Exploiting Hierarchical Parallelism Using UPC

by Lingyuan Wang

B.S. in Electrical Engineering, July 2007, Beihang University, China

A Thesis submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University in partial satisfaction
of the requirements for the degree of Master of Science

August 31, 2010

Thesis directed by

Tarek El-Ghazawi
Professor of Engineering and Applied Science

Acknowledgements

This dissertation would not be possible to complete without the guidance, friendship and support of so many people. My deepest gratitude is to my advisor, Prof. Tarek El-Ghazawi. I have been fortunate enough to have an advisor who gave me the freedom to explore on my own and at the same time the guidance when I was in need of direction. As the founding director of the High Performance Computing Laboratory (HPCL) at The George Washington University (GWU), co-director of the NSF Industry/University Center for High-Performance Reconfigurable Computing (CHREC) and director of IMPACT (The Institute for Massively Parallel Applications and Computing Technologies), Prof. El-Ghazawi gave me the opportunity to be engaged in a broad range of advanced research projects funded by Arctic Region Supercomputing Center (ARSC), and NSF CHREC projects. In the meantime, his guidance and encouragement helped me shape and realize this work. The scientific skills and experiences acquired during my three years at HPCL have become an invaluable asset for my future professional career. I feel grateful to all committee members, Professor Milos Doroslovacki, Professor Shahrokh Ahmadi and Professor Guru Prasadh Venkataramani for their time and for their valuable suggestions and insights. My appreciation also extends to all my colleagues at the HPCL: Dr. Saumil Merchant, Dr. Vikram Narayana, Dr. Miaoqing Huang, Dr. Esam El-Araby, Abdullah Kayi, Olivier Serres, Ahmed Anber for their kind help. Finally, I thank my parents their continuous support.

Abstract

Exploiting Hierarchical Parallelism Using UPC

High-Performance Computing (HPC) systems are increasingly moving towards an architecture that is deeply hierarchical and heterogeneous, comprised of multicore processors and hardware accelerators. This architectural shift has added significant programming complexity as the end users must now understand and exploit parallelism at multiple levels. Unfortunately the single-level parallelism execution model embodied in the legacy parallel programming models falls short in exploiting the available multi-level parallelism opportunities in these architectures. This makes the use of richer execution models imperative in order to fully exploit hierarchical parallelism. This thesis explores multi-level parallelism opportunities in such architectures at the language and application levels by investigating possible extensions to the Unified Parallel C (UPC) parallel programming language and integrating it with other complementary programming paradigms. UPC is a parallel extension of ISO C and supports the Partitioned Global Address Space (PGAS) programming model. It presents a globally shared address space that enables one-sided communication constructs for ease-of-use to programmers. In addition, being a PGAS language, it also provides data locality awareness for higher performance. This research identifies programming language features and important runtime-support that can help improve programmability and application performance on hierarchical architectures. It proposes two approaches. The first approach orchestrates computations on multiple sets of thread groups, whereas the second approach extends UPC with nested, shared memory multi-threading for the facilitation of expressing hierarchical execution. This thesis formally proposes these approaches and evaluates their applicability through high performance implementations of several parallel application benchmarks, such as the NAS FT and the Unbalanced Tree Search benchmark.

The results demonstrate that the explicit hierarchical programming model is better positioned for modern HPC systems.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Today most mainstream computing systems feature a highly hierarchical design, mostly comprised of multi-socket multi-core shared memory compute nodes connected via high-performance interconnects. Intranode architectural details such as the ccNUMA (cache coherent Non-Uniform Memory Access), Simultaneous Multithreading (SMT), number of cores per socket/ccNUMA domain, shared and private caches, plus I/O bottlenecks further add to the complexities. Such pervasive architectural shift has shouldered a great burden on programmers who wish to write efficient parallel software for today's high performance computing systems. Proper application to architecture mapping is the general rule of thumb for optimal performance. But oftentimes developers find that the dominant parallel programming languages fall short of exploiting memory and execution hierarchies effectively on hierarchically structured hardware, as very few of them support explicit hierarchical parallelism. Previously, significant amount of work has been invested in compiler and runtime optimizations for the evolving hardware. However, utilizing richer programming models that enable expressing portable, hierarchical parallelism at the application level is the first logical step to address these challenges.

1.2 Motivations

Partitioned Global Address Space (PGAS) languages have recently emerged as a promising alternative to the traditional message passing model. Developed as parallel extensions for popular sequential programming languages, PGAS languages such as Unified Parallel C (UPC), Titanium, and Co-Array Fortran (CAF) offer higher programmability by providing a global address space that is logically partitioned across the processors. The global address space allows programmers to create sophisticated distributed data structures, and higher performance can be achieved as programmers gain control over the memory locality within the global address space. It also allows a processor to directly read and write remote memory locations without interrupting the execution on the remote processor. Such one-sided communication model have been shown to be able to both improve performance and productivity by removing the receive operations presented in a two-sided model. PGAS languages, thus, offer a more productive shared memory programming style than explicit message passing model languages, such as the Message Passing Interface (MPI).

In this research, we demonstrate how Unified Parallel C (UPC), as one of the most popular PGAS languages, can be aided with application level hierarchical programming language constructs and optimizations to achieve better performance, while offering easier programmability. Unlike previous efforts centered on runtime optimizations that encapsulate hardware topology information within the software infrastructure [1–3], our goal is to expose optimization opportunities at the application level, thereby enabling users to exploit hierarchical parallelism based on algorithmic needs and/or hardware characteristics freely. Optimization at the programming language level in general enables all lower software layers such as compilers and run-time systems a more efficient execution with the awareness of such optimizations.

1.3 Problem Statement and Approach

PGAS languages provide a balance between programmability and performance. UPC in particular has been known to strike such balance that was recognized by a number of HPC challenge awards marking this ability. UPC, however, does not support expressing hierarchical parallelism at the application level. Therefore, two complementary approaches are formally proposed and studied in this research work. The first relies on programmers to group UPC threads together to best match the execution hierarchy for a given platform. We also propose minor extensions to the language to aid in thread grouping and efficient shared memory access and cooperation of threads with in a group. The second approach extends the first effort by providing a true thread execution hierarchy. It does so by augmenting the traditional SPMD model of UPC with children threads managed under UPC language threads. This can be implemented either with a native runtime support or via hybrid programming that incorporates UPC with OpenMP or Cilk++. As we will demonstrate, the hierarchical threading approach allows users to dynamically spawn light-weight *sub-threads* to exploit the escalating power of multicore processors. The convenience of the PGAS model also extends to sub-threads, allowing them to directly access the distributed data structures residing in the shared memory, which is not supported by the similar MPI/threads counterpart. The proposed techniques of application level hierarchical parallelism have been proven successful in a wide variety of scientific applications such as spectral methods (3D FFT), exhaustive parallel search on highly unbalanced search space (tree traversal), etc. We believe that this work has significant values and makes contributions on several fronts, which are summarized as follows:

1. We formally present two alternative methods to support hierarchical parallelism using the UPC/PGAS language paradigm. While the hybrid programming problem has been well studied in the MPI community such as MPI/OpenMP, to the best of our knowledge this is the first such research study to understand and exploit the perfor-

mance and productivity benefits of hierarchical threads in the context of UPC.

2. For the proposed methods, we describe the implementations in details with application examples and synthetic benchmarks. It should be noted here that even though current UPC specifications do not support hierarchical parallelism constructs, a UPC application developer can benefit from applying our proposed techniques immediately to express and exploit the benefits of hierarchical parallelism at the application level.

3. We use our proposal and experimental results to showcase the need for language extensions to aid in expressing hierarchical parallelism and compiler/runtime support for interoperability of UPC/threads, such as compilation of mixed UPC/OpenMP, with proper data scoping and thread safety. The observations and techniques from this study can aid future efforts and possibly help influence the directions of the UPC consortium towards inclusion of hierarchical programming constructs as part of the language specifications.

1.4 Structure of this Thesis

Chapter 2 lays out the background for our work. It starts with introduction of the system used in our experiments and discusses its design details revealing different mechanisms for communication between processing cores and the deepening non-uniform memory access effect. Further it introduces the UPC language and its implementations issues that are relevant to this work. Related previous research work are discussed in the last section. Chapters 3 and 4 detail our two approaches to express hierarchical parallelism in UPC. They also present experimental results and discuss their implications on performance and scalability. Chapter 5 concludes the thesis with a summary of the points raised and possible directions for future work.

Chapter 2

Background

2.1 Hardware Architectural Trends and Experimental Platforms

Past few decades have witnessed an exponential increase in computational power and density of silicon based processing devices. However, with the saturation in clock frequencies as a result of unsustainable thermal and power overheads, the performance improvements are now being delivered by continued escalation in transistor densities enabling packing multiple cores into a single silicon chip. Future trends point to ever increasing core counts per chip, with hundreds of cores per chip likely in the near future. As a result, today's HPC systems are built with thousands or up to 100s of thousands of processors and relying on parallelism (as expressed by the programmer) to deliver the performance. Meanwhile, exotic computing architectures such as GPUs are gaining strong foothold in the HPC arena by offering orders of magnitude of performance improvements at relatively lower energy consumption and costs. Further, consequences of the multicore, ccNUMA (cache coherent Non-Uniform Memory Access) and heterogeneous accelerators trends are resulting in complex hardware architectural hierarchies with deep non-uniform memory access effects. Such complex and hierarchical topologies have strong impact on the application performance. Developers must take as much as possible of all this into account when trying to exploit the actual hardware performance.

Our experiments were conducted on two clusters, Lehman and Pyramid hosted at the

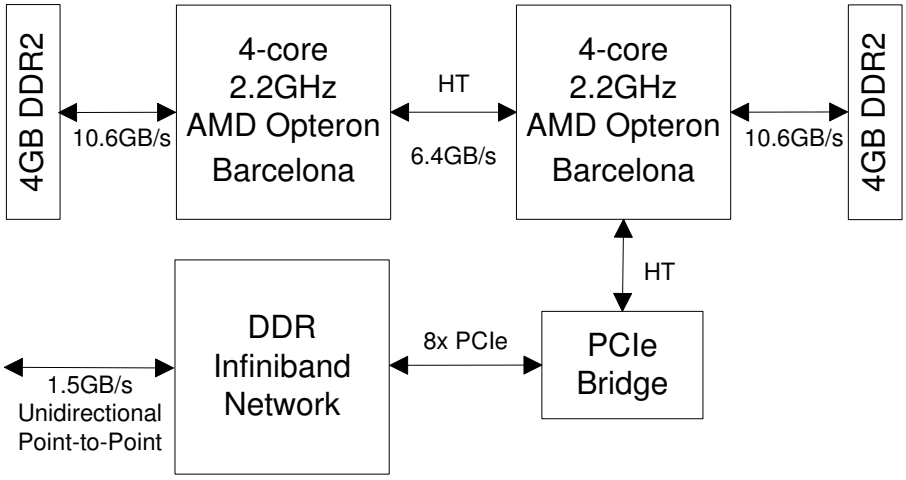


Figure 2.1: Cluster Pyramid Node Architecture

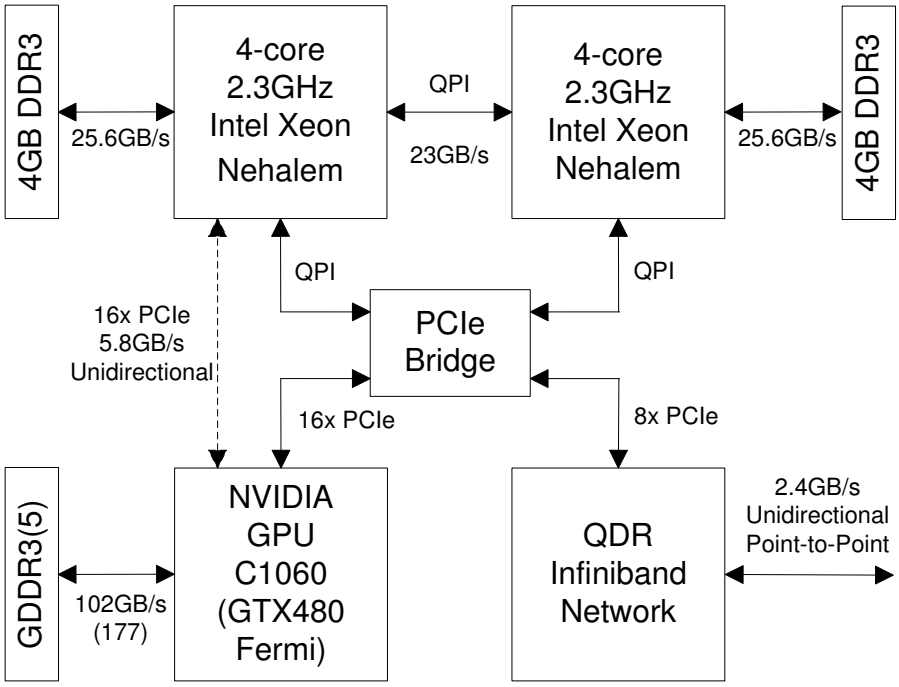
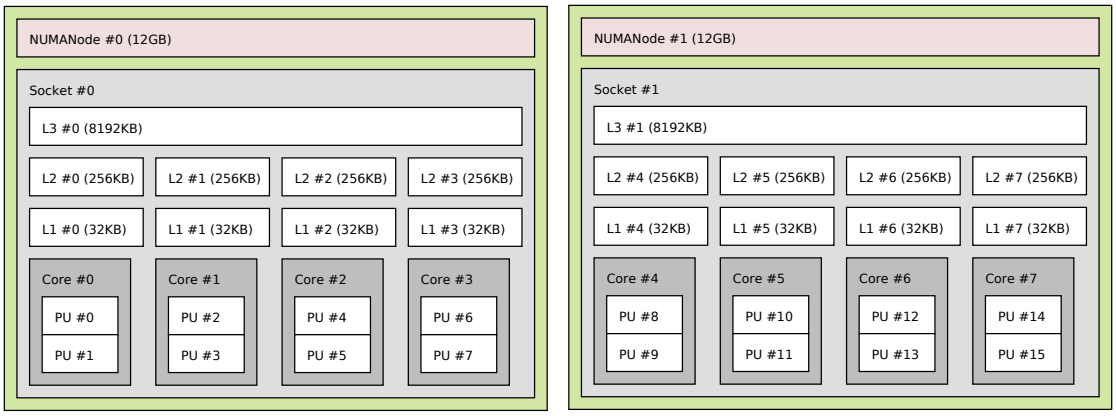


