Distributed Control in HPX

A Report Submitted in Partial Satisfaction of the Requirements for the Ph.D Qualifying Examination in Computer Science, Indiana University.

Thejaka Amila Kanewala
thejkane@iu.edu
School of Informatics and Computing
Indiana University, Bloomington, IN, USA

ABSTRACT

Tasks in an unordered algorithm can be performed in any order and the final result does not depends on the task processing order. However, prioritizing tasks improve the efficiency of the algorithm. In our earlier work, we proposed a work scheduling mechanism for unordered distributed algorithms, called "Distributed Control" (DC). In our prior work we compared DC performance by implementing a DC based Single Source Shortest Path (SSSP) algorithm. We compared performance of popular Δ-Stepping based SSSP implementation with DC based SSSP. Our results showed that DC performance is much better compared to Δ-Stepping. Our previous implementation was based on a runtime environment called AM++. In this paper, we discuss a DC implementation based on HPX-3; a distributed runtime environment for ParalleX execution model. We will discuss implementation challenges encountered and various strategies used to overcome those and also a performance comparison between our previous implementation and current implementation.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Algorithms, Performance

Keywords

Parallelism, HPC, Scheduling, SSSP

1. INTRODUCTION

Pingali et al. [10] classify algorithms into two main categories of ordered and unordered algorithms. *Ordered algorithms* require ordering of tasks for correctness whereas *unordered algorithms* do not depend on any order of tasks. Parallelizing ordered algorithms are challenging as the parallel execution must still maintain the ordering. On the other hand, unordered algorithms are easier to parallelize as tasks can be executed in any order. *Distributed Control* (DC) focuses on unordered algorithms and DC's performance mainly relies on the *priority execution* of tasks and the *overhead* incurred during execution. The underlying runtime plays a vital role when it comes to the performance of DC.

Our previous implementation (cited in [14]) of DC was based on AM++ [12], an implementation of the Active Pebbles model [13], in that, algorithm was expressed using unbounded-depth active messages. Our results showed that DC based SSSP implementation performs much better compared to Δ-Stepping implementation of SSSP. In this paper we are focused on implementing DC on *High Performance ParalleX* (HPX) [6] platform which is based on Parallex [4] runtime specification. ParalleX is a new and experimental execution model for future high performance computing architectures. HPX is the freely available, open source implementation of the ParalleX execution model. There were several implementation attempts of HPX. The HPX version which is used in this paper is called HPX-3, but in this paper we will refer HPX-3 as HPX. The API exposed by HPX is modelled after the interfaces defined by the C++11 ISO standard and adheres to the programming guidelines used by the Boost [2] collection of C++ libraries.

We implemented DC on HPX from the scratch[1]. The implementation of DC on HPX is quite different from the DC implementation on AM++. In this paper we will discuss in detail about DC implementation on HPX and the lessons learned during the implementation. Further we will compare the implementation wise differences between AM++ based DC and HPX based implementation of DC. Paper also presents performance results of DC for HPX based implementation and AM++ implementation.

However, at the time of writing this report we are facing a blocking issue with HPX platform. This issue is raised as a bug report in HPX source repository and it can be referenced using URL `https://github.com/STEllAR-GROUP/hpx/issues/1315`. Though there is a single ticket opened it reports about 3 issues with the HPX platform.The first 2 issues in the bug report is due to the high message rate generated by DC. The third issue is an intermittent bug within HPX platform. Due to this blocking issue we were unable to collect results for HPX platform for higher scales at the time of writing this report.

We will start our discussion with a summary of Distributed Control. In Section 2 we will discuss about AM++; the runtime environment we used for our previous implementation. Section 3 summarises HPX (HPX-3). All the implementation details are discussed in Section 4. Section 4 also compares the implementation on AM++ and HPX whenever appropriate. Afterwards we will present the results we gathered so far. Finally we will conclude the paper with a summary and future work.

2. DISTRIBUTED CONTROL

Distributed Control (DC) is a work scheduling method for distributed unordered algorithms that benefit from priority. A distributed system consists of *workers* divided into several shared memory *domains*. Each domain contains a part of the global data, and workers in a domain perform tasks on the data allocated to that domain. Processing a task on one domain may generate more

[1]The source for the implementation is available at https://gitlab.crest.iu.edu/thejkane/hpx-sssp/tree/master

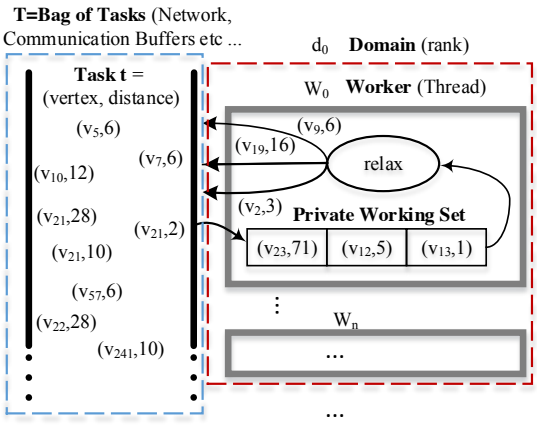


Figure 1: An Overview of *Distributed Control*, using SSSP as an example. (Referenced from [14])

tasks that depend on data on other domains, and remote tasks are communicated through the *global task bag* where every worker puts the tasks it generates. Whenever the bag contains tasks to be executed on a domain, any of the workers in the domain can retrieve any of these tasks. Workers collect the tasks they retrieve in *private working sets* that are ordered according to task priority.

In our previous work we implemented *Single Source Shortest Path* (SSSP) using DC. While Dijkstra's algorithm is the optimistic algorithm (that does minimal amount of useless work) for SSSP, it is highly suitable for sequential execution due to the use of a priority queue to order distances. In the context of SSSP, a *task* is the pair vertex and distance. In a distributed setting use of a global data structure that acts as a priority queue degrades performance significantly due to communication costs and synchronization costs. The *Chaotic SSSP* is an unordered algorithm which represents the class of algorithms with large available parallelism [5] but generates lot of work. DC based SSSP algorithm sits in between Chaotic SSSP algorithm and Dijkstra's algorithm and attempts to achieve best performance. DC based SSSP implementation orders work locally as much as possible without incurring much overhead. Further DC relies on underlying runtime environment to deliver tasks to an appropriate local priority queue as soon as possible. By maintaining local priority queues and rapidly delivering tasks to local queues as soon as they released from one queue, DC tries to build an implicit priority queue that is approximated to priority queue used in Dijkstra's algorithm. Further DC avoids any form of intermediate synchronization and due to that DC also avoids straggler effects and allows all workers to execute tasks to their maximum capability. The only synchronization performed by DC is during termination.

