Faculty of Science and Technology
Department of Computer Science

Space-Bounded Async Scheduling

A UPCxx extension

—

Nishant Verma

INF-3990 Master's Thesis in Computer Science - May 2017

Abstract

It is estimated that computers and mobile devices use more than 2% of the total energy consumed. That means a lot of energy is going in powering our cpu,display and gpu. In this paper we are trying to optimize the power consumed by cpu in partitioned global address space environment. Recent research suggests that there is scope in improving cpu power usage by having a better scheduler. Simhadri et al.[14] concluded that space bounded scheduler can improve the efficiency by 60% in parallel environment in shared global address space.

In this paper we introduce data centric approach to PGAS, particularly UPCxx[17], referred to as DUPC, inspired by space bounded scheduler. The scheduler determines the cache hierarchy which allows it to make intelligent decisions to schedule task given the size of task is known beforehand.Hwloc library is used to detect the cache hierarchy. Once the cache hierarchy and sizes of each cache is known, we can track available cache sizes. With this information in hand, and if task size is known, we are able to provide better cache locality. We use PGAS for high performance computing. We use UPCxx, which exploits power of PGAS.

Acknowledgements

I want to thank my advisor Phuong H. Ha for his idea, encouragement and support. Weekly meetings with Mr. Phuong helped me keep on track and move in the right direction. My fellow student friends deserves my thanks, without them it would be impossible to complete this thesis who contributed with their insights and discussions.

Table of Contents

List Of Figures

Chapter 1

Introduction

Moore’s law predicts that the computing power doubles every 18 months. That has been the case for last 100 years from the time of birth computers until recently when we are unable to do so because of the heating effect of the cpus. Parallel computing comes to rescue as we try to increase computation power.

Having a smart scheduler will be one way to utilise cpu efficiently. There is a lot of research on this topic which there is still room for improvement. One of such scheduling model is hierarchy aware scheduler. Such type of schedulers have shown promise as suggested by Simhadri et al.[14] in his paper. He concluded that a space bounded scheduler will be 60% more efficient than a normal work stealing scheduler. The cache hierarchy is known beforehand and user needs to specify the task size. The cache hierarchy is determined using hwloc[12] library. Based on space available on cache scheduler can make smart decision to place the tasks to the most suitable core. We investigate Simhadri theory by creating our own scheduler based on UPCxx[17] in a shared memory based parallel environment.

PGAS provides such abstraction over shared memory environment. UPCxx is an extension over PGAS. It uses operator overloading, lambda functions and several other clever techniques to provide simple apis to interact with the environment. The library is open source and we take it to our advantage and implement a space bounded scheduler.

1.1 Problem Definition

The goal of the thesis is to create a scheduler which will be able to assign task to cores available on a cpu based on the amount of available space in memory cache and thus able to improve performance of a cpu in a parallel shared address space. To achieve our goal we have to tackle two major problems:

1. Discover hardware topology : We need to gain information about how memory hierarchy is being laid out.
2. Implement a scheduling algorithm : The task must be assigned to a core based on space bounded scheduling defined later.

1.2 Method and Approach

We followed agile software development approach called scrum in developing the thesis. The agile software development model encourages more face to face communication unlike waterfall development model. Scrum is iterative and incremental agile software development approach.

Scrum methodologies contains two backlogs called product backlog and sprint backlog. The product owner adds entries to the product backlog based on the user requirement. Developers can pick up items from product backlog and try to accomplish the task in a sprint which can be 2-4 weeks long. The task acquired from product backlog becomes the part of sprint backlog. Each sprint ends with sprint review and identify the progress and lesson learned from the previous sprint.

Scrum methodology defines three roles mainly product owner, Scrummaster and team. Product owner should be a person with vision, authority and availability. Scrummaster acts as a facilitator and works to remove obstacles that are stopping the team from moving forward. Scrummaster however does not manage the team. The Team is usually self managing and can consists of developers, designer and quality assurance members.

1.3 Outline

The thesis consists of the following chapters:

Chapter 1 - Introduction

Chapter 2 - Background
This chapter presents related work on space bounded scheduling and work on scheduling in UPCxx domain.

Chapter 3 - Design and Analysis
This chapter discuss about design UPCxx, GASNet. This provides base for discussion of design of DUPC.

Chapter 4 - Implementation
Discuss in depth explanation of the implementation of the design.

Chapter 5 - Experimental Evaluation
Evaluates the functionality and performance of the DUPC.

Chapter 6 - Conclusion
Concludes the research and results achieved.

Chapter 7 - Future Work
Discuss about areas of improvement.

Chapter 2

Background

This chapter presents previous work which are relevant to the thesis. We look into some work done in scheduling in UPCxx[17] and similar work on space bounded scheduling. While there has been a lot of work in making scheduling task more efficient, there has not been enough research in space bounded scheduling. Simhadri et al.[14] did analysed the scope of space bounded schedulers and found scope in making a smart scheduler based on memory hierarchy. There has been a lot of research on popular work stealing schedulers. Habanero UPCxx brings work stealing schedulers to UPCxx. We bring space bounded scheduler to UPCxx.

2.1 Introduction to PGAS

PGAS (Partitioned global address space) is a parallel programming model which assumes the global memory address space is logically partitioned and portion of which is local to each process. There has been a lot of developments like chapel, X10, etc. This only shows the popularity of PGAS and shared memory as a whole in high performance computing referred to as HPC. Message passing used to be popular choice for HPC and it is impossible to draw conclusions and find reasons to prefer shared memory over message passing. Shared memory presents the challenge of concurrency while certain protocols need to laid out. Serialization of data is another challenge that is needed to overcome in message passing. The problems does not stop here as different machines may have different operating system, architecture and hardware. The most common problem occurs when two systems have different endianness. Memory laid out by one system can be different from other as some system chose to most significant byte first in the memory while others chose to write least significant byte.That being said, most of the problems has been handled already at system level. Shared memory approach are often difficult to scale unlike message passing model. While there has been a lot of research and it's difficult to reason against any of them, in certain scenarios one can be preferred to other. What we need is a hybrid approach. PGAS provides such a hybrid approach and combine the power of both as this has both shared and local memory.

