionality in DART is implemented using the underlying communication library such as MPI (backends for additional communication libraries are work in progress at the time of this writing). The integration of lightweight thread libraries remains as future work.

It should be noted, however, that the approach proposed in this work is not limited to DASH. Rather, it is a general concept that can be implemented on top of any PGAS implementation that supports global pointers (UPC++ for example).

5.2. Scheduler Implementation

At the core of the scheduler is a pool of POSIX Threads (pthreads) that execute tasks that are available for execution. Although libraries such as QThreads and Argobots for handling user-level threads (ULTs) have been proposed in the past, the use of a custom scheduler implementation provided a high degree of control over the behavior of both the individual pthreads and the decision the scheduler takes. A further investigation and integration of these abstractions into the DASH runtime remains as future work.

Tasks that become runnable are enqueued into a two-level queue implementation that favors a fixed-size fast thread-local queue but falls back to a global queue if the local queue is full. Threads that have completed the execution of a task first try to dequeue a task from their local queue before looking at the global queue and eventually attempting to steal from other thread-local queues if the global queue is empty. Threads generally perform depth-first execution of the task graph, i.e., tasks that are released for execution upon completion of the current task will be favored as next tasks over tasks that already exist in the queue. This strategy is meant to improve cache locality by assuming that consecutive nodes in the task graph have a higher probability of re-using data of previous tasks. However, the scheduler also

support priorities that take precedence in the global queue and provide users with a way of indicating the urgency of a specific task over other tasks. The scheduler will automatically increase the priority of nodes in the task graph that have at least one outgoing edge cutting across the local process boundary, causing communicating tasks to be scheduled as early as possible in an attempt better hide communication latency.

5.2.1. Task Discovery Throttling

As described in Section 3.2, phases are used to reliably match remote dependencies. The transition into a new phase is done through a call to `dash::async_fence`, which simply increments the phase counter. In that same call, however, matching of task dependencies *may* occur: the user may specify the size of a window of active phases through an upper and a lower bound. When the upper bound is reached, matching is triggered and tasks will be executed until the lower bound is reached, at which point the discovery of the task graph is resumed by the application's main threads.

This throttling of tasks is a common technique in task-based schedulers as it avoids excessive resource usage by the scheduler. However, throttling based on the number of tasks is not possible due to the required matching and the constraints of the matching process, namely that all tasks of a phase have to be discovered before matching the respective phase. While it may be possible to dynamically adjust the size of the window by collectively determining a high-water mark on the number of tasks across processes, the granularity of phases has been sufficient so far to curb excessive resource usage.

5.2.2. Task-Yield

Tasks (or the ULTs executing them) in this implementation are non-preemptable, i.e., the execution may not be interrupted or aborted unless the task itself yields the underlying resources. However, the implementation fully supports tasks yielding the current thread to allow it to execute another—potentially higher-prioritized—task while waiting for an event to occur. Yielding is an instrumental aspect in the quest for hiding latencies that may occur for example due to I/O or communication activity.

The implementation supports *cyclic* task-yield, i.e., it guarantees that tasks are rescheduled after a yield and can be executed by other threads [109]. Hence, a ULT has to maintain the state of each task independently of the thread executing the task, i.e., by managing independent execution contexts. For sake of simplicity, the current implementation employs the POSIX `setcontext` family of functions to manage

ULTs [110]. As will be shown in Section 6.3.1, the overhead of context switching with this facility is in the order of a few thousand CPU cycles, an overhead that is certainly not negligible and which may be reduced by employing more advanced ULT implementations such as Qthreads [111] or Argobots [34].

5.2.3. Tasklets

The overhead of maintaining separate execution contexts may be prohibitively high for tasks that carry only a small workload. Abstraction such as Argobots thus offer so-called *tasklets* in addition to ULTs. Tasklets share their execution state with the state of the thread executing them and thus may not yield the thread to other tasks [34]. An implementation of tasklets has been included in DASH to explore their benefits. They can be used by replacing the call to `async` with `tasklet` and by using `taskletloop` instead of `taskloop`.

5.2.4. Blocked and Detached Tasks

Tasks that communicate typically initiate one or more communication operations and immediately wait for their completion. In order to avoid busy waiting through a test-yield-cycle—which may incur significant overheads due to the frequent context switches—a task may be *blocked* in the scheduler to wait for one or more communication operations to complete. The scheduler regularly checks the state of active transfers and upon completion will resume the execution of the task.

A task that issues a communication operation at the end of its execution, e.g., to write a result to global memory, is not required to resume execution after the transfer completes. In that case, the task may be *detached*, which causes the task to complete as normally with the exception that the release of its dependencies is deferred until all transfers have completed. Detached tasks are a concept taken from OpenMP [10, §3.5.1], with the difference that the integration of task scheduler and communication library allows for a simpler interface in which the user passes handles to active communication operations to the runtime directly instead of coordinating the release of dependencies manually.

Blocked and detached tasks would be another use case for the callback-based notification mechanism for requests in MPI that is proposed in Appendix C as the asynchronous notification could reduce the management overhead inside the scheduler.

5.2.5. Progress Thread

As described in Chapter 4, scheduler instances communicate dependency information through active messages. While the RMA-based queues may not require the participation of the CPU in the *reception* of messages, their *processing* still has to be done by a thread running on the CPU. Hence, worker threads regularly call into the message queue to process incoming messages. Alternatively, a *progress thread* can be enabled that will perform this task in a tight loop and handle all outgoing messages. If enabled, worker threads will enqueue their outgoing messages into a command queue that is processed by the progress thread, enabling a simple message coalescing scheme: all messages in the command queue will be buffered and eventually flushed out to the respective targets.

Moreover, the progress thread also handles outstanding communication operations of blocked or detached communication operations (Section 5.2.4) and releases tasks whose operations have completed. This task is interleaved with the handling of the inter-scheduler communication.

The use of a progress thread has proven advantageous in applications such as the tiled linear algebra algorithms discussed in Section 6.4 due to the shorter delays in inter-scheduler communication. However, for applications in which the schedulers only communicate infrequently, the overhead of scheduling an additional thread may cause more harm than good. Thus, the progress thread has to be enabled explicitly by the user.

5.2.6. NUMA-aware Scheduling

In contrast to static scheduling in OpenMP work-sharing constructs, the mapping of tasks onto threads is not known in advance. This may cause the computation on the same chunk of two consecutive `taskloop` statements to be executed on different Non-Uniform Memory Access (NUMA) nodes, potentially causing significant performance degradation.

The two-level work-stealing approach described on page 89 above can be extended to include NUMA aware scheduling. Instead of a single global queue, the scheduler maintains one global queue per NUMA node. Threads enqueuing runnable tasks put these tasks into the global queue assigned to the respective NUMA node to which the task has been assigned. This assignment currently is based on the first dependency encountered for that task but a more elaborate locality selection should be implemented in future versions.

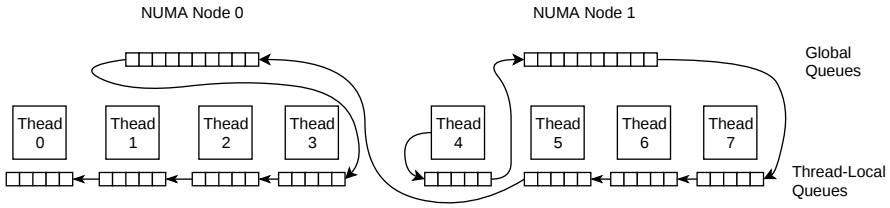


Figure 5.3.: NUMA-aware work-stealing algorithm. Arrows depict the order in which the different queues are traversed by Thread 4 to find a runnable task.

NUMA-aware work-stealing follows the scheme depicted in Figure 5.3: a thread first checks its local queue followed the NUMA node's global queue and the other threads on that NUMA node. If none of these queues contained a runnable task, the thread continues to query first the global queue on a neighboring NUMA node followed by the thread-local queues on that NUMA node.

NUMA-aware scheduling may incur additional overheads due to the additional book-keeping necessary. It is thus disabled by default and may be requested by the user.

5.2.7. Exception Handling

One of the core language features of C++ are exceptions, which can be considered objects that signal errors by traversing up the call-stack until caught. It is the application's responsibility to catch any exception that may occur during the lifetime of the application as uncaught exceptions will cause the application to terminate abnormally.

The same holds true on the level of tasks: any uncaught exception escaping a task is considered erroneous. The implementation will handle such a case correctly by informing the user about the escaping exception and properly terminating all processes in the application. More graceful ways of handling exceptions at the layer of DASH are possible but would require substantial improvements to DASH that are beyond the scope of this work. Among the most important areas for improvements to fault tolerance in DASH are the handling of distributed structure de-allocation in case of errors and proper broadcasting of locally occurring errors to other processes.

5.2.8. Cancellation

The implementation supports cancellation of tasks, both locally or collectively. Task cancellation is an important tool for handling application-level errors gracefully—such as dumping memory buffers when encountering an anomaly in a simulation—and to stop further task graph exploration upon meeting a termination criterion.

The implementation supports two modes of cancellation: collective cancellation and broadcast cancellation. *Collective cancellation* has to be triggered on all processes and will lead to the cancellation of all remaining tasks. Since tasks that are currently being executed are not forcefully interrupted, they will either run to completion or be aborted upon entering a *cancellation point*. Cancellation points in this implementation are any functions that call into the scheduler to change the state of the task, e.g., functions that create child tasks or yield the current thread. The process of collective cancellation includes a barrier to ensure that all running tasks are aborted and all remaining tasks are removed from the queues on all processes before returning.

The second supported mode—*broadcast cancellation*—is triggered by a single process that broadcasts a request for cancellation to all other processes, which then perform similar steps as in the case of collective cancellation.

5.2.9. Copyin Dependency

The `copyin` dependency described in Section 2.2 has been implemented by means of internal tasks that handle the data transfer and carry the appropriate dependencies for the source memory location. As soon as a previously unseen `copyin` dependency is encountered, the scheduler creates a new internal transfer task that will perform the data transfer and add an edge in the task graph between the user-defined task and the transfer task. Any later task carrying the same `copyin` dependency will be inserted into the task graph similarly but without creating a new transfer task.

The transfer itself can be performed as either a `get`, which is executed as soon as the input dependency of the task is resolved, or in terms of a send/recv pair. In the latter case, a task carrying the necessary input dependency information is injected at the remote process. This task will issue a send as soon as the input dependency is released. Locally, a task is created to received the data and release the `copyin` dependency. This scheme is especially useful on platforms that do not support RDMA or where traditional send/recv communication still outperforms RMA in terms of bandwidth or latency. By injecting the send-task at the source, a two-sided communication operation can be performed internally while maintaining a purely one-sided programming model. The user is not required to issue the send explicitly.

The scheme to be used is configurable by the user. In both cases, the transfer task is detached after starting the communication operation (as described in Section 5.2.4) to achieve low-overhead communication in the scheduler.

5.2.10. Tool Support

The availability of a tool infrastructure is critical for analyzing and debugging the behavior of applications developed using the task data dependencies presented in this work. In order to integrate different tools with the scheduler, a tools interface has been developed as part of a Master's thesis [112].[22] With the help of this interface, it is possible to generate information on executed tasks to be recorded in the Extrae tool, whose traces can be visualized using Paraver[23]. Moreover, the Temanejo task-debugger [57] can be attached to an application at run-time and used to visualize the global task graph as discovered by the scheduler as well as to step through its execution.

[22] The thesis was supervised by the author of this work.

[23] Extrae and Paraver are tools for recording and visualizing the behavior of parallel applications. They are available for download at `https://tools.bsc.es` (last visited March 23, 2020).

Chapter 6
Results

The evaluation of the proposed global task data dependency model will be done in several steps. First, basic measurements of task and dependency handling overheads are presented in Section 6.3. Following that, implementations of two numerical algorithms are discussed in Section 6.4. These are especially interesting for the evaluation of the ability to discover and process large distributed task graphs. A block-tridiagonal solver is then used to demonstrate the ability of the proposed approach to exploit the inherent concurrency available in algorithms used in CFD applications (Section 6.5). Last but not least, the proxy application LULESH is used to investigate the use of DASH Tasks on a more complex application using irregular meshes in Section 6.6.

6.1. Test Setup

All experiments have been performed on two systems: a *Cray XC40* installed at Höchstleistungsrechenzentrum Stuttgart (HLRS) in Stuttgart, Germany and a Bull Linux cluster called *Taurus* installed at the Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH) of the University of Technology Dresden in Dresden, Germany. The details of the configuration of both systems can be found in Appendix A. The details on the used software configurations can be found there as well and will be further specified in the respective sections.

6.2. A Note on the State of the Software Ecosystem

Over the course of this work, numerous issues in various third-party software packages have been discovered, reported, and in some cases fixed by the author.

The bulk of issues can be classified as either issues with MPI-RMA implementations, the safety of multi-threaded MPI usage, or both. Even at the time of this writing, no MPI implementation provides full hardware support in a fully stable state.

Modern MPI implementations are a vastly complex software system consisting of tens of thousands lines of code and there is no single person or organization to blame for the issues that arose during this work. The same holds true for other software packages such as compilers, performance analysis tools, hardware drivers, and various runtime systems, all of which have been found to be unstable or functionally incorrect at some point during this work.

The value of open source software cannot be overstated in this context. While most experiments have been performed on a Cray system, for example, almost all experiments on that system have used Open MPI instead of the vendor-provided implementation. This can be attributed to the fact that reporting a potentially complex issue against a closed-source software package requires significant resources in an attempt to reverse engineer a representative example, which may simply become a futile exercise. As an open source project, Open MPI has the advantage of allowing the use of debuggers for analysis and the submission of patches to fix issues that were found during experiments.

While even to this date, there appear to be issues in all major MPI implementations that directly affect the work presented here, it has at least been possible to create stable and well-performing setups using Open MPI. Even though the situation seems to be improving overall, the usage of truly multi-threaded MPI is still a fragile exercise. However, the author believes that continued and rigorous testing will eventually lead to a stable state of the major MPI implementations.

It will also become clear that performance measurements are merely a snapshot of the current state of a software package and its supporting ecosystem. The numbers reported here should thus not be taken as absolute values but rather be seen as an attempt to put the results of this work into perspective.

6.3. Task Micro-Benchmarks

In order to assess the performance of the implementation of the proposed global task synchronization scheme it is important to understand the fundamental overhead involved in managing tasks and their dependencies.

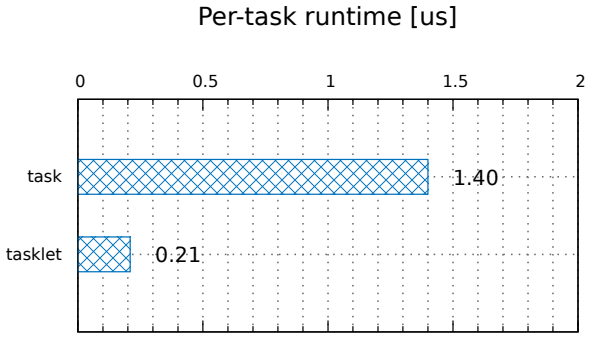


Figure 6.1.: Overhead of handling the creation and execution of tasks and tasklets.

6.3.1. Task Management Overhead

In a first benchmark, the creation and execution of tasks and tasklets without dependencies has been measured. The results depicted in Figure 6.1 are the mean time per task or tasklet that is required to create and execute 100,000 tasks on the Cray system, all created and executed by a single thread. All benchmarks were repeated five times and the standard deviation has been found below 0.1. As is to be expected, using tasklets instead of tasks incurs significantly lower overheads, by almost a factor of seven.

A tasklet incurs an overhead of 0.21 μs, which includes the allocation of approximately 256 B on the heap used for storing the task information and the invocation of the task action *on the stack of the thread*. In separate measurements, the tasklet handling has been determined to require approximately 870 instructions and 700 cycles on the Cray system. In contrast, the 1.4 μs for handling a single task involve additional costs. Most significantly, the POSIX functionality `setcontext` [110] used to enter a distinct context for each task saves and restores all necessary registers and the signal masks, the latter of which appears to incur a system call on Linux systems. Around half of the approximate 4,000 cycles are spent in kernel space. Hence, not all of the additional overhead of tasks can be attributed to the system call. While setting the signal mask is not strictly required it may be helpful for applications using custom signal handlers. Other studies have confirmed the additional costs of handling separate task execution contexts, even without the signal management overhead [34].

From these measurements, a certain minimum work inside a task required to amortize the overhead of task management can be derived. For tasklets, achieving a 10% overhead requires a workload of at least 7,000 cycles while for tasks this number is around 40,000 cycles.

6.3.2. Dependency Management Overhead

In order to determine the overhead involved with managing the dependencies between tasks, a second benchmark is used, in which the number of dependencies is increased from one to 32 in powers of two. The data presented reflects a similar setup as in the previous experiment.

Three patterns of dependencies have been chosen for these benchmarks: for the local dependencies, each tasklet carries the specific number of output dependencies, each referencing a distinct memory address. Since all tasklets use the same output dependencies, their execution is serialized due to the scheduler's treatment of write-after-write dependencies. Since the benchmark is executed by a single thread, the numbers presented in Figure 6.2 represent all overhead incurred, including the fundamental handling of tasklets. The overhead was found to be between 0.35 μs for a tasklet with a single output dependencies up to 5.8 μs for a tasklet with 32 output dependencies. Given the 0.2 μs fundamental tasklet overhead presented above, the local latency handling exhibits an almost linear increase in handling time over the range presented here. Moreover, the numbers are close to the overheads presented for different production-level OpenMP compilers presented in [113].

In order to assess remote dependencies, two benchmarks have been used: in the `remote-single` benchmark, all but the first process create and eventually execute tasklets with the respective number of input dependencies referencing global memory locations at the first process. This benchmark reflects the overall cost incurred by sending the remote dependency request to the first process, which then tries to match the dependency and sends back a release. Since there are no output dependencies in this benchmark, the release occurs immediately. In the absence of both a progress thread and coalescing of release messages disabled, the latency was 3 μs for a single dependency and again an almost linear increase to 84 μs for 32 dependencies. Surprisingly, the presence of a progress thread does not significantly lower the latency.

However, enabling coalescing of release messages reduces the mean tasklet overhead to 0.5 μs for a single dependency and 12 μs for 32 dependencies. In this case, the coalescing amortizes the active message queue communication. Interestingly, the addition of a progress thread handling the communication incurs additional overheads, presumably due to the required communication between the two threads through a lock-free data structure and the resulting higher pressure on shared CPU features and memory, even though the progress thread was bound to a separate CPU core. It should be noted, however, that the benchmark presented here is an extreme case, in which the main thread does not perform any useful work. If on

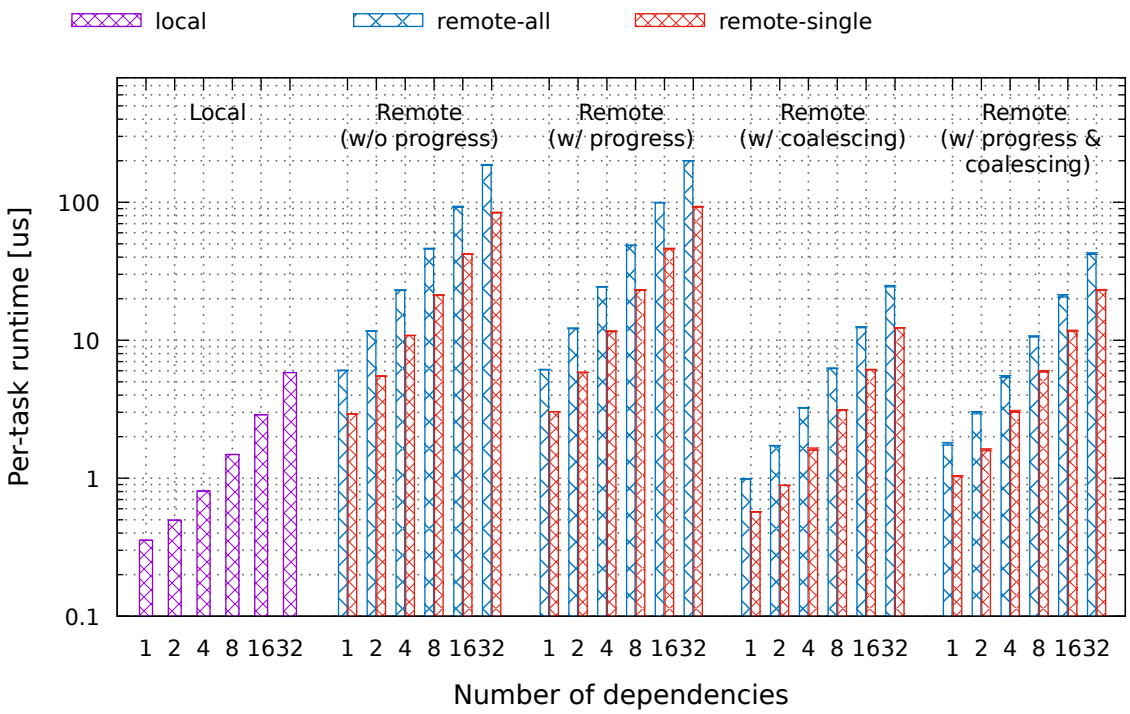


Figure 6.2.: Local and remote dependency latencies on a Cray XC40. The `local` benchmark was conducted by one process with a single thread on an exclusive node. The remote benchmarks involved 16 processes running on distinct nodes. For the `remote-all` benchmark, all processes create tasklets with input dependencies referencing global memory locations on all other processes. For the `remote-single` benchmark, all but the first process create tasklets with input dependencies referencing memory locations at the first process. The standard deviation is included in the form of error bars.

the other hand, the main thread does perform computation inside the tasks, the short response time provided by the progress thread amortizes the cost of additional resources. This is especially true for applications with large amounts of remote dependencies such as the tiled linear algebra algorithms that will be discussed in Section 6.4.

A similar picture emerges for the `remote-all` benchmark, in which all processes create tasklets with dependencies that reference global memory locations at all or a subset of all other processes, selected in a round-robin fashion. In this benchmark, all processes are required to create tasks and handle incoming remote dependencies. The resulting overheads lead to an increase in overheads by approximately a factor of two. This benchmark is closer to the reality of distributed applications developed using the proposed task synchronization, where all processes create tasks or tasklets with local and remote dependencies. With coalescing enabled, the mean overheads

range from 1 μs for a single dependency to up to 24 μs for tasklets with 32 remote input dependencies. However, these levels are still within the range of experimental tasking implementations such as OmpSs [113].

6.4. Tiled Linear Algebra Algorithms

Many problems in computational scientific and engineering applications can be expressed as systems of linear equations of the form $Ax = b$, which are used to represent networks and graphs or to model real-world phenomena. Using a set of well-known decomposition algorithms (e.g., LU, QR, Cholesky), the matrix A can be decomposed into products of two (or more) matrices that can be used to numerically solve these systems. The growing size of these matrices requires increasing computational capacities, which in turn mandates the parallelization of these algorithms.

Software projects such as ScaLAPACK have long provided distributed parallel implementations of such algorithms [114]. The current state of the art, however, lies in the use of *tiled algorithms*, where the matrix A is stored in memory as a set of tiles and lower-level mathematical operations (commonly referred to as Basic Linear Algebra Subprograms (BLAS) level 2 and level 3 [115]) are applied to these tiles in a well-defined order [59]. The order in which BLAS operations are applied onto the tiles can be expressed in the form of DAGs, enabling the use of task-schedulers to coordinate their execution. In addition, distributed runtime systems such as PaRSEC handle the communication of tiles across distributed memory domains where necessary [29].

In this section, two representative algorithms used for matrix decomposition will be discussed: Tiled Cholesky Decomposition and Tiled QR Decomposition. The distribution of large matrices in the global address space is at the heart of the DASH software package (see Section 5.1) and the implementation of tiled algorithms on top of DASH data structures may benefit from the PGAS model and inherent one-sided communication. Moreover, the distributed task discovery inherent to the model proposed in this work may prove advantageous in the scalable discovery of DAGs required to handle large matrices.

6.4.1. State of the Art Implementations

The implementation of both the Cholesky and QR algorithms using the task model proposed in this work will be compared to some of the state of the art software packages: PaRSEC and Chameleon.

6.4.1.1. PaRSEC

The PaRSEC programming system provides an abstraction for efficiently expressing distributed algorithms as DAGs encoded as Parameterized Task Graph (PTG) through the use of a data-flow language called JDF [29]. Algorithms are expressed on a high level, leaving out details such as data transfer and explicit concurrency management. Instead, the data-flow between nodes in the DAG is left to the runtime system to coordinate both execution and communication. An algorithm written in JDF will be translated into low-level languages such as C before being compiled into machine code. The Parameterized Task Graph (PTG) is encoded directly into the resulting binary, requiring no up-front task graph discovery and dependency matching at run-time. Instead, the completion of a task triggers the discovery of the next set of tasks with both local and remote dependency information immediately available. PaRSEC PTG has been the basis for the implementation of Distributed Parallel Linear Algebra Software for Multicore Architectures (DPLASMA) [116].

However, porting applications to the JDF language requires a complete rewrite of the high level algorithm.[24] In order to lower that barrier, an interface for Dynamic Task Discovery (DTD) on top of the PaRSEC runtime system has been proposed, with which the *global* DAG can be expressed using a C interface [51]. As the name suggests, the discovery happens at run-time, with all processes discovering the global task graph. Affinity tags are provided by the user to determine the process eventually executing a task. With this scheme, processes know the tasks they will execute as well as their task's positions in the global task graph. Thus, each process can infer the local and remote dependencies on the fly, without extra communication.

However, the global task graph discovery has two major drawbacks: i) all processes have to discover the full task graph, thus diverting from traditional SPMD programming style; and ii) the full discovery of the global task graph may become a scalability bottleneck in weak-scaling scenarios.

For both PTG and Dynamic Task Discovery (DTD), the runtime system performs the communication of tiles and scheduling information using two-sided MPI operations. For all experiments, git version `64574e9a9d0d` has been used from the PaRSEC repository[25].

6.4.1.2. Chameleon/StarPU

Chameleon is a dense linear-algebra library designed on top of the StarPU distributed and heterogeneous tasking system [31, 117, 67]. Similar to PaRSEC DTD, the high-

[24] PaRSEC allows tasks to call C functions that implement the actual logic of a task.

[25] `https://bitbucket.org/icldistcomp/parsec`, last visited March 22, 2020.

Algorithm 6.1: Tiled Cholesky Decomposition (after [116]).

```
1 for k ← 0 to NTILES − 1 do
2     A[k][k] ← POTRF(A[k][k]);
3     for m ← k + 1 to NTILES − 1 do
4         A[m][k] ← TRSM(A[k][k], A[m][k]);
5     for n ← k + 1 to NTILES − 1 do
6         A[n][n] ← SYRK(A[n][k], A[n][n]);
7         for m ← n + 1 to NTILES − 1 do
8             A[m][n] ← GEMM(A[n][k], A[m][k], A[m][n]);
```

level tile-based algorithms are expressed using a global task graph discovery scheme. However, Chameleon supports tracking and moving of the ownership of tiles, which allows the runtime system to trim the task graph on the fly to discard any information on tasks that are irrelevant for the tasks owned by the specific process. Thus, even though the discovery is global (the algorithm is expressed with a global view) each process only stores and manages the local portion plus information on remote dependencies. The resulting information stored at each process is similar to the extended local task graph each process possesses in the tasking model proposed in this work (see Section 3.2).

