UiT THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology

Department of Computer Science

Space-Bounded Async Scheduling

A UPC++ Extension

—

Christian Bergvoll Vik

INF-3981 Master thesis in Computer Science… June 2016

Abstract

As increasing awareness of climate changes and surging power costs for big data centers today energy efficiency becomes increasingly important. In addition to that we carry mobile devices that depend on battery technology that is falling behind the rapid evolution of transistor technology and ever increasing power demands. At the same time there is an understanding that computer resources are not efficiently used. One solution to this is the proposed Space-Bounded scheduling, a scheduler that schedules tasks with the goal of achieving better cache locality. At the other side there is also a rise in HPC (High Performance Computing) popularity and a rising demand for powerful and easy-to-implement systems that are portable yet still customizable. For this demand the PGAS (Portable Global Address Space) model fits well and UPC++ is one of the newest editions to that category. Implemented as a C++ library it is both portable, powerful and easy to use.

We combine the advantages of Space-Bounded scheduling with the performance and simplicity of UPC++ to create Space-Bounded Async Tasks: A UPC++ extension that schedules async tasks with consideration of cache locality.

Acknowledgements

I want to thank my advisors Phuong H. Ha and Otto Anshus. Weekly meetings with Phuong has helped a lot in guiding me in the right direction. My fellow lab companions at the Arctic Green Computing lab deserve thanks for all discussions, advice and for keeping up with me. Tommy Oines deserves special thanks for helping to create an atmosphere called AGC party during late night sessions, these late night sessions always had lot of music keeping the energy levels high and the motivation up. Finally I want to thank all my fellow students, for all encouragement and all the quality tanning time at the bench outside.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1

1.1 PGAS 2

1.2 UPC++ 4

1.2.1 Shared objects and global pointers 4

1.2.2 Bulk data transfers 5

1.2.3 Async tasks 5

1.3 Space-bounded scheduling 6

1.4 Context 6

1.5 Targeted applications 7

1.6 Contribution 7

1.7 Methodology 7

1.8 Outline 8

2 Idea 9

3 Related Work 11

3.1 HabaneroUPC++ 11

4 Architecture 13

4.1 GASNet 13

4.1.1 Core API 14

4.1.2 Extended API 15

5 Creating the Space-Bounded Scheduler Extension 17

5.1 Async tasks 18

5.2 Core affinity 19

5.3 Discovering hardware topology 20

5.4 Space bounded scheduler 20

5.4.1 Tree creation 21

5.4.2 The Scheduler 22

5.5 Inter-process communication and synchronization 23

5.5.1 Active Messages 23

5.5.2 Alternative Global Adress space approach 26

6 Experimentation 27

6.1 Functionality evaluation 27

6.1.1 Running UPC++ tests 29

6.2 Performance testing 30

7 Results 33

8 Conclusion **35**

References **37**

Appendices

Appendix A Readme **41**

List of Figures

4.1 GASNet Architecture 14

4.2 UPC++ Architecture 15

5.1 Async scheduler design 19

5.2 Example tree layout for an Intel i5-2600 V3 21

5.3 Scheduler communication flow 24

Chapter 1

Introduction

Energy awareness is an increasing trend in modern computing systems today. Not only does datacenters have high operating costs but battery life in mobile devices also suffer from increasing demands in energy that battery capacity development is not able to follow. In conjunction to this there is also an understanding that computer resources can be used more effective.

One approach to that is the use of hierarchy aware schedulers[5][2][17][13][7]. These types of schedulers have proven to be work well but some of them are not optimized for the more advanced cache hierarchies that we see are trending. Now there is a trend with NUMA (Non Uniform Memory Access) architectures and other more advanced cache hierarchies that some of the hierarchy aware schedulers like Priority Work-Stealing[13] are not specialized for. Out of the schedulers the space-bounded[5] type of schedulers are the one that seem most aware of this trend. With these schedulers the tasks need to specify the size of their memory footprint. This is then matched to the size of a cache and only scheduled on cores that have affinity to that cache. Simhadri et al.[14] prove through their experimental analysis that that this type of scheduler indeed is more efficient when the complexity of the cache hierarchy rises.

Parallel to that we see the rise of Partitioned Global Adress Space (PGAS) abstractions for different parallel programming languages. Compared to the Message Passing model the Shared Memory model allows for the system to have more control over communication and how data is shared. UPC++[17] is one of those systems. UPC++ is a PGAS Extension for C++ which interesting for many reasons. Not only is C++ a widely used language for

many scientific fields but also because of the way the UPC++ extension is built. It is built entirely on top of C++ as a library. Using templates and clever operator overloading it enables the functionality of UPC[15] (Unified Parallel C) without having a specific compiler or preprocessor.

We present an extension to the UPC++ implementation that allows Async tasks to be scheduled according to the principles of Space-Bounded Schedulers. More specifically we allow async tasks to be rescheduled to another rank that will better preserve its cache locality. The original implementation of UPC++ introduces Async tasks which allow for an otherwise SPMD program to execute asynchronous tasks. With Async any rank[1] can specify a function with arguments and the rank where it wants to run it. With this RPC (Remote Procedure Call) feature the programmer can also specify event-driven executions where one async task has to wait until completion of another before it starts. Our addition extends this functionality by allowing tasks to be rescheduled to another rank that the scheduler deems better. The scheduler tries to match the memory footprint of the task to that of a cache. When it has found a cache with enough room to accomodate this task it "allocates" that space and lets the task be run on a rank that runs on a core that is a subset to that cache. This is backed up further by using Process Shared Memory (PSHM), a feature in UPC++ that allows sharing of memory between processes. With PSHM activated the potential consequences of rescheduling between ranks on the same physical computer is reduced. Because now the rescheduling has less impact when the programmer has allocated memory specifically for the rank it wanted to execute the async task on.