Figure 1 graphically describes DC scheduling for SSSP. Each worker works on its private working set and generates new tasks by relaxing edges for vertex in the selected task. The generated tasks may depend on data residing in a different domain, and they need to be sent to the domain that can satisfy their data dependencies. Therefore, there are tasks that are in the network or in buffers (the task bag in the figure) and have not yet reached a private priority queue. Underlying runtime characteristics are used to deliver tasks in buffers and in network to an appropriate priority queue as soon as possible.

It is also interesting to explore the work profile of DC. Figure 2 shows how DC generates tasks over time.

At the start of the execution, when SSSP begins from the source, there are few tasks to keep workers busy. In an environment where available parallelism is higher than required parallelism SSSP

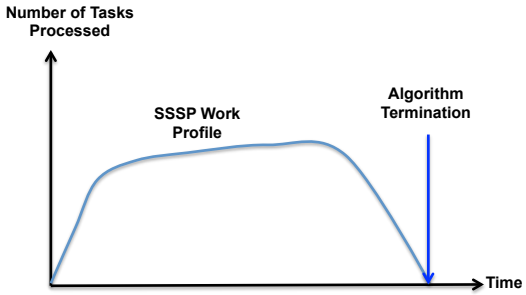


Figure 2: SSSP work profile.

algorithm would utilize tasks according to work profile. But with limited resources, as the algorithm proceeds, worker utilization increases because of the increasing amount of vertices in the input graph that are explored and a limited number workers to consume them. Then, towards the end of execution, worker utilization decreases as most of the work is completed. In our previous implementation we used coalescing of messages; i.e. several tasks are bundled together to create a single message before sending through the wire. An interesting fact about DC is, it generates lot of unfilled coalesced buffers (i.e. buffers which are not fully filled) at the start of the algorithm as well as at the end of the algorithm.

3. AM++

AM++ [12] is a runtime based on lightweight active messages that are sent explicitly but received implicitly. In AM++ each lightweight message is called a *pebble*. The implicit receive mechanism is based on *handlers*, which are user-defined functions that are executed in response to the received pebbles. AM++ is a library interface that can be executed by many workers (threads). Each worker can execute independently, and when it calls AM++ interfaces it may execute tasks from the AM++ *task queue*, which schedules tasks such as network polling, buffer flushing, and pending handlers. At the lowest level, pebbles are sent and received using transports that encapsulate all low level AM++ functionality such as network communication and termination detection. Currently, the low-level network transport of AM++ is built atop of MPI, but none of the MPI interfaces are exposed to the programmer, and AM++ has supported other transports before. In order to send and handle active pebbles, individual message types must be registered with the transport. A transport, given the type of data being sent and the type of the message handler, can create a complete message type object. To increase bandwidth utilization, AM++ performs *message coalescing*, combining multiple pebbles sent to the same destination into a single, larger message. In the current implementation, a buffer of messages is kept by each node for each message type that uses coalescing and for each possible destination. The size of coalescing buffers is determined by the maximum number of pebbles (vertex-distance pairs in DC) to be coalesced and the pebble size. Messages are appended to the buffer, and the entire buffer is sent when it becomes full or it is flushed when there is no more activity. Message coalescing increases the rate and decreases the overhead at which small messages can be sent over a network.

AM++ also provides a concept called *epoch*. Epoch is the period where we are allowed to send and receive messages. AM++ provides 2 forms of epochs as depicted in Figure 3.

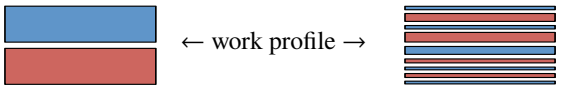
In scoped epochs we need to make sure all messages have finished sending and receiving within the scope where epoch is defined. At the end of each scoped epoch AM++ performs termination detection. Figure 3b shows how we can try to end epoch manually. In this

Listing (1) Graph generation based on graph500

```
{
  epoch e;
  // work
}
```

Listing (2) Graph generation based on graph500

```
epoch e;
while(!e.end()) {
  // work
}
```



(a) Scoped epoch with two distinct phases of work.

(b) End-epoch test with interleaving of work and progress.

Figure 3: Two epoch execution models and the interleaving of work (blue) with AM++ progress (red). Referenced from [14]

case at the end of each epoch test, AM++ sends the pebbles in side partial buffers. Out of above two strategies we chose 3b approach for our previous implementation as it helped us to send more small messages quickly.

AM++ is capable of running multiple threads. In general, AM++ workers perform two kinds of work: the worker's *private* work, and AM++ progress that can occur any time an AM++ interface is called. A thread's private work includes tasks such as local bookkeeping or preparing for an epoch.

4. HPX

4.1 ParalleX

Most of the recent high performance distributed applications are based on message passing interface (MPI). The MPI is an implementation of the *Communicating Sequential Processes* (CSP) model. *Parallex* is a process model proposed as an alternative to CSP. High Performance Parallex (well known as HPX) [7] is a ParalleX compliant runtime system implementation. In this section we will briefly summarize ParalleX execution model.

Exascale systems expected to emerge at the end of the next decade and it will require investigating multiple ways of parallelizing applications to achieve best performance. Thus there are many scaling impaired applications that do not give better performance using conventional distributed programming models such as MPI. [1] discuss in detail challenges in new generation HPC systems. In the following, we summarizes those challenges;

1. *Starvation* - due to lack of usable application parallelism and means of managing it
2. *Overhead* - reduction to permit strong scalability, improve efficiency, and enable dynamic resource management
3. *Latency* - from remote access across system or to local memories
4. *Contention* - due to multicore chip I/O pins, memory banks, and system interconnects

The ParalleX execution model introduces number of concepts to overcome above challenges. Some of the important ParalleX execution model concepts or *management principles* are *ParalleX Processes*, *Active Global Address Space* (AGAS), threads and their management, *parcel transport* and parcel management and *Local Control Objects* (LCOs). Most of these features are already implemented in HPX system. In the following, we briefly discuss HPX runtime in related to DC implementation.

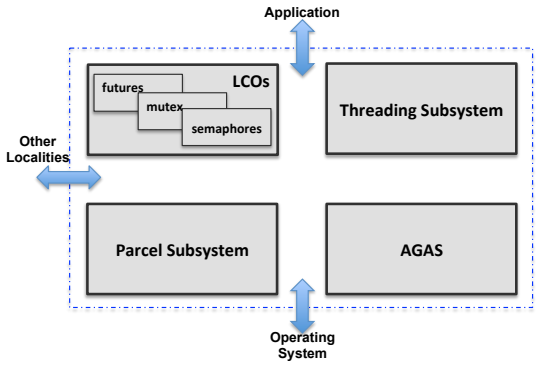


Figure 4: HPX Overview.