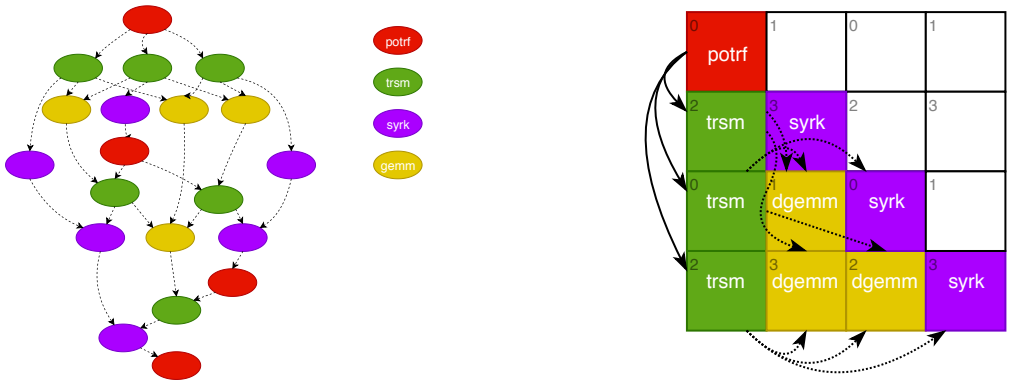
Similar to PaRSEC, MPI is used for communication of tiles and scheduling information. The used version of Chameleon was 0.9.2[26] with StarPU at version 1.2.9 [27].

6.4.2. Tiled Cholesky Decomposition

The Cholesky decomposition factorizes a Hermitian (or symmetric) positive-definite matrix A into a lower-triangular matrix L, such that the product of that matrix with its conjugate transpose yields A, i.e., $A = LL^*$. The tiled version of this algorithm employs four BLAS routines: POTRF, TRSM, SYRK, and GEMM, as depicted in Algorithm 6.1. In each iteration, the block on the diagonal ($A[k][k]$) is factorized using a non-tiled Cholesky decomposition (POTRF) and the result is applied to factorize all tiles in the same column of the trailing part of the matrix (TRSM). The result, in turn, is then multiplied with the tiles of the trailing lower triangular matrix (GEMM and SYRK). The complexity of the tiled Cholesky decomposition is in $\mathcal{O}(\frac{1}{3}n^3)$, which is

[26] Available online at `https://gitlab.inria.fr/solverstack/chameleon/-/releases`, last visited March 8, 2020.

[27] Available online at `http://starpu.gforge.inria.fr/files/`, last visited March 8, 2020.



(a) Task graph.

(b) Flow of tiles in iteration $k = 0$.

Figure 6.3.: Tiled Cholesky Decomposition of a matrix with 4×4 tiles. Red nodes/blocks depict POTRF, green TRSM, purple SYRK, and yellow GEMM operations. The gray numbers indicate the process owning the tile in a 2D-cyclic tile distribution across 4 processes. The white tiles are not used in the algorithm.

a significant advantage over QR and LU decomposition for matrices that meet the outlined criteria.

In a distributed implementation, each operation requires at most two tiles as input and produces at most one tile as output. With the exception of POTRF, operations might require input tiles to be fetched from other processes, depending on the data distribution and the position of the tile. The fact that each operation only produces one output tile allows the mapping of the task to follow the distribution of the output tiles. The process owning the output tile (and thus at least one of the input tiles) will execute the task.

The task graph of this algorithm is depicted in Figure 6.3a for a matrix with 4×4 tiles, depicting the typical fork-join pattern with sharply increasing concurrency at the top and slowly decreasing concurrency towards the bottom. In a similar fashion, Figure 6.3b depicts the flow of blocks needed as input for operations in the first iteration of the algorithm.

From the algorithm depicted in Algorithm 6.1, the number of communication operations can be determined to be in $\mathcal{O}(n^2)$: in each iteration of the outermost loop, the block on the diagonal and the blocks in the column below the diagonal have to be communicated from the processes owning them to the processes requiring them as input to SYRK and GEMM.

A simplified implementation of this algorithm using the tasking model proposed in this work was presented in Listing 2.5 and discussed in the ensuing section.

6.4.2.1. Experimental Setup

The following experiments were conducted as weak-scaling experiments with a fixed per-process matrix size. The number of processes was scaled from 1 to 144, choosing square numbers. Each process was mapped to a distinct compute node. In addition, the size of the tiles has been chosen as 320^2, 256^2, and 192^2. A larger tile size means more work in the underlying BLAS routines, potentially improving the efficiency of these algorithms on the given hardware[28]. However, smaller tile sizes are interesting to consider in the context of this work because they require a higher number of tasks at the same global matrix size and thus put more pressure on the tasking subsystem, including the communication of tiles.

Unfortunately, the DASH tiled distributed matrix does not support under-filled tiles, which are tiles at the boundary that are smaller in either direction due to the matrix size not being a multiple of the tile size. Thus, the matrix sizes have been chosen to accommodate the tile sizes: for tiles of size 320^2, a per-process matrix size of $25,600^2$ (or $25k^2$) elements was selected while for tile sizes of 256^2 and 192^2 a per-process matrix size of $24,576^2$ (or $24k^2$) was selected. While the support for underfilled tiles in DASH would be desirable to work towards a full-featured linear algebra library based on DASH, their implementation was beyond the scope of this work.

For the two extreme ends on this scale (192^2 and 320^2), the number of tasks are plotted in Figure 6.4. The smaller tile size configuration increases the total number of tasks by a factor of four, leading to a total of approximately 605 million tasks for $NB = 192^2$ compared to 147 M tasks for $NB = 320^2$ at 144 processes.

A cap of ten minutes was imposed on the run-time of each of the benchmarks to avoid excessive CPU usage. Thus, missing data points mean that the benchmark did not complete due to either crashing or reaching this limit. For reference, the run-time of PaRSEC PTG with a tile size of 320^2 took 100 seconds at 144 nodes. The cut-off was thus 6 times higher than this reference, or roughly 110 Gflop/s at 144 nodes.

6.4.2.2. Results: Cray XC40

The results for the three different tile sizes on the Cray XC40 are depicted in Figure 6.5. The performance is given as *per-node performance*, i.e., the overall performance divided by the number of nodes. Ideal scaling thus would be reflected by a constant per-node performance across the range of nodes. Both PaRSEC and the

[28] In fact, a tile size of 320^2 appeared to yield the best single-thread performance on the used Intel Haswell processor.

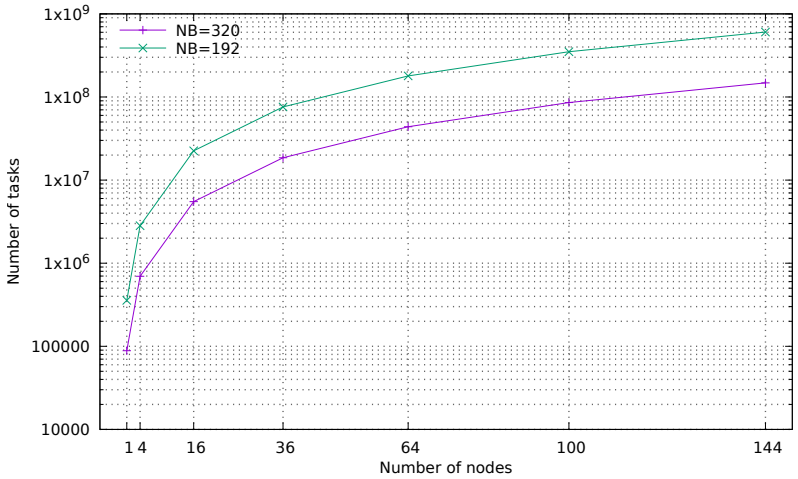


Figure 6.4.: Number of tasks in a Tiled Cholesky Decomposition for $NB = 320^2$ and $NB = 192^2$ depending on the number of nodes used.

StarPU variants were run using Open MPI and Cray MPICH, which yield significantly different results in some cases. The DASH Tasks were run only using Open MPI due to software stability issues in the vendor-provided implementation (see Section 6.2).

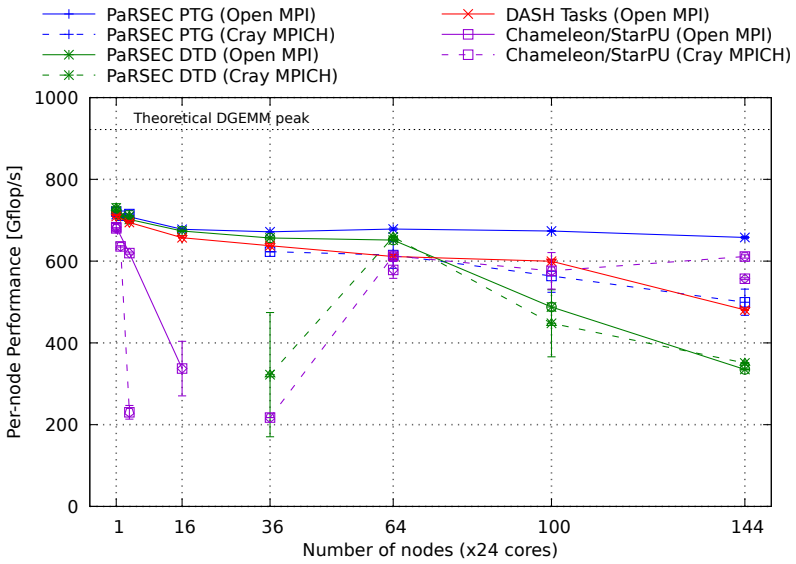
The best performance is delivered by PaRSEC at a tile size of 320^2 (Figure 6.5a) when running with Open MPI, reaching a peak per-node performance of 720 Gflop/s on a single node and 660 Gflop/s on 144 nodes. For smaller node numbers, PaRSEC PTG is trailed by PaRSEC DTD and the DASH Tasks. At 64 nodes, the DASH Tasks achieve 90% of the performance of PaRSEC PTG while at 144 nodes that number drops to 75% (and only slightly less than PaRSEC PTG running with Cray MPICH). A drop in per-node performance is visible for DASH after 100 nodes.

At the same time, the performance of PaRSEC DTD drops off sharply after 64 nodes, reaching only 50% of the performance of PaRSEC PTG at 144 nodes. However, at 64 nodes DTD still outperforms the DASH Tasks version of the benchmark.

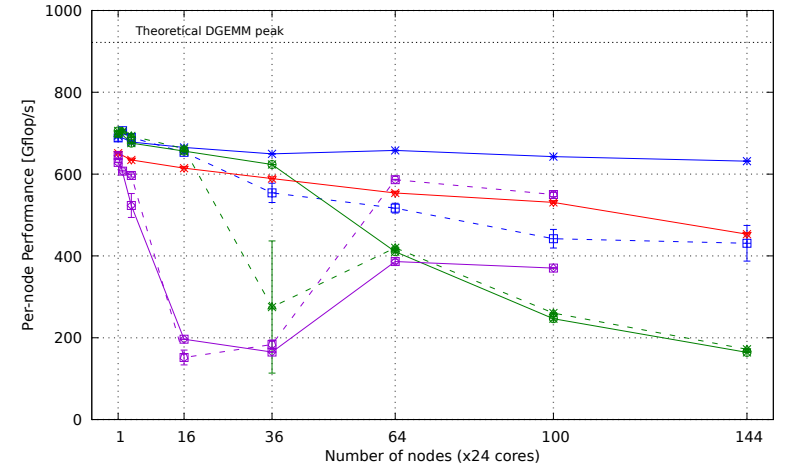
Unfortunately, the numbers for Chameleon/StarPU are inconsistent, with missing data points at 16, 36, and 100 nodes. However, at 144 nodes Chameleon still achieves 92% of the performance of PaRSEC PTG (using Cray MPICH).

At a tile size of 256^2 (Figure 6.5b), the DASH Tasks performance follows a slight downward trend that again ends up at 75% of the PaRSEC performance at 144 nodes. The drop for DTD begins earlier, leading to only 25% of the PTG performance at 144 nodes. Again, the Chameleon performance is unstable, with a significant drop at 16 nodes and a slight rebound at 64 and 100 nodes.

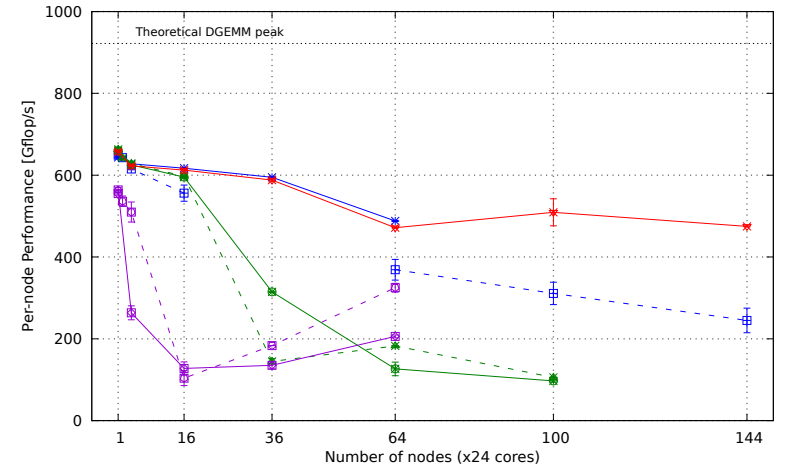
With a tile size of 192^2 (Figure 6.5c), the scaling of PaRSEC is impaired, with the Open MPI run not finishing within the time limit and the Cray MPICH runs dropping to 220 Gflop/s at 144 nodes. The DASH Tasks variant drops to 470 Gflop/s



(a) N=$25k^2$/node, NB=320^2

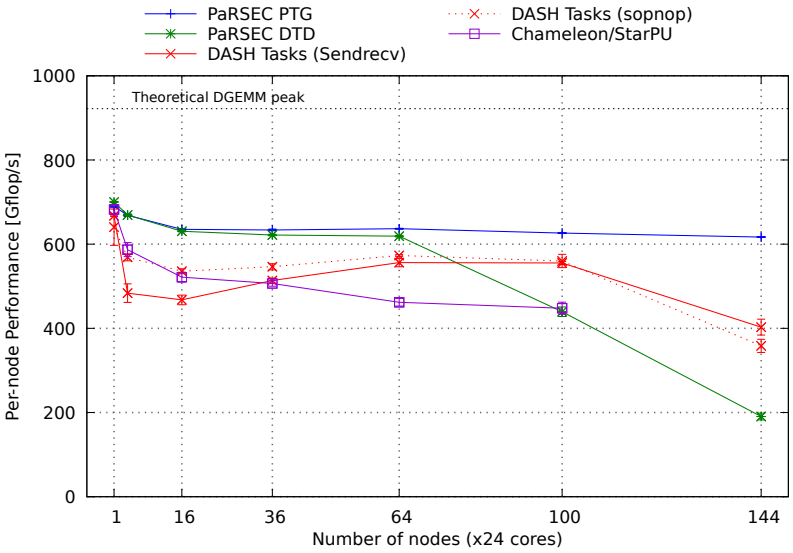


(b) N=$24k^2$/node, NB=256^2

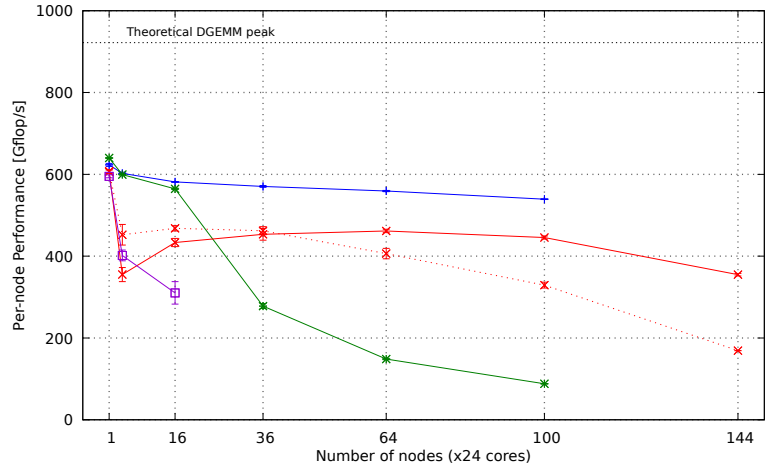


(c) N=$24k^2$/node, NB=192^2

Figure 6.5.: Performance of Tiled Cholesky Decomposition on the Cray XC40 in different configurations. Error bars represent the standard deviation of at least five runs. Missing data points represent runs that did not finish in under 10 minutes.



(a) N=$25k^2$/node, NB=320^2



(b) N=$24k^2$/node, NB=192^2

Figure 6.6.: Performance of Tiled Cholesky Decomposition on the Taurus cluster in different configurations. Error bars represent the standard deviation of at least five runs. Missing data points represent runs that did not finish in under 10 minutes.

at 64 nodes but remains mostly stable up to 144 nodes. Both PaRSEC/ DTD and Chameleon drop off sharply after 16 nodes, with at least Chameleon making a short rebound at 64 nodes.

6.4.2.3. Results: Taurus

On the Taurus Linux cluster, a mostly similar pattern emerges, as depicted in Figure 6.6. For both tile sizes of 320^2 and 192^2, PaRSEC PTG yields the best performance across the range of nodes while the performance of PaRSEC DTD again

drops after 64 and 16 nodes, respectively. An exception are the runs with a tile size of 192^2 at 144 nodes, where both PaRSEC PTG and DTD did not complete.

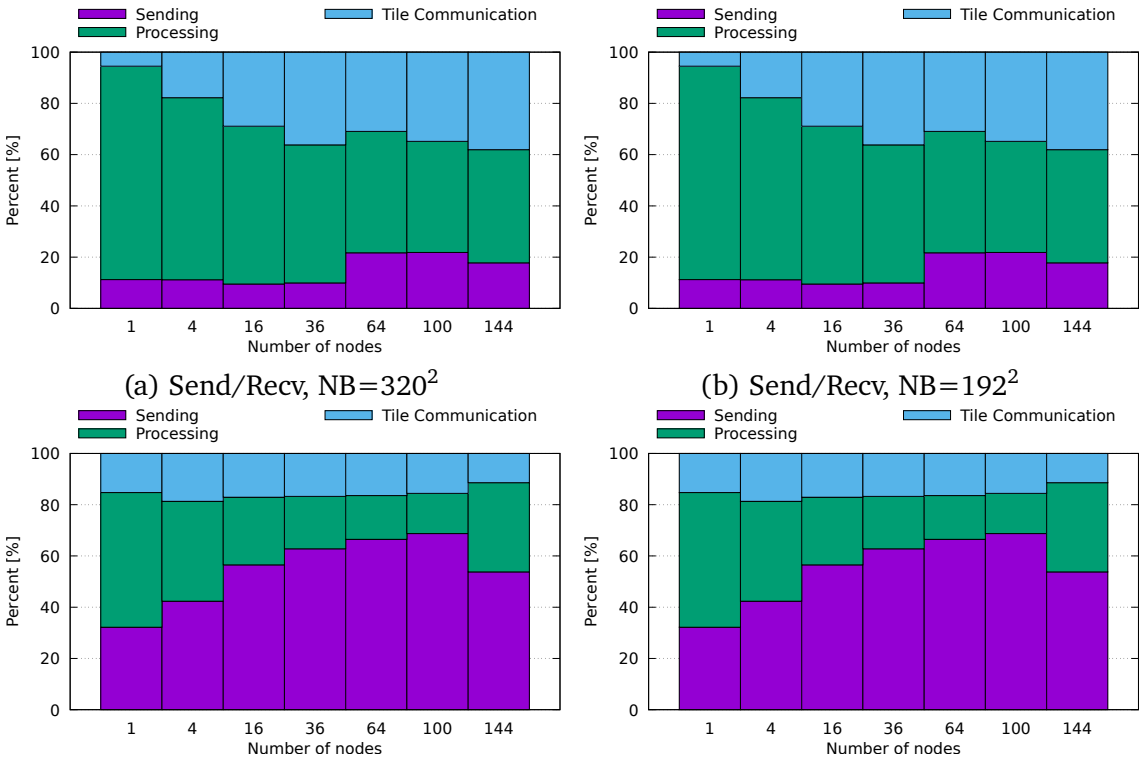
Chameleon yields a more stable performance, albeit lower than PaRSEC and DASH, until 100 nodes for 320^2 tiles. At 144 nodes as well as after 16 nodes for tiles of size 192^2 the runs fail with errors from the UCX library indicating resource exhaustion. It is beyond the scope of this work to investigate these issues.

For the DASH tasks variant, two different configurations are depicted in Figure 6.6: one using the send/recv-based queue and one using the `sopnop` queue, both of which have been discussed in Section 4.2. For both tile sizes, the `sopnop`-based queue yields slightly higher performance at smaller node numbers while the send/recv-based queues yield better performance at larger scales. The latter effect is in accordance with expectations derived from the observations in Section 4.3: the higher latency of sending (coalesced) messages to other processes and the limited AMO rate become a bottleneck with the `sopnop` queue for larger process numbers.

This effect is depicted in Figure 6.7, where the percentage of time spent by the progress thread on different operations is plotted. Here *Sending* represents the percentage of time spent on sending messages, *Processing* the time spent on receiving and processing messages, and *Tile Communication* the time spent on completing tile transfers triggered as part of the `copyin` dependency. For the `sopnop` queue, the sending of messages becomes dominant with an increasing number of nodes, going up to 70% of time spent for sending messages. When using the send/recv queue on the other hand, the progress thread spends 70–80% of the time on processing (and receiving) of messages for tile sizes of 320^2. It is interesting to note that the testing for completion of tile communication on the send/recv queue becomes more significant with increasing numbers of nodes, reaching almost 40% for tile sizes of 192^2. Although the transfer is done using RMA operations, testing for completion will likely trigger the MPI implementation's progress engine and thus progress the send/recv operations of the message queue.[29]

This analysis indicates that on this system, the scheduler interaction may benefit from a message queue that yields low latencies but does not interfere with other MPI operations. In Section 4.3.2.1, a message queue has been proposed that has the potential to yield sufficiently low latencies at scale while not requiring send/recv operations that transcend into other MPI operations. Unfortunately, the current

[29] As per [7, §11.7.3], the MPI standard mandates that processes progress *any* communication operation they are involved in.



(a) Send/Recv, NB=320^2

(b) Send/Recv, NB=192^2

(c) Sopnop, NB=320^2

(d) Sopnop, NB=192^2

Figure 6.7.: Percentage of time spent on operations by the progress thread when executing Tiled Cholesky Decomposition using different active message queue implementations.

software environment appears to be insufficient to test this hypothesis. [30] However, these results motivate further work in this direction.

6.4.3. Tiled QR Decomposition

The tiled QR decomposition takes as input a tiled $M \times N$ matrix A and factorizes it into an orthogonal $M \times M$ matrix Q and an upper-triangular $M \times N$ matrix R such that $A = QR$ [118]. The algorithm of the tiled QR decomposition is presented in Algorithm 6.2. The most notable difference compared to the tiled Cholesky decomposition is the fact that some operations (namely TSQRT and SSMQR) produce more than one output tile. In these cases it is not guaranteed that all output tiles are owned by the same process. Thus, either the programmer or the runtime system has to chose where this computation is to be performed. In the case of SSMQR,

[30] A preliminary investigation has indicated that the integration of UCX causes Open MPI to become non-reentrant in some circumstances specific to multi-threaded usage of the queues. Thus, a deadlock occurs if the EGREQ progress callback calls back into MPI.

Algorithm 6.2: Tiled QR Decomposition (after [116]).

```
1 for k ← 0 to NTILES − 1 do
2     A[k][k], A[k][k] ← GEQRT(A[k][k]);
3     for m ← k + 1 to NTILES − 1 do
4         A[k][k], A[m][k], T[m][k] ← TSQRT(A[k][k], A[m][k], T[m][k]);
5     for n ← k + 1 to NTILES − 1 do
6         A[k][n] ← ORMQR(A[k][k], T[k][k], A[k][n]);
7         for m ← k + 1 to NTILES − 1 do
8             A[k][n], A[m][n] ← SSMQR(A[m][k], T[m][k], A[k][n],
                A[m][n]);
```

it is interesting to observe that the tile $A[k][n]$ is updated in each iteration of the inner-most loop, effectively serializing all SSMQR operations inside that loop.

The complexity of tiled QR decomposition is in $\mathcal{O}(\frac{4}{3}n^3)$, significantly higher than the tiled Cholesky decomposition. In distributed memory, the number of communication operations of this algorithm has been determined to be in $\mathcal{O}(n^3)$ [119]. This constitutes a significantly higher volume compared to the Cholesky decomposition.

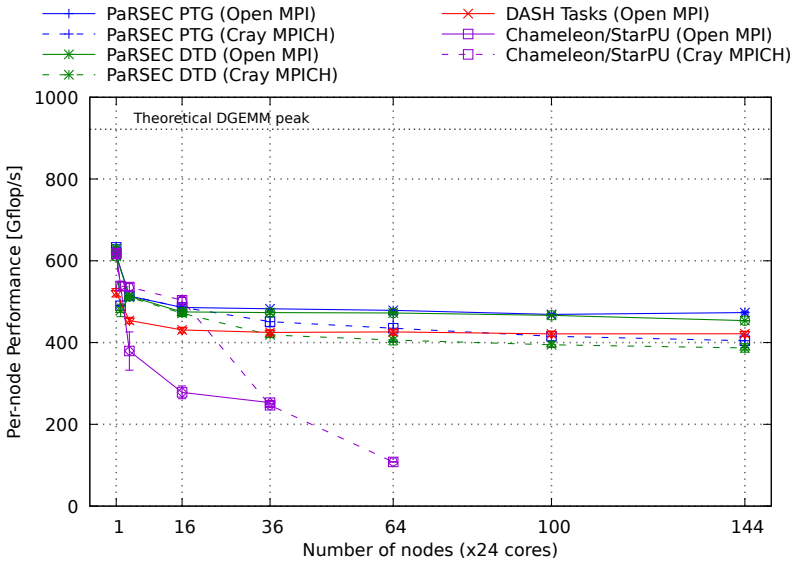
The amount of tiles to be communicated across shared memory boundaries can be dramatically reduced by introducing so-called super-tiles, which are clusters of neighboring tiles inside a process. This helps reduce the communication required for both the ORMQR and SSMQR tasks.

The implementation of the QR decomposition using the DASH Tasks interface is listed in Appendix B. DASH does not have native support for super-tiles, but that feature can be emulated using a four-dimensional matrix where the first two dimensions represent the super-tiles and the second two dimensions represent the actual tiles. The matrix is then only distributed over the first two dimensions. The size of the super-tiles was fixed for both PaRSEC and DASH at 16 in the first dimension and 1 in the second dimension as that appeared to yield the highest performance.