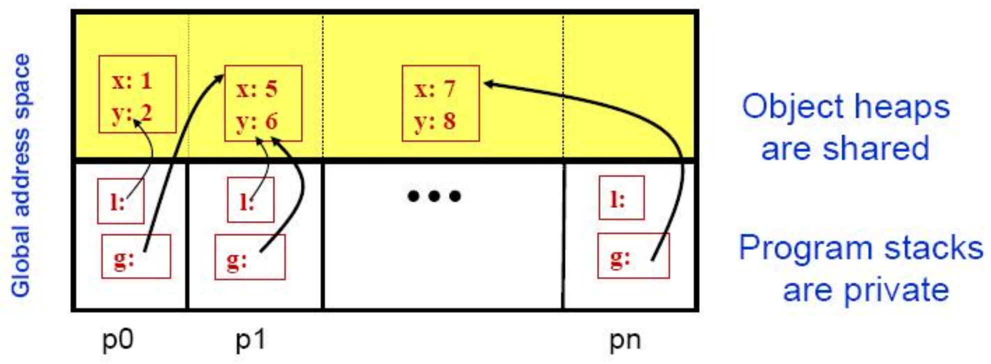


Fig 2.1 PGAS memory layout

The above figure[20] shows how the memory hierarchy looks like in PGAS. While the program stacks are private to the processes, the allocated memory in heap is shared among the processes.

The biggest advantage of shared memory is sharing of data structure. Our benchmarks share data across nodes. It's easy to parallelize programs by annotating or dividing the loop. An example in openmp

```
#pragma omp parallel for
 for(int i =0; i<n ; i++ )
```

This makes it really easy to convert sequential code into parallel code. This also means the parallel code is very close to what sequential code will look like.

2.2 Introduction to UPCxx

We need a simple programming language to express and implement our ideas. UPCxx[17] closes the GAP between HPC and object oriented programming by providing a PGAS implementation in C++. It provides other superior features and bring other parallel programming model likes MPI and openMp to PGAS. The other good thing about UPC++ is that it is library extension to C++ which is very lightweight. This does make developer life little harder as compile time errors can be frequent. The library approach allows it to provide interoperability with other popular libraries.

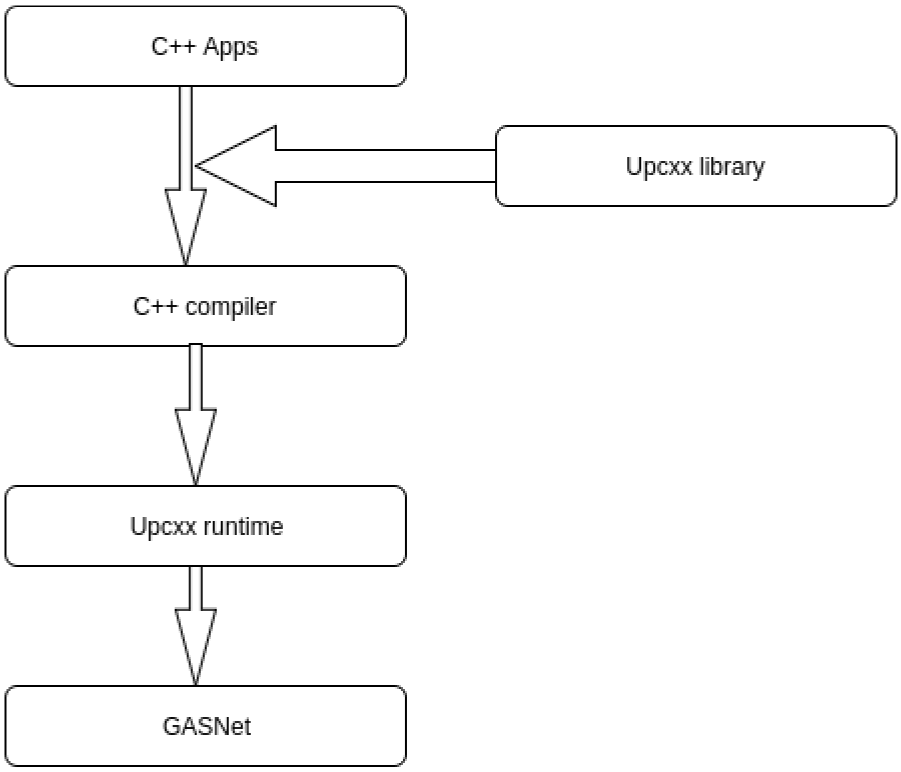


Figure 2.2 UPCxx Design

2.3 UPCxx programming constructs

2.3.1 Shared variables

Shared variables can be read and write across ranks and nodes. Shared variable has to be explicitly declared like

```
UPCxx:: shared_var<Type> shardVar;
```

Shared variable can be declared in global space, so its lifetime is execution of the program.
Similarly shared arrays are defined in UPCxx.

```
UPCxx:: shared_arrray<T,BS> shardArr(size);
```

T is the element type and BS is the block size. The data is accessible with [ ] operator similar to scalar array variables. This is achieved by overriding [ ] operator. The shared array can also be initialized at run time as follows:
sharedArr.init(size)

Where size is number of threads, which allocates block cyclically distributed global address space.

2.3.2 Global pointers

UPC++ considers the address space of each nodes into one single virtual global address space. The shared object created in global address space can be referenced by global pointer. The global pointer encapsulates the local address and the thread id.

```
UPCxx::global_ptr<Type> ptr;
```

The global pointer contains the rank information in addition to the logical address. The global pointer arithmetic logic is same as any normal pointer. The rank information is constant.

```
int rank = ptr.where()
```

Where returns the rank of the node which owns the pointer. Similarly we can get raw pointer using raw_ptr() method.

```
Type *local_ptr = ptr.raw_ptr();
```

The local address can also be obtained by typecasting to regular C++ pointer.

```
Void * local_ptr = (void *)ptr;
```

2.3.3 Dynamic memory management

Similar to malloc and calloc in c++, UPCxx[17] has allocate() method for dynamic memory allocation.

```
UPCxx:: allocate<T>(uint32_t rank, size_t count);
```

Here ranks is thread id or the node on which the memory will be allocated on for count number of elements of type T. The allocate does not call constructor explicitly, which can be achieved by using new operator. With the use of global_ptr we can escalate a private object into a shared object. We can free the memory using deallocate.

2.3.4 Bulk Data transfer functions

Copying data in bulk is more efficient than copying in chunks. UPCxx[17] provides blocking and non-blocking apis to achieve the same.

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

Source and destination buffer are supposed to be contiguous. async_copy() is asynchronous version of the copy with option to provide a callback method.

```
async_copy(global_ptr src,global_ptr dst, size_t count);
```

User can also register for an event with async_copy.