4.2 The HPX Runtime

The HPX runtime starts with the application that is using HPX. Similarly when application terminates the HPX runtime also terminates. HPX implements most of the ParalleX features which we discussed above. An overview of HPX runtime is given in Figure 4. We will go through briefly each sub component in HPX runtime.

4.2.1 AGAS

In HPX, entities that are separated by the network are called *localities*. Usually a locality refers to a node. AGAS implements a *global address space* which spans across all localities in the system. In HPX implementation AGAS consists of set of components. Each component is given a *global identifier*. Two components can communicate with each other by sending messages using global identifiers. Further the components are designed to move among localities and yet communicate with other components using global identifiers. HPX also allows user to define aliases to components so that a component is created by one locality and component *actions* can be invoked from another locality either using alias or global identifier. An action is a function that can be invoked remotely.

HPX components have a client and a server. The server is the actual entity that gets created in HPX AGAS. The client is more like a stub; client can be used to wrap server's actions and allow remote locality invoke actions.

4.2.2 Parcels

A parcel is a form of an active message [11] that can be sent from one locality another locality. All the network communications in HPX are based on parcels. Parcels are mainly used to encapsulate remote method calls. Usually a parcel contains the destination global address of an object to act on, a reference to objects method, arguments to method and optionally a continuation. Further parcels are transmitted asynchronously.

4.2.3 Threading

Upon receiving a parcel, locality converts the parcel into a method call and creates a *lightweight thread*. HPX comes with several types of light weight thread schedulers. The configured scheduler will select the thread and execute the method. If parcel destination is same as origin, HPX will just create a thread instead of sending the parcel. We can think of parcel as a method of spawning a remote (or local) thread. We can have many HPX threads mapped to a single OS thread. This model enables *fine grained parallelization*.

HPX lightweight threads does not require kernel calls during thread context switching. Further HPX threads are scheduled *cooperatively*, i.e. they are never preempted by the scheduler. But threads may voluntarily suspend themselves during IO, synchronization, locking. HPX thread scheduler also implements *work stealing* to load balance

among threads.

4.2.4 *Local Control Objects (LCO)*

LCOs provide a means to control parallelization and synchronization in HPX applications. LCOs attempt to avoid use of global barriers for synchronization and try to utilize computing power as much as possible with constraint-based synchronization, whenever necessary. In the following, we briefly discuss some of the important LCOs in HPX.

1. *Futures* - Futures are the mostly used LCOs in HPX (as per our experience). Futures are proxies for results that are not yet known. Future is associated with a function that returns value and can be created using *hpx::async* function. Value associated with future can be retrieved by calling *future::get()*. If the value is not available at the time of call then calling thread will be suspended until value is available.

2. *Dataflow* - Dataflow objects manage data dependencies without barriers. A dataflow LCO waits for a set of futures to become ready and triggers a predefined function.

3. *Concurrency control tools* - Mutexes, Semaphores, Spin Locks, Condition Variables are also exposed as LCOs. These are synchronization, mutual exclusion mechanisms used in traditional multithreaded programming.

4.2.5 *The API*

The HPX API strictly conforms to the C++11 standard. HPX implements all interfaces defined by the C++ standard related to multithreading (e.g:- future, thread, mutex, etc.). HPX provides several ways to execute a function;

1. *Synchronous function execution* - The caller waits till calling function finishes its execution. This is the most natural way of executing the function.

2. *Asynchronous without Synchronization* - This is *fire and forget* invocation. In this execution the caller does not wait for function to finish its execution. This type of execution is achieved through *hpx::apply* call.

3. *Asynchronous with Synchronization* - The caller of the function will invoke the function and the call will immediately returns but may not return the results immediately. The caller may invoke some other functions upon returning. Then when caller actually needs the result it can wait for invocation to finish (if not completed). This type invocation is carried out using futures.

5. DC IMPLEMENTATION IN HPX

In this section we will discuss DC implementation on top of HPX in detail. Whenever appropriate we will compare design choices with our previous implementation that was based on AM++.

Both AM++ based *Parallel Boost Graph Library* (PBGL2) implementation and MPI based PBGL(1) implementations uses *Boost Graph Library* (BGL) based graph implementations. Both implementations are coupled with their underlying infrastructures (e.g :- AM++ based PBGL implementation make use of epoch like concepts during graph distribution). Therefore we implemented a graph structure from the scratch to work with HPX implementation. This approach gave us lot of flexibility and control over some of the aspects such as graph generation and graph distribution. In Section 5.1 we will discuss about graph structure we used in detail.

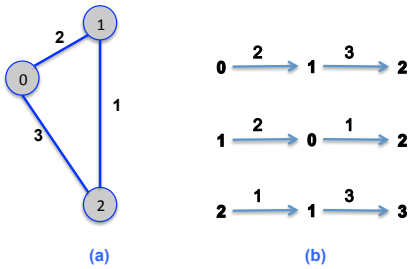


Figure 5: Adjacency List representation.

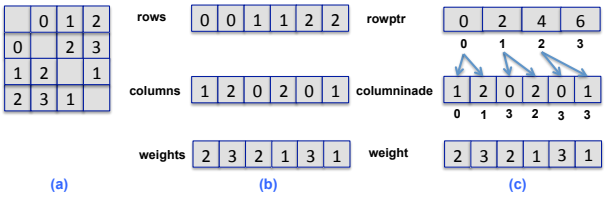


Figure 6: CSR graph representation.

HPX runtime offers rich set of functionalities to implement parallel applications. We realized, there are multiple ways to implement DC based SSSP algorithm. Section 5.4 and onwards we discuss DC-SSSP implementation on HPX.

5.1 Graph Representation

As mentioned above we implemented graph representation for HPX from the scratch. There are 2 widely used methods to represent a graph. 1. As an *adjacency list* 2. As a *compressed sparse row (CSR)* graph.

An adjacency list maintains a linked list of edges per each vertex. This is depicted in Figure 5(b). In a distributed setting adjacency lists will be distributed among multiple localities.