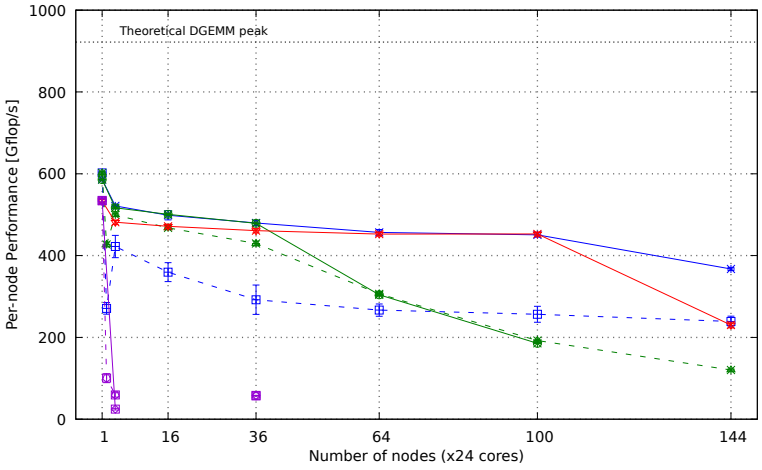
6.4.3.1. Results

The results are shown in Figure 6.8. Similar to the Cholesky decomposition results presented above, both Open MPI and Cray MPICH have been used for the PaRSEC and Chameleon benchmarks. The per-process matrix size has been chosen at $15k^2$ elements and the tile sizes as 320^2 and 192^2.

For a tile size of 320^2, the DASH Tasks variant achieves about 89% of the performance of PaRSEC PTG running on Open MPI but outperforms PaRSEC running on



(a) N=15k/node, NB=320



(b) N=15k/node, NB=192

Figure 6.8.: Performance of Tiled QR decomposition on the Cray XC40 in different configurations. Error bars represent the standard deviation of at least five runs. Missing data points represent runs that did not finish in under 10 minutes.

Cray MPICH (Figure 6.8a). The performance of both PaRSEC and DASH appear stable over the range of 16 to 144 processes. For Chameleon, the performance drops off sharply starting at only a few processes. The reason for this loss in performance is likely to be found in an apparent lack of super-tiles, which causes significantly higher amounts of communication with rising numbers of processes.

With a smaller tile size of 192^2, the communication overhead and the number of tasks again increases significantly, causing PaRSEC DTD to drop in performance after 36 nodes (Figure 6.8b). Both DASH and PaRSEC PTG (running under Open MPI) yield similar performance at 64 and 100 nodes after which both drop off, with DASH achieving only 230 Gflop/s at 144 nodes, a value similar to the performance of PTG running under Cray MPICH. It is notable that the performance of the latter is only at about 50% of PTG using Open MPI for the majority of the process range.

6.4.4. Discussion

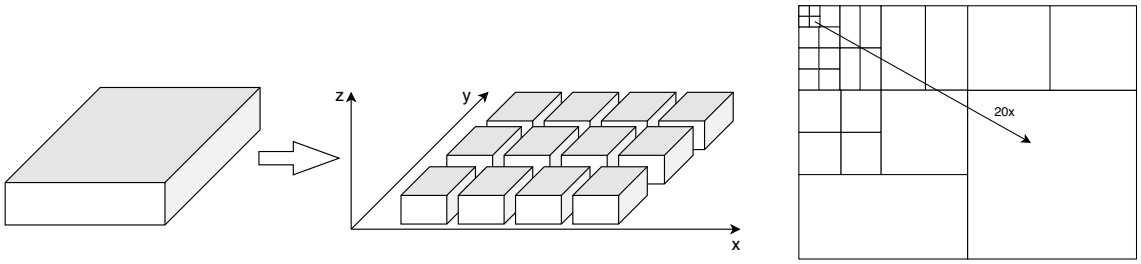
The results presented in this section show that the distributed task synchronization approach proposed in this work has the potential to scale reasonably well with applications containing large distributed task graphs. While the performance of PaRSEC PTG (which was specifically developed to target this class of applications) remains largely unrivaled, the approach of this work has been shown to outperform approaches that use global-view task graph discovery approaches. This matches the expectation expressed earlier that distributed task graph discovery avoids the bottleneck of global-view discovery and thus may improve scalability.

Nevertheless, especially at higher node numbers the performance still suffers from some degree of performance degradation. This may be attributed to the large volumes of dependency information that have to be exchanged. A new message queue design might help mitigate the impact by providing low latency high throughput message delivery. However, this design was not yet ready for application in this scenario due to software stability issues.

Overall, the results demonstrate that task-based PGAS approaches are a viable alternative for tiled linear algebra algorithms even though some further optimizations may be required.

6.5. Multi-Zone Block Tri-Diagonal Solver

The NAS Parallel Benchmarks (NPB) are a collection of benchmarks designed to reflect different computational aspects of CFD simulations, both through simple kernel and proxy applications [120]. The original version of these benchmarks



(a) 3D multi-zone domain decomposition [123].

(b) 2D cut through the zones depicting differences in size [124].

Figure 6.9.: Depiction of the NPB multi-zone grid.

exhibit mostly fine-grained concurrency that could be exploited using OpenMP worksharing constructs in combination with MPI to provide a set of parallel benchmarks for machines expected for the end of the last century. The original definition of the benchmarks was purely algorithmic and came with a set of rules for potential implementations. Parallel implementations in Fortran-77 using MPI [121] and OpenMP [122] have been provided by NASA, which are still maintained to this day. The reference implementations provided both an input generator and a verification step.

Three of the solver benchmarks—namely Lower-Upper Symmetric Gauss-Seidel (LU), Scalar Penta-diagonal (SP), and Block Tri-diagonal (BT) with blocks of 5×5 elements—solve discretized versions of the unsteady, compressible Navier-Stokes equations. These benchmarks operate on a structured 3D mesh with a single mesh distributed across all processes.

In order to better reflect complex realistic applications [62], these benchmarks have been extended to incorporate multiple overlapping meshes (or *zones*), as depicted in Figure 6.9a. This so-called *composite overset method*—or Chimera method—is a technique used in CFD simulations to handle complex and moving geometries, e.g., moving control surfaces in maneuvering aircraft [60]. It is used in production-level CFD codes such as the DLR's TAU code [61].

In the NPB multi-zone benchmarks, the zones are generated with a range of different sizes, with the largest zone being twenty times the size of the smallest zone, as depicted in Figure 6.9b. A simple static load-balancing algorithm is used to distribute the resulting work evenly among the participating processes [124].

The zones can be solved independently in each timestep before information on bordering zones is collected and exchanged. The multi-zone implementations (LU-MZ, SP-MZ, and BT-MZ) aggregate the boundary information of all zones and perform the communication in a single step to reduce the number of communication opera-

Algorithm 6.3: Simplified timestep algorithm of the reference implementation of NPB BT-MZ.

```
1 for it ← 1 to num_steps do
2     exch_qbc(zones);
3     for nz ← 1 to num_local_zones do
4         // fine-grained concurrency
5         // inside each function
6         x_solve(zones[nz]);
7         y_solve(zones[nz]);
8         z_solve(zones[nz]);
```

tions. In the reference implementation, the computation of the local zones is done sequentially, and fine-grained loop-based concurrency using OpenMP is exploited within each zone. This is depicted in Algorithm 6.3, where each timestep consists of the exchange of zone boundaries (`exch_qbc`) and the solution of uncoupled systems of equations in all three directions x, y, and z.

In the remainder of this section, the BT-MZ benchmark will be used as a representative of the NPB multi-zone benchmarks. In the OpenMP-based reference implementations, the solutions in both x and y directions allow for the parallelization over the z dimension while the solution in z direction is parallelized over the y dimension. Table 6.1 lists the grid sizes for different problem sizes (called *classes*) of the BT-MZ benchmark. It shows that the grids are scaled such that the z direction is the smallest dimension. It will be shown later that this may lead to scalability limitations in the number of threads that may be employed within each process.

Table 6.1.: Overview of the problem sizes available in the NPB BT-MZ benchmark (excerpt from [125]).

Class	**Number of Zones**	**Iterations**	**Grid Size** $[x \times y \times z]$	**Total Memory**
S	2×2	60	$24 \times 24 \times 6$	1 Megabyte (MB)
W	4×4	200	$64 \times 64 \times 8$	6 MB
A	4×4	200	$128 \times 128 \times 16$	50 MB
B	8×8	200	$304 \times 208 \times 17$	200 MB
C	16×16	200	$480 \times 320 \times 28$	0.8 Gigabyte (GB)
D	32×32	250	$1632 \times 1216 \times 34$	12.8 GB

6.5.1. Implementations

Several different implementations of NAS Parallel Benchmarks (NPB) BT-MZ have been created, whose performance will be compared in Section 6.5.2. Starting from the reference Fortran implementation, over a port to C++, to task-based versions based on the C++ port, these implementations will be briefly described in the following sections.

6.5.1.1. Reference Implementation

The reference implementation of BT-MZ is written in Fortran and combines MPI for inter-process data exchange with OpenMP work-sharing constructs for loop-level parallelism. Version 3.3.1 of the implementation is available online [31].

6.5.1.2. C++ Implementation

Since DASH is a project based on C++, a C++ version of the reference implementation was created to facilitate the porting to this and other programming models. To simplify the adaptation from Fortran to C++, a matrix implementation has been created using C++-17 features that allows for Fortran-style column-major (vs. native C++ row-major) and one-based indexing. Usage of both OpenMP and MPI is similar to the Fortran reference implementation. This version serves as the basis for all following variants of the BT-MZ benchmark.

6.5.1.3. DASH Implementation

The DASH-based implementation uses DASH containers instead of MPI messages to exchange boundary information in a style similar to the reference implementation. After computation of the zones the boundary data for each zone is stored in a distributed four-dimensional array. An entry in the distributed matrix is indexed using the tuple $(p, z_{p,j}, f, e)$ with p being the process, $z_{p,j}$ the zone j on process p, f being one of the four faces of a zone (west, east, south, north), and e being the linear element index inside the face. The distribution of data is limited to the first dimension so that each process owns exactly one slice of the remaining three dimensions (z, f, e). The computation itself is performed on local arrays.

The data exchange is done using the DASH algorithm `dash::async_copy`, which initiates a `get` and returns a `future` that can be used to wait for completion of the

[31] `https://www.nas.nasa.gov/assets/npb/NPB3.3.1.tar.gz`, last accessed February 14, 2020

Algorithm 6.4: Task-based timestep algorithm of NPB BT-MZ.

```
1 for it ← 1 to num_steps do
2     // Coarse-grain concurrent execution of zones with
3     // fine-grained concurrency inside
4     // each function
5     for nz ← 1 to num_local_zones do
6         exch_qbc(zones[nz]);
7         x_solve(zones[nz]);
8         y_solve(zones[nz]);
9         z_solve(zones[nz]);
```

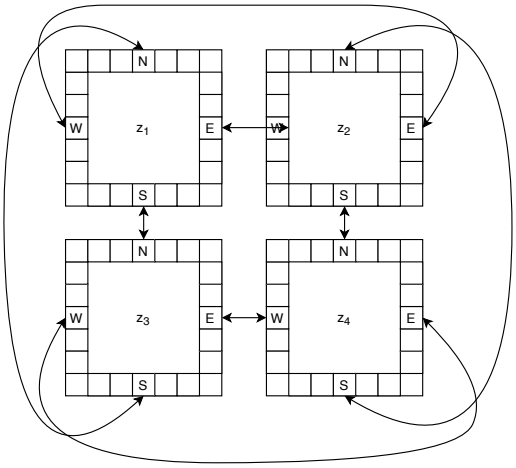
transfer. This allows to overlap all transfers, similar to the MPI-based implementation. The synchronization needed to ensure consistent reads and writes is done using `dash::barrier`, a coarse-grain collective synchronization primitive to make sure that no `get` occurs before the local writes have completed and vice versa.

6.5.1.4. DASH Tasks Implementation

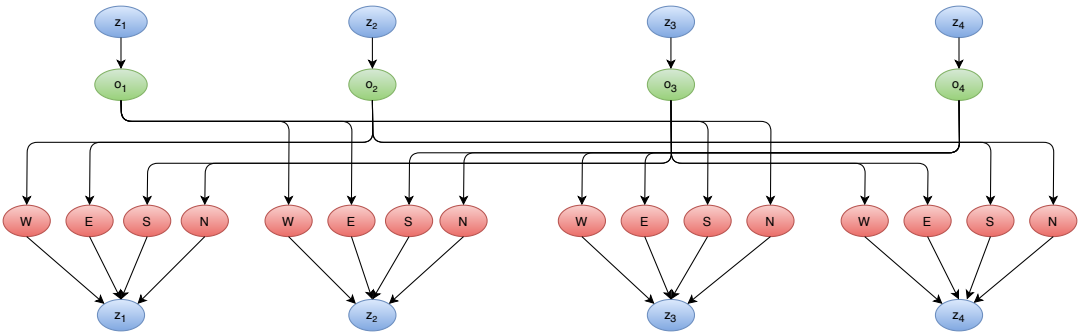
This version of BT-MZ is the first task-based implementation discussed here. Instead of OpenMP work-sharing constructs, the fine-grained concurrency is implemented using `dash::async` and `dash::taskloop`. For the latter, chunks of iterations of consecutive loops in the code are chained using dependencies between the loops, which reduces the number of synchronization points to a few sensitive points, e.g., between `y_solve` which is parallelized over the z dimension and `z_solve` which uses the y dimension for parallelism.

More important, however, is the fact that zones are computed truly independently. In each timestep, tasks are created for each zone z_i, which i) compute the solution of that zone (with task-based loop-level parallelism nested inside, as described above); ii) copy the boundary data into the global DASH array; iii) copy all relevant boundary data of neighboring zones and integrate the data into the local zone. The algorithm is displayed in Algorithm 6.4. Data dependencies are used to coordinate the execution of these tasks both locally and globally. After all four faces have been updated, the next timestep for zone z_i can be computed.

As an example, the dependencies between four zones with cyclic boundaries—which is the case for BT-MZ—are depicted in Figure 6.10. Figure 6.10a shows the communication dependencies: with only four zones, zone z_1 requires both the north and south boundary from zone z_3 and both west and east boundary of zone z_2. The resulting task dependency graph is depicted in Figure 6.10b: the blue tasks



(a) Dependencies between zone faces in a 2D cyclic domain decomposition.



(b) Task dependency graph for two iterations (blue: computation tasks; green: tasks packing boundary data into global data structures; red: tasks incorporating the boundary data of neighboring zones into local zone.

Figure 6.10.: Dependencies between zones and the tasks computing them.

represent the computational tasks, the tasks that copy the boundary data into the DASH array are depicted green, and the tasks that incorporate the boundaries into the local zone are colored red.

The nested parallelization scheme allows for the exploitation of the maximum degree of concurrency in the multi-zone benchmark. All runnable tasks are put into the same task-pool, which supplies all threads with work, reducing thread idle-times as much as possible. The data transfers are expressed through `copyin` dependencies and handled by the runtime system.

6.5.1.5. MPI with OpenMP Tasks

In addition to loop-level work-sharing constructs, OpenMP also provides task-based concurrency primitives. The author of this work has shown in [109] that the implicit dependencies between tasks communicating through MPI may pose an unsolvable

challenge to correctness of MPI-parallel applications using OpenMP tasks: While OpenMP provides a `taskyield` directive, it may simply be ignored by the runtime. This in turn may lead to situations in which insufficient numbers of communication operations are posted to guarantee that all already active communication operations progress, resulting in a deadlock. The number of zones in BT-MZ (cf. Table 6.1) may easily exceed the number of threads available to OpenMP so task-parallel communication cannot be implemented using OpenMP.

Instead, the OpenMP variant of the benchmark performs the nested parallelization described in the previous section for the *DASH Tasks* variant but requires the completion of all high-level tasks before the accumulated communication step that is taken from the reference benchmark. Thus, this variant only exploits coarse-grain concurrency within a single timestep without leveraging the potential concurrency across multiple timesteps. Threads are thus left idle during the communication phase.

6.5.1.6. MPI with OmpSs-2 Tasks

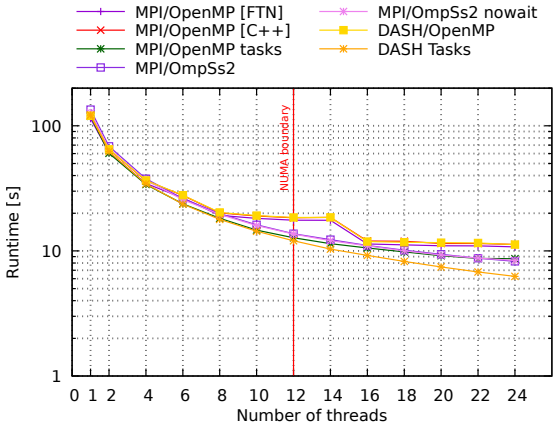
The OmpSs-2 programming model is a redesign of the OmpSs and OpenMP task-based programming models [126]. In a first step, the OpenMP variant discussed in the previous section has been adapted to the subtle differences between OpenMP and OmpSs-2 (mainly missing support for C++ 17 and differing pragma names). This variant is otherwise identical to the OpenMP variant described in Section 6.5.1.5.

6.5.1.7. TAMPI with OmpSs-2 Tasks

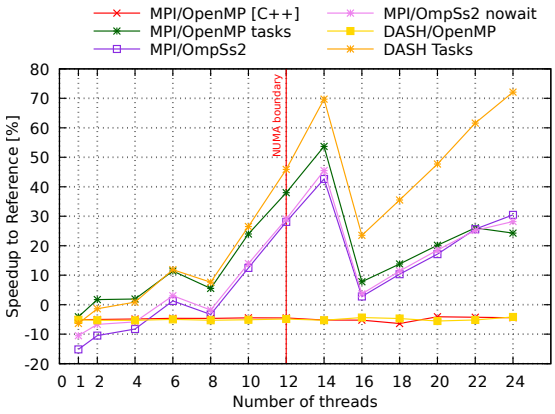
The Task-Aware MPI (TAMPI) library provides a shim layer around MPI and uses advanced features of the OmpSs-2 runtime to block the execution of tasks communicating through MPI and rescheduling them as soon as the communication operation(s) have completed [127]. Using this library, it is possible to avoid the central synchronization point between computation and communication and exploit coarse-grain concurrency between zones across timesteps, similar to the DASH Tasks variant.

6.5.2. Evaluation

In order to compare the different implementations, the evaluation will first consider the scaling behavior with respect to the number of threads on a single-node. Afterwards, node-level scaling will be considered for two different problem sizes (C and D).



(a) Runtime.



(b) Speedup relative to reference implementation.

Figure 6.11.: Runtime and speedup of the different BT-MZ variants with increasing number of threads on a single node.

6.5.2.1. Cray XC40: Thread-level Scaling

The first step in evaluating the different variants of BT-MZ is a thread-level scaling study in which a single process is running on a Cray XC40 node. The range of the number of threads is one to 24 with a 2-step increment between two and the upper limit. For this experiment, the problem size class B has been chosen as it provides reasonable run times across the full spectrum of the number of cores.

The run time of the benchmarks with increasing numbers of threads are depicted on a logarithmic scale in Figure 6.11a. Here it is notable that significant differences only start to occur around ten threads when the OpenMP work-sharing constructs start to level out before reaching the NUMA boundary, where the number of threads start exceeding the number of cores available on a single socket. On the right side of the NUMA boundary, the OpenMP work-sharing variants match the performance of the task-based variants only at 16 threads before leveling off until 24 threads. This effect can easily be explained by the limited concurrency exposed by the majority of the nested loops parallelized over the z direction: for class B, the number of blocks in the z dimension is only 17, causing any additional threads to remain idle. Thus, this effect is not a direct limitation of the OpenMP runtime but a limitation of the benchmark itself.

However, this effect can be seen as an indirect limitation of the loop work-sharing constructs, which do not allow the exploitation of coarse-grain concurrency within one timestep. As can be seen when looking at the speedups over the reference implementation depicted in Figure 6.11b, the task-based variants clearly outperform

the work-sharing variants at higher numbers of threads. Single-threaded runs are an exception because all task-based variants suffer from the task-management overhead that is not present in the traditional OpenMP variants.

The OpenMP task-based variant achieves up to 55% speedup at 14 threads over the reference implementation while the DASH Task implementation achieves up to 72% speedup at 24 cores. For both OmpSs-2 variants, it is notable that no benefit can be observed for thread counts lower than ten and that the remaining trajectory matches that of the OpenMP task-based variant.

The drop in speedup observable in Figure 6.11b in the middle of the range can be explained by the improvements in run-time observable for the work-sharing variants between 14 and 16 threads in Figure 6.11a.

It is notable that the DASH Task variant achieves significantly higher speedups at higher thread counts than the other task-based variants. This appears to be a quality-of-implementation issue as no communication happens in the single-process case and thus no difference between traditional MPI and PGAS should be notable.

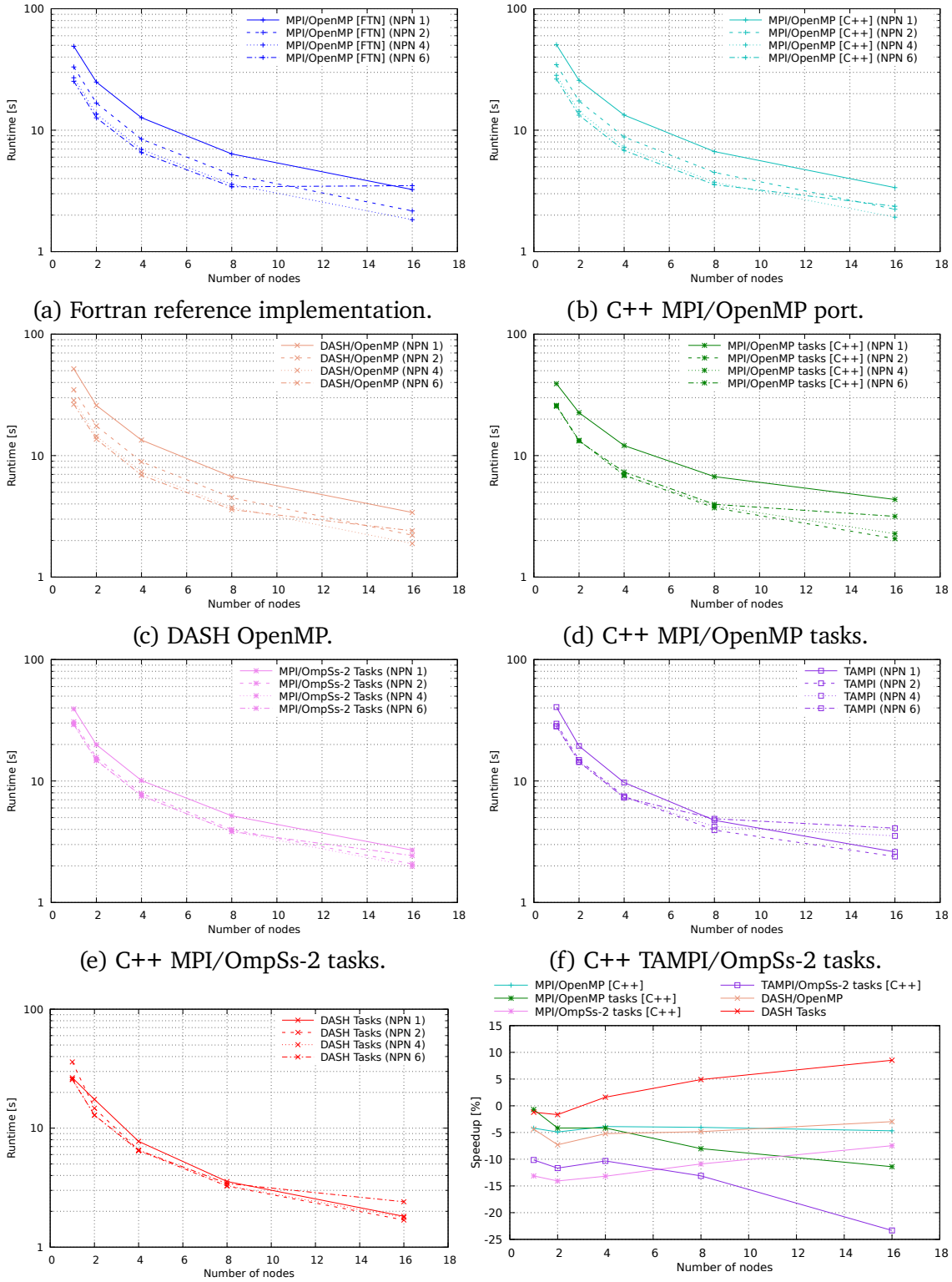
It should also be noted that the loop work-sharing variants implemented in C++ exhibit a slight performance penalty of around 5% across the full range. While this may hint at some deficiencies in the implementation of the multi-dimensional array used to port the application from Fortran to C++ (or of the C++ compiler used to compile the resulting code) it should not impact the conclusions drawn here and in the following sections.

6.5.2.2. Cray XC40: Node-level Scaling

The previous section has established that the different variants of BT-MZ exhibit different scaling behavior with an increasing number of threads. It is important to factor this in when scaling the number of compute nodes. In the following experiments, all variants were thus run on a set of nodes with different number of processes per nodes (NPNs), ranging from one over two and four up to six, i.e., using 24, 12, 6, and 3 threads per process.

While the discussion in the previous section may suggest that a configuration with eight threads per process (NPN = 3) may yield the highest performance with OpenMP work-sharing variants, it will likely lead to NUMA issues due to one process spanning across the NUMA boundary. This configuration has thus been ignored.

For the BT-MZ class C benchmark, the node scaling was done in powers of two ranging from one to 16. The results are depicted in Figure 6.12. For all OpenMP work-sharing variants (Figures 6.12a–c), the case of NPN = 1 appears to be the slowest configuration, which can be explained by the findings presented in the



(a) Fortran reference implementation.

(b) C++ MPI/OpenMP port.

(c) DASH OpenMP.

(d) C++ MPI/OpenMP tasks.

(e) C++ MPI/OmpSs-2 tasks.

(f) C++ TAMPI/OmpSs-2 tasks.

(g) DASH Tasks.

(h) Speedup of best configuration over Fortran reference implementation.

Figure 6.12.: Scaling behavior and speedup of different implementations of BT-MZ using class C on Cray XC40.

previous section. More interesting, however, is that the configuration NPN = 6 appears to be the fastest configuration up until eight nodes, an observation that is most significant in the Fortran reference implementation. At 16 nodes, the configuration of NPN = 4 exhibits the highest performance, which may be explained by larger load imbalances with NPN = 6 due to a smaller number of (differently sized zones) per process.

Contrasting that, the task-based variants (Figures 6.12d–g) perform best at NPN = 2 (twelve threads per process) as this avoids NUMA issues while better exploiting the coarse-grain concurrency within (and across) timesteps.