1.1 PGAS

Partitioned Global Address Space is a parallel programming model in HPC (High Performance Computing) where the main goal is to increase productivity while still keeping performance up. It uses a shared memory model that distinguishes between local and shared data. For a long time the dominant model in HPC has been the Message Passing model. The message passing model works by having a message passing interface (ex. MPI) that the parallel processes use coordinate activity. The communication between processes normaly consist of sharing results or to synchronize tasks. The

[1]Note: rank and node will be interchangable throughout this thesis

message passing model has been very popular for several reasons. One reason is that its quite easy to implement. It is easy to see how an extension to an already popular programming language can implement a message passing interface that delivers the requirements needed for this model to work. Secondly the programmer is in full control of the execution of the parallel application. The programmer knows not only how the data is divided but also the pattern of communication. This degree of control gives a high degree of customization for each application, something that is important in HPC. The Message Passing model is however losing momentum. As DeWael et al.[6] argues there are several reasons for this. The first being that there is a trend towards more complex computer architectures. Where there earlier used to be simple heterogenous systems there is now an increase in homogenous systems. Examples of this include both NUMA (Non-Uniform Memory Architectures) used in more advanced CPU arcitechtures, as well as the use of co-processors or GPU to leverage their advantages, which in turn give better utilisation of resources. With these system being more common it is harder to specialize each application for individual system and certainly doesn't improve portability.

Another point that DeWael et al.[6] points out is that with internet and the cloud the supercomputers that are often used in these HPC applications become more available. This in turn increases the popularity among other fields than computer science which also has use of these systems. Lusk and Yelick(2007)[10] point out in their DARPA HPCS Project report that programmer productivity is becoming an issue and that *"there exists a critical need for improved software tools, standards, and methodologies for effective utilization of multiprocessor computers."*. Meaning that with the rise of complexity and popularity there is a need for models that are easy to learn yet powerful enough to compete with the most customizable models in terms of performance. One answer to this seems to be PGAS based models. Where there before used to be only Message Passing (ex. Open MPI) and Shared Memory (ex. OpenMP) there is now the PGAS model which sits is in the middle of these two. With the PGAS model processes have both local and shared memory. Giving some of the customizability and control of Message Passing while keeping the simplicity and scalability of Shared Memory.

1.2 UPC++

In the paper *UPC++: A PGAS Extension for C++*[17] Zheng et al. present UPC++ with the following goals. A) To provide object oriented PGAS to be available in C++. B) To allow useful idioms such as asynchrnonous tasks and multidimensional arrays that were previously unavailable in UPC[15]. C) To offer PGAS that has interoperability with other parallel programming systems such as OpenMP, CUDA and MPI. As the implementors of UPC[15](Unified Parallel C) Zheng et al. implemented UPC++ as a library extension to C++. Using templates and operator overloading they implemented most of the features from UPC plus some additional features such as asynchronous tasks and multidimensional arrays. This "compiler-free" approach does not use a proprietary compiler or preprocessor and thus keeps interoperability with other popular parallel programming libraries.

1.2.1 Shared objects and global pointers

By default all variables in UPC++ are declared private so shared variables have to be declared excplicitly. Shared variables are declared by

shared_var<Type> s;

Which are generally stored in thread 0 (Rank 0) but accessible by all. The same applies to shared_array<Type> which also overloads the [] subscriptor to allow the same interface as normal. Once declared these shared variables act like any other variables for the user. Though as one would suspect shared variables have variable access-time depending on where they are physically located.

Global pointers reference an object in the global address space. Meaning that once created they contain both the local address and the thread id. Global pointers are declared with the form:

global_ptr<Type> sp;

Using operator overloading and templates to encapsulate the object they allow the global pointers to have the semantics of normal pointers. In addition to semantics of normal pointers they also contain a where() method that gives information about where it is stored. In addition to the template

types specified they also allow for void pointer types so that it can point to basically any user-defined type.

Memory in the global address space is allocated by calling:

allocate<Type> (int rank, size_t sz);

Where rank is the thread id to allocate memory on and size is the size of the type. Similar to malloc syntax the allocated shared memory has to be freed. This is done by calling deallocate on the pointer.

1.2.2 Bulk data transfers

UPC++ also allows for bulk data transfer using copy and async_copy. The syntax for this is:

copy(global_ptr src, global_ptr dst, size_t count);

With copy being blocking and async_copy being non-blocking. The syntax for both being the same except async_copy has an option to specify an event to trigger which can be used for synchronization later. Additional synchronization is implemented in async_copy_fence which waits for all async_copy to be done.

1.2.3 Async tasks

Async tasks offer RPC (Remote Procedure Calls) in UPC++. A feature that was missing in UPC. Inspired by the C++ Async library it is possible to start remote asynchronous tasks by using the syntax:

async(place)(function, args...);

In addition to normal asynchronous tasks there is also the possibility to include event triggers which are added to the async call. These events can then later be used for event-driven async tasks in the function async_after. The async_after function takes in a pointer to the event and starts executing the async task only after the previous async task is finished and has triggered the event.

1.3 Space-bounded scheduling

The idea of a hierarchy aware scheduler is not new and has been proposed in several papers[5][2][17][13][7]. Most of these are of a work-stealing type that work well for simple memory hierarchies, as the trend towards more advanced hierarchies and homogenous systems increase there is a need for another type that allows utilization of the new emerging types of architectures. A proposed type of scheduler to remedy this are the so called space-bounded schedulers[5][2]. Furthermore this is explored by Simhadri et al.[14] which did a comparison between space-bounded schedulers and the industry standard work-stealing and its hierarchy-aware versions. It is the space-bounded scheduler implemented in [14] that formed the basis for the one implemented in this thesis.

Simhadri et al. define space-bounded schedulers as:

> In a space-bounded scheduler it is assumed that the computation has a nested (hierarchical) structure. The goal of a space-bounded scheduler is to match the space taken by a subcomputation to the space available on some level of the machine hierarchy. For example if a machine has some number of shared caches each with m-bytes of memory and k cores, then once a subcomputation fits within m bytes, the scheduler can assign it to one of those caches. The subcomputation is then said to be pinned to that shared cache and all of its subcomputations must run on the k cores belonging to it, ensuring that all data is shared within the cache.