To explain CSR format we will go through the example in Figure 6. Figure 6(a) shows how the example graph can be represented as a matrix. To represent the matrix in Figure 6(a) in a program one could use tuples. Tuples should record source vertex, target vertex and the weight associated with the edge. As an example tthe matrix in Figure 6(a) can be represented as tuple set (0,1,2), (0,2,3), (1,0,2),(1,2,1), (2,0,3),(2,1,1). We can further refine the representation of tuples by creating 3 arrays to represent the rows, columns and weights as shown in Figure 6(b). But rows array in Figure 6(b) has same value repeated. To further decrease the space occupied by the graph we can maintain row pointers to column array as depicted in Figure 6(c). The row pointer array contains elements equal to number vertices + 1. The column index array and weight arrays will remain same.

Due to two reasons we chose our algorithm to be based on CSR format. 1. The space accommodated by CSR graph structure is less compared to adjacency list representation (due to compressing). Due to same space issue [3] shows that CSR format graphs gives better performance in distributed setting compared to equivalent adjacency list graph. Also less space requirement enables to generate larger graph for testing in a single node. 2. Our previous implementation (i.e. AM++ PBGL implementation) is based on the CSR graph format. Therefore for a 1-1 comparison we would need the same graph structure.

In summary the CSR graph representation involves 3 arrays as depicted in example in Figure 6(c). They are row pointer array, column index array and weight array.

5.2 Graph Distribution

There are 2 main methods to distribute graphs. 1. 1D - distribution

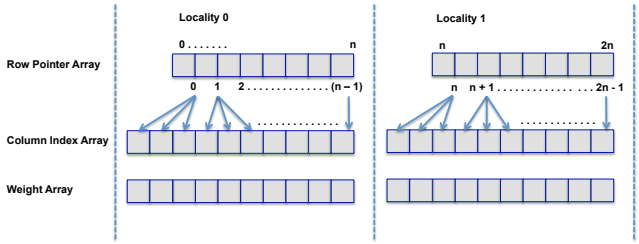


Figure 7: CSR graph distribution.

- where vertices are distributed among localities, 2. 2D distribution (also called edge based distribution) where edges are distributed among multiple localities. For HPX based implementation we used vertex based distribution which was the same distribution strategy used in AM++ based PBGL implementation.

As discussed in previous section the graph is represented using 3 arrays (row pointer array, column index array, weight array). Vertex based graph distribution involves dividing row pointer array and allocating portions to each locality. During allocation we also need to allocate appropriate sections of column index array and weight arrays. In the following, we briefly explain our approach to allocate arrays to localities.

First, the *number of vertices per locality* (num_vertices_per_locality) is calculated as follows;

$$num\_vertices\_per\_locality = total vertices / total localities$$

Then vertex number ranges are assigned to each locality. i.e. Locality 0 is responsible for vertices (0, num_vertices_per_locality-1), locality 1 is responsible for (num_vertices_per_locality, 2 x num_vertices_per_locality -1), locality 2 is responsible for (2 x num_vertices_per_locality, 3 x num_vertices_per_locality -1), etc. If there is a remainder after dividing total vertices by localities that remaining vertices are allocated to last locality. This involves partitioning column index arrays and weights arrays and allocating them to localities where their respective source vertices (in corresponding edge) belong to. Figure 7 shows how arrays are divided among localities.

5.3 Graph Generation

All the graphs generated conforms with Graph500 [9] specification and graphs are generated based on reference implementation of Graph500. The Graph500 reference implementation randomly generates edges according to two input seed values.

PBGL2 generates the graph in a single node and then permutes the graph and distribute among localities. Initially we also followed the same approach. But we experienced out of memory errors when generating graphs of scale 29 and onwards. We are not yet certain what caused high memory usage. Our assumption is HPX runtime, memory wise takes more space compared to AM++ runtime. Finding cause for higher memory usage is in our future plan. To generate higher scale graphs, graph generation was distributed. In the following, we explain how we distribute graph generation.

To make graph generation distributed we followed a novel but simple approach. For a given scale using graph500 reference implementation we create iterators. When incrementing an iterator graph500 generator generates a new edge. This logic is briefly summarised in Listing 3.

Listing 3: Graph generation based on graph500

```
1  typedef boost::graph500_iterator<vertex_t, edge_t>
       Graph500Iter;
2  double edgefactor = 16;
3  ...
4  vertex_t m = (unsigned long long)(1) << scale;
5  edge_t n = static_cast<edge_t>(floor(m * edgefactor));
6  ...
7  uint64_t a = rand_64(gen);
8  uint64_t b = rand_64(gen);
9  ...
10 Graph g_500(m, n, true);
11 g_500.addEdges(Graph500Iter(scale, 0, a, b), Graph500Iter
       (scale, n, a, b), die);
```

In above listing, the relevant code fragments to current discussion are in lines 1, 7 and 11. When creating iterators we need to specify scale and a, b seeds. The number n represents total number of edges. Seeds are generated using random number generator. An interesting observation is, If the seeds are equal we get the same set of edges. We use this fact to distribute the graph.

The rank 0 (master node) decides upon seed values. These seed values are sent to other localities during graph generation call. Each locality creates graph500 iterators as shown in Listing 3. But a generated edge is actually populated into CSR structure only if source of the generated edge lies in the vertex range assigned to current locality.

Above approach may not gain much performance compared to graph generation in a single node, but certainly it reduces the amount of memory usage and allows to generate much larger graphs. Further this approach also simplifies the graph distribution process. Now the graph distribution only needs assigning vertex ranges and distributing seed values for graph generation. In summary graph distribution is achieved by *distributing graph generator (Graph500) iterators*.

5.4 DC-SSSP Implementation

DC-SSSP algorithm is listed in [14]. We will not go through the algorithm pseudo code in this paper but will summarise the algorithm. To describe the algorithm we assume vertices in the graph are distributed as explained in previous section. In addition we assume there is an array in each locality that records the shortest distance from the source and before algorithm starts the array values are initialized to infinity. This array is called *distance array*.

5.4.1 The Algorithm

A key feature in DC scheduling is use of a *local priority queues* to order tasks (vertex-distance pairs). During relaxation of an edge new vertex distance value will be calculated for the target of the edge and newly calculated value is sent to the locality where vertex resides (The locality where the vertex belongs, the *owner* of the vertex). When particular locality receives a vertex-distance pair, algorithms compares the received vertex distance with the distance recorded in the distance array. If the received distance is better than what is recorded in distance array algorithm update the array with new distance and put vertex-distance pair to a priority queue structure. When processing the priority queue algorithm gets the vertex-distance pair with shortest distance first and checks whether vertex distance is updated to better value since it was pushed into the queue. If not the retrieved vertex's neighbours are relaxed. The algorithm continues until there are no more vertex-distance pairs to be relaxed. When there are no relaxation algorithm conclude the execution.