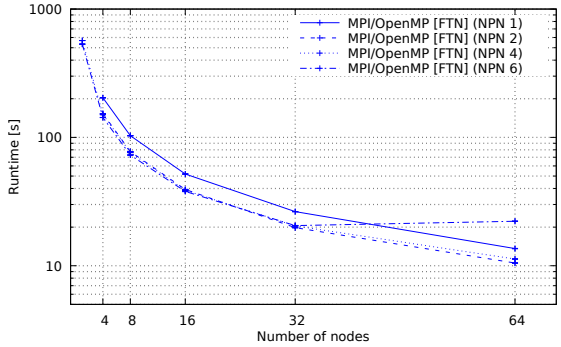
The speedup of all variants over the Fortran reference implementation is depicted in Figure 6.12h, which compares the best configuration at each data point, the configuration with the lowest run-time. The aforementioned 5% performance penalty of the C++ port is visible for the C++ MPI/OpenMP and DASH variants.

Other than the DASH Task variant, no other version of the benchmark outperforms the reference implementation, with the former achieving a 9% speedup over the latter at 16 nodes. It is remarkable that for this problem size, both OpenMP tasks and TAMPI lose ground against the reference implementation, with the latter yielding negative 24% at 16 nodes.

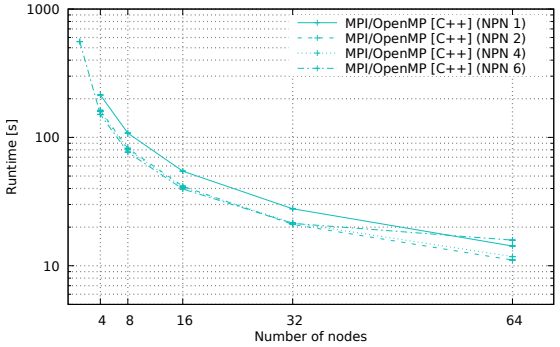
However, this benchmark class is rather small (at 16 nodes, the run time of the reference implementation is slightly below 2 s). Thus, these experiments have been repeated with class D, which entails a significantly larger grid, more zones, and a higher number of timesteps (cf. Table 6.1). The results of these experiments are depicted in Figure 6.13, where the numbers of nodes range from two to 64.

For the OpenMP work-sharing variants (Figure 6.13a–c), the best configuration at scale is NPN = 2, closely followed by NPN = 4. Similar to the observation made for NPN = 6 above, the scaling is impeded at 64 nodes compared to the other configurations. For the task-based variants (Figure 6.13d–g, with the exception of TAMPI), NPN = 2 yields the best performance at scale. For TAMPI, the best configuration at scale appears to be NPN = 1.

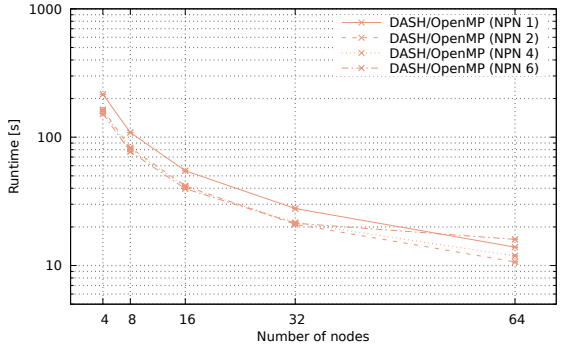
Looking at the speedup depicted in Figure 6.13h, it can again be seen that the OpenMP task variant does not yield any significant speedup over the reference implementation (even when accounting for the 5% slowdown stemming from the C++ port) and instead yields an additional slowdown at 64 nodes. The DASH OpenMP variant, on the other hand yields a slight uptick in speedup at 64 nodes, suggesting a slight benefit of using PGAS over traditional message passing at this point.



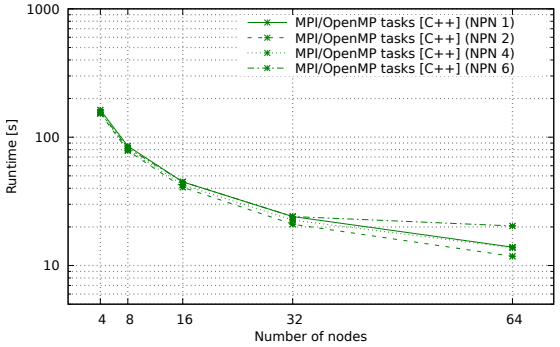
(a) Fortran reference implementation.



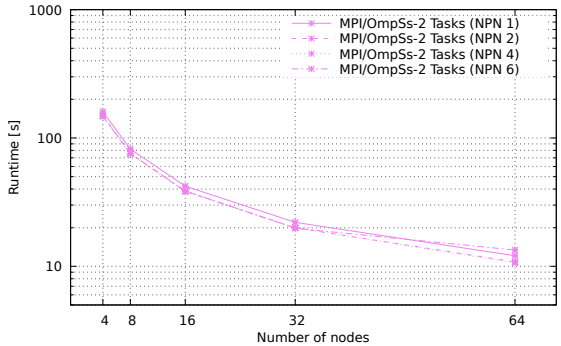
(b) C++ MPI/OpenMP port.



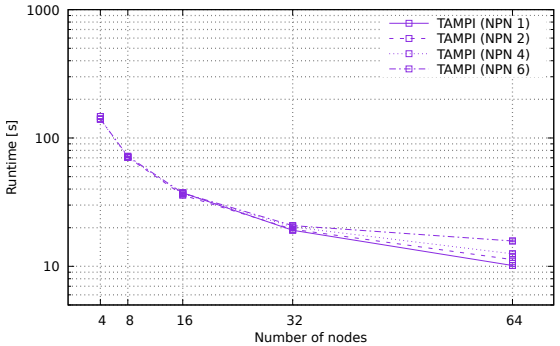
(c) DASH/OpenMP.



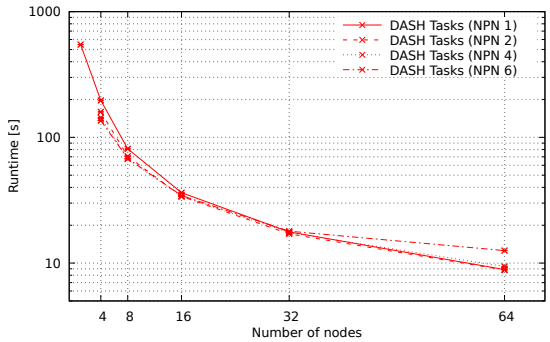
(d) C++ MPI/OpenMP tasks.



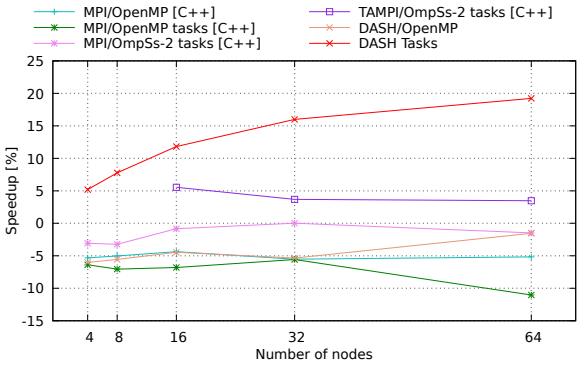
(e) C++ MPI/OmpSs-2 tasks.



(f) C++ TAMPI/OmpSs-2 tasks.



(g) DASH tasks.



(h) Speedup of best configuration over Fortran reference implementation.

Figure 6.13.: Scaling behavior and speedup of different implementations of BT-MZ using class D on Cray XC40. Error bars represent the standard deviation of at least five runs.

For this larger problem size, the TAMPI variant outperforms both the reference implementation and the OmpSs-2 port, suggesting that an improved coupling between the task runtime system and MPI can be beneficial.

Finally, except for the runs at two nodes, the variant using DASH Tasks outperforms all other variants and yields a speedup of almost 20% at 64 nodes.

6.5.2.3. Taurus: Node-level Scaling

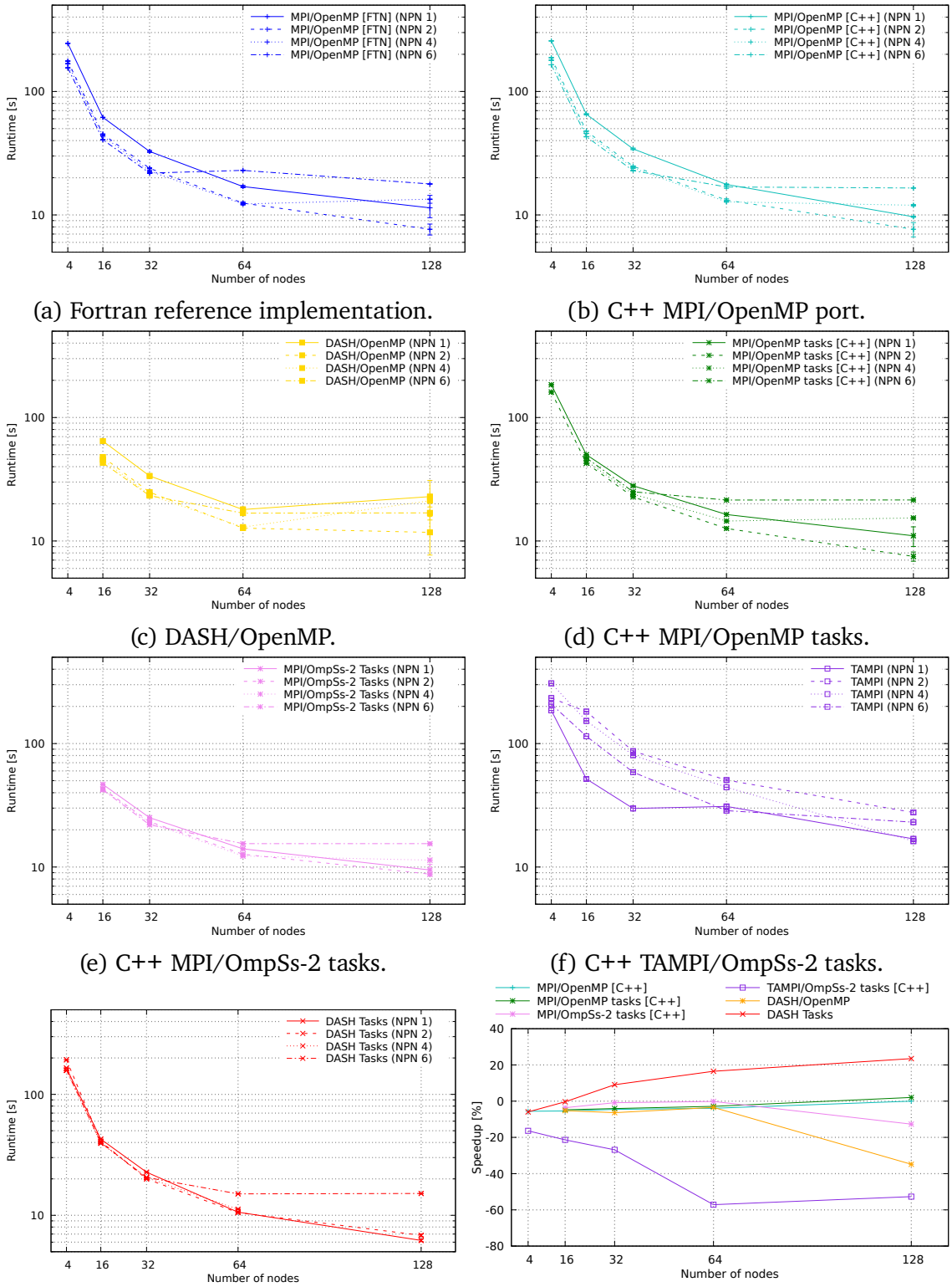
On the Taurus cluster, a similar performance pattern for BT-MZ class D emerges for the OpenMP-based implementation and the DASH Tasks implementation, as depicted in Figure 6.14. With node numbers ranging from 4 to 128 nodes, the DASH Tasks implementation again yields up to 20% speedup over the reference implementation. While the DASH/OpenMP variant shows similar performance as the MPI/OpenMP variant, the run-time curve flattens after 64 nodes (Figure 6.14c), confirming that the use of collective synchronization mechanisms poses a challenge for scalability.

An interesting observation can be made for the OmpSs-2 variant using TAMPI (Figure 6.14f). In contrast to the Cray system, the performance at higher node counts is significantly lower than the OmpSs-2 variant using regular MPI. Some preliminary investigation hinted at a problem in the interaction between OmpSs-2 and TAMPI.[32]

6.5.3. Discussion

The results discussed in this section indicate that neither the isolated use of PGAS nor the use of tasks by itself yields significant benefits over the reference implementation using MPI and OpenMP. It appears insufficient to simply replace OpenMP work-sharing constructs with tasks as the latter incurs additional overhead and the use of intra-timestep coarse-grain concurrency does not improve the use of available hardware resources. Due to a lack of proper task scheduling control in OpenMP, the communication phase can still only involve a single thread, leaving all other

[32] TAMPI may use two different modes provided by OmpSs-2: in *blocking* mode, a task is blocked until the communication has completed, which for OmpSs-2 means that the thread is blocked and another thread is created/activated on that core to continue computing. The resulting context switches and task migrations may cause significant overhead. Once all relevant communication has completed, the thread is reactivated to allow the execution of the task to complete and to release its dependencies. In TAMPI's *non-blocking* mode, the task is detached after its execution has completed, i.e., the task finishes execution but the dependencies are not released until all outstanding events of the task have been signaled. In this mode, however, the release of the dependencies is serialized on the thread testing for completion of MPI communication. This mode has yielded better performance on Taurus but it still appears that the dependency release incurs significant overheads.



(a) Fortran reference implementation.

(b) C++ MPI/OpenMP port.

(c) DASH/OpenMP.

(d) C++ MPI/OpenMP tasks.

(e) C++ MPI/OmpSs-2 tasks.

(f) C++ TAMPI/OmpSs-2 tasks.

(g) DASH tasks.

(h) Speedup of best configuration over Fortran reference implementation.

Figure 6.14.: Scaling behavior and speedup of different implementations of BT-MZ using class D on Taurus. Error bars represent the standard deviation of at least five runs.

threads idle. Similarly, using PGAS (DASH) as a replacement for MPI does not improve the communication overhead that leads to thread idle times during the communication phase as RMA communication rarely outperforms blocking message-based communication.

However, the combination of both in the form of the DASH Tasks variant yields significantly improved scaling by hiding the communication latencies through the use of coarse-grain concurrency across timestep boundaries. With this approach, it is possible to maximize the utilization of threads and thus minimize idle times.

A similar strategy of hiding communication overhead of traditional message passing through the use of inter-timestep coarse-grain concurrency has been attempted with the TAMPI variant of the benchmark. On the Cray system, it appears that the overhead of message passing limits the scalability of this approach. A potential reason may be the higher transfer efficiency of RDMA used in DASH compared to the repeatedly required calls into the MPI library to progress non-blocking message-based communication. On Taurus, however, the performance of OmpSs-2 appears to suffer from severe overheads incurred by the dependency management.

6.6. Livermore Unstructured Langrangian Shock Hydrodynamics (LULESH)

The Livermore Unstructured Langrangian Explicit Shock Hydrodynamics (LULESH) proxy application is developed at Lawrence Livermore National Lab (LLNL) and is part of the CORAL benchmark suite [63]. Using an unstructured mesh similar to the one depicted in Figure 6.15, LULESH simulates a Sedov blast on a three-dimensional staggered mesh of hexahedrons. The algorithmic details are described in [63]. For the discussion of the usage of tasks, it suffices to distinguish the *nodes* and *elements* (formed by eight nodes) in the mesh. With version 2.0, LULESH has also added the notion of *regions*, which are formed by an arbitrary number of elements [128]. The work presented here is based on version 2.0, which will henceforth be referred to simply as LULESH.

The time integration of LULESH performs four major steps: (i) determining the time increment for the next timestep; (ii) advancing the nodal quantities; (iii) advancing the element quantities; and (iv) computing the material properties of the regions. The first step requires a collective operation over all processes in order to apply the Courant-Friedrichs-Lewy constraint (CFL) and determine the width of the next timestep. However, part of the second step (advancing nodal quantities) is independent of the time increment and thus could partially be overlapped with the

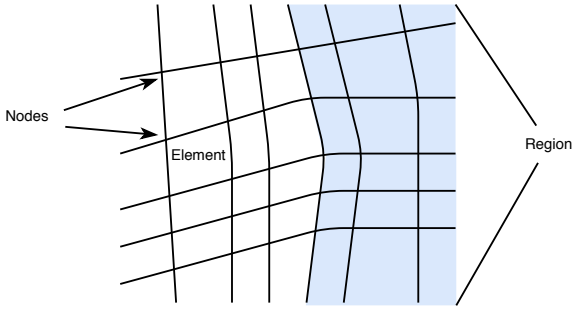


Figure 6.15.: Two-dimensional depiction of the structure of the 3D mesh used by LULESH (after [63])

collective operation (namely calculating the forces on the nodes and the resulting acceleration). In the third and fourth steps, results from the previous steps are used as input so that the computation of element quantities relies on previously computed nodal quantities, which in turn are required for the region computation. There, LULESH uses dynamically allocated memory to store temporary results during the computation.

The reference implementation of LULESH combines MPI and OpenMP work-sharing constructs for distributed and node-local parallelism. Communication occurs in 27 directions (faces, edges, vertices) and is not overlapped with any computation. An initial port of LULESH to DASH (without the use of tasks) has been presented in [37]. This version has replaced the use of MPI two-sided communication with DASH global containers, which are used to pack and unpack boundary data. The authors of [37] show no significant difference in performance between the two versions but claim a reduction in code complexity due to the one-sided communication style. Both variants of LULESH only allow for a cubic number of processors in order to simplify the process grid generation.

In contrast to the inherent strong-scaling approach taken by the NPB benchmark discussed in Section 6.5, LULESH employs an inherent *weak-scaling* approach: the problem size is defined in terms of elements per process. Thus, scaling the number of processes also automatically scales the problem size. In the following discussions, this weak-scaling approach has been used and no attempt has been made to perform strong-scaling experiments. Any problem size definition in the following discussion refers to the problem size per process.

6.6.1. Porting Strategy

LULESH is a fairly complex application, which is already written in C++, a fact that made porting easier compared to the NPB benchmark. Adapting the existing

DASH port of LULESH to use the DASH tasking model was done in three steps. As a first step, the OpenMP worksharing constructs have been replaced by DASH taskloop constructs followed by a call to `complete` in order to ensure correctness in the absence of dependencies. In a second step, taskloop chunks have been chained using dependencies wherever possible, as described in Section 2.3. At this point, the communication steps are still protected using barriers so that all tasks have to complete before the communication step may commence.

Given the nature of the unstructured mesh used in LULESH, partially overlapping communication with computation would require significant restructuring of the computational kernels to map nodes and elements to faces, edges, and vertices in the local subdomain. While high-level programming abstractions exist for mesh-based computations [47], such a transformation has not been applied here. However, using dependencies on the global data structures allows for the overlapping at least some parts of the application. Specifically, using output dependencies on tasks packing the output buffers and `copyin` dependencies to transfer the data, transfers from neighboring processes may be initialized even while the local computation has not completed yet. The local packing in tasks carrying the output dependencies, however, is only done as soon as all previous local computation tasks have completed.

This scheme has been implemented using a two-level approach where high-level tasks are used to handle the control flow between computation and communication steps and tasks nested within them are used to provide concurrency between communication events. Nevertheless, there is some potential for overlapping communication and computation: as described above, determining the Courant-Friedrichs-Lewy constraint (CFL) constraint and thus the next timestep width may be performed concurrently with some of the computation of nodal quantities. Using task dependencies, this can be expressed simply through dependencies and occurs naturally with the runtime system scheduling any available tasks for execution. The task-graph of a single timestep in LULESH is provided in Appendix D.

One of the design parameters when using taskloops is the number of tasks to be generated. While a smaller number of tasks reduces the task management overhead, a larger number of tasks allows the scheduler to distribute work more evenly among threads, especially in the presence of hardware-induced imbalances such as NUMA effects. In the task-based LULESH variant, the number of tasks is not fixed. Instead, it is a function of the number of available worker threads multiplied by a Task Loop Factor (TLF) that can be controlled by the user. Its impact will be discussed later in this section.

Even though it would be possible to concurrently perform the computation on the different regions mentioned above [128, §2], this potential has not been exploited in

order to keep the memory requirements in line with the reference implementation. However, a future version could explore this as an option to expose more concurrency to the scheduler.

6.6.2. Issues Encountered

In the following sub-sections, two issues will be briefly discussed that were discovered during the work on the task-based variant of LULESH.

6.6.2.1. Dynamic Memory Management

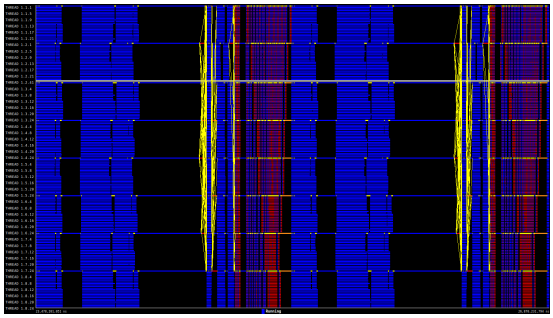
As mentioned above, some of the computation requires dynamic memory allocation and deallocation. During the experiments, it became clear that allocation and deallocation of large quantities of memory can become a costly operation. Both the reference implementation and the task-based version are impacted by this issue, although the more dynamic nature of the task-based version allows for some hiding of these latencies.

This is exemplified by traces depicted in Figure 6.16, which were obtained using Extrae and visualized using Paraver. The reference implementation (Figure 6.16a) shows rather long idle times (black) between dense computational areas. In the task-based version, however, threads are able to perform some useful work during these times (Figure 6.16c).

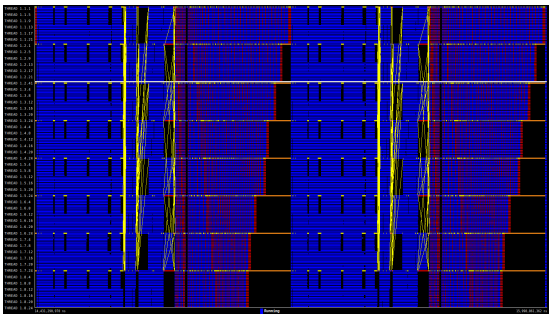
In order to mitigate this issue and provide for a fair comparison, the TCMalloc package[33] has been used to avoid expensive calls to `malloc` and `free`. To do this, TCMalloc intercepts calls into these functions and caches the memory allocations to avoid repeatedly calling the underlying system functions. Some fine-tuning was necessary to completely avoid system calls within TCMalloc, namely setting the environment variables `TCMALLOC_DISABLE_MEMORY_RELEASE=t` to prevent TCMalloc from ever returning memory to the operating system (which it would otherwise do periodically) and `TCMALLOC_AGGRESSIVE_DECOMMIT=f` to prevent it from marking memory areas as not needed (so the operating system could page them out more easily). Both operations would have otherwise introduced latencies similar to the original calling of `malloc` and `free`.

The resulting behavior is depicted in Figure 6.16b (reference implementation) and Figure 6.16d (DASH Tasks). The resulting execution trace is much more dense with significantly less idle times.

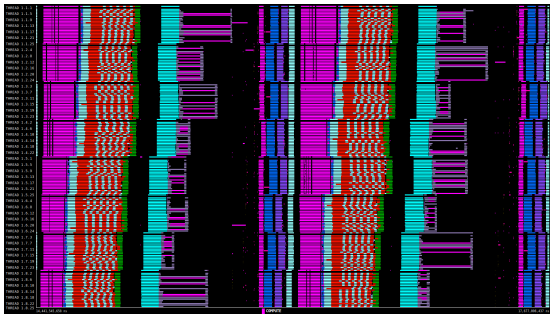
[33] TCMalloc is available as open source software at `https://github.com/google/tcmalloc` (last accessed March 5, 2020).



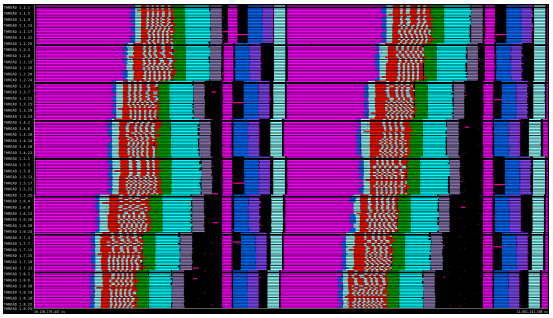
(a) MPI/OpenMP without Thread-Caching Malloc (TCMalloc).



(b) MPI/OpenMP with TCMalloc.



(c) DASH Tasks without TCMalloc.



(d) DASH Tasks with TCMalloc.

Figure 6.16.: Extrae screenshot visualzed using Paraver of two timesteps of LULESH with 200^3 elements on 8 nodes with 24 threads each, with and without the use of TCMalloc. Black areas mark idle threads, colored regions mark areas of computational or communication activity.

These traces also depict an interesting side effect of task-based parallelism: while there are regular short gaps visible in the blue (computational) areas in Figure 6.16b, such gaps are not visible in the task-based version in Figure 6.16d. These gaps in the former variant stem from NUMA effects that cause half of the threads within a process to complete their computation earlier than the other half that suffers from slower memory accesses. While the author of this work has proposed some memory-related optimizations for LULESH[34], the NUMA issues still exist in LULESH.

6.6.2.2. Optimization of Code within C++ Lambdas

A second issue that needed significant investigation is the lack of certain optimizations of code inside non-inlined lambda constructs, i.e., lambdas that are stored as part of a task for later execution. It appears that some mainstream C++ compilers

[34]Github pull request: `https://github.com/LLNL/LULESH/pull/8`, last accessed March 5, 2020.

(tested up to GCC 8 and Intel compiler 19.0) assume aliasing[35] between pointers captured inside a lambda, even if these pointers have been marked with the `restrict`[36] keyword *outside* of the lambda. This results in additional load instructions being issued by the compiler and potentially missed vectorization opportunities. This issue does not seem to occur for pointers captured in OpenMP statements. The impact of this issue was especially noticable with small problem sizes in LULESH.

A workaround that has been applied is to mark all pointers with the `restrict` keyword *inside* the lambda. At the time of this writing, it appears that only GCC has fixed this issue in version 9 of the compiler collection. However, this compiler was not available at the time the experiments were conducted.

6.6.3. Evaluation

All experiments have been carried on the Cray XC40 and all codes were compiled using the Intel compiler. Open MPI has been used for all experiments. Similar to previous experiments, a thread-level scaling is first conducted followed by a node-level scaling study.