While this is the overlying concept of the scheduler a detailed description of how the scheduler implemented can be found in Section 5.4

1.4 Context

The research in this thesis was conducted in the Arctic Green Computing groups research in Green Computing. Specifically it covers research on energy-efficient run-time systems.

1.5 Targeted applications

The extension built for UPC++ targets to improve locality awareness for asynchronous tasks where the programmer hasn't excplicitly specified or made informed choices as to where an asynchronous task should be run. That is the programmer hasn't set up allocated memory that the task will use. Because if that is the case then rescheduling will incur more overhead than the programmer intended because of memory transfers between ranks. For now it is sufficient to only include rescheduling where the programmer has no requirements other than running a task asynchronously. In the future the extension could be further expanded to overwrite async_copy and PGAS memory allocation. This will account some of the optimizations the programmer did, but as it is hard to predict which ones are related to async this is reserved for future work.

It will also only try to reschedule between ranks sharing the same memory, that is ranks on one system. Because the goal here is to increase locality awareness by integrating space-bounded scheduling[2] to asynchronous tasks and the locality awareness concept in space-bounded scheduling is not easily applied across network.

1.6 Contribution

The research in this thesis contributes with a UPC++ extension that allows asynchronous tasks to be scheduled according to hardware specific information. This preserves locality in the sense that tasks are scheduled at a level in the memory hierarchy that allows for better cache utilization. With these conditions in place the task will (in theory) occur fewer cache misses and better use of bandwidth.

1.7 Methodology

To achieve this we are going to analyze the different parts of UPC++. It is neccessary to gain a full understanding of how the different components in UPC++ interract and what consequences there are with altering these. Especially important is the understanding of the asynchronous task execu-

tion. Only when this is understood can one begin to form an idea of where such a scheduler as described in [2] can be implemented. These type of schedulers require information about the system so it is important to figure out what kind of information UPC++ has access to. If such information is not available one has to acquire it by other means. One option is through proprietary libraries such as hwloc[12] where it is possible to extract system specific details such as memory hierarchy and processing components. With such knowledge one can then begin to implement the ideas that is the goal for this research. And only when implemented can one see the consequences such a scheduler has to the system. Not only does one see how it impacts other parts of UPC++ but also if it is possible to do it. How it compares to the original implementation is also important. All this will then need to be tested to see that what was created is not only practical but also an improvement to what was there originally.

1.8 Outline

The remainder of the thesis is structured the following way:

Chapter 2 describes the general idea that was behind this research

Chapter 3 presents Related Work. It introduces alternative PGAS languages as well as another UPC++ approach similar to ours.

Chapter 4 describes the architecure of GASNET .

Chapter 5 covers detailed implementation specifics. Specifically how UPC++ works, how each component was made to fit into UPC++ and how they communicate to create the finished scheduler.

Chapter 6 describes how the system was tested. What experiments were made and how to reproduce them.

Chapter 7 evaluates the results achieved by the experiments.

Chapter 8 concludes the research and the results achieved.

Chapter 2

Idea

To combine the principles of space-bounded scheduling with asynchronous tasks of UPC++. Research[2][14][17] on these subjects show promising results and offer what seems to be the state-of-the-art in both these fields. It is therefore interesting to contribute to the further development by integrating space-bounded scheduling into UPC++ async tasks.

We believe this is a useful feature for several reasons. One reason is the continuing trend towards more complex computer architectures. We believe that with the trend of homogenous systems it is neccessary to use the resources available in a more efficient manner. With this type of scheduling the tasks of async are scheduled in accordance to how the memory hierarchy is laid out, we believe this a positive step forward in more efficient utilization of system resources. Another reason to do this is the rising popularity in HPC. As HPC is more available it is used for other fields in science where programmability becomes an issue. Not only because of programming knowledge among users but also because applications become more advanced and are harder to implement. PGAS is already showing signs of enabling easier implementation while still being able to support a high degree of customization. With the addition of space-bounded scheduling programmers can choose to let the system handle more of the decisions that are made when creating parallel applications. The system knows the hardware specific details so it is able to make informed decisions that the user will benefit from. This also makes it more portable, because the system will adapt to different architectures, thus enabling the application to run efficiently on different types of hardware without recustomizing the application. With the scheduler choosing for you we believe it will be both simple and efficient.

Chapter 3

Related Work

There are several languages that implement Partitioned Global Adress Space. Unified Parallel C[15], Coarray Fortran[11],Fortress, Chapel, X10 and SHMEM[8] are some of them.

Unified Parallel C[15] was designed by the UPC Consortium. The original concept and specifications were published in a paper by Carlson et. al[4]. UPC is an extention to the C programming language and was the predecessor to UPC++. The main features include a partitioned global address space, the use the SPMD (Single Program Multiple Data) model and data communication via shared memory.

The DARPA High Productivity Computing Systems (HPCS) program[1] was a government project in the U.S that contributed to an increased popularity of PGAS languages. The project invited big players such as IBM, Cray, Hewlett-Packard, Silicon Graphics, and Sun Microsystems to produce concept studies for next-generation supercomputers. Some languages that spawned out of this project are Fortress, Chapel and X10.

3.1 HabaneroUPC++

HabaneroUPC++[9] is an exciting PGAS implementation that is based on UPC++. In addition to the functionality of UPC++ they implement dynamic task parallelism. The authors describe HabaneroUPC++ as a li-

[1]http://www.darpa.mil/about-us/timeline/highhroductivity-computing-systems

brary that uses UPC++ to implement PGAS communication and function shipping while HabaneroUPC++ implements intra-place work-stealing. HabaneroUPC++ is highly based on their previous implementation of Habanero-C++ and Habanero-C which are dynamic tasking libraries. HabaneroUPC++ therefore combines the dynamic tasking of Habanero-C++ with the PGAS functionality of UPC++. In doing this HabaneroUPC++ introduces concurrent scheduling of async task on a compute node. Where the current UPC++ implementation only ran one task at a time habanero is able to run multiple async tasks at once. This is done by creating their own versions of async as well as specialized join tasks that join concurrently run asyn tasks. Their results show that it outperforms standard UPC++ on some benchmarks and that it scales up to 6k cores running on an Edison Supercomputer at NERSC.

