FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Implementation of Data Flow for the ParalleX Execution Model

Thomas Heller

Master's Thesis

Implementation of Data Flow for the ParalleX Execution Model

Thomas Heller

Master's Thesis

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dr. K. Iglberger
Bearbeitungszeitraum:	undetermined

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 16. April 2012 .....

Abstract

In the prospect of the upcoming exascale era with millions of execution units, the question of how to deal with this level of parallelism efficiently is of time-critical relevance. State-of-the-Art parallelization techniques such as OpenMP and MPI are not guaranteed to solve the expected problems of starvation, growing latencies, and contention. On the other hand, new parallelization paradigms promise to efficiently hide latencies and contain starvation and contention.

In this thesis an insight into ParalleX, a new, experimental parallel execution model, is given. ParalleX defines semantics and modes of operations for parallel applications ranging from large heterogeneous cluster to shared memory systems. Despite the discussion of mechanisms for distributed as well as shared memory systems, this thesis will be restricted to implementations and performance measurements on shared memory machines. The focus of this thesis lies in the implementation of a dynamic Data Flow processor within the ParalleX execution model, which presents a novel parallelization strategy. The viability of this approach is evaluated for a standard stencil-based problem, the Jacobi Method. Two applications of this algorithm will be discussed, one based on a regular and another based on an irregular grid. In order to evaluate the developed mechanisms, implementations with HPX (High Performance ParalleX), the first feature-complete, open-source implementation of ParalleX, and OpenMP, the quasi-standard for programming paradigm for shared memory machines, are compared. The performance of these applications will be evaluated on a multi-socket NUMA node.

Acknowledgments

I would like to thank Dr. Hartmut Kaiser from the Center of Computation and Technology at the Louisiana State University for making it possible to conduct the research abroad. I would like to thank my advisor from the Friedrich-Alexander University of Erlangen Nuremberg (FAU), Dr. Klaus Iglberger, for his support. Additionally I would like to thank Prof. Dr. Ulrich Rüde from FAU to offer this thesis. I acknowledge all the various other people who supported me in writing this thesis. Last but not least, my graditute goes to my wife Steffi, who gave me all the moral support to make this thesis happening.

Contents

List of Figures

List of Listings

1. Introduction

High Performance Computing (HPC) is currently undergoing major changes, provoked by the increasing challenges of programming and managing increasingly heterogeneous multicore architectures and large scale systems. Estimates show that at the end of this decade Exaflops computing systems consisting of hundreds of millions of cores and exposing billion-way parallelism may emerge. This thesis describes an experimental execution model, ParalleX, that addresses these challenges through changes in the fundamental model of parallel computation from that of the communicating sequential processes [1] (e.g. MPI) to an innovative synthesis of concepts involving message-driven work-queue execution in the context of a global address space.

The focus of this thesis lies in the description of a Data Flow extension for the High Performance ParalleX (HPX) runtime system [2]. HPX is the first open source implementation of the concepts of the ParalleX execution model for conventional systems (SMPs and commodity clusters).

As an application for the developed Data Flow extension, implementations of the Jacobi Method for shared memory systems is presented. One for a regular and one for an irregular grid. In order to evaluate the viability of this approach, implementations of the same algorithm with OpenMP [3] will be presented. OpenMP is the de-facto standard for shared memory systems, and exhibits good performance behavior for the discussed applications. This allows the comparison and analysis of this novel and experimental approach against a well-known, highly optimized solution. The benchmarks will be conducted on a multi-socket NUMA node with a total of 48 cores. Unfortunately, the scaling behavior on more cores could not be evaluated due to the lack of appropriate hardware.

The remainder of this thesis is structured as following: Chapter 2 gives an introduction to the ParalleX execution model. Chapter 3 gives an insight in the ParalleX implementation, the High Performance ParalleX runtime system. Chapter 4 deals with the implementation of Data Flow within the HPX runtime system. Chapter 5 introduces the Jacobi Method, describes the examined applications and provides a detailed explanations of the reference OpenMP and the novel HPX implementation. Chapter 6 provides an analysis of the performance behavior of the implementations discussed in the previous sections, and Chapter 7 concludes the thesis and provides suggestions for future work.

2. The ParalleX Execution Model

This section discusses the specific elements of the ParalleX execution model and derives the need for such a novel approach.