6.6.3.1. Scaling the Number of Threads

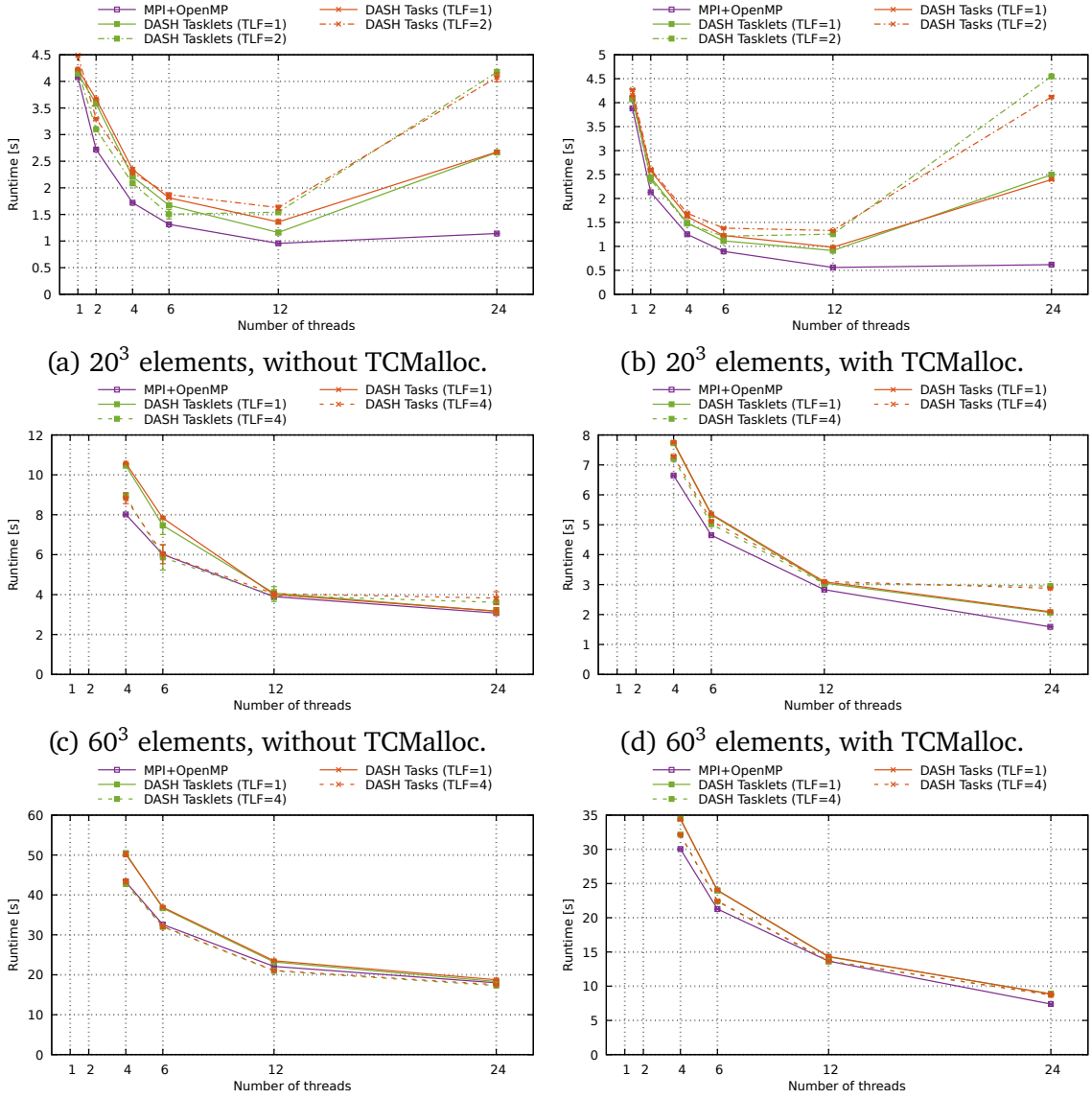
In a first set of experiments, the scaling of the number of threads under different problem sizes will be discussed. The results are depicted in Figure 6.17. The number of threads have been scaled from one (or four for larger problem sizes) threads to 24 threads, one thread per physical core at maximum. The experiments have been conducted with two task-based implementations of LULESH, one using full-fledged tasks and another using tasklets.

For a small problem size of 20^3 elements in the local mesh, the reference implementation clearly outperforms the task-based variant by a significant margin. As can be seen in Figure 6.17a (without TCMalloc) and Figure 6.17b (with TCMalloc), the reference implementation runs almost twice as fast with TCMalloc at higher thread counts. However, there is no gain in using more than 12 threads at this problem size.

More interesting, however, is the behavior of the task-based variants. While the performance is slightly worse than the reference until 12 threads, it significantly

[35] (Potential) Aliasing of pointers requires the compiler to issue load instructions because it cannot prove that writing to memory through one pointer does not change the value pointed to by another pointer. Thus, repeated read accesses through the second pointer require the compiler to issue load instructions because values previously loaded into registers may have become stale.

[36] The `restrict` keyword explicitly marks a pointer as unaliased. The keyword is part of the C standard and supported by many C++ compilers.



(a) 20^3 elements, without TCMalloc.

(b) 20^3 elements, with TCMalloc.

(c) 60^3 elements, without TCMalloc.

(d) 60^3 elements, with TCMalloc.

(e) 100^3 elements, without TCMalloc.

(f) 100^3 elements, with TCMalloc.

Figure 6.17.: Intra-node scaling of LULESH with different problem sizes on a Cray XC40. 500 iterations were executed for 20^3 elements, 100 iterations otherwise. Error bars represent the standard deviation of five repetitions.

degrades for 24 threads. Considering the small problem size, it is worth looking at the number of tasks that are being created: for the runs with a single thread, 91,001 tasks are being created, resulting in an average of 44.83 μs per task (including task discovery and management). Given the discussion in Section 6.3.1 and the heterogeneous nature of tasks (some tasks are rather short while some tasks have significantly more work) the overhead can be estimated to be in the order of 10%. For higher thread counts, however, the work per task only decreases and thus the overhead of the task management becomes even more significant.

It should also be noted that threads polling for work may impact the performance of the main thread discovering the task graph. While measures were taken in the scheduler to mitigate this impact (avoiding tight-loop busy polling for example) the overhead of threads competing for small chunks of work is significantly larger than with OpenMP, where each thread is *guaranteed* exactly one chunk of work in a statically scheduled worksharing construct. This is especially astute for 24 threads, where the execution time of the tasks is so small that threads almost exclusively compete for the next available task. The problem gets even worse when going from a Task Loop Factor (TLF) of one to a TLF of two, which effectively doubles the number of tasks.

For larger problem sizes (60^3 and 100^3 elements), the performance of the task-based variants with a TLF of four is mostly on par with the reference implementation if TCMalloc is not available. Otherwise, similar performance can only be observed at 12 threads with the gap increasing again at 24 threads. It stands to reason that the inherent communication and synchronization within the scheduler and the resulting NUMA effects further degrade the overall performance.

It is notable, however, that the performance increases with a higher TLF. This is due to the larger number of tasks leading to improved load balancing among the threads: the master thread first creating the tasks and then executing its share of the taskloop leads to less imbalance in case of more fine-grained work distribution.

The difference between tasks and tasklets seen at the smallest problem size is non-existent at larger problem sizes. This matches the expectations from the discussions in Section 6.3.1: at larger problem sizes, the task workload becomes dominant and the task management overhead is negligible. Moreover, while the performance at larger problem sizes of the task-based variant matches the performance of the reference implementation at 12 threads, there is still a significant gap of about 30% for 100^3 elements at 24 threads. Nevertheless, in the following experiments, only a process per node with 24 threads has been chosen to maximize the hardware utilization.

6.6.3.2. Scaling the Number of Nodes

In the following experiments, the number of nodes is scaled from a single node to 125 nodes with a single process per node and 24 threads per process. The experiments have been conducted with 100^3, 150^3, 200^3, and 300^3 elements per process. The run-times together with the speedup of the DASH Task version over the reference implementation are depicted in Figure 6.18.

Starting with 100^3 elements, a picture emerges that is similar to the single-node experiments discussed in the previous section: while without TCMalloc the performance of the two variants is almost close (Figure 6.18a), a gap of around 30% emerges across the range of nodes if TCMalloc is enabled (Figure 6.18b). At 150^3 elements, the DASH Tasks variant already outperforms the reference implementation without TCMalloc (Figure 6.18c) and yields only a minor slowdown with TCMalloc enabled (Figure 6.18d).

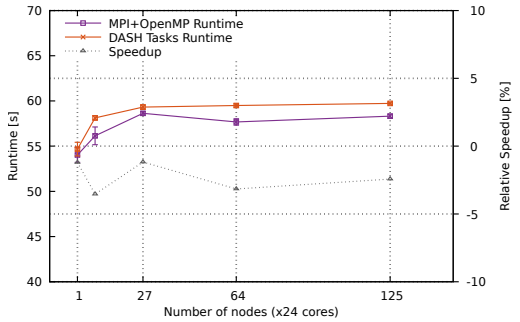
At 200^3 and 300^3 elements, the DASH Tasks variant yields a speedup of up to 7% at 125 nodes with TCMalloc enabled (Figure 6.18f and Figure 6.18h). Unfortunately, due to time constraints it was impossible to perform experiments with this configuration at larger node counts.

6.6.3.3. Large-scale Experiments

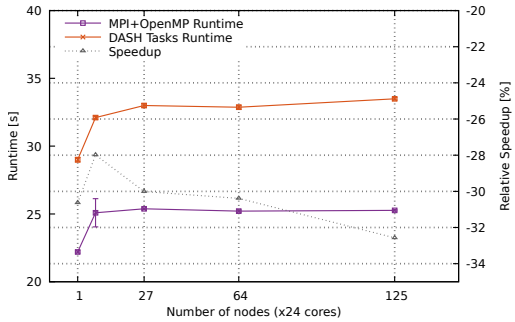
The following measurements were performed using an earlier version of Open MPI (v3.1.2) on the Cray system and were published in [64]. The results for runs up to 1000 nodes are depicted in Figure 6.19. While for 200^3 the same number of iterations have been measured, the runs with 300^3 elements computed 100 iterations instead of the previously discussed 30 iterations at that size. The performance of the DASH Tasks variant at both problem sizes matches the performance discussed in the previous section[37]. However, the performance of the reference implementation diverges significantly from what has been observed for 300^3 elements in the previous section. At 125 nodes, the run-time of the reference implementation reported in Figure 6.19b is about 30% slower than the run-time reported in Figure 6.18h.

This stark contrast can only be explained with a regression in Open MPI v3.1.2 that was fixed in later versions. The author of this work was successful in reproducing these numbers with the older version of Open MPI at the time the experiments in the previous section were conducted, which underpins the conclusion that this was

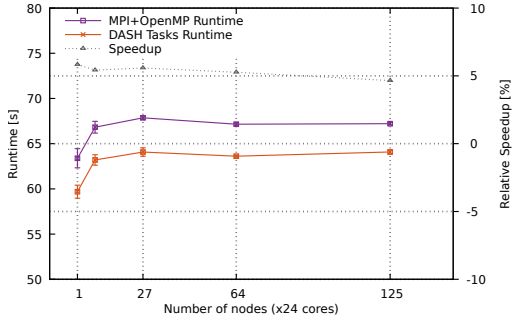
[37] For 300^3 elements the previously discussed run-time was about 82 s for 30 iterations (see Figure 6.18h). The run-time observed for 100 iterations in Figure 6.19b is 274 s, a factor of 3.3 that matches the difference in iterations.



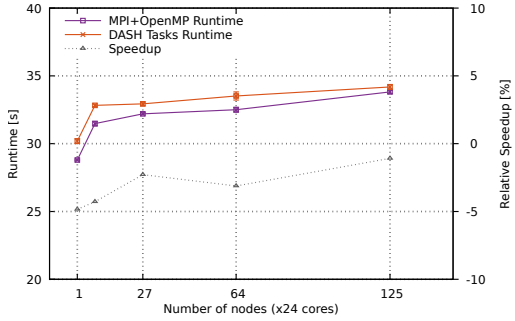
(a) 100^3 elements, without TCMalloc.



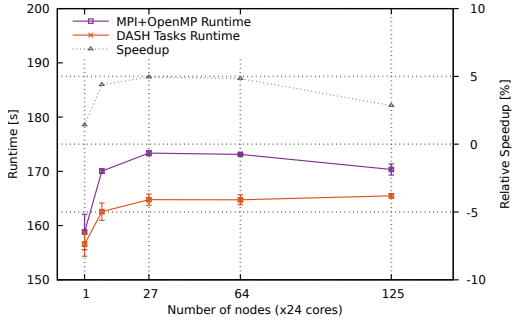
(b) 100^3 elements, with TCMalloc.



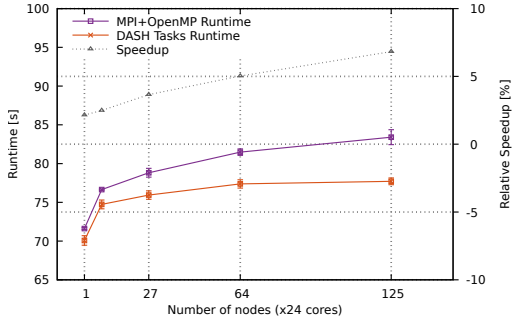
(c) 150^3 elements, without TCMalloc.



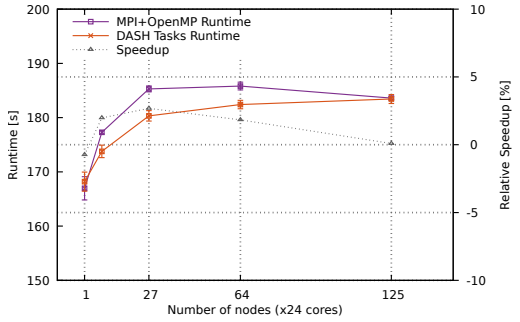
(d) 150^3 elements, with TCMalloc.



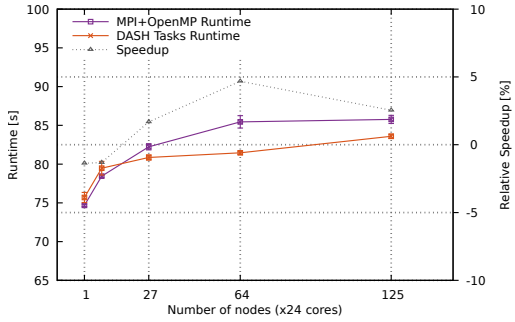
(e) 200^3 elements, without TCMalloc.



(f) 200^3 elements, with TCMalloc.

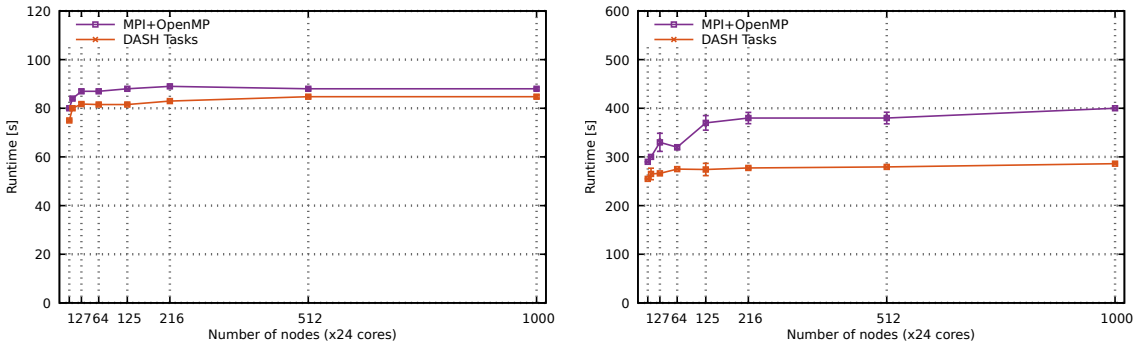


(g) 300^3 elements, without TCMalloc.



(h) 300^3 elements, with TCMalloc.

Figure 6.18.: Multi-node scaling of LULESH with different problem sizes on a Cray XC40. TLF is the *Task Loop Factor*, i.e., the oversubscription factor used in determining the number of tasks created in a taskloop. The runtime refers to the main y-axis. The speedup depicted is the speedup of the DASH task-based version over the reference implementation (secondary y-axis, negative values represent a slowdown). The standard deviation of the time measurements are plotted as error bars. The TLF was chosen as 8. The iteration count from top to bottom: 300, 100, 100, and 30.



(a) 200^3 elements (b) 300^3 elements

Figure 6.19.: Large-scale LULESH run-time for 100 iterations on the Cray XC40 using Open MPI version 3.1.2. The TLF was chosen as six. One process was used per node with 24 threads each. TCMalloc was used in both cases.

an issue in the mentioned Open MPI release. The run-times of the DASH Tasks are equivalent to the numbers presented in the previous section.

The run-times presented in this section should thus not be used to make general assumptions about the efficiency of the DASH Tasks implementation over the reference implementation but are instead an artifact of the MPI implementation used at the time. However, these measurements still serve as a demonstration that the run-time of the DASH application remains stable even up to 1000 nodes.

As mentioned earlier, it was impossible to repeat the measurements at this scale due to constraints on the availability of the machine.

6.6.4. Discussion

The results presented in this section confirm that the tasking model proposed in this work is applicable to complex real-world applications. Some of the observations made here confirm expectations described earlier, e.g., the benefit of tasklets over tasks in low-workload scenarios. At larger problem sizes, the tasking model is able to deliver a speedup of around 5% over the reference implementation even though not all optimization potential has been tapped. For example, the task-parallel handling of regions in LULESH could be implemented in a future version with the potential to further reduce imbalances. Moreover, the experiments presented here hint at an issue inside the scheduler with threads competing for work, especially when working across NUMA domains.

Nevertheless, the measurements have demonstrated that the proposed model can deliver sustained performance at scale when provided enough work. When constrained to a single NUMA domain, the break-even point seems to be located

at about 100^3 elements on that particular machine. It has also been demonstrated that even when running on a full node, the tasking approach is able to better compensate for significant outside noise such as the latencies caused by system memory management.

Chapter 7
Related Work

A number of node-local and distributed tasking systems have been proposed over the years, some of which have found traction in the community. The system proposed in this work builds on a number of previously published ideas but is different in some aspects from what has been proposed so far. As briefly touched on in Section 1.1, existing tasking systems can be coarsely classified into three categories: *process-local* task synchronization, *distributed locality-agnostic* task-based programming models, and *distributed locality-aware* models. While the scheme proposed in this section clearly falls into the latter category, models from the other categories are certainly relevant for comparison.

7.1. Process-local Tasking Models

In the category of *process-local* tasking systems, OpenMP [10] and OmpSs [56] (with its successor OmpSs-2 [126]) are likely the most prominent. Both are language extensions for Fortran, C, and C++ that allow users to annotate an application's source code to mark regions that are executed within the context of a task. While OpenMP had originally focused on work-sharing constructs (dividing the work of a loop into chunks processed by one or more threads), support for tasks has been added in version 3.0. The expression of data dependencies on local variables or memory locations serves to coordinate the relative order of execution between tasks. This implicit synchronization has been discussed in Section 1.1 and served as the inspiration for the data dependencies on global memory locations proposed in this work.

Cilk Plus extended the C and C++ languages with keywords to express concurrent execution, e.g., spawning asynchronous activities, loop-level parallelism, and array

notation for vectorization [129]. The Intel thread building blocks (TBB) provide a C++ interface that allows application developers to create a task graph by explicitly specifying nodes and the edges connecting them [130]. Nodes may communicate either directly (data-flow graph) or through shared memory (dependency graph) with each other, offer a high degree of flexibility to application developers. Additionally, TBB offers task-parallel loop constructs and thread-safe containers.

Since C++ 11, the language standard offers the concept of futures and promises that act as channels between tasks [131]. This allows for basic synchronization and communication among asynchronous activities spawned using `std::async`. A technical specification exists that extends the existing interface to allow for more complex relationships between futures, e.g., continuations that become ready as either one or all input futures become ready [132].

SuperGlue is a data-centric shared memory programming model that uses handles to memory locations to build a DAG of tasks [133]. Internally, the handles are versioned to determine the relative execution order of tasks. This versioning scheme, while not directly related to the phases used in this work, has been inspirational for this work.

7.2. Distributed Tasking Models

The HPX programming system provides an implementation of the Active Global Address Space (AGAS), which is a derivative of the PGAS paradigm [28]. Applications are expressed using asynchronous activities that communicate through futures with the ability to form complex synchronization patterns using continuations. HPX is generally a global view programming model in which algorithms are expressed independently of the available parallelism and the runtime system will distribute the work across the available resources.

Chapel is a high-level PGAS-based programming language developed by Cray [68]. Similar to HPX, it follows a global view model in which the algorithms and data structures are expressed independent of available parallelism. With a single thread coordinating the application's execution, constructs expressing concurrency distribute work among the available processes. Chapel, however, also supports lower-level SPMD-style programming constructs such as locality control. Structured and unstructured task parallelism is enabled through various language keywords with ad-hoc data-centric synchronization available through synchronization variables.

Charm++ provides a highly dynamic programming system in which so-called chares (C++ objects representing state and operations defined on it) are used for

over-decomposition of applications [134, 135]. The operations on the chares are invoked through messages sent between chares that are managed by the runtime system. Moreover, chares can be migrated between nodes, enabling both load-balance and the ability to recover from faults.

XcalableMP (XMP) is a language extension to C/C++ and Fortran for PGAS-based programming [136]. Similar to OpenMP, parallelism and data distribution are expressed through compiler pragmas. XMP supports both global and local view programming and an extension has been proposed to integrate task synchronization into the local view [137]. The synchronization primitives are modeled after two-sided message semantics with explicit specification of process/tag pairs in wait-/release statements, effectively introducing message semantics into an otherwise one-sided programming model.

The X10 PGAS language supports dynamic task creation, called asynchronous activities through the async-finish model [30]. Using a hierarchical model, `finish` statements are used to wait for the termination of tasks that are spawned locally or through remote task invocation. Tasks may be spawned locally or through remote task invocation and the `finish` statement is used to wait for completion of activities. X10 supports `futures` used to wait for the result of an activity. For dynamic (ad-hoc) synchronization, X10 initially offered so-called `clocks`, which offer barrier capabilities to a varying number of activities. The *phases* created by clocks are strict in the sense that *all* tasks in one phase of a clock have to be completed before tasks from the next phase may be executed. This is in contrast to the notion of phases used in this work, which represent a global ordering of tasks with dependencies between them.

As an extension to X10, *Phasers* are a concept to model more flexible ad-hoc synchronization mechanisms between tasks, including barrier and point-to-point synchronization [58]. A phaser acts as a broker of signals between tasks, allowing tasks to register as producers, consumers, or both. Similarly to X10 clocks, the set of tasks synchronized through phasers can dynamically grow and shrink, making it a versatile dynamic synchronization mechanism.

Habanero-Java is a derivative of X10 with advanced constructs for concurrency and work-stealing capabilities, with the async-finish model at its core [138]. An implementation of Habanero in the C programming language, which supports so-called Data Driven Tasks (DDT) are synchronized through Data Driven Futures (DDFs), which can be used for synchronization and communication between tasks (similar to futures in C++) [139]. It has been integrated with Unified Parallel C (UPC) PGAS programming model (Habanero-UPC), which among other things supported global work-stealing, as well as UPC++ (Habanero-UPC++), a C++ implementation

of UPC [140, 141]. UPC++ natively supports remote task invocation but lacks thread-parallel execution [27]. Synchronization of tasks across process boundaries in (Habanero-)UPC++ is thus only possible through ad-hoc synchronization mechanism (barriers, locks) or between tasks that originated at the same process.

Habanero was built on top of the Open Community Runtime (OCR) [142], which is an effort to provide a standardized low-level runtime system for task-based programming models. The OCR defines abstractions for tasks, data blocks, and events and requires all communication between tasks to be handled through these data blocks in order to allow for task migration and inter-process communication. However, this model is too restrictive for a tasking system such as the one proposed in this work, which relies on communication through global shared memory.

An integration of Habanero with the OpenSHMEM PGAS model [143] allows tasks to be blocked waiting for a state change in global memory (called the symmetric heap in OpenSHMEM) and to hide communication latencies by rescheduling blocked tasks [44]. Habanero has also been integrated with MPI into Habanero-C MPI (HCMPI) where MPI communication is handled asynchronously in the form of tasks and can be referenced by DDFs to synchronize tasks and MPI communication [144].

The PaRSEC programming system allows the expression of problems as DAGs using a custom language expressing dependencies between tasks, called JDF [29]. This global-view model has been the basis for the implementation of parallel linear algebra algorithms, called DPLASMA [116]. The PaRSEC runtime system manages the data storage of matrices and tiles as well as the necessary communication and synchronization between the tasks in the DAG that are executed in local and distributed memory. The code generated from the JDF description is translated into an application that implicitly contains the task graph in the form of a Parameterized Task Graph (PTG), through which the global task graph is discovered on-the-fly and not in advance. On top of this runtime system, a dynamic tasking interface has been implemented, called PaRSEC Dynamic Task Discovery (DTD) [51]. In this global-view model, all processes discover the global task graph in full in order to find all communication partners. Both PaRSEC PTG and DTD have been used as references in Section 6.4.

StarPU provides a distributed task-based programming model [31]. Similar to PaRSEC PTG, the StarPU Sequential Task Flow (STF) model offers a global-view expression of algorithms with the runtime system responsible to map tasks to processes and perform the necessary communication and synchronization [66]. Dependencies between tasks are expressed through data dependencies similar to the PaRSEC interface. A second model provides SPMD-style task-based programming through the use of so-called MPI tasks, which encapsulate MPI two-sided communication

and thus allow the scheduler to hide communication latency. In this model, local task dependencies are expressed through data dependencies while dependencies between tasks reaching across process boundaries are handled ad-hoc through the MPI tasks.

The Uintah framework is a high-level programming system for Adaptive Mesh Refinement (AMR) with a sophisticated runtime system [145]. Computation is expressed in the form of high-level abstract tasks with inputs and outputs defined by the user. From this description, the runtime system creates concrete tasks and infers both local and global (communication) dependencies. In order to avoid the discovery of the global task graph at each process, processes only discover their local task graph plus that of the neighboring processes to infer the required communication operations. The concept of phases has been introduced to avoid deadlocks in the use of collective MPI operations but these phases are independent of the dependency matching.

The Regent programming language is a high-level abstraction for applications using regions as a data representation, e.g., unstructured meshes [47]. The compiler infers global task dependencies on local and global memory regions and translates the high-level code into the Legion tasking runtime system, which uses hierarchical regions to form DAGs of tasks [146].

More general unidirectional approaches towards synchronization have been proposed in the context of PGAS programming models. Co-Array Fortran (CAF) is a Fortran-based PGAS language extension [147] that provides so-called events, which can be used to signal the availability of data and can be implemented using both MPI two-sided and one-sided communication [42]. In the context of MPI-RMA, notification schemes have been proposed that can be used to wait for remote write accesses [46, 45]. However, building complex task synchronization patterns (DAGs with a high number of edges) with these devices is tedious and error-prone since protecting data accesses requires signaling in both directions, i.e., to signal the availability of buffer space (after a previous data has been completed) and of data after a completed write. Such low-level signaling schemes can nevertheless be useful for the creation of more complex synchronization schemes, e.g., inside a distributed tasking library.

To the best of the author's knowledge, the work presented here is the first approach towards synchronization of tasks at a global scale using a local-view task discovery scheme and data dependencies in both the local and global address space.