Figure 2.2: GPU Cluster Lehman Node Architecture

High Performance Computing Laboratory (HPCL) at GWU. Pyramid is composed of 128 Sun X2200 compute nodes, each node configured with dual socket 2.2GHz quad-core AMD Opteron 2354 (Barcelona) CPUs with 8GB of RAM; for a total of 1048 cores. The cluster can deliver a theoretical peak of 9.2 TFLOPS. The nodes are interconnected via Mellanox DDR InfiniBand network with full bandwidth. A simple block diagram of the Pyramid cluster node architecture is shown in Figure 2.1. Lehman is a heterogeneous cluster comprised of both traditional CPUs and accelerators namely GPUs. As shown in Figure 2.2, each node contains two quad-core Intel Xeon E5520 (Nahalem) processors with 8GB of memory, an NVIDIA Tesla GPU (which was not used in this thesis) and a Mellanox ConnectX QDR Infiniband network adapter.

Both platforms can be categorized as a typical cluster of SMPs, where multi-socket multi-core shared-memory compute nodes are coupled via high-speed interconnects. Inside the node, CPU sockets are connected together using a ccNUMA setup, with Intel's Quick-Path Interconnect (QPI) or AMD's Hyper-Transport interface between the chips. Both processors feature on-die memory controllers hence sockets are connected to their own memory at a higher bandwidth. Thus, even though all the cores within a node have access to the entire memory space on that node through a cache coherent shared memory system, there is a notion of locality, namely ccNUMA domain. Accesses to memory attached to the same socket will be about 15% to 40% faster than accesses to another socket's memory.

Both the AMD Opteron and Intel Xeon processors used offer four cores on the same die, with each core featuring an out-of-order super-scalar microarchitecture. In addition to scalar units, it also has 128bit wide SIMD units for floating point operations. These processors also feature a similar hierarchical cache at three levels. Each core has separate L1 caches dedicated for instructions and data, and a unified L2 data cache. All four cores further share an L3 data cache. In addition, the Intel Xeon (Nehalem) processors take multithreading even further by running two threads in each core. This is what Intel calls Hyper-Threading. Therefore, the Intel Xeon processor with four cores can run eight threads



(a) (b)

Figure 2.3: Memory Hierarchy of the Dual Socket Nehalem System

simultaneously - two in each core, as shown in Figure 2.3. This multi-socket with multi-core featuring Simultaneous Multi-Threading (SMT) trend will continue to evolve into the foreseeable future, broadening the available range of parallelism per node even when looking at cost-effective server systems. Overall, Table 2.1 provides a summary of the platform information.

2.2 Overview of UPC and GASNet

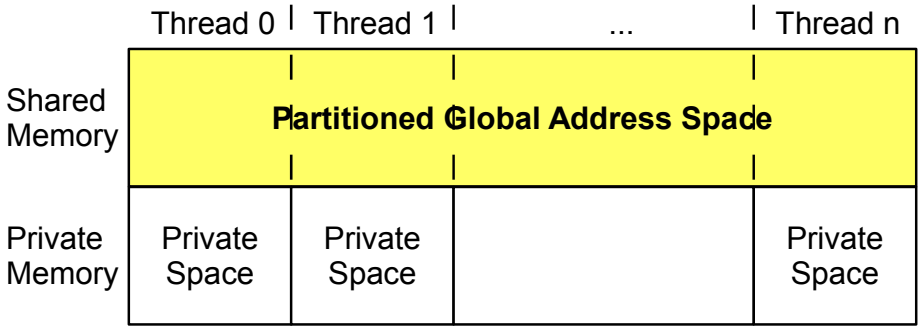


Figure 2.4: The UPC Memory and Execution Model

Table 2.1: Platform Characteristics

Machine Name	Lehman	Pyramid
Machine Location	GWU HPCL	GWU HPCL
Processor Type	Intel Xeon (Nehalem)	AMD Opteron (Barcelona)
Clock Rate (GHz)	2.27	2.2
L1 Cache/Core	32KB(D)+32KB(I)	64KB(D)+64KB(I)
L2 Cache/Core	256KB	512KB
L3 Cache/Processor	8MB	2MB
# Threads/Core	2	1
# Cores/Processor	4	4
# Processors/Node	2	2
# Cores/Node	8	8
# Threads/Node	16	8
Peak Perf. /Node (GFlops)	72	70.4
# Nodes	12	128
Network BW (GB/s)	5 (QDR)	3 (DDR)

UPC is a parallel extension of the ISO C99. The language adopts the Single-Program-Multiple-Data (SPMD) programming model, where each thread runs the same program cooperatively but keeps its own private local data. In UPC, each thread has a unique identifier as the MYTHREAD variable, and THREADS represents the total number of threads, which can either be a compile-time constant or specified at run-time. In addition to each thread's private address space, UPC provides a shared memory area to facilitate communication among threads. While a private object may only be accessed by its owner thread, all threads can read or write data in the shared address space. The shared space is further logically divided into portions each with affinity to a given thread. This mechanism provides a more global view of the user's distributed data structures, yet it allows locality-minded programmers to maintain locality control over data placed in the local shared memory, as illustrated in Figure 2.4. UPC also supports pointers into the partitioned global shared address space, as pointers may be declared to be private (local to a thread) or shared, and may

point to data that is also either private or shared.

The work presented in this thesis uses the latest Berkeley UPC compiler [4] 2.10.2. The compiler translates remote shared accesses into calls to the GASNet communication layer [5]. GASNet is the portable runtime layer that is used for the Berkeley UPC compiler as well as other PGAS languages such as Titanium [6], Co-Array Fortran [7], and Cray's Chapel [8]. It is designed to provide a network-independent, high-performance communication framework tailored for implementing parallel global address space SPMD languages. The low level interface provided by GASNet is primarily intended as a compilation target and for use by runtime library writers as opposed to end users.

2.3 Related Work

In addition to the Berkeley UPC compiler, there are several UPC implementations available on a variety of platforms. These compilers include the GCC-based Intrepid Compiler [9], the experimental MuPC runtime system [10], the Cray UPC Compiler [11], IBM [12] and HP [13] compilers. El-Ghazawi *et al.* [14–16] have evaluated the performance of some of the above compilers with NAS parallel benchmarks, and demonstrated the potential of UPC as a viable programming language and showed their potential performance advantages over MPI as well as areas for further improvements.

There have been some efforts in programming language design to support nested parallelism. For example, Sequoia [17] is an experimental programming language designed to effectively manage data allocation within and transfers throughout the machine's memory hierarchy. When programming in Sequoia, the application parallelism can be aggressively decomposed into arbitrary levels in order to best match the underlying hardware hierarchy. MPI-2 [18] defines dynamic process management, which allows programmers to spawn multiple MPI parallel/communication groups at runtime. OpenMP allows nested parallel regions, as defined in v2.5 [19], but performs poorly without runtime tuning and explicit

control over thread layout and data affinity [20].

Programming models research is a complex problem and there have been many efforts over the years aimed to improve the productivity of parallel programmers. The Hierarchical Tiled Array (HTA) approach investigated in [21] uses array operations to directly manipulate data tiles. Similar to the concept used in Sequoia [17], data tiling is used as an effective mechanism to enhance expression of locality and hierarchical parallelism. Among all the newly developed parallel programming languages and models, we'd like to highlight a new class of Asynchronous PGAS model based languages, these are Chapel [8] and X10 [22]. These languages are still experimental languages designed to enable highly productive programming on emerging petascale computers with a wealth of features and rich abstractions. For example, Chapel adds the 'locales' concept as an abstract object where all threads within a locale have uniform access to memory. Chapel supports programmer control of locality by allowing explicit control of affinity of both tasks and data to locales. Similarly, a 'place' in X10 is a collection of resident activities and objects with uniform memory access within it. Such language-level primitives offer higher abstractions to exploit fine-grained concurrency through hierarchical parallelism and asynchronous tasks. However the high performance implementation of the powerful execution models of these new languages is still a challenging problem.

In other UPC centered studies, Nishtala *et al.* [23] have investigated extending UPC with explicit distribution of shared arrays using Cartesian layouts, combined with Multi-dimensional blocking of UPC shared arrays proposed in [24]. Nishtala [25] further conducted comprehensive study on harnessing GASNet via automatically tuned collectives for ad hoc network interconnections. Such auto-tuned and data driven collective libraries offer high productivity by encapsulating network topology information within the language runtime. El-Ghazawi *et al.* [26] explored hierarchical programming using UPC for high performance reconfigurable computers. Duell investigated two implementation strategies to provide shared memory for GAS languages within shared memory multi-processor sys-

tems [27]. The performance advantages of one-sided communication models are studied in [28]. The one-sided communication and fine grained communication/computation overlapping mechanism have been applied to benchmarks such as NAS FT and shown to scale on large systems such as the IBM BlueGene [29]. Moreover, Dinan *et al.* [30] introduced a parallel task management system that can be fitted with UPC using one-sided communication. The core of the framework adopts a similar work stealing and load balancing scheme the one used in the UTS benchmark later in this thesis. In addition to MPI+OpenMP [31] hybrid programming, recent results also illustrated the benefits of hybrid MPI+UPC [32] programming for scientific applications.

Chapter 3

Exploiting Hardware Topology Using Thread Groups

The complex hierarchical architectures arising from the multicore processors and different interconnection networks have significant impact on the application performance. To effectively utilize these hierarchical systems, many popular parallel languages and software packages provide runtime optimizations to automatically tune itself to the underlying hardware architecture. These efforts are aimed to encapsulate most of the ad hoc hardware information in the runtime. While this problem has been well studied in the MPI and OpenMP community, the efforts in UPC and PGAS languages are relatively new. In addition, providing means for optimizations that can be utilized by developers who are fully aware of the application needs can further increase the performance benefits.

3.1 UPC Runtime Optimizations for Multicore Architectures

Most of the UPC compilers implement UPC language threads as UNIX processes. Access to the global shared memory space is then realized by calling network APIs for inter-process communications, even when processes are located on the same node. Though network specific optimizations for intra-node traffic could always be applied in the lower network service layer independent to the language. In order to improve performance, two mechanisms are used in the Berkeley UPC distributions for shared-memory awareness in the UPC runtime and GASNet communication layer [27]. The first one is to run multiple

threads within a single process. Because threads within the same process share the address space, the UPC runtime can perform shared memory optimizations whenever possible, for example, bypassing intra-node put/get operations to local memory accesses without sorting to GASNet. This is achieved by native support of pthreaded communication clients of the GASNet and significant modifications in the UPC runtime. These modifications are applied in order to comply strictly with the language's memory and execution model, thereby all language threads visible to the end users are still defined at the same level of parallelism. The second mechanism is named as inter-Process SHared Memory (PHSM), which instead of using pthreads, establishes shared memory among intra-node processes by cross mapping memory segments using shared memory map (mmap). In PHSM enabled GASNet installations, a supernode (we define a supernode as 1 or more nodes with cross-mapped segments using PSHM support) discovery is performed internally. It is performed at the initialization stage of the runtime in order to identify which other UPC threads are capable of sharing memory, and further establish the memory map with each other's shared memory segments. With PSHM, GASNet uses two network layers, as PSHM dedicated for intra-node communication and the external network for inter-node. In summary, the shared memory awareness provided by both pthreads and PSHM should be beneficial to most UPC constructs such as barrier, memory copy functions, and collectives. Furthermore, though both methodologies provides similar benefits, PSHM and pthreads are orthogonal to each other and thus they can be applied together.