2.3.5 Memory Consistency model and Synchronization

UPCxx uses relaxed consistency model to gain more performance compared to other consistency models. This means programmer should be more careful about writing the program in a distributed parallel environment. UPCxx provides a handful of synchronization programming constructs to help developers. Synchronization in UPCxx can be achieved using barrier, fence or lock apis each having their own advantage in different use case scenarios.

2.3.6 Remote Function Invocation

This feature is inspired by X10. The functionality is clear from the title like remote procedure invocation. The function invocation means it returns a future object, which can be used to get the return value of the called function.

```
future<T> f = async(place)(function,args...);
```

Here place is thread id and function is the name of the method followed by its params. UPCxx provides two programming methods for asynchronous programming. Event driven programming keeps thread free and are less resource hungry. We can supply event object to async call as follows:

```
future<T> f = async(place, event  *ack)(function, args...);
```

To keep the learning curve for x10 programmers, UPCxx[17] provides finish block.

```
finish{
            future<T> f = async(place)(task,args...);
                }
```

Task must be completed before code exits the finish block. The implementation detail is beyond the scope of this paper.

2.4 Experimental Analysis of space bounded scheduler

In a parallel environment, processes or threads can request memory location despite being concerned about the performance cost associated with it. If a memory location is accessed that is not in the current page(a cache miss), several cpu cycles will be wasted to bring required data to cache depending on where it resides in the memory hierarchy. This means scheduling of the process or thread can have significant performance cost. The paper argues that we can improve on performance if we can improve cache hit ratio by scheduling our task based on space available in the memory hierarchy.

Space bounded schedulers are able to preserve the locality of the program resulting in fewer cache miss. The work stealing scheduler are great in load balancing and it is quite unclear how space bounded scheduler will perform against work stealing scheduler.Simhadri et al.

[14] tested space bounded scheduler on a series of divide and conquer algorithms against popular work stealing schedulers. He concludes that space bounded scheduler results in fewer l3-cache miss compared to work stealing scheduler as well as a significant improvement in runtime performance of the scheduler.

2.4.1 Scheduler Properties

A space bounded scheduler must satisfy the two properties. The first property is called anchored which means that every task should get anchored to the smallest possible cache amongst available cache. In most of the cpu architectures, L1 cache has unique cores associated with it. L2 cache might be shared among cores while it is common to note that L3 cache is shared among cores. Thus, in cases where the cache is shared, any of the core which shares the cache can be chosen as the candidate core for scheduling. Second property is called bounded. This means that sum of sizes of all subtasks must be less than the size of the cache.

2.4.2 Scheduler Implementation

The paper discusses about capturing a snapshot of memory hierarchy and a queue for each cache based on best fit policy as described earlier. It creates a tree of the memory hierarchy of the target machine with leaf nodes of the tree representing core. The paper does not go in detail and to answer how it does it. Once the tree is created, each cache in the tree is assigned a queue and variables for bookkeeping occupied space. There are locks for concurrency control in parallel environment. When a task is assigned, the scheduler first gathers all locks from the path of the tree from the core to the cache. It make a decision based on the predefined properties for a space bounded scheduler and add the task to the queue for the cache.

2.4.3 Promising results

The paper concludes that space bounded scheduler improves L3 cache hits in a memory intensive program. Space bounded scheduling comes with increased overhead for scheduling. This is reflected in computation intensive programs where work stealing scheduler outperforms the space bounded scheduler slightly. The primary reason for that due to the small scheduling overhead incurred by work stealing schedulers.

2.5 Habanero UPCxx

This work is relevant as the work is related to scheduling and is based on UPCxx. While the scheduling idea is different (work stealing rather than scheduling based on memory hierarchy), HabaneroUPCxx[9] brings a hybrid approach to task parallelism and integrates intra task parallelism and inter-task parallelism in one PGAS based library. HabaneroUPCxx implements intra space work stealing in addition to function shipping. It maintains a worker pool for scheduling incoming tasks, preferably on idle processors and unburdening the processors which are overloaded. A worker thread pool is a great idea to have in our own implementation, but it is also costly to maintain pool of workers waiting for incoming tasks. Features like joining async tasks, collective communications are highlight of HabaneroUPCxx apart from well known async remote copy and async remote function invocation which are familiar to UPCxx users.

Chapter 3

Design and Analysis

In this chapter we present design of the various components required to make our space bounded scheduling work.

3.1 Overview

Space bounded scheduling requires the program to know the memory hierarchy beforehand and then it can make a scheduling decision. Once it has made the decision it can alter the rank. After the program is concluded we have to make appropriate adjustment to reflect the actual state of the new memory hierarchy. Thus our scheduler consists of mainly three components mainly discovering the memory hierarchy, the actual scheduling algorithm and adjusting the memory hierarchy when the program concludes.

3.2 High Level Architecture

The original architecture remains the same as we have modified the existing code and the behaviour. The initial GASNet architecture is shown below which is taken from GASNet official website[18].

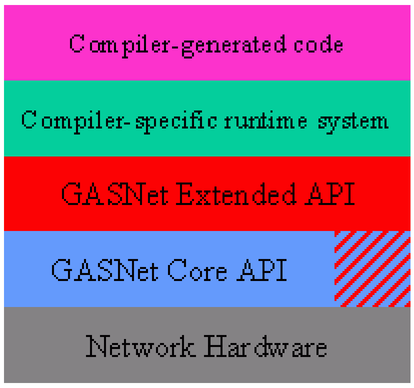


Figure 3. 1 Gasnet Architecture

GASNET is a library which implements PGAS. It basically provides functionality like remote memory access and remote method invocation which will be discussed in detail later. It is highly portable and supports a number of operating system. GASNET supports a number of networking interfaces including mpi. Programmers can write their own conduit if there is no such conduit available.

GASNet (Global address space networking) is composed of three components. The lower layer is core api. This layer is a generic interface and is responsible for remote procedure calls using active messages. The middle layer is extended api and is primarily responsible for remote put and get operations.The operations are one sided and caller provides all the information including address of data, data and length of data.

3.2.1 CORE Api

Core api consists of active messages. Active messages are basically low level remote procedure calls. An active message request will go from initiator to a node which register the request on a handler. Handler extracts the request from payload,process and prepare a response. A reply for the request can be delivered to the initiator node on response handler.