Chapter 8

Conclusions and Future Work

This work presents a novel approach towards synchronization of tasks operating on data in the global address space. By employing a local-view programming model in which processes only discover the local portions of the task graph that will later be executed by this process, this approach avoids the potential scalability bottleneck of requiring all processes to discover the full global task graph. By extending the concept of task data dependencies to the global address space, a flexible model has been proposed that allows application developers to create complex and deadlock-free synchronization schemes without the need for manually managing synchronization objects such as futures or events. At its core, this model follows the separation of synchronization and communication that is at the heart of the PGAS programming paradigm: tasks are able to read from and write to any location in the local and global memory space during their execution without relying on communication activity of other tasks. In general, the coordination of task execution both at the local and global scale is determined based on the *expected* data-flow between tasks but independent of the actual communication operations. These dependencies are provided by the application developer in a way that is similar to the description of dependencies between tasks in the OpenMP tasking model.

In order to connect the locally discovered task graphs, the scheduler instances communicate dependency information with each other and extend the locally discovered task graph by integrating any received information of remote dependencies of tasks from other processes. In order to efficiently handle the inter-scheduler communication, a new message queue has been designed that is based on one-sided RMA operations and has the potential to outperform two-sided communication at scale, provided that the underlying hardware and software supports efficient handling of atomic memory operation.

An implementation of this model has been described in the context of the DASH PGAS library. Using this implementation, it was shown that the performance of two linear algebra algorithms (Cholesky and QR decomposition) is competitive with software packages that have been specifically designed for this type of workload. Moreover, it has been shown that global-view programming models may indeed suffer from the overheads imposed by global task graph discovery, a bottleneck that is avoided in the model proposed in this work.

Using one of the NASA Aerodynamic Simulation (NAS) parallel benchmarks, it was shown that the combination of PGAS and distributed task synchronization yields performance that is superior to traditional hybrid MPI+OpenMP worksharing and task-based implementations. Using a more complex application, LULESH 2.0, it was shown that while for small problem sizes traditional OpenMP worksharing constructs may be more efficient, larger problem sizes may benefit from the dynamic behavior of task-based approaches.

Future work may include the integration of features that allow breaking up the currently static mapping of tasks to processes. For example, by enabling tasks to be explicitly shipped to other processes—while maintaining the possible synchronization through dependencies—it may be easier to implement more dynamic algorithms for which the definition of dependencies is not suitable, e.g., dynamic distributed data structures such as hash tables.

Furthermore, an investigation into dynamic load balancing approaches may improve the performance of unbalanced applications such as LULESH. However, in order to allow for the automatic transfer of tasks to other processes, the set of operations tasks are allowed to perform may have to be restricted to global memory accesses, which in turn may slow down parts of a task's execution due to indirect memory accesses.

Both the OmpSs and the OpenMP programming model offer advanced dependency types beyond input and output dependencies, e.g., concurrent and mutually exclusive execution. While concurrent execution is mostly a syntactic refinement that can be achieved using input and output dependencies, the use of the mutually exclusive dependency type can alleviate the need for atomic operations or even locking mechanisms in tasks that need mutually exclusive access to local or global memory. The inclusion of both types would help application developers to better express their intent, leaving it to the runtime system to implement efficient mechanisms to fulfill the required semantics.

This work has also laid out suggestions for additions and changes to the RMA part of the MPI standard that would help improve both the performance and usability of the RMA interface. For some of these suggestions, a discussion among the members

of the MPI forum has been started to find ways for incorporating them into the standard. While this work has provided the motivation, their full implementation and discussion in the standardization body remains as future work.

Overall, the results presented in this work show that a local-view tasking-based programming system using data dependencies at a global scale can provide competitive performance compared to (and for some applications outperforms) traditional hybrid MPI+OpenMP applications. The local-view approach taken in this work makes porting of existing MPI applications easier by avoiding the transition to a global-view programming model. Using data dependencies for the synchronization of tasks executing in the global memory space follows the path of the well-established OpenMP programming model while avoiding the intricacies connected to the synchronization required in the PGAS programming model. At the same time, the flexibility of random access capabilities in the global memory space is retained, as is the one-sided approach to communication inherent to the PGAS model. This, in turn, makes both task-based programming and the PGAS model easier to use for application developers, allowing scientific and engineering application to exploit the benefits of both approaches and ultimately leading to higher efficiency in the use of computational resources provided on today's and future HPC machines.

Bibliography

[1] W. E. Nagel, D. B. Kröner, M. M. Resch, eds. *High Performance Computing in Science and Engineering '18*. Transactions of the High Performance Computing Center, Stuttgart (HLRS). Springer International Publishing, 2018. DOI: 10.1007/978-3-030-13325-2 (cit. on p. 29).

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, A. R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511 (cit. on p. 29).

[3] M. Bohr. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper." In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 11–13. DOI: 10.1109/N-SSC.2007.4785534 (cit. on p. 29).

[4] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860 (cit. on p. 29).

[5] D. Unat, J. Shalf, T. Hoefler, T. Schulthess, A. D. (Editors), et al. *Programming Abstractions for Data Locality*. Tech. rep. Lugano, Switzerland, Apr. 2014 (cit. on p. 29).

[6] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, M. Pericás. "Trends in Data Locality Abstractions for HPC Systems." In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (Oct. 2017), pp. 3007–3020. DOI: 10.1109/TPDS.2017.2703149 (cit. on p. 29).

[7] *MPI: A Message-Passing Interface Standard*. Tech. rep. June 2015. URL: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on 04/09/2019) (cit. on pp. 30, 68, 70, 110, 177, 178).

[8] *Exascale Computing Project*. URL: https://www.exascaleproject.org/ (visited on 03/14/2020) (cit. on p. 30).

[9] D. E. Bernholdt, S. Boehm, G. Bosilca, M. G. Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, G. R. Vallee. "A Survey of MPI Usage in the US Exascale Computing Project." In: *Concurrency Computation: Practice and Experience* (Sept. 2018). DOI: https://doi.org/10.1002/cpe.4851 (cit. on pp. 30, 37).

[10] *OpenMP Application Programming Interface, Version 5.0*. OpenMP Architecture Review Board. Nov. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (visited on 03/16/2020) (cit. on pp. 30, 50, 91, 141).

[11] B. Chapman, G. Jost, R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007 (cit. on p. 30).

[12] J. M. Bull, J. Enright, X. Guo, C. Maynard, F. Reid. "Performance Evaluation of Mixed-Mode OpenMP/MPI Implementations." In: *International Journal of Parallel Programming* 38.5 (Oct. 2010), pp. 396–417. DOI: 10.1007/s10766-010-0137-2 (cit. on p. 30).

[13] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, B. Chapman. "High Performance Computing Using MPI and OpenMP on Multi-core Parallel Systems." In: *Parallel Comput.* 37.9 (Sept. 2011), pp. 562–575. DOI: 10.1016/j.parco.2011.02.002 (cit. on p. 30).

[14] M. J. Chorley, D. W. Walker. "Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters." In: *Journal of Computational Science* 1.3 (2010), pp. 168–174. DOI: https://doi.org/10.1016/j.jocs.2010.05.001 (cit. on p. 30).

[15] L. G. Valiant. "A Bridging Model for Parallel Computation." In: *Commun. ACM* 33.8 (Aug. 1990). DOI: 10.1145/79173.79181 (cit. on p. 30).

[16] C. Boneti, R. Gioiosa, F. Cazorla, M. Valero. "Using Hardware Resource Allocation to Balance HPC Applications." In: *Parallel and Distributed Computing*. Ed. by A. Ros. Jan. 2010. DOI: 10.5772/9447 (cit. on p. 30).

[17] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, M. Valero. "A Quantitative Analysis of OS Noise." In: *2011 IEEE International Parallel Distributed Processing Symposium*. May 2011, pp. 852–863. DOI: 10.1109/IPDPS.2011.84 (cit. on p. 30).

[18] M. Small, X. Yuan. "Maximizing MPI Point-to-point Communication Performance on RDMA-enabled Clusters with Customized Protocols." In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09. ACM, 2009, pp. 306–315. DOI: 10.1145/1542275.1542320 (cit. on p. 31).

[19] G. Amerson, A. Apon. “Implementation and design analysis of a network messaging module using virtual interface architecture.” In: *2004 IEEE International Conference on Cluster Computing*. Sept. 2004, pp. 255–265. DOI: 10.1109/CLUSTR.2004.1392623 (cit. on p. 31).

[20] S. Majumder, S. Rixner. “An Event-driven Architecture for MPI Libraries.” In: *In Proceedings of the 2004 Los Alamos Computer Science Institute Symposium*. 2004 (cit. on p. 31).

[21] C. Keppitiyagama, A. Wagner. “Asynchronous MPI messaging on Myrinet.” In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. Apr. 2001. DOI: 10.1109/IPDPS.2001.924989 (cit. on p. 31).

[22] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, F. W. Atos. “The BXI Interconnect Architecture.” In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. Aug. 2015, pp. 18–25. DOI: 10.1109/HOTI.2015.15 (cit. on p. 31).

[23] G. Almasi. “PGAS (Partitioned Global Address Space) Languages.” In: *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1539–1545 (cit. on p. 32).

[24] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, T. Wen. “Productivity and Performance Using Partitioned Global Address Space Languages.” In: *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*. PASCO ’07. ACM, 2007, pp. 24–32 (cit. on p. 32).

[25] G. M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.” In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560 (cit. on p. 33).

[26] M. A. Khaleel. *Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale*. Tech. rep. Feb. 2011. URL: https://digital.library.unt.edu/ark:/67531/metadc841613 (visited on 03/14/2020) (cit. on p. 33).

[27] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, K. Yelick. “UPC++: A PGAS Extension for C++.” In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. May 2014, pp. 1105–1114. DOI: 10.1109/IPDPS.2014.115 (cit. on pp. 33, 144).

[28] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey. “HPX: A Task Based Programming Model in a Global Address Space.” In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS ’14. ACM, 2014, 6:1–6:11. DOI: 10.1145/2676870.2676883 (cit. on pp. 33, 41, 142).

[29] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra. “DAGuE: A Generic Distributed DAG Engine for High Performance Computing.” In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. May 2011, pp. 1151–1158. DOI: 10.1109/IPDPS.2011.281 (cit. on pp. 33, 50, 102, 103, 144).

[30] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar. “X10: An Object-oriented Approach to Non-uniform Cluster Computing.” In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’05. ACM, 2005. DOI: 10.1145/1094811.1094852 (cit. on pp. 33, 143).

[31] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures.” In: *Concurrent Computing: Practice and Experience* 23.2 (Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631 (cit. on pp. 33, 103, 144).

[32] F. Darema. “The SPMD Model: Past, Present and Future.” In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Y. Cotronis, J. Dongarra. Springer Berlin Heidelberg, 2001 (cit. on p. 33).

[33] D. Molka, D. Hackenberg, R. Schöne, W. E. Nagel. “Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture.” In: *2015 44th International Conference on Parallel Processing*. Sept. 2015, pp. 739–748. DOI: 10.1109/ICPP.2015.83 (cit. on p. 35).

[34] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, P. Beckman. “Argobots: A Lightweight Low-Level Threading and Tasking Framework.” In: *IEEE Transactions on Parallel and Distributed Systems* 29.3 (Mar. 2018), pp. 512–526. DOI: 10.1109/TPDS.2017.2766062 (cit. on pp. 35, 91, 99).

[35] A. Shpiner, E. Zahavi, O. Dahley, A. Barnea, R. Damsker, G. Yekelis, M. Zus, E. Kuta, D. Baram. “RoCE Rocks Without PFC: Detailed Evaluation.” In: *Proceedings of the Workshop on Kernel-Bypass Networks*. KBNets ’17. ACM, 2017, pp. 25–30. DOI: 10.1145/3098583.3098588 (cit. on p. 36).

[36] C. Barthels, S. Loesing, G. Alonso, D. Kossmann. “Rack-Scale In-Memory Join Processing Using RDMA.” In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. ACM, 2015. DOI: 10.1145/2723372.2750547 (cit. on p. 37).

[37] K. Fuerlinger, T. Fuchs, R. Kowalewski. "DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms." In: *2016 IEEE 18th International Conference on High Performance Computing and Communications*. Oct. 2016. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0140 (cit. on pp. 37, 87, 129).

[38] J. Schuchart, R. Kowalewski, K. Fuerlinger. "Recent Experiences in Using MPI-3 RMA in the DASH PGAS Runtime." In: *Proceedings of Workshops of HPC Asia*. HPC Asia '18. ACM, 2018. DOI: 10.1145/3176364.3176367 (cit. on p. 37).

[39] P. Schmid, M. Besta, T. Hoefler. "High-Performance Distributed RMA Locks." In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '16. ACM, 2016, pp. 19–30. DOI: 10.1145/2907294.2907323 (cit. on pp. 39, 70).

[40] A. Gómez-Iglesias, D. Pekurovsky, K. Hamidouche, J. Zhang, J. Vienne. "Porting Scientific Libraries to PGAS in XSEDE Resources: Practice and Experience." In: *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. XSEDE '15. ACM, 2015, 40:1–40:7. DOI: 10.1145/2792745.2792785 (cit. on p. 39).

[41] A. Dragojević, D. Narayanan, O. Hodson, M. Castro. "FaRM: Fast Remote Memory." In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. USENIX Association, 2014, pp. 401–414 (cit. on p. 39).

[42] A. Fanfarillo, J. Hammond. "CAF Events Implementation Using MPI-3 Capabilities." In: *Proceedings of the 23rd European MPI Users' Group Meeting*. EuroMPI 2016. ACM, 2016. DOI: 10.1145/2966884.2966916 (cit. on pp. 40, 145).

[43] *OpenSHMEM Application Programming Interface Version 1.4*. Open Source Software Solutions, Inc. Dec. 2017. URL: www.openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf (visited on 04/09/2019) (cit. on pp. 40, 84).

[44] M. Grossman, V. Kumar, Z. Budimlić, V. Sarkar. "Integrating Asynchronous Task Parallelism with OpenSHMEM." In: *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments*. Ed. by M. Gorentla Venkata, N. Imam, S. Pophale, T. M. Mintz. Springer International Publishing, 2016, pp. 3–17. DOI: 10.1007/978-3-319-50995-2_1 (cit. on pp. 40, 144).

[45] M. Sergent, C. T. Aitkaci, P. Lemarinier, G. Papauré. "Efficient Notifications for MPI One-sided Applications." In: *Proceedings of the 26th European MPI Users' Group Meeting*. EuroMPI '19. ACM, 2019, 5:1–5:10. DOI: 10.1145/3343211.3343216 (cit. on pp. 40, 145).

[46] R. Belli, T. Hoefler. “Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization.” In: *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*. IPDPS ’15. IEEE Computer Society, 2015, 871–881. DOI: 10.1109/IPDPS.2015.30 (cit. on pp. 40, 145).

[47] E. Slaughter, W. Lee, S. Treichler, M. Bauer, A. Aiken. “Regent: a high-productivity programming language for HPC with logical regions.” In: *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2015, pp. 1–12. DOI: 10.1145/2807591.2807629 (cit. on pp. 41, 130, 145).

[48] L. A. Meyerovich, A. S. Rabkin. “Empirical Analysis of Programming Language Adoption.” In: *SIGPLAN Not.* 48.10 (Oct. 2013), pp. 1–18. DOI: 10.1145/2544173.2509515 (cit. on p. 41).

[49] N. Gustafsson, A. Laksberg, H. Sutter, S. Mithani. *N3857: Improvements to std::future<T> and Related APIs*. Tech. rep. Jan. 2014 (cit. on p. 41).

[50] J. Bachan, D. Bonachea, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, B. van Straalen, S. B. Baden. “The UPC++ PGAS Library for Exascale Computing.” In: *Proceedings of the Second Annual PGAS Applications Workshop*. PAW17. ACM, 2017, 7:1–7:4. DOI: 10.1145/3144779.3169108 (cit. on p. 41).

[51] R. Hoque, T. Herault, G. Bosilca, J. Dongarra. “Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime.” In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ScalA ’17. ACM, 2017, 6:1–6:8. DOI: 10.1145/3148226.3148233 (cit. on pp. 41, 103, 144).

[52] J. M. Pérez, R. M. Badia, J. Labarta. “A flexible and portable programming model for SMP and multi-cores.” In: *Barcelona Supercomputing Center, Barselona, Technical report* 3 (2007) (cit. on p. 42).

[53] J. E. Smith, G. S. Sohi. “The microarchitecture of superscalar processors.” In: *Proceedings of the IEEE* 83.12 (Dec. 1995), pp. 1609–1624. DOI: 10.1109/5.476078 (cit. on p. 42).

[54] *OpenMP Application Programming Interface, Version 4.5*. OpenMP Architecture Review Board. Nov. 2015. URL: http://www.openmp.org/mp-documents/openmp-4.5.pdf (visited on 03/16/2020) (cit. on p. 42).

[55] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, G. Zhang. “The Design of OpenMP Tasks.” In: 20.3 (Mar. 2009), pp. 404–418. DOI: 10.1109/TPDS.2008.105 (cit. on p. 42).

[56] A. Duran, A. Eduard, R. M. Badia, J. Larbarta, L. Martinell, X. Martorell, J. Plana. "OmpSs: A proposal for programming heterogeneous multi-core architectures." In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193. DOI: 10.1142/S0129626411000151 (cit. on pp. 42, 141).

[57] S. Brinkmann, J. Gracia, C. Niethammer, R. Keller. "TEMANEJO - a debugger for task based parallel programming models." In: *Applications, Tools and Techniques on the Road to Exascale Computing – Proceeding of the International Conference on Parallel Computing (ParCO)*. Advances in Parallel Computing (2011). URL: http://arxiv.org/abs/1112.4604 (visited on 03/18/2020) (cit. on pp. 42, 95).

[58] J. Shirako, D. M. Peixotto, V. Sarkar, W. N. Scherer. "Phasers: A Unified Deadlock-free Construct for Collective and Point-to-point Synchronization." In: *Proceedings of the 22Nd Annual International Conference on Supercomputing*. ICS '08. ACM, 2008. DOI: 10.1145/1375527.1375568 (cit. on pp. 42, 143).

[59] A. Buttari, J. Langou, J. Kurzak, J. Dongarra. "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures." In: *Parallel Computing* 35 (Jan. 2009), pp. 38–53. DOI: 10.1016/j.parco.2008.10.002 (cit. on pp. 43, 102).

[60] R. L. Meakin. "Composite Overset Structured Grids." In: *Handbook of Grid Generation*. Ed. by J. F. Thompson, B. K. Soni, N. P. Weatherill. 1998 (cit. on pp. 44, 115).

[61] D. Schwamborn, T. Gerhold, R. Heinrich. "The DLR TAU-code: Recent applications in research and industry." In: *ECCOMAS CFD*. Jan. 2006 (cit. on pp. 44, 115).

[62] M. J. Djomehri, R. Biswas, M. Potsdam, R. C. Strawn. *An analysis of performance enhancement techniques for overset grid applications*. Tech. rep. NAS-03-008. NASA Advanced Supercomputing (NAS) Division, 2003 (cit. on pp. 44, 115).

[63] *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory*. Tech. rep. LLNL-TR-490254. Livermore, CA. URL: https://computing.llnl.gov/projects/co-design/spec-7.pdf (visited on 03/18/2020) (cit. on pp. 44, 128, 129).

[64] J. Schuchart, J. Gracia. "Global Task Data-Dependencies in PGAS Applications." In: *High Performance Computing*. Ed. by M. Weiland, G. Juckeland, C. Trinitis, P. Sadayappan. Springer International Publishing, 2019 (cit. on pp. 47, 136).

[65] J. Järvi, J. Freeman. "C++ lambda expressions and closures." In: *Science of Computer Programming* (2010). Special Issue on Object-Oriented Programming Languages and Systems (OOPS 2008), A Special Track at the 23rd ACM Symposium on Applied Computing, pp. 762 –772. DOI: 10.1016/j.scico.2009.04.003 (cit. on p. 48).

[66] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S. Thibault. *Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model*. Research Report RR-8927. June 2016. URL: https://hal.inria.fr/hal-01332774/file/RR-8927.pdf (visited on 04/09/2019) (cit. on pp. 50, 144).

[67] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov. "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs." In: *GPU Computing Gems*. Ed. by W. mei W. Hwu. Vol. 2. Morgan Kaufmann, Sept. 2010. URL: https://hal.inria.fr/inria-00547847/file/gpucomputinggems_plagma.pdf (visited on 04/09/2019) (cit. on pp. 50, 103).

[68] B. L. Chamberlain, D. Callahan, H. P. Zima. "Parallel Programmability and the Chapel Language." In: *International Journal of High Performance Computing Applications* 21 (3 Aug. 2007), pp. 291–312 (cit. on pp. 50, 142).

[69] Y. Robert. "Task Graph Scheduling." In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Springer US, 2011, pp. 2013–2025. DOI: 10.1007/978-0-387-09766-4_42 (cit. on p. 55).

[70] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. "Active Messages: A Mechanism for Integrated Communication and Computation." In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ISCA '92. ACM, 1992, pp. 256–266. DOI: 10.1145/139669.140382 (cit. on p. 65).

[71] D. Bonachea. "GASNet Specification, v1." In: *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207* (2002) (cit. on p. 65).

[72] D. Bonachea. *AMMPI: Active Messages over MPI*. URL: https://gasnet.lbl.gov/ammpi/ (visited on 05/06/2019) (cit. on p. 65).

[73] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, R. Thakur. "An Implementation and Evaluation of the MPI 3.0 One-sided Communication Interface." In: *Concurr. Comput. : Pract. Exper.* (Dec. 2016), pp. 4385–4404. DOI: 10.1002/cpe.3758 (cit. on p. 65).

[74] D. Bonachea, P. H. Hargrove. "GASNet-EX: A High-Performance, Portable Communication Library for Exascale." In: (Oct. 2018). DOI: 10.25344/S4QP4W (cit. on p. 65).

[75] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, D. Arnold. "Improving MPI Multi-threaded RMA Communication Performance." In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. ACM, 2018, 58:1–58:11. DOI: 10.1145/3225058.3225114 (cit. on pp. 65, 85).

[76] J. J. Willcock, T. Hoefler, N. G. Edmonds, A. Lumsdaine. "AM++: A Generalized Active Message Framework." In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. ACM, 2010, pp. 401–410. DOI: 10.1145/1854273.1854323 (cit. on pp. 65, 66).

[77] A. M. Mainwaring, D. E. Culler. *Active Message Applications Programming Interface and Communication Subsystem Organization*. Tech. rep. EECS Department, University of California, Berkeley, Oct. 1996. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1996/5768.html (visited on 05/06/2019) (cit. on p. 66).

[78] X. Zhao, P. Balaji, W. Gropp, R. Thakur. "MPI-Interoperable Generalized Active Messages." In: *2013 International Conference on Parallel and Distributed Systems*. Dec. 2013, pp. 200–207. DOI: 10.1109/ICPADS.2013.38 (cit. on p. 66).

[79] S. Jana, T. Curtis, D. Khaldi, B. Chapman. "Increasing Computational Asynchrony in OpenSHMEM with Active Messages." In: *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments*. Ed. by M. Gorentla Venkata, N. Imam, S. Pophale, T. M. Mintz. Cham: Springer International Publishing, 2016, pp. 35–51 (cit. on p. 66).

[80] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, B. Steinmacher-Burrow. "PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer." In: *IEEE 26th IPDPS*. May 2012, pp. 763–773. DOI: 10.1109/IPDPS.2012.73 (cit. on p. 66).

[81] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, C. J. Archer. "The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer." In: *Proceedings of the* 22^{nd} *Annual International Conference on Supercomputing*. ICS '08. ACM, 2008, pp. 94–103. DOI: 10.1145/1375527.1375544 (cit. on p. 66).

[82] P. Balaji. *Generalized RMA Atomics*. PR. MPI Forum, 2018. URL: https://github.com/mpi-forum/mpi-standard/pull/93 (visited on 04/09/2019) (cit. on pp. 69, 76).

[83] *Manual page: pthread_rwlock_rdlock*. The Open Group Base Specifications 7. IEEE and The Open Group. 2018. URL: http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_rwlock_rdlock.html (visited on 04/29/2019) (cit. on p. 69).

[84] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, N. Zeldovich. *Non-scalable locks are dangerous*. 2012. URL: https://pdos.csail.mit.edu/papers/linux:lock.pdf (visited on 05/06/2019) (cit. on p. 70).

[85] G. Barnes. “A Method for Implementing Lock-free Shared-data Structures.” In: *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’93. ACM, 1993, pp. 261–270. DOI: 10.1145/165231.165265 (cit. on p. 72).

[86] G. L. Peterson. “Concurrent Reading While Writing.” In: *ACM Trans. Program. Lang. Syst.* (Jan. 1983), pp. 46–55. DOI: 10.1145/357195.357198 (cit. on p. 72).

[87] G. L. Peterson, J. E. Burns. “Concurrent reading while writing II: The multi-writer case.” In: *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. Oct. 1987, pp. 383–392. DOI: 10.1109/SFCS.1987.15 (cit. on p. 72).

[88] M. P. Herlihy. “Impossibility and Universality Results for Wait-free Synchronization.” In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. PODC ’88. Toronto, Ontario, Canada: ACM, 1988, pp. 276–290. DOI: 10.1145/62546.62593. URL: http://doi.acm.org/10.1145/62546.62593 (cit. on p. 72).

[89] M. Herlihy, J. E. B. Moss. “Transactional Memory: Architectural Support for Lock-free Data Structures.” In: *SIGARCH Comput. Archit. News* 21.2 (May 1993), pp. 289–300. DOI: 10.1145/173682.165164 (cit. on p. 72).

[90] H. Sundell. “Efficient and practical non-blocking data structures.” PhD thesis. Göteborg: Chalmers University of Technology and Göteborg University, 2004 (cit. on p. 72).

[91] J. D. Valois. “Lock-free Data Structures.” UMI Order No. GAX95-44082. PhD thesis. 1996 (cit. on p. 72).

[92] A. Fogg. *4. Instruction tables – Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. Aug. 2019. URL: https://www.agner.org/optimize/instruction_tables.pdf (visited on 10/06/2019) (cit. on p. 72).

[93] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, A. Selikhov. “MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes.” In: *SC ’02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Nov. 2002, pp. 29–29. DOI: 10.1109/SC.2002.10048 (cit. on p. 73).

[94] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, J. Dongarra. “Post-failure recovery of MPI communication capability: Design and rationale.” In: *The International Journal of High Performance Computing Applications* (2013), pp. 244–254. DOI: 10.1177/1094342013488238 (cit. on p. 73).

[95] Gengbin Zheng, Lixia Shi, L. V. Kale. "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI." In: *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935)*. Sept. 2004, pp. 93–103. DOI: 10.1109/CLUSTR.2004.1392606 (cit. on p. 73).

[96] Jiuxing Liu, B. Chandrasekaran, Weikuan Yu, Jiesheng Wu, D. Buntinas, S. Kini, P. Wyckoff, D. K. Panda. "Micro-benchmark level performance comparison of high-speed cluster interconnects." In: *11th Symposium on High Performance Interconnects, 2003. Proceedings.* Aug. 2003, pp. 60–65. DOI: 10.1109/CONECT.2003.1231479 (cit. on p. 74).