Chapter 4

Architecture

4.1 GASNet

Underneath UPC++ is the GASNet[3] Communication Layer. GASNet provides a portable network communication layer that is specialized for PGAS languages. It is designed to be portable and light. With GASNet the network specific details of a PGAS language is abstracted to GASNet, ensuring that the language is portable across different networking interfaces. GASNet supports a number of networking interfaces such as Aries, Gemini, IBV, MPI, MXM, PAMI, Portals4, SHMEM, SMP and UDP[1]. GASNet implements this support by using different conduits for each network type. When setting up GASNet the conduit for that network type is chosen. If such a conduit doesn't exist there are templates available so that programmers can implement it themselves, thus enabling support for additional network interfaces.

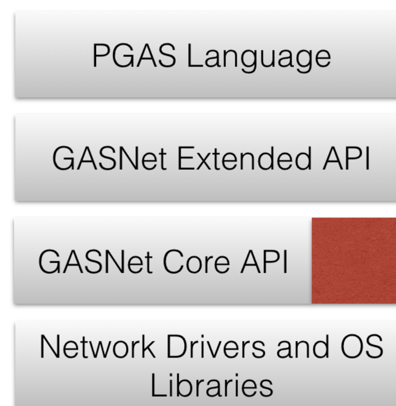
GASNet also aims to provide performance by using low-level lightweight communication primitives. Performance is important because network performance directly impacts how parallel applications perform. In addition to networking the GASNet communication interface supports GAS (Global Address Space) by adding features that provide OS-level features such as remote memory access and collective operations.

The GASNet Communication interface is split into two layers, the Core API

[1]Further information on supported networks can be found on the GASNet project website: https://gasnet.lbl.gov/#spec

and Extended API. The GASNet Core API is the lowest level layer which provides the networking while GASNet Extended API is built on top of the Core API and provides features together with the OS such as higher level shared memory and collective operations. This is designed so that developers of a PGAS language using GASNet can bypass the extended level if they prefer to implement the functions directly to improve performance or to use platform specific hardware support.

Figure 4.1: GASNet Architecture



4.1.1 Core API

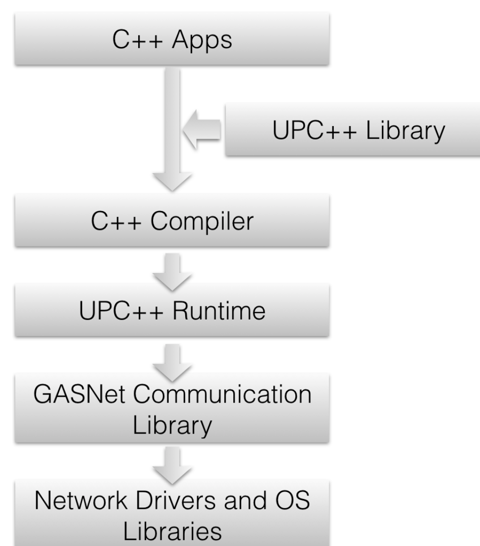
The Core API is based on the Active Messages (AM) Paradigm[16]. Active messages are often defined as "Messages that are able to perform computation of its own". Also defined as a "Low-level communication primitive that tries to exploit features in modern computer networks"[2]. In essence it is a low-level RPC (Remote Procedure Call) mechanism that provides unordered reliable delivery of messages. It works by having a handler attatched to each type of active message. The handler knows what to do with the message and is invoked when the active message reaches its target. In the message there is a payload that the handler extracts. Depending on the network conduit used the Active Messages can either run in a polling based manner or use hardware interrupts. The Core API is as mentioned earlier designed to be very lightweight and only implements core features. Most applications will therefore use the Extended API most of the time.

[2]By the Active Messages Specification: http://now.cs.berkeley.edu/AM/active_messages.html

4.1.2 Extended API

The Extended API sits on top of the Core API and implements most of the features that applications use. These features include more advanced communication features than those that were introduced in the previous section. Examples of features are one sided remote memory put/get and collective operations. Most of the features use Active Messages from the Core API as the base. For example a remote put could be expressed as an active message sent to the remote target where a handler registers the incoming put and places the data in its memory. The collective operations are implemented in the same manner, using communication between the nodes to synchronize. In general the Extended API implements features using the Core API but with more customizations and functionality that might not suit all applications. It is therefore recommended that applications wanting to use advanced features go directly through the Core API or the NIC. Figure 4.1.2 shows how UPC++ is implemented on top of GASNet.

Figure 4.2: UPC++ Architecture



PSHM

Process Shared Memory is a feature that allows processes running on the same memory to share adress space. This feature is only available if the

OS it is running on supports it. For UNIX systems this is provided by POSIX Shared Memory. The goal of this feature is to allow processes on the same compute node faster and more reliable communication. It lowers the communication latency by not having to go through the network API loopback. Processes running PSHM are grouped together in a team (Similar to MPI Groups). Compute nodes with shared memory are called supernodes[3]. The supernode structure contains the number of nodes sharing memory with indexes ranging from 0 to GASNET_SUPERNODE_MAXSIZE (An environment variable set in GASNet).

[3]Note that throughout this thesis the term Supernode is often used for the node with index:0 in the supernode table. This is because the nodes are often accessed as supernode + offset

Chapter 5

Creating the Space-Bounded Scheduler Extension

As the name implies UPC++ is a C++ library that implements UPC functionality using the underlying GASNET communications library with clever operator overloading and C++11 templates. Therefore it is natural to continue using the same programming language to write the extension. The extension is built into the UPC++ source code using the same structure as already in place, using source and header files in their respective places and adding these to their makefile. It should also be easy to exclude this functionality, so should one chose not to use space-bounded scheduling for async tasks then it can be deactivated easily.