While compiler and runtime optimizations are welcome features in language implementations, these efforts alone are insufficient to exploit applications specific optimizations and could significantly complicate compiler and runtime system design. For example, neither PSHM nor pthreads could effectively reduce the shared pointer address translation overhead in UPC. Such pointer casting overhead may significantly offset the overall performance [33], making it much more difficult to achieve optimal performance on shared memory architectures such as multicore processors.

3.2 Exposing Architectural Hierarchy to Applications

With today's hardware architecture becoming more and more hierarchical, programming model redesign is, at least, as important as improving the compiler and runtime libraries. To take UPC for example, in the era of uniprocessors, UPC provides support for locality by distinguishing between local shared memory and remote one, which can be treated as two non-uniform memory access levels. Thus data located in a thread local shared space is said to have affinity to the thread. However with today's multicore processors and SMP nodes, the memory hierarchy and locality have become far more complex, as multiple UPC threads within a node/socket/core could have uniform memory access to collective blocks of shared memory. In other words, multiple threads could be considered to have the affinity to a given block of memory. Exploiting local-versus-remote may be good for performance but this will require a high degree of expressivity. Moreover, modern high-end systems such as IBM BlueGene and clusters of SMPs are built as a hierarchy, in which thousands of processors communicate through an interconnection network in some topology, typically some form of a mesh or tree. As a result, increasing levels of the memory hierarchy and wide variety of machines (such as heterogeneous systems) make the prospect of tuning computation and/or communication schedules at the compiler/runtime level infeasible. In order to develop applications that effectively utilize the underlying hardware, programmers are becoming more involved in application level algorithmic optimizations that exploit hardware architectural characteristics. Even though high-level language interfaces such as collective libraries may be optimized or automatically tuned for the underlying hardware topology [25], it is very unlikely that compilers and runtime systems will automatically do a good job of distributing data or tasks to optimize the performance based on the user's computational needs. Therefore, it is important for modern parallel programming languages to provide abstractions and dedicated constructs to manage hierarchical parallelism at the application level.

3.2.1 Managing Hardware Hierarchy Using Thread Groups

In practice, the hardware topology information could usually be abstracted in different levels. For instance, hwloc [34] is an example of low level portable API designed for runtime system developers focused on providing comprehensive functionalities and high expressivity. The way to retrieve and manage hierarchical hardware topology information is generally lacking in UPC. Berkeley UPC provides a specific runtime thread layout query function to discover which threads are relatively closer together than others. For our experimental testbed, this function can only differentiate between the threads within a node and those outside the node.

With the hardware topology information exposed at the application level, the programmer would need control over work and data distribution among the processing elements within a given hierarchy. This will possibly require language constructs aided with some kind of abstraction to ensure productivity and portability. Ideally, in that way programmers can make algorithmic decisions related to data/task distribution and hierarchical execution. This can also provide the means of specifying where on the machine a specific part of the computation should be performed. In practice, applications select the most appropriate thread grouping for the underlying system by querying the hardware attributes at runtime. In that way any given application could adapt itself to a variety of architectural characteristics. Unfortunately, UPC does not provide explicit language features to manage subsets of threads, and there is no related specification to 'thread groups' defined in the UPC Language specifications. A related 'thread team' concept as GASNet team extensions has been proposed in [35]. However, at the time of writing this thesis, the interface for teams for UPC has not been decided on and the preliminary GASNet implementation of teams is not yet publicly released. GASNet teams are designed only to facilitate collective operations on a subset of threads, which is very similar in concept and definition to MPI groups/communicators. Using GASNet thread teams combined with UPC collective operations was

shown to be useful to aid in productivity according to [25].

In addition to creating thread teams as a descriptor for collective communications. The similar thread grouping concept is also featured in several new languages, such as 'locales' in Chapel [8] and 'places' in X10 [22]. As previously mentioned, 'locale' is an abstract object where all threads within a locale have uniform access to memory. A programmer has explicit control over affinity of both tasks and data to locales. Therefore, the *locale* and similar *place* concepts provide richer abstractions to potentially exploit hierarchical parallelism at the application level. Another set of examples of thread grouping are cooperative thread arrays/blocks used in CUDA [36] and very similar *workgroups* in OpenCL [37]. Unlike Chapel and X10 as PGAS languages, CUDA and OpenCL adopt the shared memory programming paradigm. Moreover, CUDA thread blocks or OpenCL workgroups are more restrictive and hardware architectural driven, as thread blocks in CUDA are always explicitly constructed based on thread identifiers and statically scheduled on processor cores. Performance tuning opportunities of faster synchronization and communication amongst threads within a thread block are left in the hands of programmers by explicitly exploiting the local storage within a core.

Although currently lacking the language support, hardware architectural driven thread grouping can be hand-coded in UPC. For example, shared memory thread groups can be identified using a suggested UPC castability extension [38]. A UPC programmer can use this functionality to explicitly identify which other UPC threads are capable of sharing memory, and further use the casted local pointers to directly access neighbors part of UPC shared memory. The overhead of obtaining this neighborhood information and pointer casting is negligible. Because the time consuming memory map or thread creation has been done internally in the runtime up front, using PSHM, pthreads or the combination of both. Similar thread grouping for other hardware hierarchies could also be realized, providing the architectural information could be correctly retrieved during runtime. We believe that the proposed thread groups should be allowed to overlap with each other, therefore multiple

hardware hierarchies could be exploited concurrently. One noteworthy fact for such explicit thread grouping practice is that thread and/or process binding on hardware processing units are required. Though the hand-code practice is still low level, it provides the means to perform shared memory optimizations within a intra-node thread group and to orchestrate parallelism for algorithmic optimizations.

3.3 Application Examples

We selected three benchmarks to demonstrate the effectiveness of application level thread grouping. The thread grouping strategy in our evaluations is limited to shared memory oriented practices. This is due to the fact that the clusters of SMPs we used only exhibit three major distinct levels, which are private memory of a core, shared memory within a SMP node and aggregated global distributed memory. The non-uniform nature of memory access times to the memory of different processors is explicitly exposed in UPC. Therefore thread groups are used to aid in expressivity on shared memory SMP nodes, where processors within a node have uniform memory access to physical shared memory. The first benchmark is the STREAM benchmark, which has a synthetic memory access pattern to show how the expensive shared pointer casting overhead can be eliminated using intra-node thread grouping. The second benchmark is the Unbalanced Tree Search (UTS) benchmark, which uses algorithmic optimizations on locality control of dynamic communications. The third is the NAS FT, which is selected to measure the effectiveness of PSHM and pthreads based runtime, and categorize the performance differences between runtime optimizations and thread group based manual optimizations for bulk memory copies. We have used the same thread grouping techniques in each of these benchmarks. Specifically, the neighborhood relationship information of a shared memory thread group is retrieved at the beginning of an application. Local pointer tables are also created at each thread. The pointer table contains pointers of base addresses of UPC shared memory segments that pro-

vide direct access to the collectively thread group shared memory without expensive shared pointer casting.

3.3.1 STREAM Benchmark

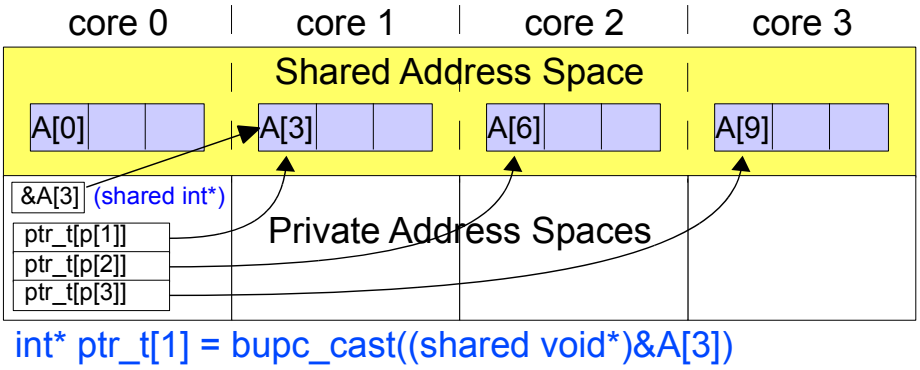


Figure 3.1: Illustration of Shared Pointer Privatization

The UPC and OpenMP implementations of the STREAM triad benchmark are modified with neighborhood odd-even exchange access patterns. Specifically, the benchmark is coded in the way that each thread first allocates and initializes its local part of arrays. As a result arrays are evenly distributed with partial affinity to different threads. During the TRIAD computation stage, a thread with the even rank accesses data located on the thread with an odd rank that is next to it, and vise versa. On a single SMP node, such a memory access pattern will not generally affect the performance of shared memory models such as OpenMP, but may incur significant shared pointer dereferencing overhead in UPC, even the UPC runtime system may perform optimizations of memory access within the node. The output from Berkeley UPC-to-C translator and baseline performance further verify our hypothesis, that every memory access involves a shared pointer translation. Without the recently proposed castability function, UPC necessitates global data re-localization operations in order to improve performance which will be served by either collective or point-to-point communication operations. By taking advantage of the cast pointers pointed

Table 3.1: Performance of the Twisted STREAM Triad

Variants*	Throughput (GB/s)
UPC baseline	3.2
UPC with re-localization	7.2
UPC with cast	23.2
OpenMP baseline	23.4

*Running 8 threads on two quad-core Nehalem with thread binding

at neighbors' data, as illustrated in Figure 3.1, the performance of UPC is equivalent to corresponding OpenMP implementations. The results are summarized in Table 3.1.

3.3.2 Unbalanced Tree Search

The Unbalanced Tree Search benchmark (UTS) represents a class of asynchronous and load unbalanced applications, which need load balancing to relocate the data throughout the execution. The communication behavior in such applications is therefore dynamic. The UPC implementation of the benchmark [39] was used to demonstrate the effectiveness of the global work-stealing in the context of PGAS, as it is usually hard to program such dynamic communications in traditional two-sided message passing models.

In the work stealing implementation of UTS, each thread maintains a steal-stack residing in the UPC shared memory, and does depth-first exploration of tree nodes on its own stack. During the process, tree nodes are popped from the head of the stack, whereas children nodes generated are pushed towards the tail. When a thread has exhausted its local stack, it mush search and steal from among its peers for surplus work. Under work stealing, a thief first randomly selects a victim and checks the victim's steal-stack to determine if any work is available. If so, the thief locks the victim's queue and transfers a certain amount of work from the tail of the victim's queue to its own.

3.3.2.1 Locality-conscious Work Stealing

The baseline UTS implementation used in our test is derived from [39]. Although there have been prior reported works on tuning this benchmark [40,41], to the best of our knowledge this is the first attempt that demonstrates the benefit of the local-stealing strategy. As previously mentioned, the original benchmark uses a randomized victim selection and work stealing scheme. Although random work-stealing is commonly used in shared memory runtime systems such as Cilk [42], the underlying algorithm can be modified to better exploit the new hierarchical architectures. The communication behavior and explicit load balancing scheme used in this benchmark make it best suited for thread grouping techniques. As such, we modified the algorithm with locality-awareness implemented by prioritized work discovery and stealing for threads with high affinity, such as intra-node threads residing within the same shared memory domain. This is because communication with remote threads could be much slower than with local ones. Figure 3.2 illustrates the state machine of each thread's execution. Accesses to peers' steal-stack are always performed directly via the local pre-cast address table. Theoretically, the steal-local optimization should increase the ratio of local steals, leading to a reduction of time spent in work discovery and work stealing.

The local-stealing mechanism does have a drawback though. Since work discoveries are always performed locally first, it would only improve the efficiency when there is stealable local work. Otherwise, it may result in competition and starving situations when there is little to no work available locally. This calls for another optimization, namely rapid diffusion that is used in [40]. In the original implementation, a thief searches and steals a fixed amount of work from a victim through a single steal operation. Previous studies have shown that the work stealing granularity parameter has a strong impact on performance. With rapid work diffusion enabled, a thief steals half of the available work on the victim's work queue as long as the victim's stack meets the particular threshold of work availability.

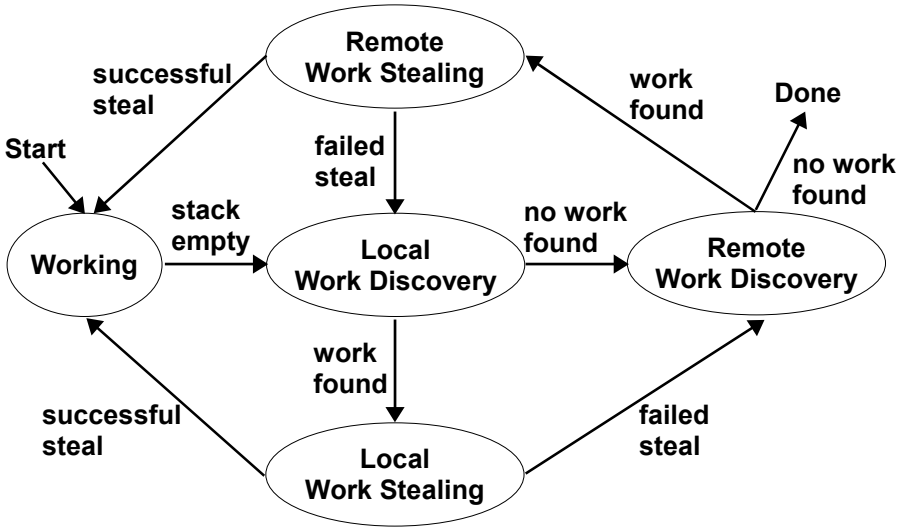


Figure 3.2: State diagram of the hierarchical stealing strategy

This results in faster diffusion of work across different thread groups as it amounts to a workload recursive bisection process. This works naturally since a thread that steals large amount of work remotely will itself become a work source to other local threads in the same group. As a result, such an optimization can further increase concurrency and largely mitigate the local starvation issue.