3.2.2 Extended Api

Above core api is extended api. Extended api provides functionality like remote put and get operation.It is basically one sided.operation where caller provides local and remote address and length of the payload. User has option to choose between blocking and non-blocking put and get operation where blocking will return only when the operation is complete and non-blocking will return immediately after the call. The non blocking operation can be explicit where it provides a handle, which can be used for completion. Alternatively we can define a region, where compiler waits for the pending operations to return(eg begin-end blocks). This type of non-blocking operation is called implicit non-blocking operation.

3.3 Discovering memory hierarchy

We must know the memory hierarchy before we can make any scheduling decision. We take advantage of UPCxx program lifecycle to determine the memory hierarchy. Every UPCxx program must start with init() call and end with finalize() call. Our call to determining memory hierarchy is closely tied to this lifecycle of a UPCxx program. On init() call, we determine the memory hierarchy. It is important to note that the memory hierarchy is obtained on a core with rank 0. A UPCxx program flow will now look like as shown in figure 3.2.

3.4 Scheduling Algorithm

The second step is to determine and the schedule the task on new rank. UPCxx[17] provides an api called async to schedule task or a method on certain rank. The DUPC scheduler tries to find the best rank based on space available on cache. This is discussed in detail in later section. UPCxx is open source library which means we can modify the async code. Without trying to complicate things, our design decision was to determine the rank by scheduler before forwarding the call to gaset.

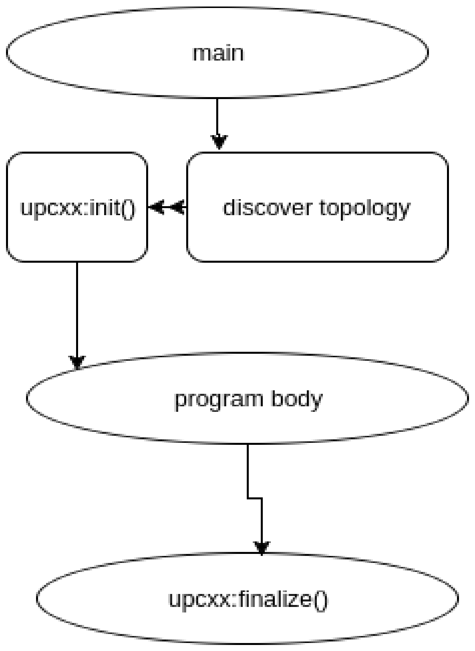


Figure 3.2 Discovering topology

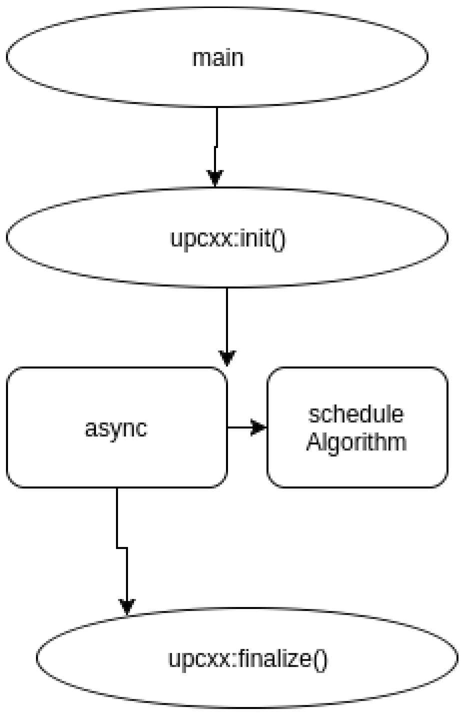


Figure 3.3 Scheduling logic in a UPCxx program

There can be scenarios when call to async is made from a non-zero rank. We use active messages to communicate with the scheduler which is based on rank zero. The high level design is shown below:

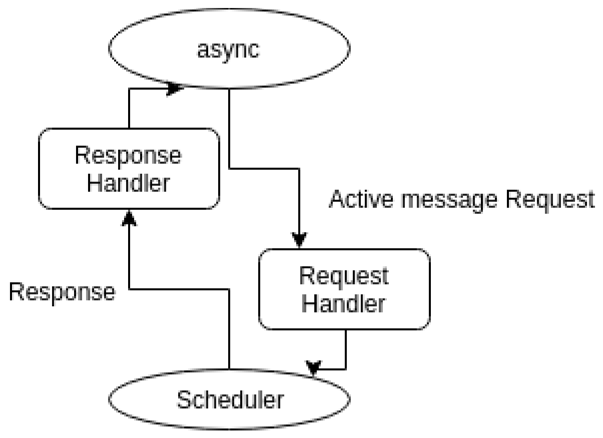


Fig 3.4 Communicating with Scheduler

Chapter 4

Implementation

Now that we have defined space bounded scheduler, we implement the scheduler which is based on UPCxx. The implementation extends and override certain api of UPCxx to achieve our target. The challenge is to find an optimal core for a task based on the size of the task and state and availability of the cache in memory hierarchy to obtain maximum cache hit.

For scheduling, the first requirement is to know the memory and cache hierarchy. A library called hwloc[12] is used to discover the topology when the UPCxx program is initialized.The implementation modifies async api of UPCxx. The async api executes a method on a core provided by the user. The modified async calls the scheduler which is based on a core with rank zero. If the async call is from a core whose rank is not zero, the communication to the scheduler is achieved using active message api provided by GASNET. The call to schedule contains rank and task size of the task. The scheduler traverses the memory topology and find an optimal rank for the current task. If the optimal rank is not the rank selected by the user, we change the rank. This is discussed in detail in the following sections.

4.1 Async task

Async task is usually referred as a short term for asynchronous tasks. In UPCxx async is an api for remote function invocation. This allows individual task to run asynchronously on different nodes. An async call in UPCxx[17] looks like

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

The async(place) call creates an async object which has an overloaded “()” operator for handling the function pointer and arguments in the second parenthesis. UPCxx then packs the function pointer and args in a continuous buffer and send it to the target node using active messages. On receiving the message, the remote node unpacks the buffer and place the request in a task queue. When the task is executed, the runtime sends a reply with return value as param.

In case when space bounded scheduling is enabled, the scheduler finds the optimized place value using the scheduling algorithm and place value is altered to be the one chosen by scheduler.

4.2 Active messages

Active messages are part of gasnet components which are responsible for communication between nodes. There are separate handlers which are responsible for handling active message requests and response and are handled by their corresponding handlers. Communication between two nodes A and B is handled as follows:

1. To send a request from A to B, A call gasnet_AMRequest*() with data payload, node index of B and index of the request handler to run on B as arguments.