To begin implementing the idea there were certain things that needed to be in place for the concept to work. To begin with there needed to be a way of discovering or specifying the size of each task. It should also be possible to alter the target rank for an async task without ruining the dependancies or limiting async tasks' functionality. Another aspect that one has to consider is the affinity of each rank to a core. Because if each rank is not associated with one core then the cache data will migrate around which is inefficient, it's not just inefficient but it also makes it impossible to predict and control scheduling of tasks. It is in fact neccessary for each rank to be fixed to one core because without that the concept of space-bounded scheduling does not work. Lastly there is the challenge of integrating all of this into the already working system.

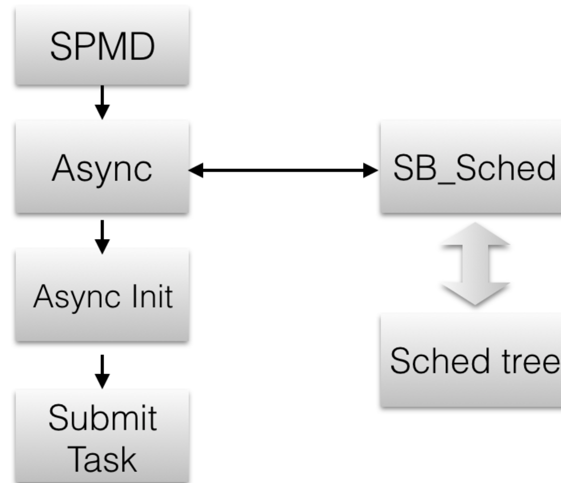
Important aspects to consider are how the individual ranks communicate

with eachother to decide where to schedule the async task, where the data structure for the scheduler stored, if the data structure is shared or migrated between cores or if there is a central scheduler that is reached through message passing. All of these implementation specific details and the chosen options will be adressed in the following sections.

5.1 Async tasks

Async tasks are the part of the framework that allows individual ranks to run tasks asynchronously. To do this it implements a few data structures that enables this to work in an otherwise SPMD (Single Process Multiple Data) model environment. To implement this they use templates to account for both single ranks and groups of ranks. They also implement something called async_after which waits until a dependancy event is triggered. Typically used when it needs to wait for another async to finish before starting to execute. Deeper down they are backed up by either C++11 supported syntax that uses lambda functions and variadic templates for defining any type of function. Should C++11 not be supported then there are also manual templates that support up to 16 arguments. The tasks are then initiated with information such as the caller, the callee, a function pointer and arguments. When initiated they are put in their corresponding queue. There is one queue for local tasks and a separate one for remote tasks. The queues are emptied regularly by the runtime where tasks in the local queue are executed and tasks in the out-queue are sent to their destination rank via active messages. It is in the step before they are selected for a queue that a check is done to see if space-bounded scheduling is enabled. If this is the case then the scheduler is invoked to see if it should be scheduled somewhere else. If that is the case the callee of the async task is altered to the one that the scheduler chooses.

Figure 5.1: Async scheduler design



5.2 Core affinity

The principles of space-bounded scheduling allows better prediction of cache locality. But for this prediction to be correct there is a need for more fine-grained control over the location of each rank. If one rank migrates between cores then that prediction will not be true most of the time. Therefore there is a need to pin each rank to one core only. To do this a library called hwloc[12] is used. This library allows both discovery of hardware topology as well as fine grained control over where threads are run. Section 5.3 will cover how hwloc[12] is used to discover hardware topology while this section will cover how cores are detected and how each rank is pinned to one core.

Each rank in UPC++ always start by running an upcxx::init(). This is the part where hwloc is included. To pin to one core it needs a topology object and a cpuset (a structure containing information on which cores to schedule on.) Therefore it begins by initiating hwloc and discovering the topology. After that is done a function is used to exclude all other objects but the cores. When that is done it is a matter of pinning each rank to its core. The core is decided by (rank % num_cores). This is done so that each rank greater than num_cores is divided equally among cores. Although this is not the point of the UPC++ extension it is supported.

5.3 Discovering hardware topology

For the space-bounded scheduler to work it is important to have information about the hardware. The important information here is the size of the different levels of cache and how they are mapped. Especially in NUMA architectures this is important because of the penalty of cache line transfers and inter-core communication. In section 5.2 hwloc[12] is used to discover the hardware topology. This information is then later used in creating the the tree data structure that the scheduler uses. The structure of the tree that hwloc creates is a bit different and more comprehensive than what the scheduler needs. Therefore it needs to be converted to match the structure of the scheduling tree. The exact layout of the tree will be covered in section 5.4

5.4 Space bounded scheduler

For the scheduler to be able to make smart decisions when scheduling tasks it is important that it is backed up by a data structure that is both simple and that represents the hardware correctly. The structure chosen is a simple tree structure based on Simhadri et al.(2014)[14]. The tree structure represents a PMH (Parallel Memory hierarchy) model as presented in[1]. Each node[1] in the tree represents a unit in the memory hierarchy. The node structure represents a single node in the tree and contains information about their size and how much space is currently occupied. This is used by the scheduler to track how much memory is currently being used by other tasks. Next is the number of children and number of siblings as well as their sibling id. This is used to iterate through siblings and children. Next are pointers to their parent, siblings and children. Finally there is a boolean value that describes whether it is a computational unit or not.

The tree data structure represents the tree as a whole. It has integers depicting the number of levels and leaves in the tree. It has a pointer to the root node as well as an array of pointers to the leaf nodes. Starting from the top the root represents main memory with size marked as unlimited. Because if all other caches are full it will get scheduled here, and then it's able to be scheduled by any core. The next level will usually be the L3 cache. In simpler quad core systems the L3 cache is usually shared by all four cores.

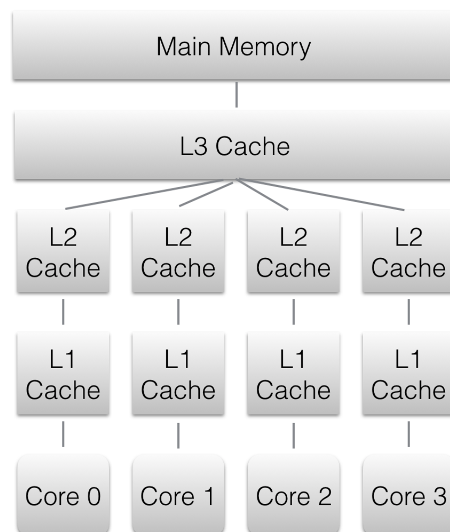
[1]In this section node will often refer to a tree-node

But if there are more then multiple L3 nodes will be created. L3 cache is then followed by a number of L2 caches depending on the number of children that the L3 has. This continues until the last level of cache L1 where they are connected to a core.