3.3.2.2 Performance Results of UTS

We compare the performance of the implementation of three variants: the baseline original implementation, locality optimization and locality optimization with rapid diffusion. The default process based compilation with PSHM is used in the particular tests, along with two sets of network conduits, the Gigabit Ethernet and the InfiniBand on the Pyramid cluster. The binomial tree used in our tests has total 4.1 million nodes. The performance results are shown in Figure 3.3. The optimized versions consistently outperform the original on both networks. Clearly, the performance gain in Ethernet is more evident with up to 2x improvement. This is reasonable since the optimization is designed to exploit locality and

Table 3.2: Profiling Results of UTS

Thread configurations* total/local	Overall† improvement	% of local stealing	
		baseline	optimized
Infiniband 32/2	3.4%	36.2%	59.0%
Infiniband 64/4	7.1%	58.1%	82.9%
Infiniband 128/8	11.2%	72.2%	90.9%
Ethernet 32/2	49.4%	18.2%	57.8%
Ethernet 64/4	66.5%	40.5%	81.1%
Ethernet 128/8	99.5%	58.1%	89.7%

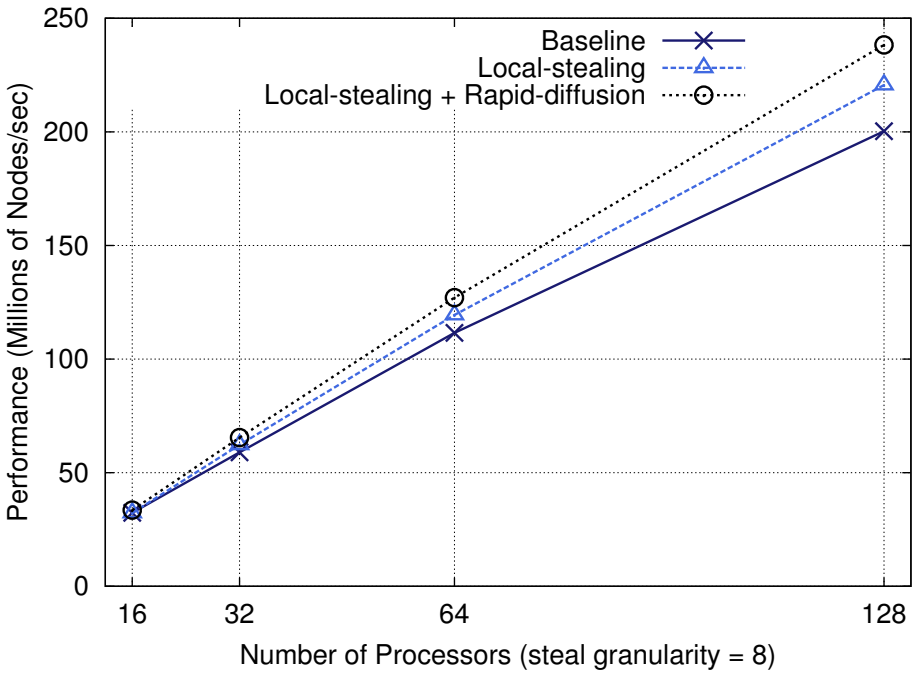
*Using fixed number of 16 nodes

†Time in seconds; Tree size = 4.1 million nodes; Optimized = local-stealing + rapid-diffusion; steal granularity: InfiniBand = 8, Ethernet = 20

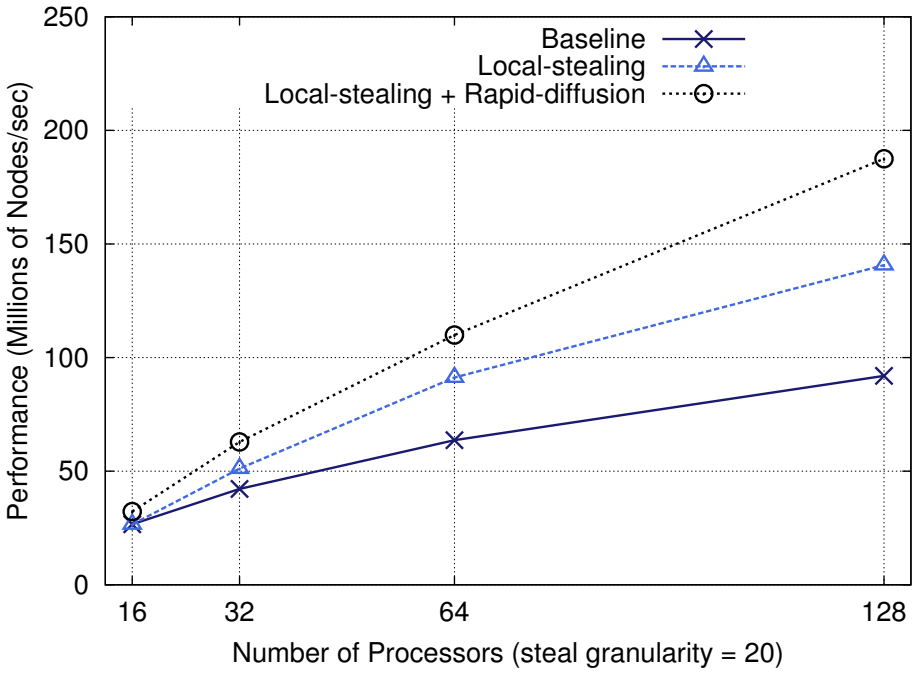
avoid remote accesses as much as possible, which has a pronounced performance impact due to the higher penalty associated with Ethernet communication. The measurements shown in Table 3.2 further reveals that the locality driven thread group optimizations can lead to competitive performance and scaling. The noteworthy fact about Table 3.2 is that local stealing ratio improves with the increasing of the number of local worker threads, even the local/remote thread configuration ratio remains unchanged. This indicates that for parallel applications with global load balancing needs, locality aware load balancing becomes even more crucial for optimal performance and scalability with the increase of local work instances.

3.3.3 NAS FT

The NAS FT benchmark solves a partial differential equation using a series of forward and inverse Fast Fourier Transforms (FFT) over three dimensions. The original implementation of NAS FT is written in Fortran MPI using a one dimensional data decomposition scheme, where two of the dimensions are computed locally. To perform the FFTs along the remaining third dimension, a global exchange involving all processors is used in order



(a) Performance on Infiniband



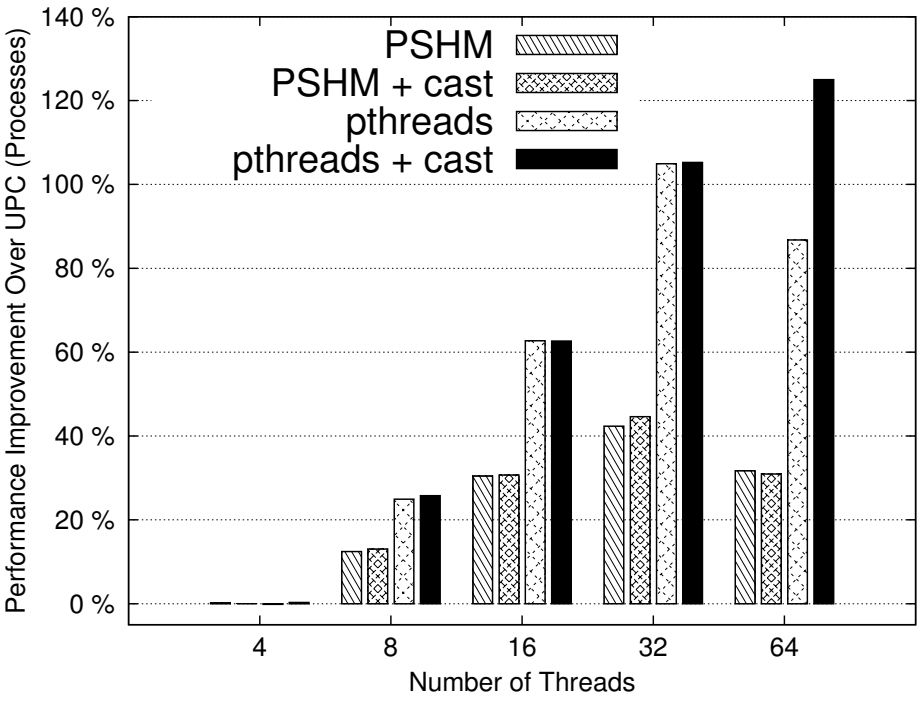
(b) Performance on Ethernet

Figure 3.3: Parallel scalability on 16 cluster nodes (8-way SMP) for the UPC implementation of UTS

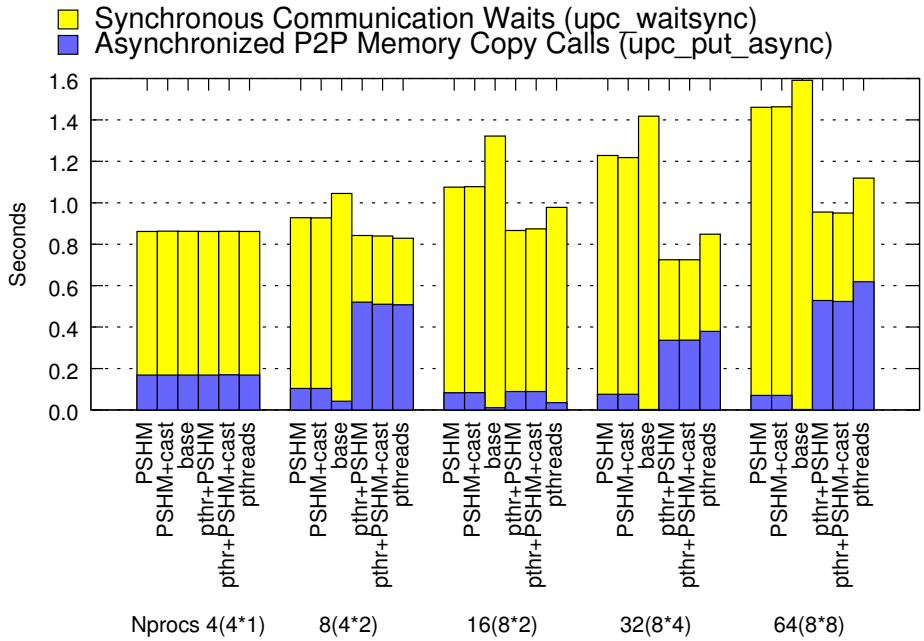
to re-localize the data, where each thread exchanges its portion of the domain with every other thread. Such bulk communication operation is known to be limited in performance by the bisection bandwidth of the network. Further details of the application algorithm and UPC implementations can be found in [28, 46].

3.3.3.1 Communication Performance with Runtime and Manual Locality Optimizations

Ideally, internal runtime optimizations discussed in Section 3.1 should automatically apply to shared memory optimizations on a variety of language constructs such as barriers, memory copy functions and collectives as the runtime system already has full knowledge of how the shared memory locations are mapped onto thread affinities. Thus, applications with static communication/computation patterns as exhibited in NAS FT should safely rely on the runtime system to exploit the heterogeneity in network performance. As a matter of fact, the Berkeley UPC distribution used in our evaluations utilizes some shared-memory awareness optimizations on most of the language constructs. In order to access and compare the performance between runtime optimization and manual optimization, we conducted an experiment to investigate the all-to-all communication performance on a variety of settings. The manual locality optimization is applied to *upc_memcpy* calls with the replacement of the standard *memcpy()* from GLibc, whenever the destination of the memory copy can be cast to local. In other cases, the code is compiled and executed with pthreads and/or PSHM support. The baseline case is also used where neither PSHM nor pthreads is applied. The performance results are summarized in Figure 3.4(a) for synchronized memory copy functions and in Figure 3.4(b) for Berkeley extended asynchronized functions. As observed, the manual optimization of pointer casting with local memory load/store yields an average of 20% of performance improvement over baseline for both blocking and non-blocking cases. However there is no performance difference between manual optimizations and runtime optimizations of PSHM and pthreads, which confirms the effectiveness of au-



(a) Performance improvement on synchronized memcpy functions (upc_memput)



(b) Communication performance using non-blocking memcpy functions (upc_memput_async)

Figure 3.4: NAS FT (class B) all-to-all communication performance with UPC runtime optimizations and hand optimizations on 4 cluster nodes

tomatic runtime optimizations. We also found that when PSHM and/or pthreads is applied, the non-blocking calls take longer time in the network, particularly in pthreads cases. We suspect this is due to the fact that local blocking load/store operations are always used for shared memory communication.