In the history of HPC, there have been different phases, which are characterized by an improvement in technology and a specific programming paradigm. These phases include vector computing, SIMD arrays and CSP. With upcoming new technology we are reaching massive multicore structures and trends suggest we reach the exascale era by the end of this decade comprising hundreds of millions of cores and multi-billion way parallelism [4]. Managing these massively concurrent systems efficiently will be one of the challenges of this decade.

Parallel efficiency is expressed by how much speedup a specific program is able to achieve, in other words how well it scales. Generally, two forms of scaling exist, strong scaling and weak scaling. Strong scaling is the time variance for a fixed problem size when adding more and more parallel resources. Weak scaling is characterized by keeping the problem size per processor fixed. Both are important metrics for parallel programs [5].

In order to utilize as much parallelism as possible an application has to support both strong and weak scaling, which requires a high percentage of parallel work executed. This implies, for the best case if a application is executed with N processor, it either runs N times faster or can handle N times more data. However, this is merely a theoretical limit. In the general case, scalability is limited by current hardware architectures and programming models. This can be summarized by the term *SLOW* [4]:

> **S**tarvation: Insufficient concurrent work to maintain high utilization of resources
>
> **L**atency: Time-distance delay of remote resource access and services
>
> **O**verhead: Work for management of parallel actions and resources on critical path which is not necessary in sequential variant
>
> **W**aiting for contention: Delays due to lack of availability of oversubscribed shared resource

In order to overcome these impeding factors, ParalleX tries to improve efficiency by reducing average synchronization and scheduling overhead, improve utilization through asynchrony of workflow, and employ adaptive scheduling and routing to mitigate contention (e.g., memory bank conflicts). Scalability will be increased, at least for certain classes of problems, through data directed computing using message-driven computation and lightweight synchronization mechanisms that will exploit the parallelism intrinsic to dynamic directed graphs through their meta-data. As a consequence, sustained performance will be improved both in absolute terms through extended scalability for those applications currently constrained, and in relative terms due to enhanced efficiency achieved. Finally, power reductions will be achieved by reducing extraneous calculations and data movements. By combining already existing ideas into a new

set of governing principles ParalleX is defining the semantics of this new model. The key aims of ParalleX are:

a) Expose new forms of program parallelism to increase the total amount of concurrent operations.

b) Reduce overheads to improve efficiency of operations and, in particular, to make effective use of fine-grain parallelism where it should occur (this includes, where possible, the elimination of global barriers).

c) Facilitate the use of dynamic methods of resource management and task scheduling to exploit runtime information about the execution state of the application and permit continuing adaptive control for best causal operation.

ParalleX is solidly rooted in the governing principles discussed in the following sections. Section 2.1 explains the preference of hiding latencies instead of latency avoidance. Fine-grain parallelism instead of Heavy-weight threads will be the focus of Section 2.2. Replacing global barriers with constrained based synchronization is the topic of Section 2.3. Section 2.4 will embrace adaptive locality control instead of static data distribution. The preference of moving work to data instead of moving data to work is the focus of Section 2.5. The chapter will be closed with the Section 2.6 dealing with the favor for message driven computation over message passing.

2.1. Latency Hiding

While it is impossible to design systems with zero latency, development of recent and past systems is going into optimizations that target the minimization of these latencies. One example for this is the development of InfiBand, which is a low latency, high-speed network technology [6]. Other examples can be found in all modern processors, which implement memory hierarchies [7].

These existing latencies are only slowing down applications whenever the need to wait for a resource to complete the operation occurs. As such, hiding these latencies through asynchronous operations, which allows the idle-time to be overlapped with other, possibly unrelated, work. There are modern systems which already implement similar techniques. For example asynchronous APIs to program web services (such as NodeJS [8]), pipelined instruction execution in processor cores, asynchronous receive and send operations in MPI [9], and many more.

Enabling and advertising asynchronous operations throughout the whole system stack is what ParalleX proposes to make latency hiding an intrinsic concept of operation in the programming model.

2.2. Fine Grained Parallelism

By demanding latency hiding as discussed above, the ideal mode of operation would be schedule asynchronous operations which might be very short-living, for example fetching the contents of a memory cell. To support this the need for very lightweight threads with extremely short context switching times arises (optimally in the range of a few cycles). Unfortunately, this is

not possible with today's architectures. However, for conventional systems, we can increase the granularity of threads by decreasing the overhead of context switches and therefore better utilize our available resources and increasing parallel efficiency.