5.4.1 Tree creation

With the information given by hwloc (Section 5.3) the scheduling tree can be created. During the initialization of upc++ the supernode[2] of each PSHM group calls on tree_init() to create the scheduling tree. Tree_init() starts with allocating memory for the tree structure and a root node. The root node represents main memory so it is initialized with unlimited space and zero siblings. Next hwloc is used to discover topology. Because this is already done earlier by each rank to decide on which core they are pinned this object can be reused. The next step is then to iterate through the hwloc tree. For each object in the hwloc tree it is checked whether it is a cache unit or a core. If this is the case a new node is created containing the size of each cache and the number of children. This continues until the whole tree is created.

Figure 5.2: Example tree layout for an Intel i5-2600 V3



[2]Supernode refers to the node in the supernode structure with index:0

Manual tree creation

Because of time constraints and complications in handling all exceptions of the hwloc tree a manual tree creation is also available. This tree creation requires the information to be known and specified at compile time. This information includes the number of cores, number of caches, size of caches and fan out at each level. When specified this creates a tree equal to that which would get produced automatically by the automatic tree creation based of the hwloc tree.

5.4.2 The Scheduler

With the underlying tree structure in place the scheduler can now start to schedule tasks. Passed along to the scheduler is the tree, the rank that the async task was scheduled on originally and the size of the task. It starts out by fetching the core of the tree by looking up the core id in the leaf array. Now at the core it moves on to the parent of that core, that is the L1 cache. At the L1 cache it checks if the total size of that level is enough to hold the size of the task. If not it continues to go up one level as each cache on this level has the same size. It continues to do this check until it reaches A) a cache level which size is able to contain the task or B) reaching Main Memory, main memory has size infinite so it will always fit here.

When reaching a level that has enough cache size to fit the task it knows that this level of cache can contain the task. First it checks whether the cache of the original core fits the task including now its occupied size as well. This is because if there are already running tasks on that core this task will also be occupying that cache so there will be contention for that level of cache leading to more cache misses. If the original cache doesn't have enough available memory it iterates through its siblings trying to find an alternative cache to pin the task to. If it fits it will schedule it at one of the cores in the subtree of that node. In the case of the Intel Xeon E5-1607 there will only be one core for L1 and L2 caches, so the decision is easy. However for higher level cache where there are more than one core the decision is harder. In this case it will be scheduled at the original rank (given that it is one of the possible candidates). Whenever a task is pinned to a core the cache that was chosen marks its occupied value with that of the task size. This is also done to all caches its connected through up to main memory as well. Because if something is scheduled on a L1 cache then its L2 cache and L3 will also have

to reserve space for this task.

In addition to marking space as occupied when a task has been scheduled there it is also important to remove this allocation of space when tasks are finished. This is done by each rank when they finish executing the task. Upon finishing a task each async task checks if there are dependancy events tied to it. Right after this check they also signal the scheduler at the supernode that they are finished. The scheduler is called with information about the rank and task size. The scheduler then goes through the occupied caches and subtracts the value of the task size from the nodes occupied counter.

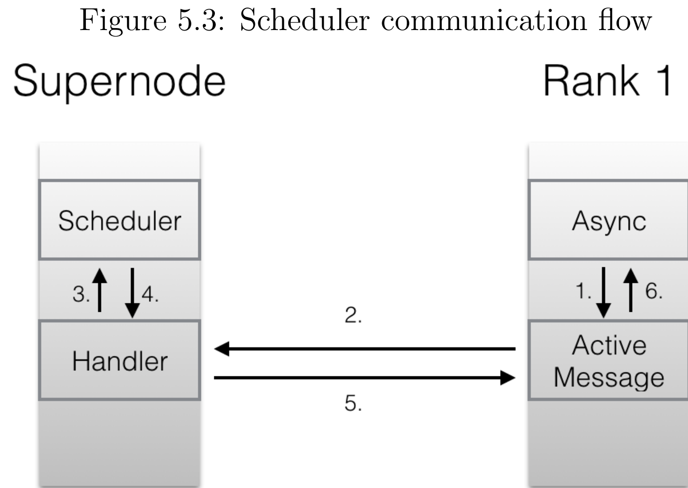
5.5 Inter-process communication and synchronization

For the scheduling to work between the ranks of the system they need to have a shared notion of the scheduler. In UPC++ and GASNET there are several ways of achieving this. One option is to have a central scheduler at the supernode that the other ranks communicate with using active messages. Another approach is to have the scheduling tree shared among all ranks using global address space features of UPC++. A final approach that is a mix between both of the previous approaches is to have the scheduling tree at the supernode with communication being in the form of mailboxes in the global adress space. The two first approaches were researched the most, with the active message approach being the currently used variant.

5.5.1 Active Messages

Active messages are the communication form available in UPC++ and GASNET. They require destination rank, handler tag, the data and the size of the data. With the active message approach the supernode handles all scheduling for its PSHM group. The rank of the supernode is already known in UPC++ by all ranks sharing the same supernode. The active message flow is discribed step-by-step in the list below.

Figure 5.3: Scheduler communication flow



Rank 1 is trying to execute an async task on rank 2:

1. Rank 1
   (a) Call to the local scheduler with rank and task_size
   (b) The scheduler prepares to send a request to group scheduler.
      i. It creates two new structs. One for the request and one for the reply
      ii. In addition to that an event is created and incremented
      iii. The request struct has pointers to reply and event which are set to the newly created objects.
      iv. The other fields of struct are rank and task_size which correspond to the rank and task_size that was specified when calling the scheduler.
      v. The Reply struct is left unchanged
   (c) The active message is sent through GASNET with the SCHED_GET tag.
   (d) Because the message is not blocking it needs to wait for an answer. It does this by waiting for the event that was passed along in the active message to be triggered.
2. Rank 0 - Supernode

(a) Receives message with tag SCHED_GET

(b) The tag is registered to the get_handler function

   i. The get handler function extracts the data from the package which is the request struct.

   ii. From the request struct it extracts the rank and task_size which it uses to call the scheduler which is local to the supernode.