3.4 Summary

This chapter introduced the cooperative thread group approach to facilitate effective mapping of application parallelism on hierarchical architectures. The thread grouping technique could be used universally for both pure application driven partitioning or combined algorithmic optimizations on various network topologies and/or multicore based shared memory systems. The thread group approach provides the means to express asynchronized, modular parallelism while still conforming to the SPMD model and a single level of parallelism.

Chapter 4

Extending UPC with Hierarchical Parallelism

In the previous chapter, the approach of grouping threads according to the hardware hierarchy as cooperative thread groups was discussed. The other important mechanism for managing hierarchical parallelism is to directly apply hierarchically organized execution instances by spawning nested language threads. The latter approach is inherently more intuitive and expressive, and sometimes more effective since execution instances are transparently organized at different levels with probably different responsibility and granularity. As in the previous chapter, we start with reasoning and description of this proposed hierarchical programming model and then discuss the details of its implementation. We further show how communication intensive application kernels such as 3D FFT can realize performance improvements leveraged by the hierarchical threading approach discussed in this chapter. Towards the end of the chapter we discuss the pros and cons of each approach, reasoning how and why both approaches complement each other.

4.1 Extending UPC with Hierarchical Sub-threads

4.1.1 The Approach

Today's parallel languages tend to support only a single level of parallelism explicitly. Recently, there have been some efforts in programming language design to support nested

parallelism. For example, Sequoia [17] is an experimental programming language designed to effectively manage data allocation within and transfers throughout the machine's memory hierarchy. It enables programmers to aggressively decompose an application into arbitrary levels of parallelism in order to best match the underlying hardware hierarchy. MPI-2 [18] defines dynamic process management, which allows programmers to spawn multiple MPI parallel/communication groups at runtime. However, MPI-2 dynamic processes are not well suited for expressing finer-grained, nested parallelism. This is because the programmers are still confined within the message passing paradigm hence hard to achieve performance gains from the use of shared memory optimizations. OpenMP allows nested parallel regions defined in v2.5 [19], but performs poorly without runtime tuning and explicit control on thread placement and data affinity [20]. UPC unfortunately does not support nested parallelism. The most straightforward way to add nested execution in UPC may require recursively partitioning the global address space according to multiple levels of thread layouts. But the inherited SPMD model makes it difficult to express dynamic, nested parallelism. In addition, there is a foreseeable programming complexity to such a solution due to variable scoping and consistency. This may end up requiring substantial changes to the language, which is not desirable for a mature language like UPC.

Since legacy programming models such as message passing and shared memory can not directly address the hierarchical nature of the emerging architectures, hybrid solutions have emerged, e.g. MPI/OpenMP [31]. In most hybrid programming approaches, the parallelism is abstracted in two different programming models at different execution levels. Inspired by this practice, we propose our notion of extending UPC with hierarchical parallelism by combining the strengths of PGAS and shared memory multithreading within a single framework. In our proposed approach, sub-threads are layered on each SPMD UPC thread and coordinate within the local shared memory address space. This is unlike the Berkeley UPC pthreads implementation, where each pthread is treated as a UPC thread and is statically initialized by the UPC runtime. In our approach, a given UPC thread and

its spawned sub-threads work in a master-worker pattern. The master UPC thread can fork and join sub-threads dynamically at arbitrary points within a program.

4.1.2 Comparing to the MPI/Threads Hybrid Programming Model

When comparing to the MPI/threads hybrid programming model, the major difference is that sub-threads in UPC can access the remote distributed memory directly via the global address space provided in the UPC language. This is because UPC as a PGAS language emphasizes global shared arrays as the primary constructs for parallel programming, while MPI provides no support for distributed data structures. The MPI-2 standard [18] added support for one-sided messaging that allows remote processes to access exposed memory windows without any explicit interaction from the host. However, while the MPI-2 provides similar capability to access remote memory in a one-sided manner like GAS languages, it is more restrictive than a global address space in terms of memory coherence, access granularity and synchronization characteristics [32, 43, 44]. Another advantage of the proposed hierarchical UPC/threads model over MPI/threads hybrid may be contributed to its consistency and intuitiveness in both concept and adaptation, as the PGAS model could be considered as a superset of the shared memory programming paradigm, the two programming models are more tightly integrated.

4.1.3 Expected Hierarchical Parallelization Benefits

The proposed UPC with sub-threads programming model preserves the SPMD execution model of UPC on the top. Additionally it adds mechanisms to express fine-grained, dynamic parallelism on sub-threads which may not necessarily be confined to SPMD. Ideally, depending on the runtime implementations, sub-threads scheduling should be encapsulated in a runtime system similar to Cilk or OpenMP. Therefore dynamic scheduling can be abstracted via simple language constructs such as *cilk_spawn* or parallelization clauses such

as *omp for* compiler directives, without explicit data movements and thread identity based work assignments required in UPC/MPI hybrid model. This decouples programmers from explicit management on sub-threads, and allowing node-level dynamic task parallelism to be used within the context of a static data parallelism computation.

In addition to load-balancing and task parallelism on sub-threads, there is a broad range of advantages of the hierarchical UPC/sub-threads approach. The most obvious benefit is the ability to be able to express multiple levels of parallelism, as application developers no longer have to shoe-horn application parallelism into a single-level of execution. Such additional levels of parallelism can not only be used to capture the natural parallelism of an algorithm, but also take advantage of the compute resources such as multicore processors and accelerators such as GPUs. Secondly, it reduces software overhead such as shared pointer dereferencing by removing the PGAS virtualization on sub-threads. Moreover, unlike the UPC pthreads backend, since the global and the static data are now process scoped, thread-local data detections and conversions [27] are no longer needed for sub-threads. Further opportunities such as lower communication overhead will be detailed in Section 4.3.1.

4.2 Implementation Details of the Hierarchical UPC/sub-threads Model

The hierarchical UPC/sub-threads programming approach is new to UPC, therefore the interaction between UPC and user spawned 'sub-threads' are not yet defined in the language. As such support or any specific definitions of such interoperability is lacking in current UPC implementations. In order to realize our proposed hierarchical programming model, we experimented with external shared memory multithreading models such as OpenMP and Cilk++ mixed with UPC. The native support is certainly of interest so we also developed a pthread based shared memory runtime for extensive evaluations.

4.2.1 The Interoperability of UPC with Cilk++ and OpenMP

As mentioned earlier, UPC/threads model allows sub-threads access to the entire GAS. This is possible only when sub-threads are reachable in the UPC context. For example, Cilk++ as an extension to C++ makes it impossible to be mixed with UPC in a same source file, but Cilk++ parallelized functions with ”extern C” signatures are callable from UPC. Such a hybrid configuration requires the least compliant runtime/compiler support from both languages, since the parallel control flows are completely separated, and sub-threads initialized in Cilk++ are only able to access the master UPC thread's private and local shared memory.

Ideally, in the case of UPC/OpenMP hybrid, as shown in Figure 4.1, it should simply be UPC source codes with additional OpenMP directives/pragmas. With OpenMP parallel constructs embedded in the UPC SPMD contexts, sub-threads would have direct access to the global address space and use the syntax freely. However the Berkeley UPC and GCC UPC [9] compilers' front-end as of now do not support such a kind of hybrid compilation. To explain in detail, the Berkeley UPC uses a source-to-source compilation strategy that translates UPC code to C intermediate code first with inserted calls into the UPC runtime. However the front-end translator is oblivious to OpenMP pragmas. This subsequently makes the OpenMP parallel constructs meaningless in the translated code (which is not intended to be incrementally modified) and produces errors in the next compilation stage. We manually-modified the translated C codes to correctly put OpenMP constructs in place for our experimental evaluations. As mentioned above, the main goal of this work is to exploit and evaluate hierarchical programming techniques using UPC. We first want to demonstrate their usefulness, and consider the specific compiler work around of this issue to be outside the scope of the research.

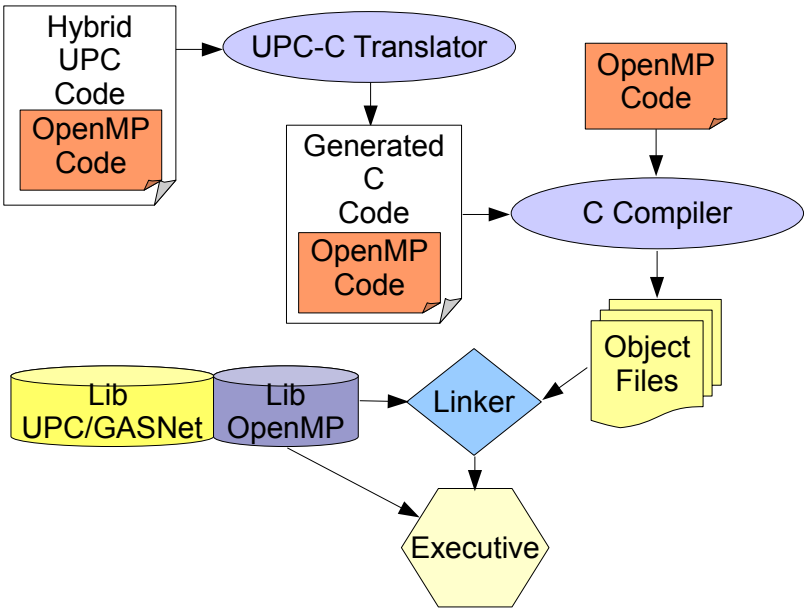


Figure 4.1: Illustration of Compilation Flow of UPC/OpenMP hybrid

4.2.2 Prototype Implementation of the Native Sub-threads Support in UPC

The in-house prototype runtime library that we implemented uses the thread pool pattern as the strategy to minimize the overhead associated with creating and destroying threads. It specifies a central task queue associated with a pool of threads. The task queue allows the execution engine to automatically balance supply and demand for threads across multiple tasks. At the application level, the user is responsible to spawn independent tasks/activities to the thread pool, which is similar to *cilk_spawn*. It also comes with supportive interfaces such as barriers and management functions for sub-threads.

4.2.3 UPC Interaction with User-spawned Threads

As mentioned earlier, there is currently no definition in UPC for interaction with threads. The MPI-2 standard defines the following levels of thread safety:

1. MPI_THREAD_SINGLE: Only one thread will execute.

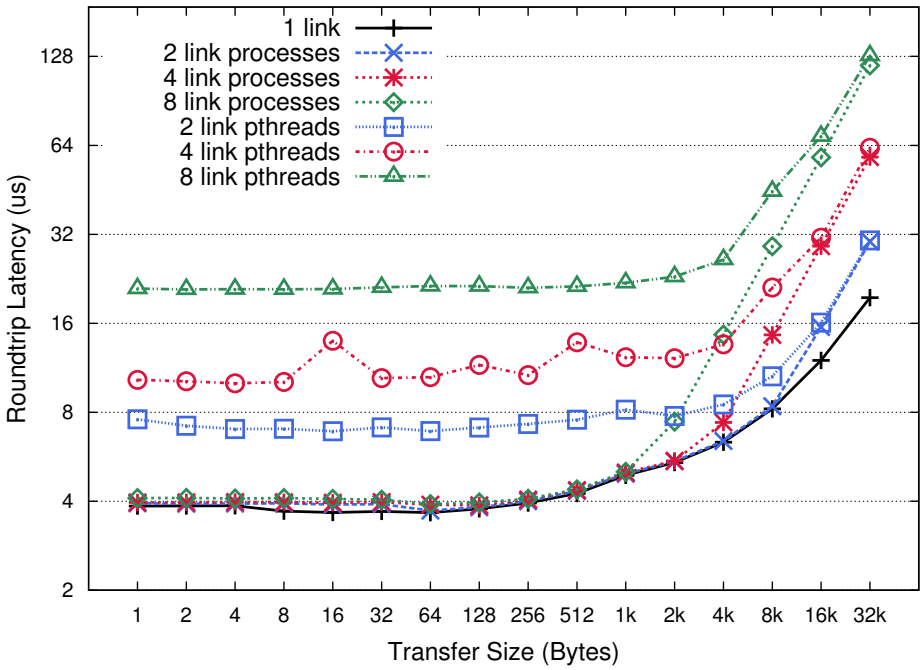
2. MPI_THREAD_FUNNELED: The process may be multithreaded, but only the main thread will make MPI calls.