[97] B. Alverson, E. Froese, L. Kaplan, D. Roweth. *Cray XC Series Network*. Tech. rep. Cray Inc., 2012. URL: www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf (visited on 04/09/2019) (cit. on pp. 74, 82, 84).

[98] R. Latham, W. Gropp, R. Ross, R. Thakur. "Extending the MPI-2 Generalized Request Interface." In: *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. PVM/MPI'07. Springer-Verlag, 2007 (cit. on p. 80).

[99] J.-B. Besnard, J. Jaeger, A. D. Malony, S. Shende, H. Taboada, M. Pérache, P. Carribault. "Mixing Ranks, Tasks, Progress and Nonblocking Collectives." In: *Proceedings of the 26th European MPI Users' Group Meeting*. EuroMPI '19. ACM, 2019, 10:1–10:10. DOI: 10.1145/3343211.3343221 (cit. on p. 80).

[100] W. P. McCartney, N. Sridhar. "Stackless preemptive multi-threading for TinyOS." In: *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*. June 2011, pp. 1–8. DOI: 10.1109/DCOSS.2011.5982136 (cit. on p. 80).

[101] Mellanox Technologies. *Connect-IB: Architecture for Scalable High Performance Computing*. Tech. rep. 2013. URL: http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf (visited on 03/12/2020) (cit. on p. 82).

[102] Unified Communication Framework Consortium. *UCX: Unified Communication X API Standard v1.6*. Unified Communication Framework Consortium. 2019. URL: https://github.com/openucx/ucx/wiki/api-doc/v1.6/ucx-v1.6.pdf (visited on 04/09/2019) (cit. on pp. 84, 85).

[103] M. Flajslik, J. Dinan. "On the Fence: An Offload Approach to Ordering One-Sided Communication." In: *2015 9th International Conference on Partitioned Global Address Space Programming Models*. Sept. 2015, pp. 1–12. DOI: 10.1109/PGAS.2015.9 (cit. on p. 85).

[104] T. Patinyasakdikul, D. Eberius, G. Bosilca, N. Hjelm. “Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs.” In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2019, pp. 1–11. DOI: 10.1109/CLUSTER.2019.8891015 (cit. on p. 85).

[105] J. Dinan, M. Flajslik. “Contexts: A Mechanism for High Throughput Communication in OpenSHMEM.” In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS ’14. ACM, 2014, 10:1–10:9. DOI: 10.1145/2676870.2676872 (cit. on p. 85).

[106] A. Bouteiller, S. Pophale, S. Boehm, M. B. Baker, M. G. Venkata. “Evaluating Contexts in OpenSHMEM-X Reference Implementation.” In: *OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence*. Ed. by M. Gorentla Venkata, N. Imam, S. Pophale. Springer International Publishing, 2018, pp. 50–62 (cit. on p. 85).

[107] Cray Inc. *XC^{TM} Series GNI and DMAPP API User Guide (CLE 6.0.UP05) S-2446*. URL: https://pubs.cray.com/content/S-2446/CLE%206.0.UP05/xctm-series-gni-and-dmapp-api-user-guide (visited on 12/09/2019) (cit. on p. 86).

[108] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, J. M. Squyres. “A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency.” In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. Aug. 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19 (cit. on p. 86).

[109] J. Schuchart, K. Tsugane, J. Gracia, M. Sato. “The Impact of Taskyield on the Design of Tasks Communicating Through MPI.” In: *Evolving OpenMP for Evolving Architectures*. Ed. by B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, J. Labarta. Awarded Best Paper. Springer International Publishing, 2018, pp. 3–17. DOI: 10.1007/978-3-319-98521-3_1 (cit. on pp. 90, 119).

[110] The Open Group Base. *getcontext, setcontext - get and set current user context*. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/getcontext.html (visited on 02/14/2020) (cit. on pp. 91, 99).

[111] K. B. Wheeler, R. C. Murphy, D. Thain. “Qthreads: An API for programming with millions of lightweight threads.” In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. Apr. 2008. DOI: 10.1109/IPDPS.2008.4536359 (cit. on p. 91).

[112] P. Franczak. “Design and Proto-typical Implementation of an Analysis Tool Interface for a Task-based PGAS Runtime.” Supervised by the author of this work. Master Thesis. Universität Stuttgart, Dec. 2019 (cit. on p. 95).

[113] J. Schuchart, M. Nachtmann, J. Gracia. "Patterns for OpenMP Task Data Dependency Overhead Measurements." In: *Scaling OpenMP for Exascale Performance and Portability*. Ed. by B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, M. S. Müller. Springer International Publishing, 2017, pp. 156–168 (cit. on pp. 100, 102).

[114] J. Choi, J. J. Dongarra, R. Pozo, D. W. Walker. "ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers." In: *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*. Oct. 1992, pp. 120–127. DOI: 10.1109/FMPC.1992.234898 (cit. on p. 102).

[115] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley. "An Updated Set of Basic Linear Algebra Subprograms (BLAS)." In: *ACM Trans. Math. Softw.* 28.2 (June 2002), pp. 135–151. DOI: 10.1145/567806.567807 (cit. on p. 102).

[116] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, J. Dongarra. "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA." In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. May 2011. DOI: 10.1109/IPDPS.2011.299 (cit. on pp. 103, 104, 112, 144).

[117] I. national de recherche en sciences et technologies du numérique (INRIA). *Chameleon—A dense linear algebra software for heterogeneous architectures*. URL: https://project.inria.fr/chameleon/ (visited on 03/08/2020) (cit. on p. 103).

[118] G. Quintana-Orti, E. S. Quintana-Orti, E. Chan, R. A. v. d. Geijn, F. G. V. Zee. "Scheduling of QR Factorization Algorithms on SMP and Multi-Core Architectures." In: *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. Feb. 2008, pp. 301–310. DOI: 10.1109/PDP.2008.37 (cit. on p. 111).

[119] F. Song, H. Ltaief, B. Hadri, J. Dongarra. "Scalable Tile Communication-Avoiding QR Factorization on Multicore Cluster Systems." In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2010, pp. 1–11. DOI: 10.1109/SC.2010.48 (cit. on p. 112).

[120] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al. "The NAS Parallel Benchmarks." In: *International Journal on High Performance Computing Applications* 5.3 (Sept. 1991), pp. 63–73. DOI: 10.1177/109434209100500306 (cit. on p. 114).

[121] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0*. Tech. rep. NAS-95-020. NASA Advanced Supercomputing (NAS) Division, 1995 (cit. on p. 115).

[122] H. Jin, M. Frumkin, J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and its Performance*. Tech. rep. NAS-99-011. NASA Advanced Supercomputing (NAS) Division, 1999 (cit. on p. 115).

[123] R. F. V. der Wijngaart, H. Jin. *NAS Parallel Benchmarks, Multi-Zone Versions*. Tech. rep. NAS-03-010. NASA Advanced Supercomputing (NAS) Division, 2003 (cit. on p. 115).

[124] H. Jin, R. F. Van der Wijngaart. "Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks." In: *Journal on Parallel and Distributed Computing* 66.5 (May 2006), 674—685. DOI: 10.1016/j.jpdc.2005.06.016 (cit. on p. 115).

[125] NASA Advanced Supercomputing Division. *Problem Sizes and Parameters in NAS Parallel Benchmarks*. URL: https://www.nas.nasa.gov/publications/npb_problem_sizes.html (visited on 02/13/2020) (cit. on p. 116).

[126] B. S. Center. *OmpSs-2 Specification*. 2019. URL: https://pm.bsc.es/ftp/ompss-2/doc/spec (visited on 02/14/2020) (cit. on pp. 120, 141).

[127] K. Sala, X. Teruel, J. M. Perez, A. J. Peña, V. Beltran, J. Labarta. "Integrating blocking and non-blocking MPI primitives with task-based programming models." In: *Parallel Computing* 85 (July 2019), 153–166. DOI: 10.1016/j.parco.2018.12.008 (cit. on pp. 120, 178).

[128] I. Karlin, J. Keasler, R. Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, Aug. 2013. URL: https://computing.llnl.gov/projects/co-design/lulesh2.0_changes1.pdf (visited on 03/18/2020) (cit. on pp. 128, 130).

[129] A. D. Robison. "Composable Parallel Patterns with Intel Cilk Plus." In: *Computing in Science Engineering* 15.2 (Mar. 2013), pp. 66–71. DOI: 10.1109/MCSE.2013.21 (cit. on p. 142).

[130] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly & Associates, 2007 (cit. on p. 142).

[131] *ISO/IEC TS 19571:2016: Programming Languages – C++*. Standard. Geneva, CH: International Organization for Standardization, Sept. 2011. URL: https://www.iso.org/standard/50372.html (visited on 03/20/2020) (cit. on p. 142).

[132] *ISO/IEC TS 19571:2016: Programming Languages – Technical specification for C++ extensions for concurrency*. Standard. Geneva, CH: International Organization for Standardization, 2016 (cit. on p. 142).

[133] M. Tillenius. “SuperGlue: A Shared Memory Framework Using Data Versioning for Dependency-Aware Task-Based Parallelization.” In: *SIAM Journal on Scientific Computing* 37 (Jan. 2015). DOI: 10.1137/140989716. URL: http://epubs.siam.org/doi/10.1137/140989716 (visited on 10/25/2016) (cit. on p. 142).

[134] L. Kalé, S. Krishnan. “CHARM++: A Portable Concurrent Object Oriented System Based on C++.” In: *Proceedings of OOPSLA'93*. Ed. by A. Paepcke. Sept. 1993, pp. 91–108 (cit. on p. 143).

[135] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, L. Kale. “Parallel Programming with Migratable Objects: Charm++ in Practice.” In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2014, pp. 647–658. DOI: 10.1109/SC.2014.58 (cit. on p. 143).

[136] J. Lee, M. Sato. “Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems.” In: *2010 39th International Conference on Parallel Processing Workshops*. Sept. 2010. DOI: 10.1109/ICPPW.2010.62 (cit. on p. 143).

[137] K. Tsugane, J. Lee, H. Murai, M. Sato. “Multi-tasking Execution in PGAS Language XcalableMP and Communication Optimization on Many-core Clusters.” In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. HPC Asia 2018. ACM, 2018, pp. 75–85. DOI: 10.1145/3149457.3154482 (cit. on p. 143).

[138] V. Cavé, J. Zhao, J. Shirako, V. Sarkar. “Habanero-Java: The New Adventures of Old X10.” In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Association for Computing Machinery, 2011, 51–61. DOI: 10.1145/2093157.2093165 (cit. on p. 143).

[139] *Habanero-C*. 2013. URL: https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C (visited on 03/12/2020) (cit. on p. 143).

[140] S. jai Min, C. Iancu, K. Yelick. “Hierarchical work stealing on manycore clusters.” In: *In Fifth Conference on Partitioned Global Address Space Programming Models*. 2011 (cit. on p. 144).

[141] “HabaneroUPC++: A Compiler-free PGAS Library.” In: PGAS '14. DOI: 10.1145/2676870.2676879 (cit. on p. 144).

[142] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, N. Vrvilo. “The Open Community Runtime: A runtime system for extreme scale computing.” In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2016, pp. 1–7. DOI: 10.1109/HPEC.2016.7761580 (cit. on p. 144).

[143] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, L. Smith. "Introducing OpenSHMEM: SHMEM for the PGAS Community." In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. PGAS '10. Association for Computing Machinery, 2010. DOI: 10.1145/2020373.2020375 (cit. on p. 144).

[144] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, Y. Yan. "Integrating Asynchronous Task Parallelism with MPI." In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pp. 712–725. DOI: 10.1109/IPDPS.2013.78 (cit. on p. 144).

[145] Q. Meng, J. Luitjens, M. Berzins. "Dynamic task scheduling for the Uintah framework." In: *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*. Nov. 2010, pp. 1–10. DOI: 10.1109/MTAGS.2010.5699431 (cit. on p. 145).

[146] M. Bauer, S. Treichler, E. Slaughter, A. Aiken. "Legion: Expressing locality and independence with logical regions." In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. Nov. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71 (cit. on p. 145).

[147] B. Long. "Additional Parallel Features in Fortran." In: *SIGPLAN Fortran Forum* (July 2016), pp. 16–23. DOI: 10.1145/2980025.2980027 (cit. on p. 145).

[148] H. Lu, S. Seo, P. Balaji. "MPI+ULT: Overlapping Communication and Computation with User-Level Threads." In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. Aug. 2015, pp. 444–454. DOI: 10.1109/HPCC-CSS-ICESS.2015.82 (cit. on p. 177).

[149] B. Elis, D. Yang, M. Schulz. "QMPI: A next Generation MPI Profiling Interface for Modern HPC Platforms." In: *Proceedings of the 26th European MPI Users' Group Meeting*. EuroMPI '19. 2019. DOI: 10.1145/3343211.3343215 (cit. on p. 178).

[150] M.-A. Hermanns, N. T. Hjlem, M. Knobloch, K. Mohror, M. Schulz. "Enabling Callback-driven Runtime Introspection via MPI_T." In: *Proceedings of the 25th European MPI Users' Group Meeting*. EuroMPI'18. ACM, 2018, 8:1–8:10. DOI: 10.1145/3236367.3236370 (cit. on p. 178).

[151] E. Castillo, N. Jain, M. Casas, M. Moreto, M. Schulz, R. Beivide, M. Valero, A. Bhatele. "Optimizing Computation-communication Overlap in Asynchronous Task-based Programs: Poster." In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. ACM, 2019, pp. 415–416. DOI: 10.1145/3293883.3295720 (cit. on p. 178).

Appendix A

Test Configurations

A.1. System Overview

The experiments presented in this work (Chapter 4 and Chapter 6) were all conducted on two different systems. The detailed configuration information for both systems are listed in Table A.1.

The Cray XC40, called *Hazel Hen*[38], was installed at HLRS in Stuttgart, Germany and has been decommissioned at the time of this writing. It consisted of 7712 dual-socket compute nodes employing Intel Haswell CPUs and offering 128 GB of main memory per node. The second system is a Bull Linux cluster, called *Taurus*[39], that is installed at the ZIH of the University of Technology Dresden in Dresden, Germany. The nodes used in the experiments contained similar CPUs the ones that were available in the Cray XC40, with only 64 GB of main memory.

[38] `https://www.hlrs.de/systems/cray-xc40-hazel-hen/`, last accessed March 22, 2020.

[39] `https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareTaurus`, last accessed March 22, 2020.

Table A.1.: Systems under test.

Name	Hazel Hen	Taurus
System	Cray XC40	Bull Linux Cluster
CPU	Intel Xeon CPU E5-2680 v3 12C (Haswell) @2.50 GHz	Intel Xeon CPU E5-2680 v3 12C (Haswell) @2.50 GHz
Node memory	128 GB	64 GB
Interconnect	Cray Aries	Mellanox Connect-IB
Batch system	PBS Pro / ALPS	SLURM
Compiler	Intel 19.0.1.144	Intel 19.0.1.144, GCC 7.3.0

A.2. Used Software

A.2.1. Message Queue Benchmarks

The different MPI implementations and their configurations used in the experiments discussed in Chapter 4 are listed in Table A.2. Both MPICH and Open MPI on Taurus used UCX as a network abstractions.

On Taurus, it was necessary to announce to UCX that the Bull-specific NIC is in fact a Connect-IB device by setting the environment variable `UCX_IB_DEVICE_SPECS=0x119f:4113:ConnectIB:5`. This has since been added to the UCX release.

The Open MPI version used on Taurus contains patches by the author of this work to the UCX one-sided communication component that add support for the window info key `acc_single_intrinsic`, which has been discussed in Section 4.2.2. A pull request to integrate this feature into the mainline Open MPI repository has been opened at `https://github.com/open-mpi/ompi/pull/6980` (last accessed March 22, 2020).

For these experiments, all software packages have been built using the Intel compiler in version 19.0.1.144.

The experiment code for low-level operation latencies is available online at `https://github.com/devreal/mpi-progress` (last visited March 22, 2020). The message queue implementations together with the benchmark can be found at `https://github.com/dash-project/dash/tree/feat-dash-concurrency/` (last visited Mach 22, 2020).

A.2.2. Tiled Linear Algebra Experiments

The build-time configuration of third-party packages used in the tiled Cholesky and QR decomposition experiments on both systems are listed in Table A.3. Similar the the above, all software packages have been built using the Intel compiler in version

Table A.2.: MPI software configuration options. All software was built with the Intel compiler 19.0.1.144.

Software	Version	Configuration
MPICH	3.3.1	`--with-device=ch4:ucx`
UCX	1.6.x (`git-736d503`)	`--enable-mt --with-avx --enable-optimizations`
Open MPI (Taurus)	4.0.x (`git-69008e4`)	`--with-ucx --with-slurm --with-pmi`
Open MPI (Cray)	3.1.2	*Default*
MVAPICH	2.3.2	*Default*
Intel MPI	2018.4.274	*Default*
Cray MPICH	7.7.4	`MPICH_MAX_THREAD_SAFETY=multiple`

19.0.1.144. The runtime options used for tiled Cholesky decomposition and the tiled QR decomposition are listed in Table A.4 and Table A.5, respectively.

Table A.3.: Versions and relevant build configurations of software packages used in tiled linear algebra experiments.

Software	Version	Configuration
PaRSEC	`git 64574e9a9d0d`	`-DCOREBLAS_PKG_DIR=$PLASMAROOT/lib/pkgconfig/ -DBUILD_SHARED_LIBS=OFF -DHWLOC_LIBRARY=$HWLOCROOT/lib/libhwloc.a -DCMAKE_BUILD_TYPE=Release`
StarPU	1.2.9	`--with-mpicc=$(which mpicc) --with-mpicxx=$(which mpicxx)`
Chameleon	0.9.2	`-DCHAMELEON_SCHED_STARPU=ON -DBLAS_DIR=$MKLROOT -DCHAMELEON_USE_MPI=ON -DCMAKE_EXE_LINKER_FLAGS=-nofor_main -DCMAKE_BUILD_TYPE=Release`
PLASMA	2.8.0	*Default*
MKL	2019.1.144	*Default*
hwloc	1.11.9	*Default*

Table A.4.: Run-time options used to run tiled Cholesky decomposition benchmarks. The variables N is the total size of the matrix, BS is the tile size, and with NP the number of processes $P = Q = \sqrt{NP}$.

Software	Configuration
DASH Tasks	`DART_THREAD_PROGRESS=1 DART_THREAD_IDLE_SLEEP=0 DART_TASK_MATCHING_INTERVAL=100 DART_BIND_THREADS=1 DART_NUM_THREADS=23 DART_MATCHING_PHASE_MAX_ACTIVE=-1 DART_TASK_IDLE_THREAD=poll DART_TASK_COPYIN_WAIT=detach`
PaRSEC	`-N $N -NB $BS -v -c 23 -p $P -q $Q`
Chameleon	`-n $N --nb=$BS -P $P --nowarmup --nowarnings --threads=23` `STARPU_SCHED=dmdas`

Table A.5.: Run-time options used to run tiled QR decomposition benchmarks.

Software	Configuration
DASH Tasks	`DART_THREAD_PROGRESS=1 DART_THREAD_IDLE_SLEEP=0 DART_TASK_MATCHING_INTERVAL=300 DART_BIND_THREADS=1 DART_NUM_THREADS=23 DART_MATCHING_PHASE_MAX_ACTIVE=-1 DART_TASK_IDLE_THREAD=poll DART_TASK_COPYIN_WAIT=detach DART_MATCHING_PHASE_LB=600`
PaRSEC	`-N $N -NB $BS -v -c 23 -p $P -q $Q -SNB 1 -SMB 16`
Chameleon	`-n $N --nb=$BS -P $P --nowarmup --nowarnings --threads=23` `STARPU_SCHED=dmdas`

A.2.3. NPB BT-MZ Build-time Options

All NPB Block Tri-diagonal solver, multi-zone version (BT-MZ) implementations discussed in Section 6.5.1 were compiled with GNU Compiler Collection (GCC) compiler version 7.3.0 using the option `-O2 -mcmodel=medium -march=core-avx2`.

A.2.4. LULESH Build-time Options

All variants of LULESH evaluated in Section 6.6 were built using the Intel compiler 19.0.1.144 with the optimization flags `-O3 -xCORE-AVX2`.

Appendix B

QR Factorization implemented using DASH Tasks

The implementation of the QR decomposition algorithm using DASH Tasks is listed in Listing B.1. Some of the most important aspects should be discussed here. A performance evaluation and comparison with PaRSEC and Chameleon were presented in Section 6.4.3.

The matrix `A` is allocated as listed in Listing B.1. Since DASH currently does not support the notion of super-tiles (clusters of neighboring tiles in one process) this feature has been emulated using a four-dimensional pattern in which the first two dimensions represent the super-tiles and the last two dimension the actual tiles. The use of a of a balanced two-dimensional `teamspec` mapped into a four-dimensional `teamspec` ensures that the data distribution happens on the level of super-tiles, keeping the actual tiles local.

While the DASH matrix implementation provides the notion of blocks its implementation was never stable enough for use in this scenario. Hence, a custom

```
1  using PatternT = typename dash::TilePattern<4>;
2  using TiledMatrix = dash::Matrix<value_t, 4, dash::default_index_t, PatternT>;
3
4  dash::TeamSpec<2> ts2d(dash::size(), 1);
5  ts2d.balance_extents();
6  dash::TeamSpec<4> teamspec(ts2d.extent(0), ts2d.extent(1), 1, 1);
7  size_t nt = N / tile_size;
8  TiledMatrix A(nt, nt,          // super-tiles
9                dash::TILE(sb_m), dash::TILE(sb_n),
10               tile_size, tile_size, // tiles
11               dash::TILE(tile_size), dash::TILE(tile_size),
12               teamspec);
```

Listing B.1.: Allocation of DASH matrix with super-tiles.

'MatrixBlock' implementation has been created, which represents a tile in the matrix and offers member functions for loading and storing blocks.

It should be reiterated here, that the DASH library—unlike integrated numerical algorithm software packages such as DPLASMA and Chameleon the DASH abstraction aims at being a general purpose PGAS library. However, it is not inconceivable to create a numerical algorithms library based on DASH to further investigate the use of PGAS models in this area.

The algorithm in Listing B.1 allocates two scratch spaces in the global memory (`scratch_kn` and `scratch_kk`). These are critical for efficient communication as they allow the implementation to avoid writing back non-local output tiles to the original matrix and have tasks requiring these tiles as input fetch them from there. Instead, all output tiles that *may* be non-local are stored in the scratch space and fetched from there (Lines 54, 78, 104, 107, and 128). Only the final version of the tile is written back to the original matrix (Lines 140–174). This represents a simple mechanism for shifting ownership of these temporary tiles.

Storing the temporary tiles in global memory simplifies the task of coordinating the communication compared to two-sided communication. The locality of the temporary tiles can be determined based on the locality of other tiles, whose locality was significant in the selection of which process executes the respective task (Lines 116, 141, and 158).

Threads manage local scratch space for `work` and `tau` variables, which are allocated locally on demand (e.g., Lines 39–42). These scratch variables are passed to the underlying BLAS library functions were needed. The allocation of this scratch space could be moved into the tasking library (or some higher-level abstraction) in future versions.

As discussed throughout the course of this work, the proposed programming model is a local-view mode. Processes only discover the tasks they will execute. Similar to the Cholesky decomposition discussed in Section 2.4, this is determined based on the locality of the respective tiles (Lines 35, 61, 87, 114, 142, and 159). For tasks that have more than one tile as output in the same matrix (namely TSQRT and TSMQR), the block determined by the iteration of the inner loop is used to select the locality.

Overall, the implementation of the QR decomposition in DASH is concise and clear, despite some additional memory management that is not required in abstractions used for numerical algebra packages. However, future versions of DASH may be better equipped to handle temporary data internally, e.g., by offering abstractions for shifting tile ownership.