5.4.2 Algorithm Implementation with HPX API

In HPX API *component server* is the main abstract entry point to interact with the AGAS. A component server may contain *actions*. An action is a method that can be invoked remotely(or locally). Upon instantiation component server is assigned a global identifier (gid) which is unique across the whole system. Then later the gid can be used to access component server's data and invoke component

server's actions. The code in Listing 4 shows how a component server class looks like;

Listing 4: HPX component server implementation

```
// This is the server side representation of the data. We expose this as a HPX
// component which allows for it to be created and accessed remotely through
// a global address (hpx::id_type).
struct partition_server
  : hpx::components::simple_component_base<
      partition_server> {

  partition_server(graph_partition_data const& data)
    : graph_partition(data) {
...
  // action is simply a method with some additional preprocessor
  // macro definitions
  void relax(const vertex_distance& vd);
  HPX_DEFINE_COMPONENT_ACTION(partition_server, relax,
                              dc_relax_action);
...
private:
  graph_partition_data graph_partition;
}

// Each defined component action also needs to register to let
// HPX runtime inform that defined action is an active action
typedef partition_server::dc_relax_action
    partition_relax_action;
HPX_REGISTER_ACTION(partition_relax_action);
```

As mentioned in Section 5.2 the graph is partitioned equally among available localities (last locality maybe an exception if number of vertices is not divisible by the number of localities). Each partitioned graph is associated with a component server instance (as shown in Listing 4, the private variable graph_partion). To achieve this association an instance of a component server class is created in each locality. In HPX an instance of component server can be created in a remote locality (or local) as shown in Listing 5.

Listing 5: Instantiating component server in a remote locality

```
...
hpx::id_type where = locality[i]
graph_partition_data data = ...
...
typedef hpx::components::client_base<partition,
    partition_server> base_type;
base_type(hpx::new_colocated<partition_server>(where,
    data))
```

Variable *where* says in which locality, partion_server (component server class in Listing 4) needs to be instantiated and *data* represent the graph partition which encapsulate vertex range associated to the locality referred by *where*. Next we discuss how defined actions can be invoked.

Actions defined in Listing 4 can be invoked using a code similar to the following;

Listing 6: Invoking a server action

```
partition_server::dc_relax_action act;
hpx::apply(act, gid, vd);
```

The *gid* represents the global identifier associated with the component server object, *vd* is an instance of vertex_distance structure which represent the pair vertex and distance. As discussed in Section 4.2.5, HPX provides several ways to invoke an action. The hpx::apply invokes the action (or the component server method remotely or locally) asynchronously in a fire and forget manner.

HPX component also contains a client part. The client code is not necessary but it is a convenient to abstract out certain details about AGAS (e.g:- the global identifier). The client code usually wraps all HPX actions with a method. For example the client code for above defined server component would look like follows;

Listing 7: Wrapping client code for server actions

```
// This is a client side helper class allowing to hide some of the tedious
// boilerplate.
struct partition : hpx::components::client_base<partition, partition_server> {
  typedef hpx::components::client_base<partition,
      partition_server> base_type;
...
  // Invoke remote relax
  void relax(vertex_distance const& vd) const {
    partition_server::dc_relax_action act;
    hpx::apply(act, get_gid(), vd);
  }
...
}
```

Above we discussed the basic infrastructure of the algorithm implementation. In the following, we discuss more interesting challenges faced during the implementation and our approach of overcoming those challenges.

5.4.3 *Queues*

Priority Queues play a major role in DC scheduling. In previous implementation a priority queue was maintained per each OS thread; which avoids contention between threads to access a priority queue. Further assigned thread always populated the priority queue and the same thread consumed the priority queue (AM++ task based execution framework allowed this design). But in HPX, threads are lightweight threads that are running on top of OS threads. Therefore having a priority queue per each OS thread was not an option. Further in PBGL2 the number of priority queues were decided by the number of cores assigned, but for HPX we do not have a deciding factor for number queues as threads are lightweight. Therefore the number of queues are configurable in HPX based DC-SSSP implementation.

As discussed in Section 5.4.1 algorithm should insert incoming *better* vertex-distance pair to a priority queue after comparing the distance with the distance array. In AM++ based implementation priority queue was selected based on currently executing thread id; i.e. insert vertex-distance pair to the queue that is associated with current OS thread. But in HPX implementation queues are spread over OS threads; there is no association between OS threads and queues. Therefore in HPX implementation, during a relax a queue must be selected *randomly* and insert vertex-distance pair to the randomly selected queue. Since there is no assurance that randomly selected queue is not used by another thread, each insertion to the queue needs to be guarded with a LCO (mutex).

There were several options to implement queue consumption logic. They are;

1. Single consumer thread created during first insertion - Start a HPX lightweight thread to consume a given queue when inserting first element to the queue and let thread exit when queue ran out of elements to process. Again when an element is inserted into the queue start a HPX thread but make sure there is only one consumer thread active per queue.

2. Multiple consumer threads created during first insertion - Same as Option 1 but allow multiple consumer threads to

work on a queue.

3. Start HPX lightweight threads per each queue at the start of the program execution. Push elements to queue during relax and if queue ran out of elements wait (and allow HPX thread scheduler to schedule another thread) till different (producer) thread push elements to queue and notify.

Our very initial results showed that option 3 performs slightly better than option 1. We believe option 1 introduces extra contention during consumer thread creation; because we need to make sure there is no other thread operating on the same queue during consumer thread creation. We did not implement Option 2 but we believe option 2 introduces more contention than option 1 (Experimenting option 2 is in our future plans). Therefore the current implementation is based on option 3; i.e. start a HPX thread at the start of program execution.

Thus, the option 3 introduces a slight complication caused by the behaviour of HPX light weight threads. HPX threads are *cooperative*; i.e. they cannot be preempted by the HPX thread scheduler. If a consumer thread starts working on a queue it will execute until queue is empty. If there is a constant stream of vertex-distance tasks populating the queue the thread will not interleave with other threads. To overcome this issue we introduced a *yield count*. Whenever particular thread process number of elements equal to the yield count it will yield itself by executing *hpx::this_thread::yield();*. The instruction *hpx::this_thread::yield()* instructs HPX thread scheduler to yield current executing thread and allow another thread to be scheduled.

Option 3 lead us to another challenge; i.e. when to terminate consumer threads ? Ideally consumer threads should be terminated when there are no more relaxes to be done. In the following, subsection we discuss about termination in detail.