2. On receiving the request, node B runs the request on appropriate handler and prepare a response for the arguments supplied in the request. It calls gasnet_AMReply*() to send a response back to A.
3. After some time, A receives the response back from B and runs the response handler with data from gasnet_AMReply*().

4.3 Core Affinity

For each core there is rank assigned by GASNET. The rank value is unique to each core and have values between 0 and n-1, where n is the number of cores present in the system. UPCxx[17] has provided simple apis to detect the value of n and rank of the system on which it is running on. To get the current rank we can use myrank() from UPCxx namespace. Space bounded scheduler also need to detect the cache hierarchy and core information. This is done using a library called hwloc.

When a UPCxx program is initialized, hwloc library detects the topology and core information. Each core discovered an id between 0 to n-1 is assigned to each core, where n is the number of cores. UPCxx assign rank (a value between 0 and n-1) to each core. We need a proper mapping between id of the core and rank assigned by UPCxx before we make a scheduling decision, which the current implementation lacks.

4.4 Discovering hardware topology

For scheduler to make smart decision on where to place the task, it must know the cache hierarchy and core information. We will look in detail, how to get this information using hwloc. The library gives the topology(cache hierarchy and core information) information in a struct hwloc_topology_t. The following api calls populate the items of the aforementioned struct.

```
hwloc_topology_t  topology;

hwloc_topology_init(& topology);

hwloc_topology_load(topology);
```

Once we have the topology struct, we can obtain the cache and core information from a series of api provided by the library. The cache is represented by a struct hwloc_obj_t, which has member fields providing crucial information for our implementation like cache size, parent, child and sibling information.Following api returns such struct of type L3 cache.

```
Int L3_CACHE_DEPTH =
hwloc_get_type_depth(toplolgy,HWLOC_OBJ_L3CACHE );

hwloc_obj_t  L3_CACHE = hwloc[12]_get_obj_by_type (topology,
                         HWLOC_OBJ_L3CACHE),
                         hwloc_get_nbobjs_by_depth(topology,l3depth)-1);
```

Similarly, we can extract core, L1 and L2 cache information from topology object. Now, we have the tools to create the hierarchy tree, which the scheduler will use to make smart decisions to schedule a task from the available choice of cores.Next section discusses how we create the scheduling tree.

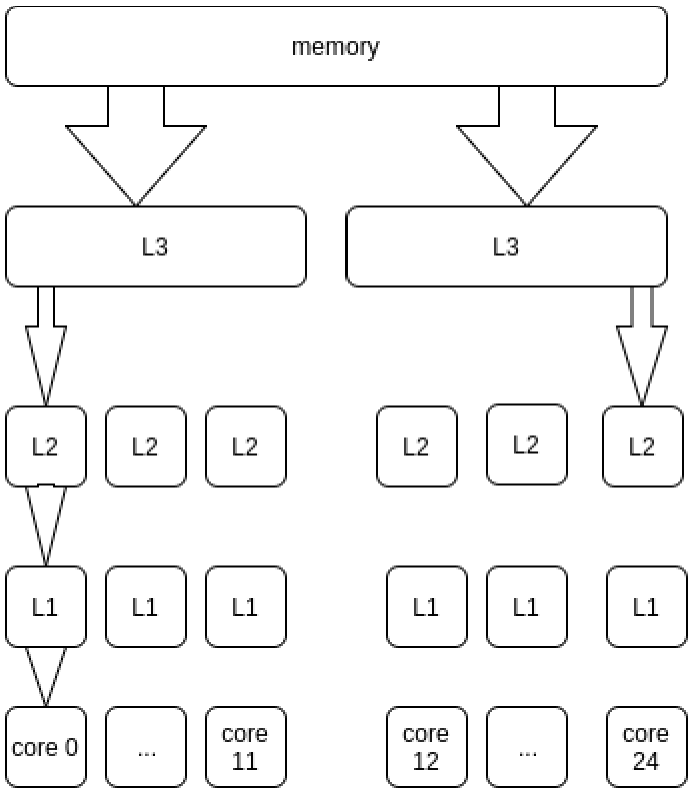


Fig 4.1 Xeon E5-2650L architecture

4.5 Tree Creation

This is a data structure which represents the cache hierarchy of the system. From previous section, we know how to get the core and cache information including its childs, parent and siblings. Using this information, we can now create the tree. The following implementation considers L3 cache to be at the top level of cache hierarchy which is consistent with most of the cpus out there but not all. First we obtain L3 cache information in a struct hwloc_obj_t. We store information like cache size and number of children and pointers to it. Using the children pointers, we traverse the L2 cache. L2 cache struct is processed in the same way and we can process the L1 cache using child pointers and so on. A struct shced_tree_t is the data structure which is used to store the cache hierarchy information. Here is a snippet of code showing how the struct is being populated after obtaining information from L3 cache(described in previous section).

```
shced_tree_t  *tree = allocate(sizeof(shced_tree_t));

int num_of_cache = hwloc_get_nobjs_by_type(
                            topology, HWLOC_OBJ_L3CACHE);

 for(int i=0; i <num_of_cache;i++ ){

         hwloc_obj_t   L3_CACHE  =   getL3Cache();  //from  previous
        section

         sched_tree_node_t  L3 =  malloc(sizeof(sched_tree_node_t));

         L3->size = L3_CACHE->attr->cache.size
         L3->num_children = L3_CACHE->arity
         L3->sibling_id = i;

for(int j=0; j<L3->num_children; j++){
//fetch l2 cache and so on
}
}
```

4.6 Task Size

Previous section described how to create the tree representing memory hierarchy. For the scheduler to make decision on core, it must know the task size beforehand. The developer must provide the size of the task when executing an async operation. There are few ways to accept the task size parameter from the user in async. We exploit the async api capability to accept events(a UPCxx struct) as param. Our implementation expects developer to provide task size in an event struct. So a typical async request should look like:

```
event * ev = new event;
ev->task_size = TASK_SIZE;

async(rank,ev)(function,args ...);
```

The task size should be in mb. If the task size is not provided, the scheduler will use a default task size of 3k mb which might result in abnormal behaviour and performance gain might seem minimal or none. While calculating a task size can be cumbersome for tedious algorithms, we can use and test the scheduler for simple algorithms. There are certain ways which can be used to calculate the size of task, and we provide some insights in future work section.