Listing B.1: Tiled QR decomposition implemented using DASH Tasks.

```
template<typename TiledMatrix>
void compute(TiledMatrix& A, TiledMatrix& T){

  using value_t   = typename TiledMatrix::value_type;
  using Block     = MatrixBlock<TiledMatrix>;
  const size_t nt = A.pattern().extent(0); // number of tiles
  const size_t ts = A.pattern().extent(2); // tile size

  // allocate scratch space
  constexpr const int num_kn_scratch = 16;
  dash::Matrix<value_t, 4> scratch_kn(dash::size(), dash::BLOCKED,
                                      num_kn_scratch, dash::NONE,
                                      nt, dash::NONE,
                                      ts*ts, dash::NONE);

  constexpr const int num_kk_scratch = 6;
  dash::Matrix<value_t, 3> scratch_kk(dash::size(), dash::BLOCKED,
                                      num_kk_scratch, dash::NONE,
                                      ts*ts, dash::NONE);

  double **scratch_tau  = new double*[dash::numthreads()];
  std::fill(scratch_tau, scratch_tau+dash::numthreads(), nullptr);
  double **scratch_work = new double*[dash::numthreads()];
  std::fill(scratch_work, scratch_work+dash::numthreads(), nullptr);

  // iterate over column of blocks
  for (int k = 0; k < nt; ++k) {
    Block a_kk(A, k, k);
    Block t_kk(T, k, k);
    int k_scratch_entry = k%num_kk_scratch;

    /**
     * Compute QR factorization of block on diagonal
     */
    if (a_kk.is_local()) {
      dash::async("GEQRT",
        [=, &scratch_kk, &A, &T]() mutable {
          int threadnum = dash::threadnum();
          if (scratch_work[threadnum] == nullptr)
            scratch_work[threadnum] = new double[ts*ts];
          if (scratch_tau[threadnum]  == nullptr)
            scratch_tau[threadnum]  = new double[ts];

          geqrt(a_kk.lbegin(), t_k.lbegin(),
                scratch_tau[threadnum], scratch_work[threadnum]);

          // copy the block to the scratch space for subsequent accesses
          std::copy(a_kk.lbegin(), a_kk.lend(),
                    &scratch_kk.local(0, k_scratch_entry, 0));
        },
        (dart_task_prio_t)((nt-k)*(nt-k)*(nt-k))/*priority*/,
        dash::out(a_kk),
        dash::out(t_kk),
        dash::out(scratch_kk.local(0, k_scratch_entry, 0))
      );
```

```
56      }
57      dash::async_fence();
58
59      for (int n = k+1; n < nt; ++n) {
60        Block a_kn(A, k, n);
61        if (a_kn.is_local()) {
62          dash::async("ORMQR",
63            [=, &scratch_kn, &A](double *a_k, double *t_k_ptr) mutable {
64              int threadnum = dash::threadnum();
65              if (scratch_work[threadnum] == nullptr)
66                scratch_work[threadnum] = new double[ts*ts];
67              // Block(k, n) is both input and output, use the original block as
68              // input and copy it to the scratch space afterwards
69              ormqr(a_k, t_k_ptr, a_kn.lbegin(), scratch_work[threadnum]);
70              // copy the block to the scratch space for subsequent accesses
71              std::copy(a_kn.lbegin(), a_kn.lend(),
72                        &scratch_kn.local(0, k%num_kn_scratch, n, 0));
73            },
74            DART_PRIO_LOW,
75            dash::copyin_r(a_kk),
76            dash::copyin_r(t_kk),
77            dash::out(a_kn),
78            dash::out(scratch_kn(dash::myid(), k%num_kn_scratch, n, 0))
79          );
80        }
81      }
82      dash::async_fence();
83
84      for (int m = k+1; m < nt; ++m) {
85        Block a_mk(A, m, k);
86        Block t_mk(T, m, k);
87        if (a_mk.is_local()) {
88          // get tile (k, k) from from previous owner's scratch space
89          value_t *a_kk_ptr = &scratch_kk.local(0, k_scratch_entry, 0);
90          int prev_kk_owner = Block{A, m-1, k}.unit();
91          dash::async("TSQRT",
92            [=, &A, &T]() mutable {
93              int threadnum = dash::threadnum();
94              if (scratch_work[threadnum] == nullptr)
95                scratch_work[threadnum] = new double[ts*ts];
96              if (scratch_tau[threadnum]  == nullptr)
97                scratch_tau[threadnum]  = new double[ts];
98
99              tsqrt(a_kk_ptr, a_mk.lbegin(), t_mk.lbegin(),
100                   scratch_tau[threadnum], scratch_work[threadnum]);
101
102           },
103           (dart_task_prio_t)((nt-k)*(nt-k)*(nt-k)) /*priority*/,
104           dash::copyin_r(scratch_kk(prev_kk_owner, k_scratch_entry, 0), ts*ts,
                a_kk_ptr),
105           dash::out(a_mk),
106           dash::out(t_mk),
107           dash::out(*a_kk_ptr)
108         );
109       }
110       dash::async_fence();
111
```

```
112       for (int n = k+1; n < nt; ++n) {
113         Block a_mn(A, m, n);
114         if (a_mn.is_local()) {
115           // get tile (k, n) from from previous owner's scratch space
116           int prev_owner = Block{A, m-1, n}.unit();
117           value_t *a_kn_ptr = &scratch_kn.local(0, k%num_kn_scratch, n, 0);
118           dash::async("TSMQR",
119             [=, &A, &scratch_kn](double *a_mk, double *t_mk) mutable {
120               int threadnum = dash::threadnum();
121               if (scratch_work[threadnum] == nullptr)
122                 scratch_work[threadnum] = new double[ts*ts];
123
124               tsmqr(a_kn_ptr, a_mn.lbegin(),
125                     a_mk, t_mk, scratch_work[threadnum]);
126             },
127             (dart_task_prio_t)((nt-k)*(nt-n)*(nt-n)),
128             dash::copyin_r(scratch_kn(prev_owner, k%num_kn_scratch, n, 0),
129                            ts*ts, a_kn_ptr),
130             dash::copyin_r(a_kk),
131             dash::copyin_r(t_mk),
132             dash::out(a_mn),
133             dash::out(*a_kn_ptr)
134           );
135         }
136       }
137     }
138     dash::async_fence();
139
140     // write back block (k,k)
141     int prev_k_owner = Block{A, nt-1, k}.unit();
142     if (dash::myid() == prev_k_owner) {
143       dash::async("WRITEBACK_KK",
144         [=, &scratch_kk, &A]() mutable {
145             Block a_kk(A, k, k);
146             a_kk.store_async(&scratch_kk.local(0, k_scratch_entry, 0));
147             // detach the task
148             dart_handle_t handle = a_kk.dart_handle();
149             dart_task_detach_handle(&handle, 1);
150         },
151         DART_PRIO_HIGH,
152         dash::in(scratch_kk.local(0, k_scratch_entry, 0))
153       );
154     }
155
156     // write back blocks (k, n)
157     for (int n = k+1; n < nt; ++n) {
158       int prev_owner = Block{A, nt-1, n}.unit();
159       if (dash::myid() == prev_owner) {
160         auto scratch_block_kn = scratch_kn(prev_owner, k%num_kn_scratch, n, 0)
              ;
161         dash::async("WRITEBACK_KN",
162           [=, &scratch_kn, &A]() mutable {
163             Block a_kn(A, k, n);
164             a_kn.store_async(&scratch_kn.local(0, k%num_kn_scratch, n, 0));
165             // detach the task
166             dart_handle_t handle = a_kn.dart_handle();
167             dart_task_detach_handle(&handle, 1);
```

```
168             },
169             DART_PRIO_HIGH,
170             dash::in(scratch_block_kn)
171           );
172         }
173       }
174     }
175
176     dash::complete();
177
178     for (int i = 0; i < dash::numthreads(); ++i) {
179       if (scratch_tau[i]  != nullptr) delete[] scratch_tau[i];
180       if (scratch_work[i] != nullptr) delete[] scratch_work[i];
181     }
182     delete[] scratch_tau;
183     delete[] scratch_work;
184   }
```

Appendix C

Callback-Driven Request Completion

The MPI standard provides non-blocking send/recv, collective, and RMA operations that return a request object, which can later be used to test the status of an operation or wait for its completion. Especially in the case of task-based programming models, it is desirable to receive notifications on the completion of communication operations blocking the execution of tasks as soon as possible to reduce the latency between blocking and resuming execution. Moreover, the management of these requests either inside the application or the scheduler itself carries significant overhead and requires inter-thread synchronization to avoid race conditions.

Previous work has shown that the direct integration of ULT-based programming models with the MPI layer can be advantageous [148]. The stated goal has been to enable the use of blocking communication operations inside user-level thread by blocking the high-level thread instead of the executing resource (pthread). However, the tight integration of different programming abstractions will likely not scale in terms of combinations of supported programming models and their implementations.

It should also be reiterated that applications relying on implementation-specific details such as integrated support for ULT packages are non-portable. Any MPI implementation *not* supporting ULT implementations and thus blocking the ULT *and* the underlying pthread is still in accordance with the MPI standard [7, §12.4]. The MPI standard further states that communication patterns relying on buffering are *unsafe* and that communication is *errant* if blocking operations are posted in an order that may deadlock [7, §3.5]. Portable applications using ULTs should thus not rely on any support for ULTs inside MPI implementations.

Other approaches such as TAMPI provide a shim layer on top of MPI that intercepts blocking MPI communication calls, initiating a non-blocking operation instead and blocking the current ULT [127]. TAMPI collects all active communication requests and periodically checks for their completion, releasing blocked ULTs whose communication operations have completed.[40]

At the same time, MPI implementations (and many underlying network abstractions) are already aware of all active communication operations and their associated requests. Thus, offering applications asynchronous notifications on the completion of *non-blocking* operations would relieve the upper layers from managing the requests while allowing ULTs to block on pending communication. It would also alleviate the need for MPI implementations to support an arbitrary number of ULT implementations.

Instead of deferring the notification to the point where the application tests the status of a request, the notifications would be issued as soon as the progress engine of the MPI implementation is triggered. By using the callback-driven `MPI_T` interface proposed for the upcoming MPI standard [150], it has been shown that the technique of asynchronous notifications can yield benefits in applications employing MPI [151]. However, the usage of the `MPI_T` interface still requires the application or runtime system to manage a mapping between MPI request objects and the tasks blocked by the associated operation.

In order to achieve *loose coupling* of programming models and alleviate the need for explicit request management in the application or runtime layer, an extension to the MPI standard is proposed here: the new function `MPIX_Request_on_completion`[41] would be used to associate a completion callback function and a corresponding callback argument with an active request. As soon as the MPI implementation detects the completion of an operation it will invoke the callback to signal its completion.

Listing C.1 provides the declaration of the function `MPIX_Request_on_completion` and the signature of the associated callback function. Listing C.2 shows an example of how this functionality can be used to block and resume a ULT, provided that the underlying runtime system provides such capabilities. After the send operation has been initiated (Line 15), the completion callback is registered in Line 17 and the

[40] A major draw-back of Intercepting MPI calls to control task execution is the use of the MPI profiling interface (PMPI) [7, §14] that is used by performance analysis and correctness tools to intercept MPI calls. This approach significantly hinders performance analysis of task-based applications. A potential replacement has been proposed in the form of QMPI, which would allow multiple tools to intercept MPI calls [149].

[41] By convention, the prefix `MPIX` is used to mark experimental extensions to the MPI standard.

```
typedef int (*MPIX_Completion_cb)(MPI_Request req, void *cbdata);

int
MPIX_Request_on_completion(
    MPI_Request req,
    MPI_Completion_cb callback,
    void *cbdata);
```

Listing C.1.: Interface of the function `MPI_Request_on_completion`. The registered callback function takes a generic pointer that is passed to `MPI_Request_on_completion` upon registration of the callback.

```
int completion_cb(MPI_Request req, void *cbdata)
{
  int flag;
  ultref_t ult = (ultref_t)cbdata;
  // testing the request is required to release the object
  MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
  // the send has completed, resume the task
  resume_ult(ult);
}

void ult_fn()
{
  MPI_Request req;
  // start the send operation
  MPI_Isend(..., &req);
  // register completion callback
  MPI_Request_on_completion(req, &completion_cb, current_task());
  // block the task
  block_ult();
  // more work to do here once the send is complete
}
```

Listing C.2.: Usage example of the function `MPIX_Request_on_completion`. The function `ult_fn` is executed inside a ULT. The functions `current_ult`, `block_ult`, and `release_ult` are provided by the ULT implementation and yield a reference to the current ULT, block the current ULT, and release the referenced ULT, respectively.

task is blocked in the scheduler.[42] As soon as the MPI implementation determines that the send operation has been completed, it will invoke the completion callback (Line 1) where the request object is released in the call to `MPI_Test`[43] before the execution of the ULT may be resumed.

This feature will be evaluated in the context of two use-cases. First, it will be used as a replacement for TAMPI in the context of the NPB benchmark discussed in

[42]The definition of `block_task` and `resume_task` used in Listing C.2 depends on the used tasking system and is not shown here.

[43]The requirement to call `MPI_Test` is a conscious design decision intended to allow the application to still call `MPI_Test` or `MPI_Wait` on the request at arbitrary points during the execution and to allow for the querying of the status object. Alternatively, `MPI_Request_on_completion` could be designed to take ownership of the request, in which case any further call to `MPI_Test` would neither be required nor legal.

Section 6.5. Second, it is employed to replace the use of EGREQs in the coordination of the sequence of RMA operations required for the pipelined `sopnop` active message queue, which was presented in Section 4.3.2.1.

C.1. Use with OmpSs-2

The callback-driven notification proposed in Appendix C can be used in combination with the ability of the OmpSs-2 runtime to block and resume the execution of tasks. A task can thus initiate a non-blocking communication operation and register a completion callback on the associated request object before blocking itself. When the communication operation has completed, the callback will be invoked by the MPI implementation and the task will be unblocked to be resumed later.[44]

In order to ensure that the release will eventually be signaled, at least one thread is required to periodically poll the MPI library for the completion of communication operations. While it may be possible for the application to spawn its own communication thread, this may have detrimental effects on the performance of the worker threads managed by the underlying runtime system. Instead, the OmpSs-2 library allows the application to register so-called *services* that are periodically invoked by any of the threads managed by the runtime.

These services can be used to periodically poll the MPI library for completed communication operations. With the callback-driven completion notification, the application is not required to manage the list of active communication operations. Instead, invoking the MPI implementation's progress engine is sufficient.

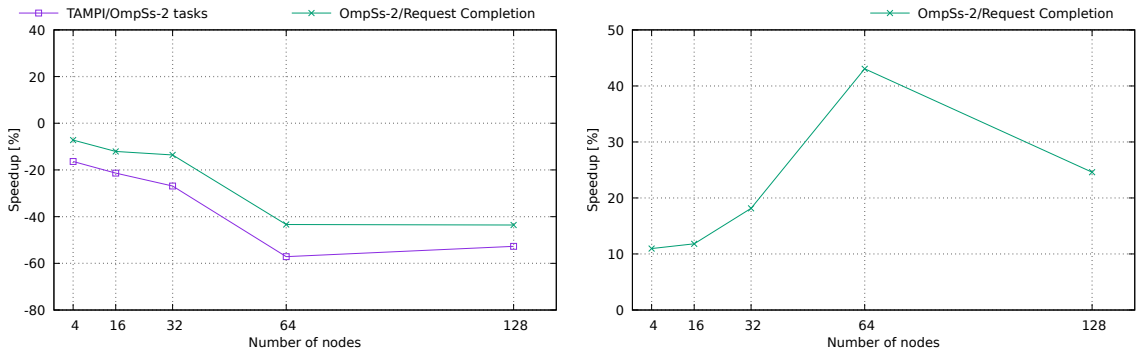
C.1.1. Implementation

Listing C.3 provides the outline of the functions necessary for using the proposed request completion callbacks in combination with OmpSs. The OmpSs-2 service responsible for triggering the MPI progress engine is listed in lines 6–15. The MPI progress engine is triggered by a call to `MPIX_Request_progress()`, which is repeated as long as requests are active and the previous call (if any) yielded completed requests. While the repeated calling is in no way required, triggering the progress engine repeatedly has shown to be beneficial for lowering the time-to-release.

[44] OmpSs-2 also supports *task external events*, which are similar to detached tasks in OpenMP 5.0. In that case a task will complete but its dependencies will not be released until all all external events have been fulfilled. The use with the request completion callback would be similar.

```
static std::atomic<int> active_reqs{0};
static thread_local int complete_reqs{0};
static std::once_flag   flag;
MPI_Request             dummy_request;

bool mpi_poll_service(void*)
{
  if (active_reqs > 0) {
    do {
      complete_reqs = 0;
      MPI_Test(&dummy_request); /* invoke progress engine */
    } while (complete_reqs > 0 && active_reqs > 0);
  }
  return false; /* signal that we need to continue to be called */
}

static int request_completion_cb(void *event_counter, MPI_Request req)
{
  int flag;
  MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
  --active_reqs;
  ++complete_reqs;
  nanos6_unblock_task(task); /* release task */
  return MPI_SUCCESS;
}

static void block_on_request(MPI_Request req)
{
  int flag;
  MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
  if (!flag) {
    ++active_reqs;
    MPIX_Request_on_completion(req, request_completion_cb, event_counter);
    void* task = nanos6_get_current_blocking_context();
    nanos6_block_current_task(task); /* block task */
  }
}

int communicate(void *buf)
{
  MPI_Request request;
  /* the service has to be registered only once */
  std::call_once(flag, [](){
    MPI_Irecv(..., myrank, ..., &dummy_request);
    nanos6_register_polling_service("MPI", mpi_poll_service, NULL);
  });

  /* a task that receives data from another process */
  #pragma oss task depend(out:buf)
  {
    MPI_Irecv(buf, ..., &req);
    block_on_request(request);
  }

  /* a task that consumes the received data */
  #pragma oss depend(in:buf)
  {
    [...]
  }
}
```

Listing C.3.: Example for using the proposed callback-driven request completion notification with OmpSs-2.



(a) Speedup of TAMPI and request completion callbacks over the reference.

(b) Speedup of request completion callbacks over TAMPI.

Figure C.1.: Speedup of BT-MZ class D using OmpSs-2 in combination with TAMPI and request completion callbacks.

The service is registered in line 45 and the usage of `std::call_once` ensures that the service is registered only once. Upon completion of a request, MPI calls the registered callback listed in lines 17–25. This callback is registered in the function `block_on_request` that is listed in lines 27–37 and called in line 52. While the initial test for completion (line 30) is not mandatory, it has proven beneficial by avoiding an unnecessary deferral of the dependency release even though the operation is already complete. Moreover, it has the potential to trigger the progress engine and thus release already blocked tasks even if the OmpSs-2 service is not currently active, e.g., because all other threads are busy executing tasks.

After the registration, the current task is block (line 35). As soon as the communication operation has completed, the task is resumed in line 23. The test for completion inside the completion callback (line 20) is necessary to release the resources associated with the request object to allow for the MPI implementation to reuse the request object. It also offers the application a way to access the status object of the completed request, if desired.

As shown here, it requires only three dozen lines of code in the application to make use of the request completion callbacks. Using this technique, external libraries such as TAMPI (and their associated draw-backs such as the interference with MPI profiling interface (PMPI)-based tools) can be avoided and a loose coupling between MPI and task-based programming models can be established in the application space.

C.1.2. Evaluation with NPB BT-MZ

While TAMPI has shown rather low performance in the NPB BT-MZ benchmark on the Taurus system as described in Section 6.5.2, it is worth testing the request completion

callback interface proposed above. The performance relationship between TAMPI and this implementation is depicted in Figure C.1. As can be seen from Figure C.1a, the scaling curve of this implementation mostly follows the trajectory of TAMPI over the range of nodes. Looking at the speedup of this implementation over TAMPI (Figure C.1b) it becomes clear, however, that the completion callbacks yield a not insignificant speedup of 10–40% over TAMPI. Unfortunately, this is not sufficient to offset the overhead incurred by OmpSs-2 itself.[45] An investigation of this overhead is pending and the developers of OmpSs-2 appear to be working on a version with improved efficiency at the time of this writing.

These numbers suggest that using the proposed request completion callbacks to achieve loose coupling can also lead to improved performance. This may be attributed to the ability of *any* thread to invoke these callbacks upon detecting a request, thus reducing the time-to-release of blocked or detached tasks.

C.2. Use in an Active Message Queue Implementation

While the discussion so far has focused on use of this interface in the context of ULTs, it is easy to see the value of such a notification scheme inside a task scheduler itself, where it may serve to release detached tasks or tasks blocked automatically by the scheduler (cf. Section 5.2.4). This interface may even be used to advance the state machine of the active message queue described in Section 4.3.2.1. Figure C.2 depicts the speedup achievable by replacing the use of EGREQs to drive the state machine depicted in Figure 4.6 with request completion callbacks. It is notable that while at two and 64 nodes a slowdown of up to 10% can be observed, the range in between yields a positive speedup of more than 15%. It also appears that for higher node counts the difference starts to diminish. A deeper analysis of the slowdown is beyond the scope of this work and this experiment is simply meant to show that request completion callbacks have the potential to provide for lower latencies compared to using EGREQs.

[45] Interestingly, the opposite of what was described in Footnote 32 on page 126 is true for this variant: the blocking mode of OmpSs-2 yielded superior performance over the non-blocking mode. The measurements presented here used the blocking mode for the request completion callbacks.

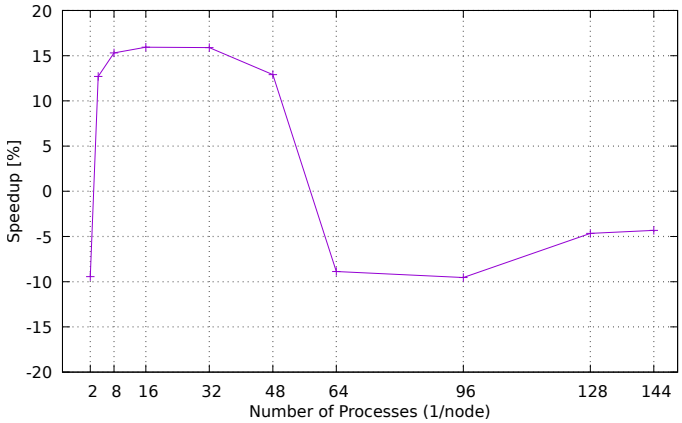


Figure C.2.: Speedup of an experimental message queue implementation using request completion callbacks to drive the state machine of the pipelined `sopnop` over the implementation using EGREQs on Taurus.

C.3. Discussion

The results presented here confirm that the callback-driven request completion notifications can be used in a versatile way i) to organize the interaction between ULT-based programming models and MPI; and ii) to coordinate a complex sequence of RMA operations. In both cases, the results indicate that the resulting performance is higher than the references TAMPI and EGREQ. In contrast to TAMPI (or direct tight integration of ULT libraries with MPI), the resulting solution is not as simple as these references but it provides for portable and standard-compliant solutions (assuming this interface will be adopted by the MPI forum).

Appendix D

LULESH Task Graph

Figure D.1 presents the task graph of one timestep in LULESH as seen by the scheduler with the current implementation. LULESH and the port to DASH Tasks were discussed and evaluated in Section 6.6.

As briefly mentioned during that discussion, the structure of the task-parallel execution in LULESH is such that a number of high-level tasks coordinate the execution of (nested) computational tasks. These tasks are depicted as white nodes in Figure D.1. The bulk of computation is handled by tasks nested inside these coordinating tasks, mostly expressed through `taskloop` constructs chained by dependencies as described in Section 2.3.

Each timestep comprises four communication steps, depicted using yellow and red nodes in Figure D.1. The boundary exchange in all 26 directions (faces, edges, and vertices of the 3D domain) is performed but individual tasks for each direction in between the high-level computational tasks. In the current implementation there is no overlap between packing of outgoing buffers and computation, e.g., the forces on all nodes have to be computed before the `SendForce` tasks are started, which pack force information on the boundary nodes into the global data structures for communication. Achieving such an overlap would require a dissection of the unstructured mesh, e.g., into patches or regions that are handled independently, and was beyond the scope of this work.

Nevertheless, incoming transfers will be started by the scheduler as soon as the data is available on the remote process. Thus, for slower processes the boundary data for integration into the local domain will at least be available already. All communication and integration of boundary data has to be completed before the next communication task has completed executing.

While the high-level task graph is mostly serialized, there is some concurrency even at this level. For example, the computation of nodal forces and accelerations may be performed before the collective communication at the bottom has completed as the width of the timestep is not relevant for them. A smaller degree of concurrency is provided by the determination of part of the Courant-Friedrichs-Lewy constraint (CFL) in `HydroConstraints`. However, the computational demand for CFL is rather small.

While the two-level approach may not expose as much concurrency as a region-based decomposition of the unstructured mesh it provided for a relatively simple adaptation of LULESH. Another alternative is to use a flat task graph in which the taskloops are at the same level as the communication tasks. While this may avoid some synchronization between computational nodes in the task graph it may increase the initial cost of task discovery and global dependency matching. An evaluation of such this approach is left as future work.

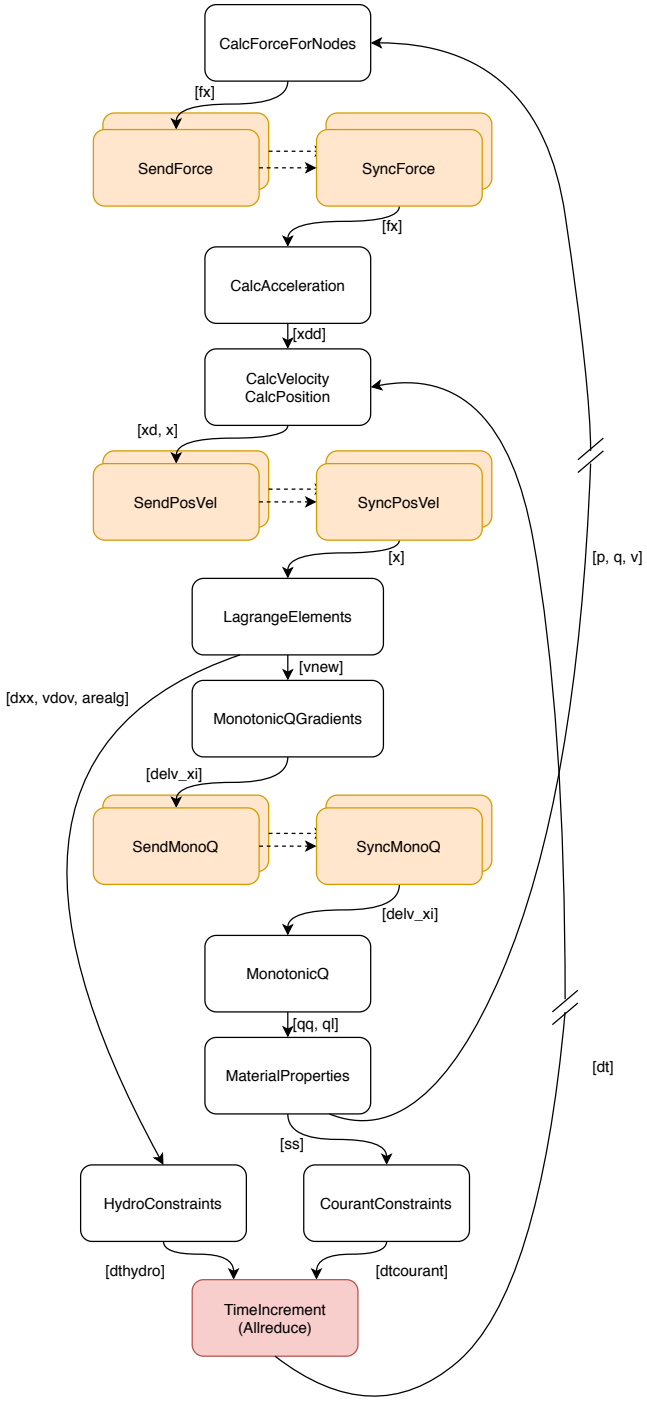


Figure D.1.: High-level task graph of a single timestep in LULESH. White Nodes represent high-level coordination tasks with nested task-graphs inside. Yellow nodes represent a set of communication tasks (packing and unpacking in all 26 directions). Local dependencies are represented through solid arrows, annotated with the referenced variable names. Broken lines symbolize dependencies between tasks in different timesteps. Dashed arrows represent a set of remote dependencies between processes used to handle communication. The red node contains a collective communication operation over all processes.

Appendix E

Curriculum Vitae

Joseph Schuchart received his German university diploma in Computer Science (Dipl.-Inf.) from the University of Technology Dresden in Dresden, Germany in 2012. His area of specialization was Computer Engineering with Prof. Dr. Wolfgang E. Nagel. In addition, he completed course work on Software Engineering (Prof. Dr. Uwe Aßmann), Theory of Programming (Dr. Monica Sturm), Architecture of Distributed Systems (Prof. Dr. Alexander Schill), and System-oriented Computing (Prof. Dr. Klaus Kabitzsch). His diploma thesis covered performance analysis of loosely coupled distributed software systems using event-based trace data collection. He has worked as a student assistant at the Center for High-Performance Computing and Information Services (ZIH, chaired by Prof. Nagel) on the VampirTrace and Score-P projects.

After one year of working at the Oak Ridge National Laboratories (ORNL), Joseph joined the Energy Efficiency research group led by Daniel Hackenberg at ZIH where he was involved in the development of the power and energy instrumentation framework HDEEM in collaboration with Bull SAS. He was also responsible for the inception and initial phase coordination of the READEX Horizon 2020 project.

In August 2016, Joseph joined the Scalable Programming Models and Tools group (SPMT) led by Dr. José Gracia at the High-Performance Computing Center Stuttgart (HLRS, chaired by Prof. Dr. Michael M. Resch) as part of the DASH project. His main responsibility was the investigation of task-based programming models and the continued development of the MPI-based communication back-end in DASH. As a visiting researcher, Joseph worked with Prof. Dr. Mitsuhisa Sato at Riken AICS in Kobe, Japan and Dr. George Bosilca at the Innovative Computing Laboratory (ICL) at the University of Tennessee, Knoxville (UTK).