5.4.4 Termination Detection

AM++ based DC-SSSP implementation controls termination through AM++ interface. AM++ termination only takes place if there are no *activities* (definition of activity broad here, but we will not go into details) in the system, in addition AM++ provided an interface (API) to increase the activity count and decrease the activity count. PBGL2 DC-SSSP implementation uses the AM++ API to increase the activity count when a priority queue transition from empty to non-empty and decrease activity count when a queue transition from non-empty to empty.

HPX also implements a similar termination strategy. HPX termination will take place when we call *hpx::finalize()* function. This function is usually called at the end of the program (at the end of *hpx_main()*). HPX decides program termination based on several factors. Some of those factors include, whether all parcels are delivered, whether all threads have finished their execution, etc. Unfortunately HPX API does not provide an interface to control termination as in AM++. Therefore we had to implement a termination detection algorithm on top of HPX to end consumer threads running on queues.

Termination detection algorithm in HPX DC-SSSP is based on *the four counter method* described in [8], Section 4. The implementation maintained 2 counters in each locality. *acitve_count* to maintain the number of vertex-distance tasks that are being active in a locality and *complete_count* to maintain number of tasks , completed processing. The active_count is incremented before sending vertex-distance message for relaxing. The completed_count is increased when a vertex-distance message gets invalidated due to a better distance in distance map or when it finish relaxing all its neighbours. As per four counter method, to detect termination; accumulation of active_count and accumulation of complete_count in the whole system must be equal twice successively. The total active_count and total completed_count is calculated using HPX reductions.

In the initial implementation the termination detection algorithm was ran continuously until counters are equal twice in a row. But with this implementation number of reductions performed to calculate total active_count and total completed_count was quite high. In Order to reduce the number of reductions, reduction function(s) was changed to wait till all the queues become empty (in a single locality). This optimization greatly reduce the number of reductions taking place and on average for scale 15 graph there were only 4 reductions.

5.4.5 Message Handling in HPX

When running DC-SSSP we experienced an extreme slowness even for very small scales like 15. Further investigations reveals that DC generates quite a lot of messages with a high message rate and HPX is unable to handle such message rate. For higher scales we were getting "MPI message truncation" errors. (As described in ticket 1315[2]). In this section we discuss various strategies we came up to overcome this issue.

HPX maps a single action invocation to a single parcel. In our very first implementation the remote action interface for relax looks like follows;

void relax(vertex_distance vd);

With above interface a parcel is created per each vertex_distance generated. On the other hand DC-SSSP is a message hog; DC-SSSP generates large number of messages at a higher rate and HPX parcel subsystem is unable to handle such a large number of messages. To alleviate this issue message coalescing was introduced.

Message coalescing is the process of bundling several messages together and sending bundled message to the destination. Message coalescing is provided as a platform functionality in AM++. But HPX is more flexible in setting the message coalescing. If message coalescing is needed it has to be implemented at the remote action interface. In HPX context this is usually referred to as setting *granularity* for an action. For coalescing strategies a new relax action is introduced. The action signature looks like follows;

void coalesced_relax(std::vector<vertex_distance> vds);

With coalesced_relax we can send multiple vertex-distance messages in a single parcel.

For DC-SSSP, granularity was implemented in 3 ways. The difference between each strategy is the way coalesced buffers were handled. In all strategies a maximum coalescing size limit was enforced. The maximum coalescing is a configurable value. In the following, we discuss each strategy. During the discussion we will use terms *partial buffer* to refer to a buffer that is not filled upto maximum coalescing size and *full buffer* refers to a buffer where number of elements is equal to maximum coalescing size. These terms are analogous to terms used in AM++.

***Strategy 1* :** In the initial approach each consumer thread maintains a buffer per each destination locality. When a buffer reaches its maximum coalescing limit HPX-SSSP will send it to appropriate destination. Whenever consumer thread's queue gets empty the messages left in buffers are sent to their destinations. This behaviour is depicted in Figure 8. For example if there are 10 queues and 5 localities there will be 5 * 10 = 50 buffers. In addition buffers are emptied whenever individual queues run out of vertex-distance pairs to process. Therefore this method generates higher number of partial

[2]https://github.com/STEllAR-GROUP/hpx/issues/1315

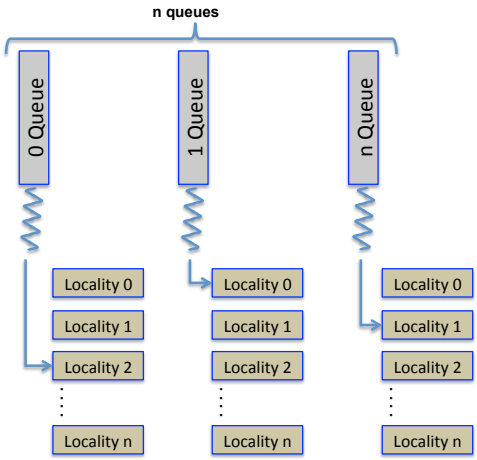


Figure 8: Coalescing implementation strategy 1 - Each queue holds set of buffers for all destinations.

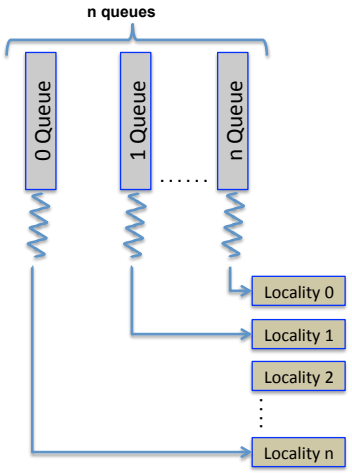


Figure 9: Coalescing implementation strategy 3 - Single set of buffers shared between all queues.

buffers (small sized) at the start of the algorithm and also towards to end of the algorithm (See work profile in Figure 2). Due to large number of partial buffers generated at higher scales the performance of HPX-SSSP algorithm was not satisfiable.[3]

Strategy 2 **:** Strategy 2 followed the same queue implementation as in strategy 1, but reduced the number of partial buffers during startup and at the end of algorithm by sending partial buffers when "all" queues are empty. The reduction operation will wait till all queues empty before calculating active_count and completed_count. Once all the queues are empty the remaining messages in all buffers are sent to their destinations. This obviously has a *straggler* effect on some threads (since we need to wait till all queues are empty), but this strategy reduces number of partial buffers. AM++ follows a similar strategy in its coalescing implementation. Still the number of partial buffers generated during algorithm execution was too much for HPX to handle and showed a sluggish execution.[4]