4.7 The Scheduler

Now we have all the ingredients ready to create the scheduler. The goal is to match a computation with available cache space. In case of UPCxx[17], user may try to run an asynchronous operation on a node, and we are trying to find a node which matches the computation with available cache size. In simpler words we are trying to find a better ‘place’ in async call.

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

Since, we have all the memory hierarchy in place in a tree data structure, all we need to do is to traverse the tree and match the computation size or task size provided by the developer as suggested in previous section. For this async implementation is modified.

The scheduling logic is handled on super node. We chose the node with rank 0 to be supernode. To keep the implementation generic, so that place variables is always chosen wisely regardless of the rank of the node it is being called from, we need to communicate with the scheduler in case where the async is called from a node whose rank is not zero. Active messages is used for communication. The modified async implementation calls a method sched_get which returns a new rank and handles the communication.

```
int new_rank = sched_get(rank_t rank, size_t task_size);
```

To communicate with the scheduler, an active message request is created. The request contains a call back event, rank of the current node,task size and a pointer to reply message.

After the request is prepared, the active message request is sent to node with rank 0 using gasnet api and we wait for response event.

```
GASNET_CHECK_RV(gasnet_AMRequestMedium(
    0,SCHED_GET,&req,sizeof(req)))
```

A handler on node 0 receives this request with data and a token to respond to. It extracts necessary data and process the request.

```
Sched_am_t  req = (Sched_am_t  *) req_buf;
    int new_rank = sched_tree_schedule(req->rank, req->task_size);
```

The overridden implementation calls sched_tree_schedule. The method is responsible for finding a new rank or core based on the computation size. It takes computation size and rank as param and return new rank. The first step is to find a cache where the task can fit on the original core provided by the user. It starts with L1 cache on that rank and L2 cache and so on, until it finds a place where the computation can fit or it reaches the main memory which will be able to accommodate task of any size. On finding such cache, it is marked occupied and we also need to add the task size. We also need to mark the cache occupied all the way upto the main memory as there is no direct line from cache to main memory.

```
      void  sched_tree_schedule(int rank, int task_size){
            sched_tree_node_t  *core = tree->leaf_array[rank]

sched_tree_node_t  *l1_node = core->parent;

for(node =l1_node; node != null; node = node->parent){
if(task_size < available size){
return rank;
}
}
```

In case, if there is no available space in memory hierarchy on the core provided by the user, the scheduler tries one of the sibling and follows the same steps as mentioned above. It might try all siblings until giving up and schedule the task on the rank provided by user.

For example, suppose a processor has two L3 cache each having n number of L2 cache each connected to a separate core. If one of the L3 cache is occupied, the scheduler will ignore all the n cores connected to this cache.

The scheduler keeps track of all the cores on which a task is scheduled and eliminate them from possible candidate from selection. Now, we have computed the new rank based on space available in cache hierarchy, we need to send a response back using active message as described earlier.

```
     response->rank = new_rank;
        GASNET_CHECK_RV(gasnet_AMRequestMedium(
          0,SCHED_REPLY,&response,sizeof(response)))
```

The requester receives the response on a handler. It process the response and broadcast the event it has received a response on scheduling decision.

```
          void sched_reply_handler(gasnet_token_t token, void
     *buf, size_t nbytes  ){

          sched_reply_t  reply = getFromBuffer(buf);
notifyResponseRecieved();
}
```

When a task is complete, we need to readjust the occupied task size value in the tree to keep the behaviour correct for upcoming async requests. For this whenever a task is finished an event is delivered to supernode with task size and rank. The scheduler then traverses the tree and make proper adjustment to reflect the correct task size.

Chapter 5

Experimental Evaluation

This chapter evaluates functionality and performance of space bounded scheduler. The first section discusses and tests functionality on a string of tests and benchmarks. The second section tests performance of the scheduler on a Intel Xeon E5-2650L processor with 24 cores.

5.1 Functionality Evaluation

Automation testing framework like Junit can be be really helpful in the course of development cycle. We will need some testing framework in c++ or UPCxx[17] in our case. We instead tested all our functionality manually and including automation testing for the project can be considered for future work. As mentioned before, we have three core functionality and each of them should work as expected.

The first functionality is to discover hardware topology and populate our tree data structure with the size, number of child and pointer information. The library used to extract the information(hwloc[12]) provides a command called ‘lstopo’ which gives a visual representation of the tree data structure as seen in figure 5.2. Using this information, we can test and make sure that our first functionality is correct.

The scheduling algorithm is the heart of what we are trying to achieve. To test this functionality we need to test the behaviour in certain scenarios. We discuss one such scenario for the cpu with following architecture:

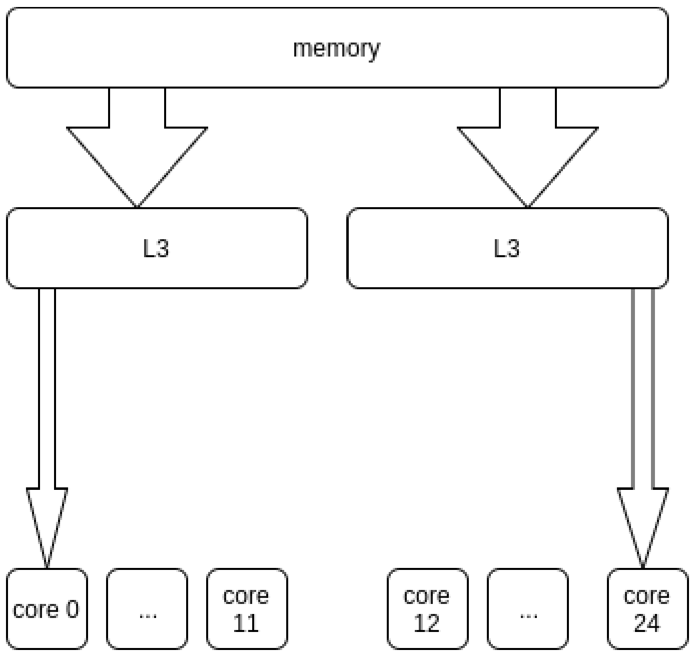


Figure 5.1 - Xeon E5-2650L Architecture

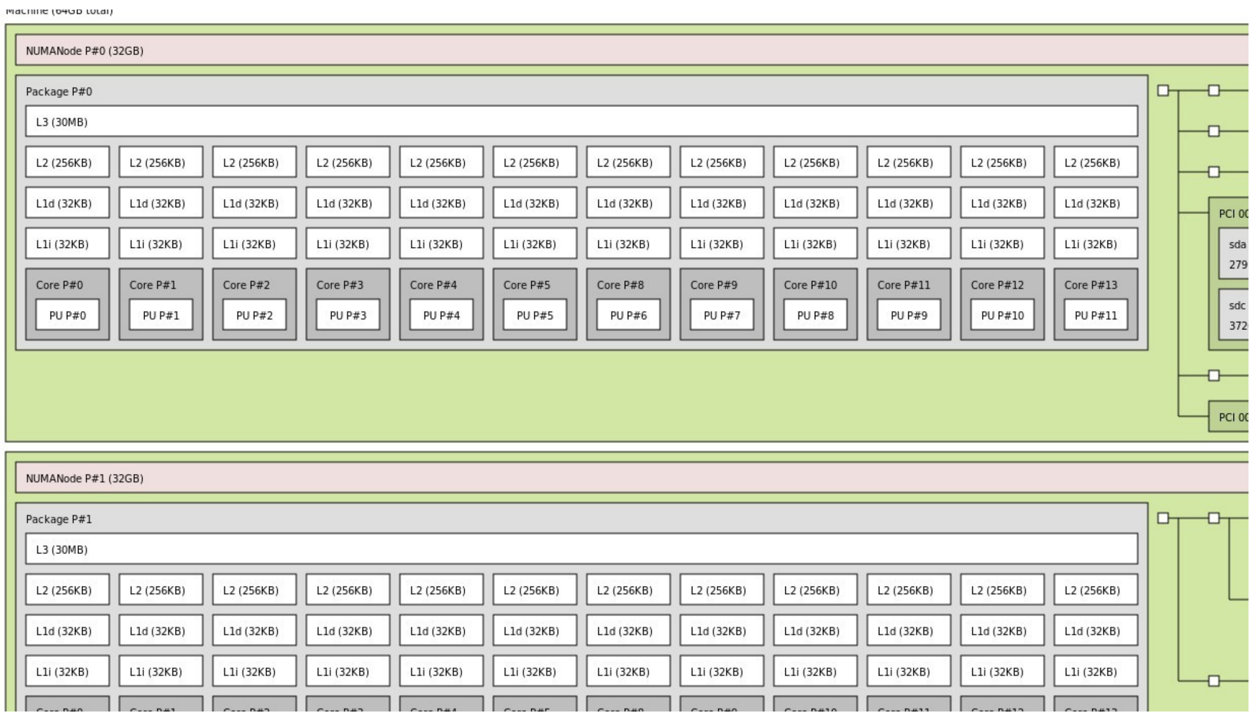


Figure 5.2 lstopo Output

We expect the rank to be changed to a core shared with the second L3 cache in case where first L3 cache can no longer hold the task. We wrote a simple program with each task size just large enough to be held by L3 cache and tested and verified this behaviour.

```
#include <upcxx.h>
#include <iostream>

#define TIME() gasnett_ticks_to_us(gasnett_ticks_now())
int number =5000000;

using namespace UPCxx;

void test(){
        int sum =0;
        int *arr = new int[number];
        printf("rank == %d\n",myrank());
        for(int i=0;i<number;i++){
                arr[i] = i;
        }
        int sw = 1;
        for(int i=0;i<number;i++){
                if(sw == 1){
                        sum += arr[number-1-i] ;
                }else{
                        sum += arr[i] ;
                }
                sw = 1-sw;
        }
```

```
          std::cout << "sum = \n"<<sum;
}

int main (int argc, char **argv)
{
  upcxx::init(&argc, &argv);

  if(myrank() == 0){
           int P = 2;
          int n = number*P;
          async_wait();
          double start_time = TIME();
          std::cout <<"start time  ="<<start_time;
          int size =  0;
          for(int i =0; i< P; i++){
                    event * ev = new event;
                    int var_size = sizeof(int);
                    ev->task_size = number*var_size;
                    size = ev->task_size;
                    async(i,ev)(test);
          }
async_wait();
std::cout <<"end time  ="<<TIME()-start_time<<" size = "<<size;
 }
UPCxx::finalize();
return 0;
}
```

5.2 Performance Evaluation

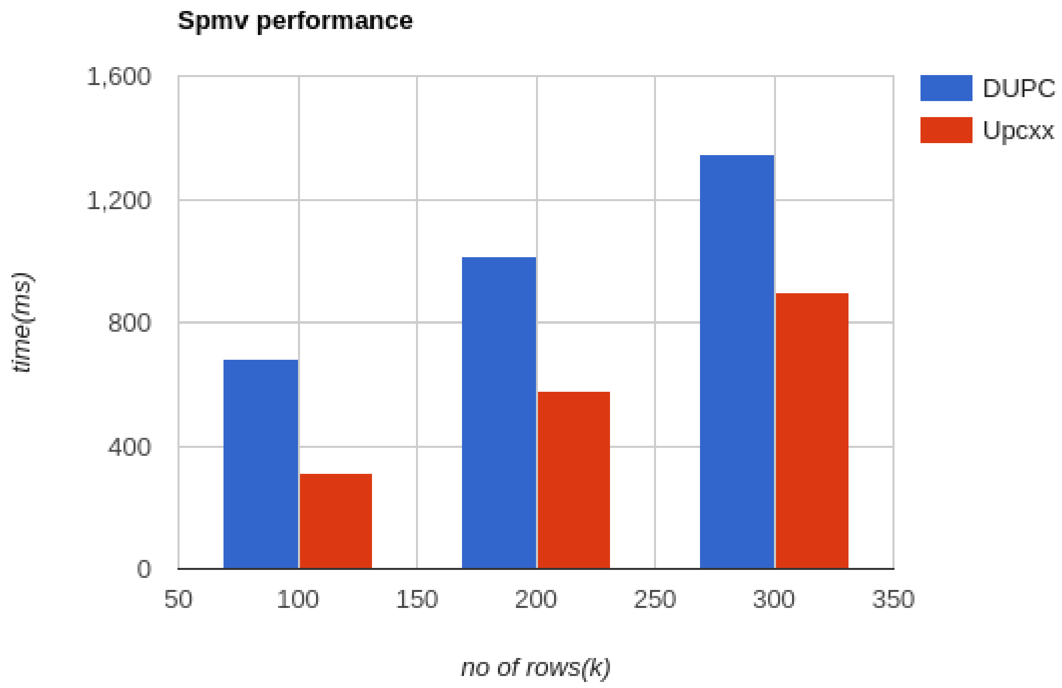
We were unable to observe any kind of performance gain in any of the benchmarks we tried. We are missing one crucial step in our implementation that can be the primary reason for this behaviour.