3. MPI_THREAD_SERIALIZED: The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time.

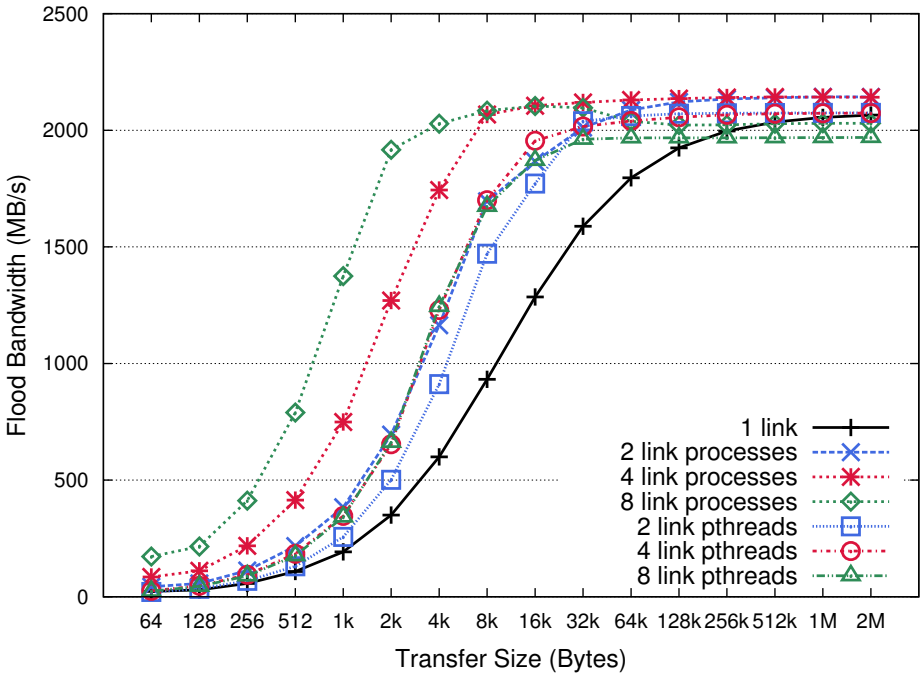
4. MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions.

In a thread-compliant implementation, a UPC thread is a process that may be multi-threaded. It is desirable that UPC calls from sub-threads be safely executed concurrently. In addition, a blocking UPC call should block only the invoking thread allowing the remaining threads in that process to continue execution. To realize all the benefits of the UPC/threads model, it is necessary to add thread safety support, e.g., THREAD_MULTIPLE. This is because unlike MPI, UPC supports implicit shared memory style remote accesses. Thus, a non-thread safe implementation necessitates manual interceptions on calls to the network to be serialized, which could be a significant distraction to programmers.

Based on our observations, the current Berkeley UPC run-time support of thread safety is analogous to THREAD_FUNNELED, also with limited constructs that are capable of THREAD_SINGLE. The Berkeley UPC runtime and GASNet communication libraries in fact provide thread-safe configurations for various network implementations to support the UPC pthreads backend. However, these thread-safe libraries rely on pthread-specific data for many important runtime variables. Due to the intrisic difference of legacy UPC pthreads implementations and our proposed UPC/sub-threads hybrid model, user-spawned sub-thread, which lacks the per-thread data used by the UPC runtime, can crash at the invocation of any functions that need the thread-specific data. Instead, it would require a thread-safe UPC runtime which implements per-UPC-thread data similar to how the regular C variables are scoped to the process [45]. We have reported and consulted this thread-safety issue to the Berkeley UPC team. At the time of writing this paper, the Berkeley UPC enhancement of thread-safety support for user-spawned threads is still experimental.



(a) Message Latency (*upc_memgut*)



(b) Unidirectional Flood Bandwidth (*upc_memput*)

Figure 4.2: Multi-link Network Microbenchmark Performance

4.3 Preliminary Evaluation

4.3.1 Microbenchmarks

With escalating parallelism within a node, contention within shared resources such as the network adapter becomes a major concern. To quantitatively see the performance impact of network contention we present the multi-link latency and bandwidth microbenchmarks. Through the first benchmark we analyze the unidirectional flood bandwidth performance and the second measures round-trip latency. The benchmark is written using various combinations of blocking and non-blocking *upc_memcpy* operations, but only the standard *upc_memput()* results are reported here. The tests are conducted on two nodes of the Lehman cluster connected via the QDR InfiniBand network. Each compute node has up to eight UPC language threads with each one initiating a series of point-to-point put operations of a specified size to its corresponding receiver thread residing on the other node. Therefore a varying number of network link-pairs are established. We vary the size of messages over powers of two and for each data point report the median bandwidth or latency achieved. To provide a further comparison, we measured the performance achieved with both process-based and pthread-based messaging link-pairs. The key differences between pthreads and processes is that pthreads share the same set of network connections/handles. Processes, on the other hand, will each have their own network connection. In our tests this results in a difference of 1 network connection per cluster node for pthreads, versus 1 per core on the node for processes.

Figure 4.2(a) and 4.2(b) show the performance measurements of multi-link latency and flood bandwidth respectively. First of all, as the plots show, the use of multiple UPC threads per node as communication link-pairs yields increases in both bandwidth and software overhead (latency). For small and mid-range message sizes, there is a significant improvement in the achievable data transfer bandwidth when more than one UPC threads are used. On the other hand, communication latency also dramatically increases starting

from the message size of 1KB. In comparison with process-based and pthread-based performance, the data show that the pthread-based link-pairs are able to extract less of the throughput than its process-based counterpart. This is mainly due to the presence of the additional network bandwidth made available through multiple network connections, especially for small message sizes. In terms of latency, latency for pthreaded messaging appears serialized with the increase of the number of UPC threads. These results could be attributed to the negative side-effect of sharing a single network connection, due to which only a single thread can enters the network at a time.

However, it cannot be definitively said at this point that the end result favors processes. On the contrary, there are reasons that UPC pthreads or user spawned sub-threads (which also use pthreads in our experiments) sharing a network connection is better positioned for today's multicore based systems. One important consequence of threads sharing a single network connection is that it allows an application to use a network with a smaller number of addressable nodes. This can potentially provide speedups and memory savings of communication buffers allocated by the runtime system, which is becoming a serious issue with the increase in the number of cores in supercomputing systems [27]. Another benefit of threads sharing a network connection is less software overhead and contention within the communication runtime system. This is because concurrent access to network resources needs to be handled at some level in the software infrastructure. When processes are used, contention in the lower network API level is likely to be slower. It is worth noticing that the end communication behavior and tradeoffs involved in the mentioned threads and processes are likely to vary across different network architectures and APIs. However, with the increasing of fine-grained parallelism such as Simultaneous MultiThreading within a node, we suspect that sharing a network connection using pthreads as UPC threads or sub-threads may be more efficient. In practice, in order to maximize performance of an application, experiments with different thread/process ratios within a node (therefore the number of network connections) may be needed to determine the optimal solution. Later in

Section 4.3.3 we will see that for the 3D FFT algorithm, the execution time is determined by the communication performance in the middle range of message sizes.

4.3.2 STREAM Benchmark

An important step in application tuning on the ccNUMA architecture involves careful placement of threads and data. The users must take data and sub-thread affinities into account when trying to exploit the actual hardware performance. Specifically, as shared arrays are always 'first touched' by UPC threads, sub-threads that tightly cooperate underneath a given UPC thread should be placed onto cores sharing a cache or confined to similar affinity masks. However, two independent UPC language threads (as processes) should probably be spread across different sockets to maximize the memory throughput and to reduce thrashing on caches.

In order to examine how hybrid thread placement can affect the performance, we modified the STREAM triad benchmark using hybrid UPC/OpenMP. The arrays in the original benchmark are allocated as shared arrays in UPC, and accessed by OpenMP sub-threads during computation. The tests are conducted on a single node of the Lehman cluster. We used *numactl* to explicitly bind UPC language threads as processes on ccNUMA nodes (as CPU sockets) in a round-robin manner. Sub-threads within a process all inherit the same affinity mask. Therefore, it is guaranteed that OpenMP sub-threads spawned by a given UPC thread run exactly within the same multicore processor as the master UPC thread.

It is worth pointing out that the characteristics of the STREAM triad benchmark render no performance gain by using hierarchical threads. This is because the benchmark simply performs a series of multiply-addition operations on three vector arrays. The performance measurements shown in Table 4.1 therefore only reveal the importance of careful thread placement while using hierarchical threads. Compared to the original UPC and OpenMP implementations, a single UPC thread with 8 OpenMP sub-threads without thread binding performs poorly achieving merely a little more than half of the optimal throughput.

By forking two UPC threads onto separate ccNUMA sockets with OpenMP sub-threads binding on the same socket, identical throughput can be achieved. Therefore, in other experiments, UPC processes are cyclically pinned to independent ccNUMA nodes (CPU sockets) using *numactl* by default.

Table 4.1: Performance of the STREAM Triad

Variants	Thread Configurations (UPC*OpenMP)	Throughput (GB/s)
UPC	8	24.5
OpenMP	8	23.7
UPC*OpenMP	1*8	13.9
	2*4	24.7
	4*2	24.7

4.3.3 NAS FT

As the previous sections show, network contention from concurrent UPC threads within a node has significant performance impact, pthreads UPC and hierarchical UPC with sub-threads can potentially mitigate this issue. In this section we see how the hierarchical UPC with sub-threads approach can improve the overall application performance while providing higher expressivity with hierarchical parallelism. We used the NAS Parallel Benchmark [46] FT as a case study.

4.3.3.1 Algorithm Sketch and Hierarchical Parallelism Adaptation

The benchmark solves a large three-dimensional Fast Fourier Transforms (FFT). Previous efforts from the Berkeley UPC group [28, 29] have demonstrated the effectiveness of one-sided communication and fine-grained computation/communication overlap. The hand

tuned UPC implementations in those studies can fully outperform MPI counterparts implemented using the same pattern.

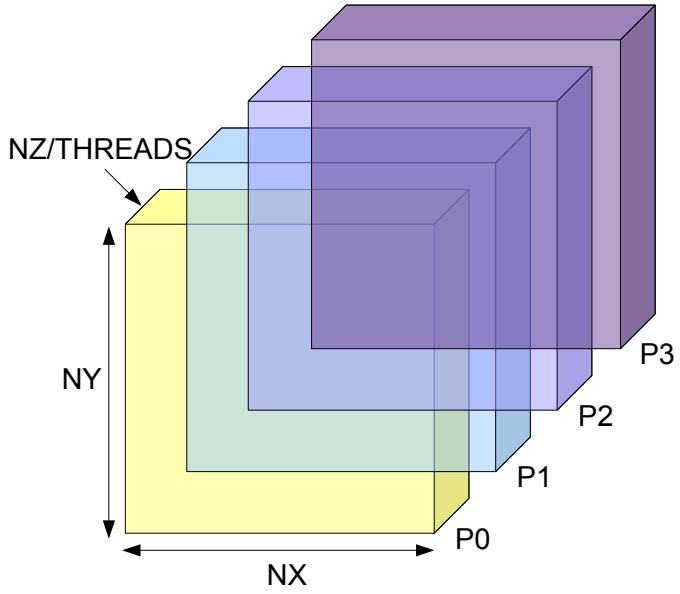


Figure 4.3: 1D decomposition for FFT

Our baseline UPC implementation of the FT benchmark derived from the Berkeley versions uses an one dimensional data decomposition scheme. As illustrated in Figure 4.3, two of the dimensions are computed locally. To perform the FFTs along the third dimension, a global exchange is used to re-localize the data, where each thread exchanges its portion of the domain with every other thread. Such all-to-all communication step is implemented using point-to-point memory copy operations rather than collectives that were used in the original MPI version. This is largely due to the fact that collective operations are not yet supported on sub-threads. The other parts of the benchmark are mostly computationally intensive, local bulk computations such as *FFTs*, *local transpose* and *evolve* exhibit data and memory intensive hierarchical parallelism opportunities for sub-threads. The domain decomposition on sub-threads follows a similar pattern by further slicing the slab into evenly distributed 2D planes on sub-threads.

Two major variants of the benchmark are used for our evaluation. The first one namely split-phase uses the same bulk synchronized pattern used in Fortran-MPI, in which computations and communications are carried out separately in distinct phases: after computing the FFT over all its planes, every thread locally transposes the computed data into an ordering suitable for the exchange operation. After the global exchange, the data is re-transposed to complete the remaining FFTs. The second variant applies the communication and computation overlap algorithm detailed in [28]. In this implementation, each thread initiates the communication on a 2D plane (using non-blocking calls) as soon as the 2D FFT computation on that given plane is finished, as there are no further dependencies. Therefore the computation of a second plane can be overlapped with the distribution of the previous computed plane. The interleaved communication and computation pattern necessitates concurrent sub-threads access to the UPC shared memory from sub-threads. This is unlike the former split-phase implementation which can be realized with the 'master-only' hybrid thread execution model, where all-to-all communications are only carried out by the root UPC thread.

4.3.3.2 Experimental Setup

Experiments were conducted on both clusters, Lehman and Pyramid. In our experiments we fix the problem size and use the NAS FT Class B (512*256*256 double complex numbers). For the evaluation we used 8 nodes from Lehman and 16 nodes from Pyramid. Since we are more interested in the strong scaling performance on multicore processors, we vary the processor count from 1 core per node to a full set of 8 cores per node. The number of cores used per node in the performance evaluation is thereby *n/8* for Lehman and *n/16* for Pyramid, as *n* being the total number of cores used within the entire system. To perform the FFTs we used the FFTW 3.2.2. The MPI implementation we used is OpenMPI v1.3. The C compiler used is GCC version of 4.3.3, which comes with OpenMP v2.5. The UPC compiler used is Berkeley UPC 2.10.2 with GASNet 1.14.2. PSHM is enabled by default

in both pure UPC process based compilations and hybrid with UPC pthreads. We also used Cilk++ build 8503 (GCC 4.2.4).