Strategy 3 **:** In this strategy a single set of buffers indexed by the destination locality were maintained. All the outputs from queues are pushed into relevant output buffer. Figure 9 summarises the implementation. Appropriate locking mechanisms were implemented to make sure two or more threads will not update the same output buffer at the same time. Compared to other strategies this approach increases contention between threads, but reduces number of partial buffers as all messages are accumulated per destination. As in previous approach the remaining messages in buffers will be sent to their destinations during a reduction, once all queues become empty. Performance results in this case are much better compared to previous two strategies.[5]

Implementing a coalescing layer on top of HPX opened paths for more possible optimizations. Some of the possible optimizations are; 1. Sorting output buffers, 2. Reductions. Out of possible optimizations mentioned above we implemented output buffer sorting. Implementing reductions involves maintaining reduction cache and coming up with policies to remove data from the cache. It is possible to implement reductions on top of current implementation but we were more focused on getting application executed for larger scales. Implementing reductions is in our future list of tasks.

[3]Code for strategy 1 is in branch `https://gitlab.crest.iu.edu/thejkane/hpx-sssp/tree/coalesc_attempt_1`

[4]code for implementation is in branch `https://gitlab.crest.iu.edu/thejkane/hpx-sssp/tree/mult_q_mult_buf`

[5]code for implementation is in branch `https://gitlab.crest.iu.edu/thejkane/hpx-sssp/tree/master`

6. RESULTS

Results we are going to present are not the best interesting results; mainly due to the issues we faced when running algorithm in many nodes and for larger scales. But we will start the section by discussing some of the vital parameters that affect the performance. Then we will present results for smaller scales.

Parameters effecting performance can be mainly divided into 2. 1. Algorithmic parameters 2. Runtime (HPX specific) parameters. Some of the algorithmic parameters are as follows;

1. Number of queues
2. Coalescing size
3. Yield count
4. Enable buffer sorting

HPX comes with number of runtime parameters[6]. Testing all runtime parameters is extremely laborious task. Therefore we tested some of HPX runtime parameters and measured the effect on performance. Some of those parameters are number of MPI connections, enable numa sensitivity, thread scheduling policy, etc.

6.1 Testing Approach

Since there are many parameters (Both algorithmic and HPX specific) effecting HPX DC-SSSP implementation, we first carried out several tests to go over parameter space and find parameters which gives best performance for each scale. Then those parameters are used in *strong scale* and *weak scale* testing.

Running DC-SSSP on more than one node showed extreme slowness. Further we were not able to consistently execute program for each test case when running in multi-nodes. We were getting intermittent issues reported in bug reports. Therefore we took results for a single node only. Scales we ran were extremely small. We present those results just to get an idea about the state of HPX DC-SSSP implementation.

All tests were executed in Indiana University Big Red II [7]. For scaling tests we used CPU queues and for other tests we used gpu queues. Below we present our findings.

[6]These parameters are listed in HPX documentation at `http://stellar-group.github.io/hpx/docs/html/hpx.html#hpx.manual.init.commandline`

[7]https://kb.iu.edu/d/bcqt

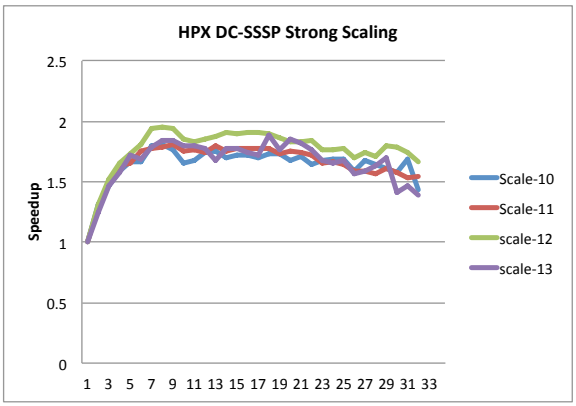


Figure 10: Strong scale results for 1-32 cores and scales 10-13.

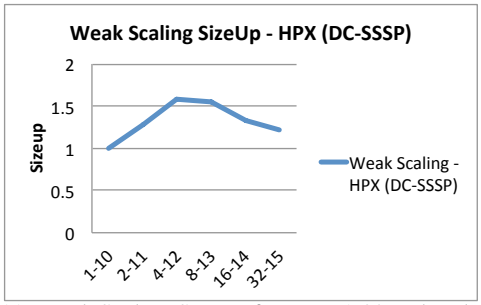


Figure 11: Weak Scaling SizeUp for cores 1-32 and scales 10-15

6.1.1 *Strong Scaling Results*

We present strong scaling speedup results for scales 10-13 in Figure 10.

As per results maximum parallel speedup is achieved when application uses 6-7 cores. Afterwards parallel speedup either stable or slightly decreases as the number of cores increases. We believe this is due to increase amount of contention between OS threads.

6.1.2 *Weak Scaling Results*

For weak scaling we calculate the *Size Up (SizeUp* value for number of cores used. Our scaling starts with 10. i.e. on a single core we ran algorithm starting scale 10. The problem size is expressed as the scale. The SizeUp calculation is given below;

Problem Size = N (Scale)
Processor Cores = K
Program executing time for scale (N) and cores (K) = T(N, K)
Let N(K) be the problem that depend on number of K cores
Then sequential time = $T\_seq(N(1), 1)$
SizeUp(N, K) = { N(K) / N(1) } *
{ T_seq(N(1), 1) / T_par(N(k), k) }

Further, we calculate Efficiency = SizeUp(N, K) / K
If SizeUp is ideal then efficiency should be 1 irrespective of number of cores used.

Graphs 11 and 12 very well depicts that HPX DC-SSSP does not scale well. One reason for poor scaling is HPX platform's inability to handle large amount of small messages in an effective way.

6.1.3 *Comparison between AM++ DC-SSSP and HPX DC-SSSP*

In Order to see the difference between AM++ implementation and HPX based implementation we compare results for AM++ implementation and HPX implementation (In Figure 13). Also the same graph shows that NUMA sensitivity did not have an impact on the collected results.

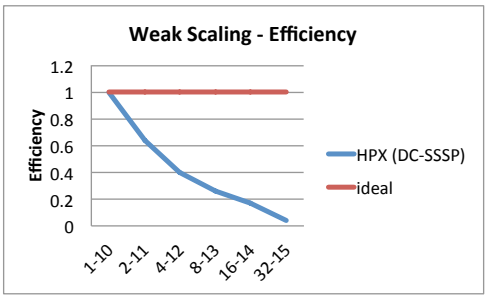


Figure 12: Weak Scaling Efficiency for cores 1-32 and scales 10-15.