For today's architectures there already exist a few libraries providing exactly this type of functionality: non-preemptive, task-queue based parallelization solutions. Examples are Intel's TBB [10], Microsoft's PPL [11], Cilk Plus [12], and many others. Of course, HPX implements its own thread management, which will be outlined in Section 3.1. The possibility to suspend a current task if some preconditions for its execution are not met (such as waiting for I/O or the result of a different task), seamlessly switching to any other task which can continue, and to reschedule the initial task after the required result has been calculated, makes the implementation of latency hiding almost trivial. On the contrary, OpenMP [3], which is used as a baseline comparison (see Section 5.1.1 and Section 5.2.1), is based on data-parallelism. However, the most recent specification adds supports for tasked-based parallelism.

2.3. Constrained Based Synchronization

The two previous sections describe the basic ingredients for the principle parallel execution of a program on a single multiprocessor machine in the ParalleX model. However, the means of organizing and synchronizing different threads of execution has not yet been discussed.

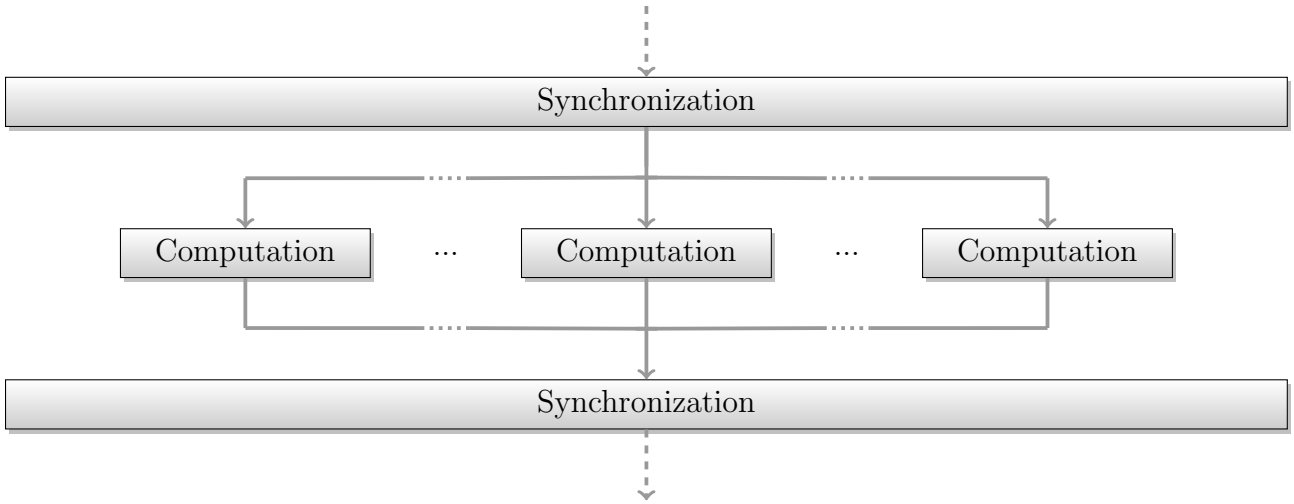


Figure 2.1.: A typical computation flow of a algorithm with global barriers.

When looking at the two dominating programming models for either shared memory systems (OpenMP) or distributed memory systems (MPI) and how they handle synchronization between threads or processes, the most frequent technique used is global barriers. Figure 2.1 shows a graphical representation of how a computation with global barriers looks like. In OpenMP for instance, an implicit global barrier is inserted after each loop parallelized as the system synchronizes the threads used to execute the different iterations in parallel. In MPI each of the communication steps imposes an explicit barrier onto the execution flow as (often all) nodes

have to be synchronized. This programming model proves to be perfect for algorithms that have uniform workload for the different parallel tasks and expose embarrassingly parallel properties when performing the task. As soon as parallelism gets too fine-grain, or synchronization in each individual parallel section is required, global barriers become inefficient. Additionally, once the execution times of the different computations varies, the parallel work is bound by the longest executing task.

A closer analysis of a couple of key algorithms used in science applications reveals that these global barriers are not always necessary [13]. In many cases it is sufficient to synchronize a small subset of the threads. Any operation should proceed whenever the preconditions for its execution are met, and only those. Usually there is no need to wait for iterations of a loop to finish before you could continue calculating other things, all you need is to have those iterations done which were producing the required results for a particular next operation.