4.3.3.3 Application Performance Results

First of all, we examine the parallel scalability of distinct steps of the benchmark in Figure 4.4, tested on the Lehman cluster. As the data show, local bulk computation kernels scale perfectly across all cores. This is mainly because the benchmark is coded in such a way that the data to be processed for a given UPC thread are always local. Unlike compute kernels, the all-to-all communication step does not scale beyond 16 UPC threads, or 2 threads per node. The major cause for the decay can be attributed to the contention on the network. Additionally, the difference from 64 threads to 128 threads shows some interesting kinks. This is because at 128 threads, two threads are running per core as HyperThreads (or SMT). As a matter of fact, the instruction units, cache, and floating point units are shared resources amongst the two SMT threads. Therefore, computation kernels such as *FFTs*, *evolve* and *transpose* demonstrated limited speedups on two SMT threads at 5% to 30%, which may be contributed to better latency hiding of memory accesses and/or branch misses.

As the all-to-all communication step dominates the overall execution of the benchmark, we highlight the communication performance collected from Lehman and Pyramid separately in Figure 4.5(a) and Figure 4.5(b). Because the problem size is fixed in our tests, the overall amount of data to be globally exchanged during the communication is unchanged. Thereby, the message size decreases but the messaging rate increases as the number of threads used increases. As the data show, for both platforms, the all-to-all communication does not scale when running more than two UPC or MPI threads per node, indicating increased software overhead resulting from network contention. When comparing the UPC implementations, UPC pthreads realizes stronger strong scaling than UPC processes. These results largely go with our experiences and expectations that originate

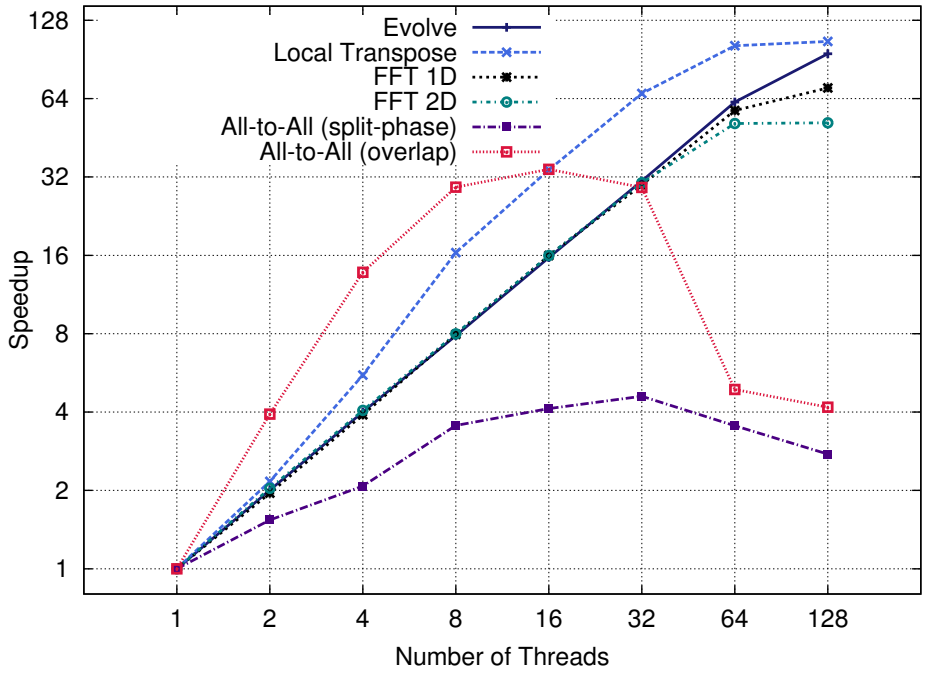
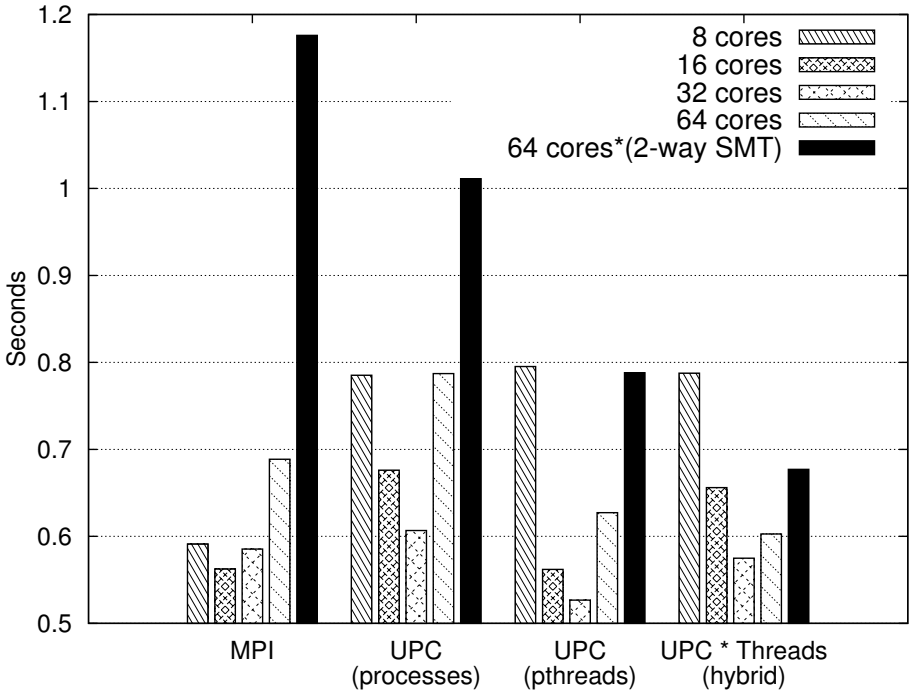
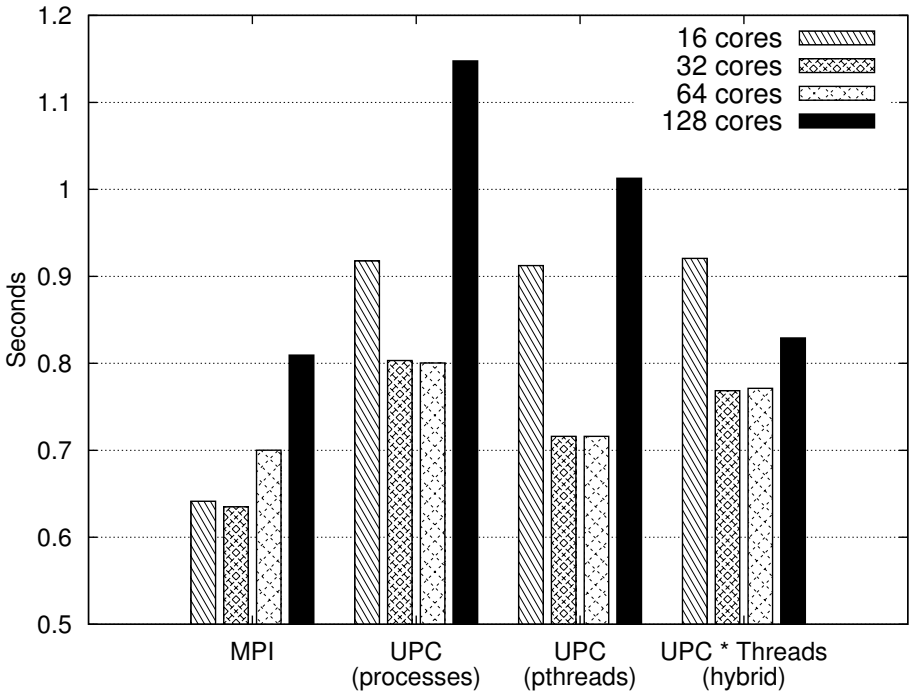


Figure 4.4: NAS FT Runtime Performance Breakdown

from the micro-benchmarks. However pthreads UPC still performs poorly when scaled beyond a certain capacity. The hierarchical sub-threads on the other hand, offer the same computational concurrency but potentially have the least software overhead. This is due to the fact that data movement is mainly handled by master UPC language threads while sub-threads as user-spawned worker threads do not implicitly poll on the GASNet communication system. As a result, though the hierarchical sub-threads may be less attentive (timely processing of incoming communication calls) than UPC threads, they scale better without the tendency of swamping the runtime and communication system, which explains the better performance of sub-threads when all cores within the nodes are utilized. Additionally, the results also show that MPI outperforms UPC on both platforms. This is largely due to the optimized collective functionalities used in the MPI-Fortran implementation. However the data points indicate that the collective operations in MPI also scale badly over 2 cores per-node.

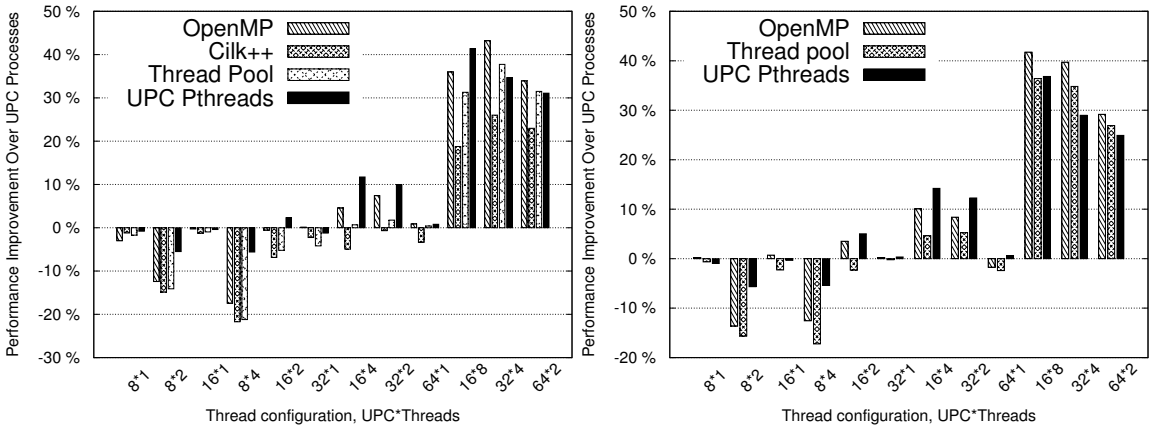


(a) Lehman



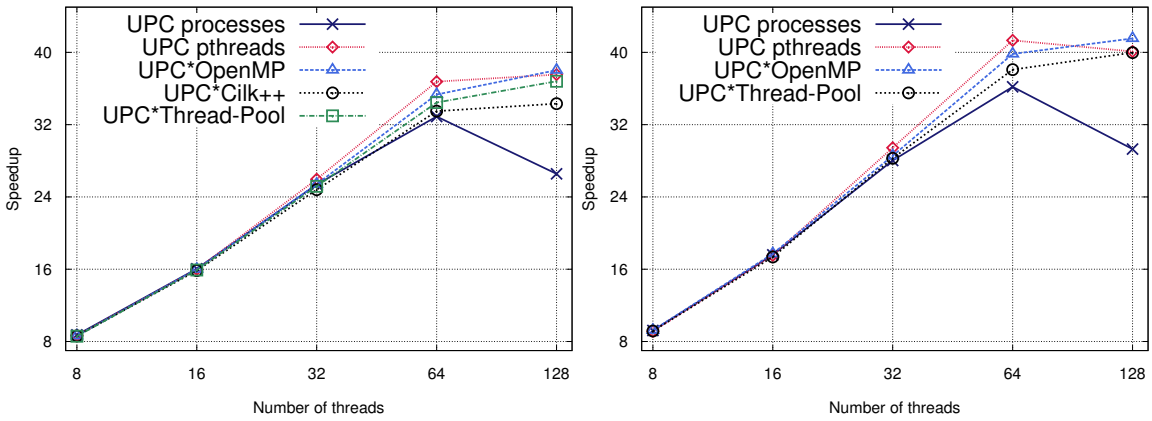
(b) Pyramid

Figure 4.5: Time spent in communication calls of the split-phase implementation



(a) Relative Performance of Split-phase

(b) Relative Performance of Overlap



(c) Scalability of Split-phase

(d) Scalability of Overlap

Figure 4.6: NAS FT Class B (512*256*256) Performance Results

The results in Figure 4.6 summarize the overall performance scaled on 8 cluster nodes of Lehman starting from 1 thread per-node to a full set of 16 threads per-node. The process/thread ratio is a major parameter that affects the overall performance. We have tested various combinations among the range of cores to the full set of possibilities. Figure 4.6(a) and 4.6(b) show the performance comparison of different thread configurations relative to pure process based UPC for the split-phase and the overlap implementations respectively. The performance decay on *8*n* configurations of hierarchical UPC with sub-threads can be attributed to unbalanced thread subscription and CPU usage. Since we bind processes on sockets in order to prevent sub-threads going off the chip, only one socket/chip is used per node for *8*n* configurations. When comparing the performance of UPC/sub-threads with pure UPC, the profiling results reveal that the synchronous communication performance is the major differentiation factor to the overall performance. This is because the NAS FT benchmark is load-balanced as the 3D prism is evenly decomposed into threads. Thereby there is no particular benefit of using shared memory among sub-threads compared to a same amount of UPC threads under PGAS, as only static work sharing are used on sub-threads. Subsequent examination further revealed that sub-threads implementations of computation kernels perform just the same as pure UPC. Among all the sub-threads variants, OpenMP hybrid performs the best, followed by the prototype runtime implementation of thread pool. Cilk++ performed the worst with up to 10% of slowdown on FFTs and a consistent 0.2 seconds of lag on single sub-thread configurations, which may be attributed to higher runtime overhead of Cilk++. Additionally, sub-threads perform generally the same as pthreaded UPC, but scales better especially for SMT threads, as shown in Figure4.6(c) and Figure4.6(d). The average speedups of hierarchical UPC with sub-threads over legacy process based UPC are about a mean of 10% for 64 threads and 30% for 128 threads. Overall, these results show the overall performance improvement achieved through using hierarchical light-weight sub-threads in the context of UPC. The results also illustrate that UPC or MPI 'everywhere' across multiple hardware hierarchies may not al-

ways be optimal. Not only are they limited to a single level parallelism, but the software overhead could be another issue that prevents performance scaling as the number of cores grow.