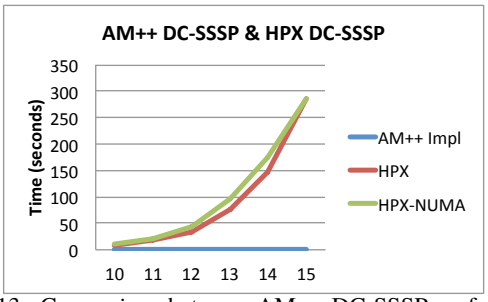


Figure 13: Comparison between AM++ DC-SSSP performance, HPX DC-SSSP performance and HPX DC-SSSP (NUMA enabled) performance, for scales 10-15.

7. SUMMARY & FUTURE WORK

HPX provides user friendly abstraction and full set of infrastructure to implement distributed high performance application. HPX lightweight threads allow us to achieve maximum parallelism. Programming model is easy to adapt and documentation is quite concise to start with.

During this exercise we reported several HPX bug reports. Some of those ticket references are as follows;

1. Unable to invoke component actions recursively - `https://github.com/STEllAR-GROUP/hpx/issues/1297`
2. Compilation error when tried to use boost range iterators with wait_all - `https://github.com/STEllAR-GROUP/hpx/issues/1301`
3. Unable to execute an application with –hpx:threads - `https://github.com/STEllAR-GROUP/hpx/issues/1308`
4. Errors while running performance tests - `https://github.com/STEllAR-GROUP/hpx/issues/1315`
5. Unable to run program with abp-priority and numa-sensitivity enabled - `https://github.com/STEllAR-GROUP/hpx/issues/1318`
6. An exception when tries to specify number high priority threads with abp-priority - `https://github.com/STEllAR-GROUP/hpx/issues/1319`

Out of all above tickets 1315 is the most critical. It is not resolved at the time of writing this report. This issue is one of the reasons for us to implement 3 types of coalescing in the level of the application. HPX also provided coalescing at runtime after reporting issue 1315 (see `https://github.com/STEllAR-GROUP/hpx/pull/1321` for

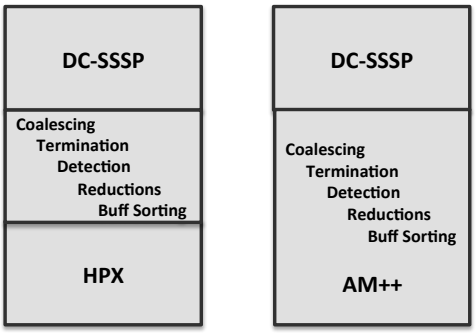


Figure 14: Comparison between AM++ and HPX as a runtime for DC-SSSP.

more details), but still 1321 does not solve all the issues reported in 1315. Ticket 1315 act as a impeding issue for collecting performance results. We were unable to run HPX applications on more nodes (generally 10 or more), even if we were able to run the application, execution showed a sluggish behaviour.

Further we realize DC-SSSP implementation on HPX is not straightforward. One of DC's main objective is to order tasks without overhead as much as possible. But HPX's abstraction layers prevent achieving thread level local ordering. We had to share priority queues among several lightweight threads and had to introduce locking synchronization mechanism. We also had to be extra cautious about message granularity used in action definitions to achieve better performance.

DC-SSSP is an algorithm which generates large number of messages. During this exercise our primary observation was lack of HPX ability to handle large number of small messages. Even though we implemented application level coalescing in 3 strategies it was not sufficient to obtain performance results comparable to AM++ based PBGL2 performance (even in smaller scales).

Further many of the runtime features offered by AM++ were not offered by HPX or those features were hidden using HPX API. We feel that we created an AM++ like layer on top of HPX with coalescing, termination and reductions (which could have implemented), etc. Figure 14 shows the situation diagrammatically.

From the experience gathered so far we feel that HPX is not carefully designed to execute unordered algorithm such as DC-SSSP. Mainly due to the lack of proper message handling, ability to interact with termination strategies. But performance wise it is hard to come to a proper conclusion until we resolve 1315 and collect results for higher scale graphs. Therefore in our future work the primary goal is to resolve 1315 with the help from HPX members. HPX rich set of functionalities allow us implement same algorithm in multiple ways. In this paper we selected only one strategy. We would like to experiment other possible methods (e.g:- data flows) to implement the same algorithm and compare performance.

8. ACKNOWLEDGEMENTS

I would like to thank all the members of STE||AR-GROUP who helped with guidance and advices. Specially, I would like to thank Hartmut Kaiser and Thomas Heller from HPX group in LSU who answered all my questions without hesitation and without a delay. Also I would like to thank Marcin Zaleswski for all the conceptual level advices. Last but not least, I would first like to thank my Ph.D advisor, Andrew Lumsdaine for his input.

This research was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU was also supported in part by Lilly Endowment, Inc.

9. REFERENCES

[1] M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling. An application driven analysis of the parallex execution model. *arXiv preprint arXiv:1109.5201*, 2011.

[2] B. Dawes, D. Abrahams, and R. Rivera. Boost c++ libraries. *URL http://www. boost. org*, 35:36, 2009.

[3] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-source shortest paths with the parallel boost graph library. *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ*, pages 219–248, 2006.

[4] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6. IEEE, 2007.

[5] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. Unordered: A Comparison of Parallelism and Work-Efficiency in Irregular Algorithms. *ACM SIGPLAN Notices*, 46(8):3–12, 2011.

[6] H. Kaiser, M. Brodowicz, and T. Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

[7] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. Hpx–a task based programming model in a global address space.

[8] F. Mattern. Algorithms for distributed termination detection. *Distributed computing*, 2(3):161–175, 1987.

[9] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray UserâĂŹs Group (CUG)*, 2010.

[10] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The Tao of Parallelism in Algorithms. *ACM SIGPLAN Notices*, 46(6):12–25, 2011.

[11] D. W. Wall. Messages as active agents. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 34–39. ACM, 1982.

[12] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Am++: A generalized active message framework. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 401–410, New York, NY, USA, 2010. ACM.

[13] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: A programming model for highly parallel fine-grained data-driven computations. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 305–306, New York, NY, USA, 2011. ACM.

[14] M. Zalewski, T. A. Kanewala, J. S. Firoz, and A. Lumsdaine. Distributed control: priority scheduling for single source shortest paths without synchronization. In *Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, pages 17–24. IEEE Press, 2014.