(c) The scheduler returns the rank the task should be be scheduled on.

(d) This is written into the same request struct as was received.

(e) An active message reply is sent back to the requesting node with the SCHED_REPLY tag. The reply has the same request object that was received but with altered rank.

3. Rank 1

   (a) Receives message with tag SCHED_REPLY

   (b) The tag is registered to the reply_handler function.

      i. The reply handler extracts the message as a request struct.

      ii. Inside the request struct there is the rank now altered by the scheduler.

      iii. It writes this rank to the address of the reply struct which is also described in the request struct.

      iv. Finally it decreases the event reference

   (c) The event triggered because the reply handler decreased the reference.

   (d) It now checks the reply struct for the rank to schedule on.

   (e) The scheduler is finished and rank is returned to async

The active message data structs look like this:

Request struct

event ∗cb_event;

rank_t rank;

size_t task_size;

void *reply;

Reply struct

rank_t rank;

5.5.2 Alternative Global Adress space approach

Another approach to scheduling is to have the scheduling tree available for all nodes. Since UPC++ is a PGAS language it allows this with the use of shared variables and global pointers. With this approach the tree is allocated in the global address space so that all other ranks have access to it. The practical way this work is by having a global pointer to the tree that is shared among all nodes using shared variables. The global pointer points to the tree struct which then in turn uses global pointers to point to the other nodes of the tree.

The general idea of this approach is good. It uses the advantages that PGAS gives to declare shared memory that all nodes have access to. Even better is it that the nature of the scheduler applies to nodes in the same PSHM system. This means that the access time of the global pointers will not have to include network delay as they share memory. The only problem with this approach is the conversion between pointers and synchronization. The synchronization part uses shared locks which works well. One global shared lock for the tree gives a little contention but not much. Further fine-grained locks enable concurrent access to the tree, but incur additional communication. The additional communication comes from the fact that locks are implemented using active messages. The problem with conversion arises due to the way these global pointers are implemented in UPC++. Template specialization allows for typecasting between the different pointer types but for custom structs like the tree the void type has to be used. Whenever casting a global_ptr <void> to void pointer the local address of a shared variable is given, this pointer then has to be typecasted to the tree structure. This then has to be done for every pointer when traversing the tree data structure. This requires massive changes to the entire scheduler as it has to do conversions back and forth every time it traverses to a new node. This works but overly complicates the construction of the tree and gives less flexibility when trying to allow for creation of generic trees using hwloc data.

Chapter 6

Experimentation

The UPC++ Space-Bounded Async Task extension was developed and tested using a Lenovo Workstation equipped with a Xeon E5-1607 v3 (2.7Ghz) Quad Core. The Xeon E5-1607 has a memory architecture with private L1 and L2 caches with L3 being shared among all cores. The size of the different caches are L1: 256KB L2: 1MB L3: 10MB.

The specs of this machine are typical for a Programming and Development Workstation but not typical for UPC++ applications. UPC++ is foremost created to run on clusters running GASNet communication layer underneath. The experiments run were not extensive and only tested conceptually that the new features worked as expected and that they did not interfere with UPC++ elsewise.

6.1 Functionality evaluation

The functionality testing aims to test that the scheduler works as it is intended. It tests that when space-bounded scheduling is activated the scheduler will reschedule tasks according to how they fit in the memory hierarchy. The first step in testing this is to test that the underlying data structure is correct. The scheduling tree data structure should be created at each supernode and should contain correct information about the size and layout of the memory architecture. Because the tree is currently implemented manually and not dynamically with hwloc it is easy to reason about and to verify that the tree is indeed created correctly. This is verified by printing the size of

each node as well as scheduling behaviour once it starts scheduling.

The second thing that needs testing is that the communication between ranks are working correctly. That the messages are sent to the correct scheduler and that the replies are received with the correct content and in the right order. Initial tests verified that once set up the active message style communication works great. Prints to terminal as well as scheduling behaviour back up this claim. Throughout all testing the communication has been working flawlessly.

Third thing that needs testing is the scheduler. With the data structure and communication in place the scheduler itself can be tested. To test the scheduler the test_async.cpp included in UPC++ was used. The original test_async.cpp tested the functionality of async. With some minor changes this test is altered to better test the space-bounded scheduler implemented. The test was conducted using 4 ranks, one for each core of the CPU and using the following source code:

```
int main(int argc, char **argv)
{
  upcxx::init(&argc, &argv);
  if (myrank() == 0) {
    printf("Rank %d will spawn %d tasks...\n", myrank(), ranks());


    for (uint32_t i = 0; i < 10; i++) {
       async(1)([=] () { printf("Rank %d n %d\n", myrank(), 1000+i);
    }

  }
  async_wait();
  upcxx::finalize();
  return 0;
}
```

This source code tests the scheduler by spawning 10 tasks on rank 1. With an artificially high task_size of 32000 the scheduler is encouraged to reschedule tasks more often. Verified by prints to the terminal it is observed that the scheduler lets the first task schedule at the original rank, 3 next tasks are rescheduled to the other cores because they have free space in their L1 caches. After all 4 cores have gotten 1 task each it has to go up one level in cache. At L2 there is enough room for the rest of the remaining 6 tasks so they are

scheduled at the original rank. The async tasks are spawned faster than they complete so there is no deduction of occupied space until after all tasks are scheduled.

6.1.1 Running UPC++ tests

With UPC++ there are a number of tests that test the basic functionality of UPC++. The tests reside in upcxx/examples.