Although we are changing the rank during scheduling according to space bounded scheduling algorithm, this does not mean that we actually schedule on the core we desired to run the thread on. UPCxx can assign ranks to core in a random manner, and all this means when we schedule a task on some rank, we are not sure which core it is going to run. We still do not have a good solution to this problem. But hopefully after this problem is solved we will be able to see some performance gain in memory intensive benchmarks.

We performed a series of test on standard algorithm and programs intended to create scenarios where it will benefit from DUPC scheduler.

5.2.1 SPMV Performance Test

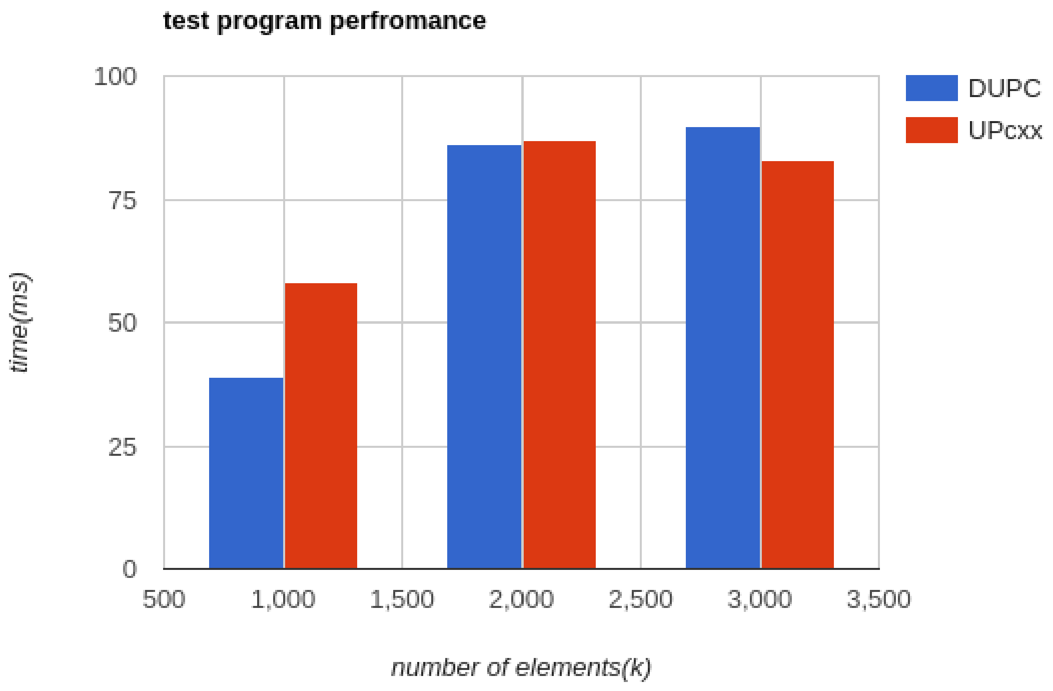
Sparse Matrix vector multiplication(SPMV) is a upcxx benchmark which uses async to perform computation. The program can be considered memory intensive if we create a matric large enough so that it can take advantage of DUPC scheduler. We set up a test environment which uses 24 cores on aforementioned cpu with following results.



The x-coordinate shows number of of rows and number of columns times 1000 in a matrix and y-coordinate represents time in milliseconds. We did not expect any performance gains and the program actually performs better without the scheduler which can be due to scheduling overhead but can not be said with certainty. We expect scheduling overhead to be small as we are traversing a small tree without any complex computation.

5.2.2 Test Program for DUPC

We created a simple program to create scenarios where it can actually take advantage of DUPC and test functionality. The program does nothing interesting, it creates an array of a predetermined size. A large array makes the program memory intensive and ideal for our test. After a series of test, it shows the following result.



The x-coordinate represents number of elements in array times 1000 and y axis represent time in milliseconds.The program does show slight improvement in two of the three scenarios. But there is no continuous trend which concretes our finding. As we did not expect any improvement in first place, all we can say that the mean of the execution time of the programs in two scenarios will approach each other when the experiment is performed enough number of times.

Chapter 6

Conclusion

In this thesis we introduce data-centric approach to UPCxx[17], inspired by space bounded scheduler. The scheduler determines the best possible rank for the task based on the amount of available space in memory hierarchy, given the task size is determined. We were able to evaluate the functionality but did not saw any performance gain.

The functionality evaluation showed that the scheduler works as expected and shows potential for performance gain as we aimed for. While the algorithm for finding the correct core is correct, the currently implementation of DUPC cannot differentiate between the core and rank assigned by UPCxx[17]. This is the primary reason, we are not able to see any performance gain in memory intensive task.

We can conclude that, the current implementation of DUPC has potential but limited to functionality and will potentially show performance gain with required changes in future.

Chapter 7

Future Work

The current implementation of space bounded scheduler has a number of areas for improvement. We present two major areas for improvement in future. The implementation lacks the ability to differentiate between a rank and a core. Apart from this, future implementation should focus on making the scheduler more developer friendly.

7.1 Differentiating core from rank

Our scheduler is not able to differentiate between a core and a rank. Even though we are able to change the rank based on space bounded scheduling algorithm, we are unable to place the task on the core we would have wanted. We expect to see some performance gains once we have added this feature to our implementation, which has already been shown in previous work.

7.2 Developer Friendly Implementation

The library should not add extra burden on developer and must remain user/developer friendly. For instance our implementation requires the developer to provide task size for our algorithm to work. The future implementation might be able to figure out the task size on its own. This does sound challenging and there is no easy way to do this. We did have a look at Dynamorio[19] to achieve the same without having the understanding of the approach.

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 speci- fication. 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 Cav´e, Zoran Budimli´c, 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-No¨el Quintin and Fr´ed´eric 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 spacebounded 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.

[18] GASNet, Global Address Space Networking, https://gasnet.lbl.gov, 2017

[19] Dynamorio, Runtime Code manipulation system and support code transformation, http://www.dynamorio.org

[20] PGAS architecture, https://mohamedfahmed.wordpress.com/2010/05/06/partitioned-global-address-space-pgas/