The solution ParalleX offers is to provide constraint based synchronization primitives, the so called Local Control Objects (LCOs) [2][4]. A LCO is an abstraction for different responsibilities in the ParalleX model. Those range from synchronization of concurrent data access that would lead to race conditions otherwise, to the event-driven creation and suspension of threads of control. In essence, LCOs are possibly user defined entities that allow event-driven creation or reactivation of threads. They are an abstraction of a multitude of different functionalities for event driven thread creation, protection of data structures and automatic on-the-fly scheduling of work. The following models are mentioned in the ParalleX model:

- Semaphores
- Mutex
- Future [14][15]
- Data Flow [16]

Section 3.2 will provide a more in-depth discussion of LCOs by providing an overview of the possible implementation with HPX. Many HPX applications, including the Jacobi Method applications detailed here, utilize Local Control Objects (LCOs) to simplify parallelization and synchronization.

2.4. Adaptive Locality Control

In order to efficiently address starvation and contention on a distributed memory machine, the need for an global address space arises. With MPI, which is today's prevalent programming model for distributed memory machines, it is the programmers responsibility to decompose the data on all the nodes the application is running and pass data through messages. Implementing an adaptive, dynamic data distribution scheme with MPI is very tedious and difficult.

An attempt to overcome this shortcoming was the development of PGAS (Partitioned Global Address Space). A couple of specialized languages and programming environments based on PGAS (Partitioned Global Address Space) have been designed to overcome this limitation, such as Chapel [17], X10 [18], UPC [19] and Co-Array Fortran [20]. However all systems based on PGAS rely on static data distribution. This works fine as long as such a static data distribution does not result in heterogeneous workload distributions or other resource utilization imbalances.

In a distributed system these imbalances can be mitigated by migrating part of the application data to different localities (nodes). One of the first attempts to solve this and related problems was the Linda coordination language [21].

To fight these limitations, the ParalleX model defines the Active Global Address Space (AGAS). It incorporates the notions of a global, possibly distributed, uniform address space and adds the capability of data migration to flexibly support dynamic and adaptive global locality control.

2.5. Moving Work to Data

When it comes to running applications in a distributed memory machine, it is inevitable to transfer bytes from one part of the system to another. It seems obvious that the size of these messages should be minimized. In the current standard MPI model, the data is exchanged from one compute node to another such that a process running on that node is able to crunch the data. The larger the amount of data gets that has to be transferred back and forth, the more time the application has to spend communicating. This leads to a decrease in utilizing the precious parallel compute resources, and therefore has a negative impact on the strong scaling characteristics of the application.

On the other hand, encoding a certain operation which is to be executed on a specific piece of data often leads to smaller messages and as such to less network traffic. For that purpose, the ParalleX model describes the notion of parcels [2]. Parcels are an advancement of Active Messages [22]. They encode a globally unique identifier provided by AGAS (see Section 2.4), which represents the object on which the action is to be executed, and an additional workload, which usually contains arguments for the aforementioned action. Additionally every parcel carries a continuation, which is used to chain multiple operations.

2.6. Message Driven Computation

ParalleX adopts a message-driven approach. The main entities communicating in a ParalleX system are localities - sets of resources with a bounded, guaranteed response time for operations (currently, this is a node in a conventional cluster). Message-driven communication is one-sided and asynchronous. The receiver does not explicitly wait to receive data, but instead reacts asynchronously to incoming messages. This asynchrony additionally emphasizes the claim to hide latency. Other systems like Charm++ [23] are adapting Active Messages and proving the validity of this concept.

3. The High Performance ParalleX Runtime System

High Performance ParalleX (HPX [2][24][25]) is the first open-source implementation of the ParalleX execution model. HPX is a state-of-the-art runtime system developed for conventional architectures and, currently, Linux and Windows-based systems; e.g. large Non Uniform Memory Access (NUMA) machines and clusters. Strict adherence to Standard C++11 and the utilization of the Boost C++ Libraries [26] make HPX both portable and highly optimized. It is modular, feature-complete, and designed for optimal performance. This modular framework facilitates simple compile or runtime configuration and minimizes the runtime footprint. HPX supports both dynamically loaded modules and static pre-binding at link time. The design of HPX as an implementation of ParalleX tries to overcome conventional limitations outlined in Chapter 2.