The following tests of upcxx/examples were found to be working with space-bounded scheduling and artificial task size (to promote rescheduling):

basic/test_event Tests that async_after works. The test spawns async tasks with following async_after tasks that waits for an event. Shows that events are still working even though they are rescheduled to a different rank.

basic/test_event2 Also uses async_after. Completes different steps of executing after the previous rank has finished and copied their data to shared memory using async_copy.

basic/test_progress_thread Tests asynchronous task execution by the progress thread. The progress thread is a module in UPC++ that can halt task execution on different ranks.

basic/test_copy_close Copies data to a neighbour and runs async.

Tests that do **not** work:

basic/test_copy_and_signal *Works without scheduling.* The copy and signal benchmark has too many assumptions as to where the data is copied and located. This test does not work with rescheduling. Likely caused by rescheduled tasks not being able to find the data that their async task tries to compute.

basic/test_am_bcast *Doesn't work without scheduling either*

basic/test_shared_array *Doesn't work without scheduling either*

spmv/spmv The sparse matrix vector multiplication algorithm does not work with rescheduling for the same reasons as test_and_signal. The algorithm is designed using dependancies that does not allow for rescheduling.

Tests that are running but doesn't use any functionality tied to the async scheduler implemented:

- basic/test_global_ptr
- basic/test_lock
- basic/test_remote_inc
- basic/test_shared_array2
- basic/test_shared_var
- basic/test_fetch_add
- basic/test_team

6.2 Performance testing

Unfortunately the benchmarks included in UPC++ does not use async in a way that will benefit from the new scheduler. For example the benchmark SPMV (Sparse Matrix-Vector multiplication) only divides its problem set on the number of ranks. So running this benchmark with 4 ranks will only spawn 4 Async tasks. With benchmarks running the same number of async tasks as number of ranks the scheduler is never really used, and certainly not used as intended. While this is a reasonable way of dividing the matrix for the old async tasks it does not take advantage of the benefits provided by the new scheduler. Instead one would have to divide the problem area in N parts and let Async schedule them as needed. Unfortunately there was no time to change these benchmarks to utilize the advantages of the scheduler. To complete the experiments there will have to be minimum of relevant benchmarks that compare the old Async tasks with the new space-bounded Async tasks. The experiments will have to compare running time, cache hits and energy consumption. Benchmarks that in theory should benefit from space-bounded scheduling are those of a memory-intensive nature.

Another important point is that the theory of the space-bounded scheduler works best when the memory hierarchy become more advanced. To put the extension to test one should test it both on a two-socket system as well as a cluster. Running it on a two-socket system would display the benefits when memory architecture become more advanced and running it on a cluster would showcase its behaviour when there are more than one supernode. Meaning that there are ranks working together that are not part of the same physical system and that they communicate via a network interface.

Chapter 7

Results

Because of the limitations in the experiments it was only possible to test functionality. The functionality evaluation show that the scheduler works the way its supposed to work. Both the communication back and forth as well as the data structure works as it should. Being able to run most of the tests included shows that the scheduler works on on simple tests that use async without specialized distribution of data. The tests that are not able to run are those that explicitly defines where the data should reside. The tests that do work show that the tests that did not run would be able to run if they are redesigned using global_ptrs and not as strict data allocation. For example one could reference a problem set with global_ptrs and have the asynchronous tasks copy over what data they needed. Eventually not copy at all because these nodes run on the same compute node and use shared memory.

For the performance testing the results would show how this new type of async scheduling would compare to the old async task execution.

Chapter 8

Conclusion

In this thesis we presented a Space-Bounded Asynchronous extension to UPC++. The scheduler is integrated into UPC++ by interrupting the normal async execution flow. With the task size known the scheduler can come up with a suggestion as to where the task should be run. If this type of scheduling is chosen then the asynchronous task will alter its destination and be sent to that rank instead. This type of scheduling alleviates the need to choose a particular rank and makes a scheduling decision based on the computers memory hierarchy and the current state of it. In theory this should cause the application to use resources more efficiently but this is hard to prove without extensive testing of performance.

The tests show that scheduling works, but that in some cases it requires the application to be redesigned so that it doesn't explicitly define data locations or wait for a reply from a fixed rank using other means than attaching events to async tasks.

The Space-Bounded Asynchronous Scheduler shows promising results for the future. With more testing and evaluation this might prove to be a really useful addition to UPC++. The theory of the space-bounded scheduler is already proven and on systems with a more advanced memory-hierarchy this type of schedulers really start to shine.

References

[1] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers, 1993. Proceedings*, pages 116–123, Sep 1993.

[2] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, New York, NY, USA, 2011. ACM.

[3] D Bonachea and J Jeong. Gasnet: A portable high-performance communication layer for global address-space languages. *CS258 Parallel Computer Architecture ...*, pages 1–27, 2002.

[4] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, Karen Warren, Lawrence Livermore, and National Laboratory. Introduction to upc and language specification introduction to upc and language specification introduction to upc and language specification. 2000.

[5] Rezaul A. Chowdhury, Francesco Silvestri, Brandon Blakeley, and Vijaya Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium*, pages 1–12, April 2010.

[6] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Comput. Surv.*, 47(4):62:1–62:27, May 2015.

[7] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the

memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[8] Karl Feind. Shared Memory Access (SHMEM) Routines. *Cray User Group*, pages 303–308, 1995.

[9] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. Habaneroupc++: A compiler-free pgas library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 5:1–5:10, New York, NY, USA, 2014. ACM.

[10] Ewing L. Lusk and Katherine A. Yelick. Languages for high-productivity computing: The darpa hpcs language project. *Parallel Processing Letters*, 17(1):89–102, 2007.

[11] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[12] Open MPI project. Portable Hardware Locality hwloc. `https://www.open-mpi.org/projects/hwloc/`, 2016.

[13] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Harsha Vardhan Simhadri, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Aapo Kyrola. Experimental analysis of space-bounded schedulers. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 30–41, New York, NY, USA, 2014. ACM.

[15] Berkeley UPC. Berkeley UPC unified parallel c. `http://upc.lbl.gov`, 2016.

[16] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[17] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. Upc++: A pgas extension for c++. In *Proceedings*

of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, pages 1105–1114, Washington, DC, USA, 2014. IEEE Computer Society.

Appendix A

Readme

How to make it work.