4.4 Discussion: Comparing the Two Approaches

As discussed in the last section, there are no native language constructs in UPC that support the expression of hierarchical parallelism, we abstract the hardware topology information either by thread groups or layering sub-threads. The thread group approach is more traditional and low level. It requires programmers to retrieve the hardware topology information during runtime, and then orchestrate application parallelism accordingly using thread groups. Ideally, the UPC runtime system should perform most of the optimizations automatically with techniques such as PSHM, pthreads backend and automatically tuned collective communications, where different communication protocols are used internally. In fact, the latest Berkeley UPC compiler that we used in this work adds shared memory awareness in most of the UPC constructs. However, thread grouping provides the means for performance minded programmers to optimize the performance based on the application's computational needs with hardware architecture awareness, for example, the local stealing strategy we applied in the Unbalanced Tree Search benchmark. Moreover, unlike the hierarchical threading approach relying on fine-grained sub-threads, thread groups can be arbitrarily mapped on any given hardware architectural levels with the potential of overlapping of multiple groups.

The hierarchical UPC with sub-threads approach is limited for some levels of shared memory parallelism within a system, such as within a multicore processor. A broad range of benefits of this approach has been detailed previously in Section 4.1.3. To realize most of the benefits, it requires runtime libraries to support both UPC and shared memory sub-threads, and a thread-safe UPC runtime library is a must. Moreover, for hybrid program-

ming with OpenMP, compiler support for mixed source code compilation is needed. The hierarchical sub-threads approach is also highly dependent on optimal process/thread configuration and placement. However, there are no standard mechanisms or APIs to achieve thread and process placement, and the optimal number of threads per process may not be known beforehand of the execution.

In practice, the hierarchical UPC/sub-threads approach would come in handy for many scientific applications that could make optimizations with multi-level domain decomposition. On the other hand, it means that only applications with multiple levels of parallelism, especially with enough fine-grained parallelism to allow shared memory sub-threads scaling to the number of cores per socket/node, can be benefited from such an approach. Some other applications can be written using simple data/task parallel abstractions without using multi-level parallelism, the thread group approach would fit better in these cases, such as UTS, Random Access, etc. For scenarios that require extensive application optimizations and scaling on large scale supercomputing systems, both thread grouping and hierarchical sub-threads could be applied together. For example, a programmer may spawn sub-threads on multicore chips while grouping multiple chips within a node for an aggregated thread group representing all the threads within a node. Thus we argue that both approaches are important and satisfy different application requirements and can be used together. We are less concerned about the exact formal specification in the language to realize the hierarchical programming approaches we exploited in this research. However, we found both approaches could use minimal language extensions that further complement the existing UPC language semantics and improve expressivity. As a result, this would allow programmers to incrementally optimize legacy UPC codes with more elaborated hierarchical parallelism, allowing more levels of parallelism in the hardware to be targeted.

4.5 Summary

By mixing with shared memory style sub-threads, UPC can support a hierarchical programming model with set of mechanisms to express hierarchical parallelism and asynchronized tasks. We believe the UPC/sub-threads hierarchical model is more suitable for addressing increasing fine-grained parallelism within a chip and potentially heterogeneous computing. Furthermore, as the computation domain is being partitioned only amongst the UPC language threads, communication behavior is more centralized resulting in less contention on the network. We discussed our experiences with the implementation of sub-threads and evaluation using synthetic and application benchmarks. The results we have observed from the NAS FT show significant communication performance improvement compare to base UPC implementation, especially when processing cores within a node are fully utilized. As a result, the hierarchical sub-threads variants of the benchmark consistently show superior scalability. In summary, the hierarchical programming model using sub-threads has the potential to achieve high performance with little incremental programming effort, and is better positioned for future processors and systems exhibiting increasingly hierarchical parallelism.

Chapter 5

Conclusions

The problems and challenges for developers in this quickly evolving computational landscape are daunting. The architectural trends have moved towards increased parallelism and hierarchy of parallelism. Software on the other hand must address those changes in order to harness the full power of these new architectures. This research addressed this with parallelism-hierarchy-conscious programming. We aim to provide extensions to UPC with application level hierarchical parallelism constructs to better exploit today's systems that exhibit multiple levels of execution and memory hierarchies. This thesis presents two approaches to facilitate portable mapping of applications to hierarchical architectures. The first mechanism groups subsets of threads according to application needs and/or underlying hardware hierarchies. The second one extends PGAS/UPC with shared memory subthreads for hierarchical memory access and thread execution. Both approaches facilitate the direct manipulation of hierarchical parallelism at the application level. The approaches complement each other for different application cases. Experimental results indicate that both approaches can enhance applications to fully exploit the power that the system offers. By employing the explicit hierarchical programming model, we have demonstrated performance improvements in several parallel application benchmarks at the negligible increase in code size and minimum cost of programming effort; NAS FT performance increases by a factor of 1.4 using hybrid nested UPC/threads, and UTS experiences a two-fold speedup

using thread groups that span on 8-way SMP nodes.

A future goal is to extend our study to express and exploit asynchronous, hierarchical parallelism to effectively leverage heterogeneous resources such as GPUs, where clusters of GPUs already exhibit rich hierarchy of parallelism. Ultimately, the UPC language itself should support hierarchical parallelism in its native semantics. It would be an interesting direction to incorporate other related efforts such as multi-dimensional array blocking and placement, global task parallelism and runtime optimizations with the hierarchical programming model investigated in this research. Furthermore, more applications need to be examined and developed in UPC especially with hierarchical parallelism. Results from such studies obtained from a wider variety of algorithms and platforms will provide more accurate and deeper understanding of various aspects of the proposed hierarchical programming model, and yield more robust arguments to influence the future language direction.

Bibliography

[1] D. Bonachea, P. Hargrove, R. Nishtala, M. Welcome, and K. Yelick, "Optimized collectives for PGAS languages with one-sided communication," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 143.

[2] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick, "Automatic nonblocking communication for partitioned global address space programs," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 158–167.

[3] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained UPC applications," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2005.

[4] *The Berkeley UPC Compiler. http://upc.lbl.gov.*

[5] *GASNet home page. http://gasnet.cs.berkeley.edu.*

[6] *Titanium home page. http://titanium.cs.berkeley.edu.*

[7] *Co-Array Fortran Language. http://www.co-array.org/.*

[8] *The Chapel parallel programming language. http://chapel.cray.com/.*

[9] *GCC/UPC Compiler, http://www.gccupc.org/*, Intrepid Technology, Inc.

[10] Z. Zhang, J. Savant, and S. Seidel, "A upc runtime system based on mpi and posix threads," in *PDP '06*, 2006.

[11] *Cray C/C++ reference manual. http://docs.cray.com/books/004-2179-001/html-004-2179-001/*, Cray Inc.

[12] *IBM XL UPC Compilers. http://www.alphaworks.ibm.com/tech/upccompiler*, IBM.

[13] *HP UPC. http://www.hp.com/upc/*, HP.

[14] F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. El-Ghazawi, "Performance monitoring and evaluation of a upc implementation on a numa architecture," in *Proc. of 17 International Parallel & Distributed Processing Symposium (IPDPS)*, 2003.

[15] T. El-Ghazawi and F. Cantonnet, "Upc performance and potential: A npb experimental study," in *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002.

[16] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity analysis of the upc language," in *IPDPS*, 2004.

[17] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 83.

[18] *MPI: A Message-Passing Interface Standard, v2.2*, MPI Forum, 2009.

[19] *The OpenMP API. http://openmp.org/wp/*.

[20] L. Huang, B. Chapman, and C. Liao, "An implementation and evaluation of thread subteam for OpenMP extensions," in *PMUP '06, Workshop on Programming Models for Ubiquitous Parallelism*, Seattle, WA, 2006.

[21] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 48–57.

[22] *The X10 programming language. http://x10-lang.org/*.

[23] R. Nishtala, G. Almasi, and C. Cascaval, "Performance without pain = productivity: data layout and collective communication in upc," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, USA, 2008.

[24] C. Barton, C. Cascaval, G. Almasi, R. Garg, and J. N. Amaral, "Multidimensional blocking in UPC," in *Technical Report RC24305, IBM*, july 2007.

[25] R. Nishtala, "Automatically tuning collective communication for one-sided programming models," Ph.D. dissertation, Computer Science Division, UC Berkeley, December 2009.

[26] T. El-Ghazawi, O. Serres, S. Bahra, M. Huang, and E. El-Araby, "Parallel programming of high-performance reconfigurable computing systems with Unified Parallel C," in *Proc. of Fourth Annual Reconfigurable Systems Summer Institute (RSSI'08)*, 2008.

[27] J. Duell, "Pthreads or processes: Which is better for implementing global address space languages?" Master's thesis, Computer Science Division, UC Berkeley, August 2007.

[28] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Proc. of 20th International Parallel & Distributed Processing Symposium (IPDPS)*, 2006.

[29] R. Nishtala, P. Hargrove, D. Bonachea, and K. Yelick, "Scaling communication-intensive applications on bluegene/p using one-sided communication and overlap," in *IPDPS '09: Proceedings of the 23rd International Parallel & Distributed Processing Symposium*, Los Alamitos, CA, USA, 2009.

[30] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan, "Scioto: A framework for global-view task parallelism," in *Proc. 37th Intl. Conf. on Parallel Processing (ICPP)*, 2008, pp. 586–593.

[31] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed, and Network-Based Processing*, 2009, pp. 427 –436.

[32] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, "Hybrid parallel programming with mpi and unified parallel c," in *Proc. 7th ACM Conf. on Computing Frontiers (CF)*, 2010.

[33] W. Zhao and Z. Wang, "ScaleUPC: a UPC compiler for multi-core systems," in *PGAS '09: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. New York, NY, USA: ACM, 2009, pp. 1–8.

[34] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE, Ed., Pisa Italie.

[35] R. Nishtala, D. Bonachea, and P. Hargrove, *GASNet Teams Specification and Design Document*, U.C. Berkeley and Lawrence Berkeley National Lab.

[36] *Nvidia CUDA Programming Guide 3.1*, Nvidia Corporation, Mar. 2010.

[37] *OpenCL. http://www.khronos.org/opencl/*.

[38] B. Wibecan, "Privatization functions for UPC," in *In UPC workshop at PGAS'09: the Third Conference on Partitioned Global Address Space Programming Models*, 2009.

[39] J. Prins, J. Huan, and W. Pugh, "UPC implementation of an unbalanced tree search benchmark," in *Technical Report 03-034, Department of Computer Science, University of North Carolina*, 2003.

[40] S. Olivier and J. Prins, "Scalable dynamic load balancing using UPC," in *Proc. of 37th Intl. Conference on Parallel Processing (ICPP)*, 2008.

[41] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *SC '09*. New York, NY, USA: ACM, 2009, pp. 1–11.

[42] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. 1998 SIGPLAN Conf. Prog. Lang. Design Impl. (PLDI 98)*, 1998, pp. 212–223.

[43] V. Tipparaju, W. Gropp, H. Ritzdorf, R. Thakur, and J. Traff, "Investigating high performance rma interfaces for the mpi-3 standard," sep. 2009, pp. 293 –300.

[44] D. Bonachea and J. Duell, "Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations," *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC)*, pp. 91–99, 2003.

[45] *Berkeley UPC bug 2808 - runtime support for user-spawned pthreads. https://upc-bugs.lbl.gov/bugzilla/*.

[46] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks," vol. 5, no. 3, pp. 63–73, 1991.