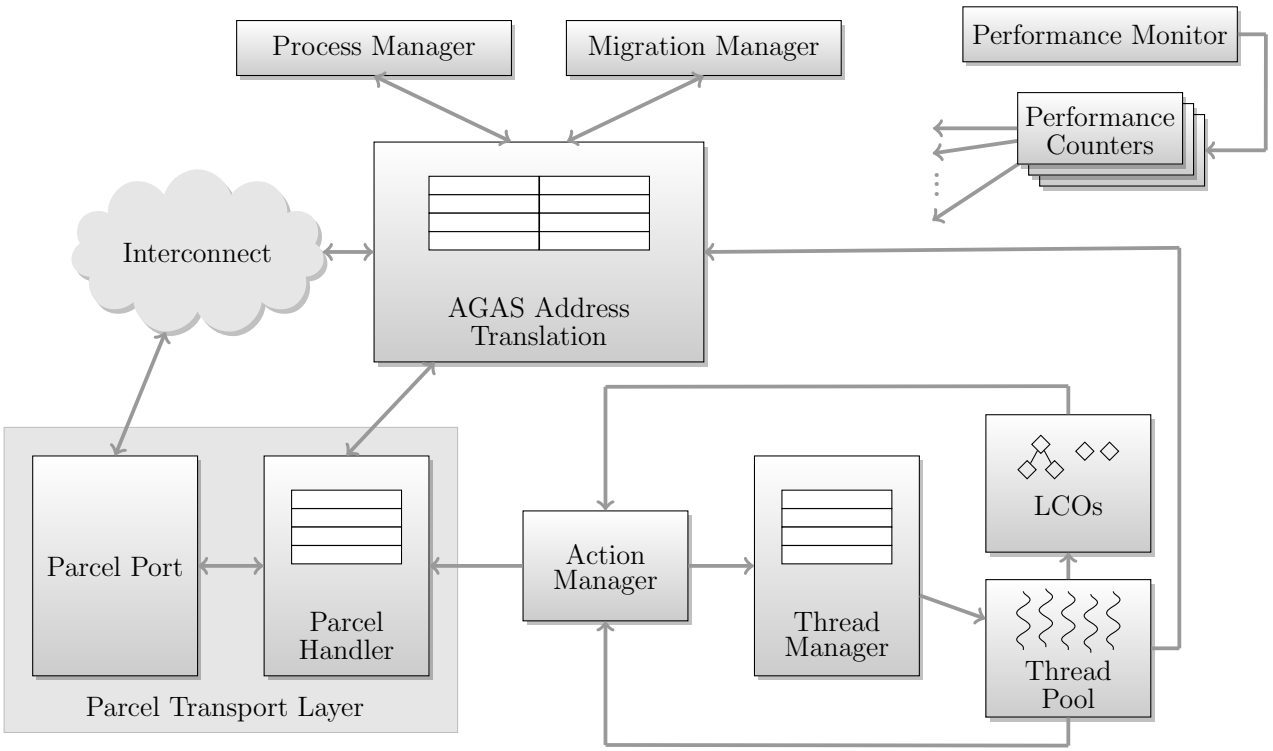


Figure 3.1.: Architecture of the HPX runtime system. HPX implements the supporting functionality for all of the elements needed for the ParalleX model: Parcels (parcel-port and parcel-handlers), HPX-threads (thread-manager), LCOs, AGAS, HPX-processes, performance counters and a means of integrating application specific components.

The remainder of this chapter will explain the key parts of the HPX runtime system. The current implementation of HPX supports all of the key ParalleX paradigms: Parcels, PX-threads, Local Control Objects (LCOs), the Active Global Address Space (AGAS), and PX-processes (see Fig. 3.1). Section 3.1 will focus on the HPX thread subsystem. Section 3.2 briefly introduces the implementation of LCOs. Although not relevant to the applications presented in this thesis a short overview of the Parcel Transport layer will be given in Section 3.3 and the AGAS service will be outlined in Section 3.4. The Performance Monitor, Process Manager and Migration Manager will not be discussed.

3.1. Threads and their Management

As outlined in Section 2.2 the efficient management of lightweight threads is one of the most important parts in a ParalleX implementation and thus the HPX runtime system. It is responsible for managing millions of thousands of threads. HPX makes that possible by implementing a lightweight user level threading module. This module implements a $M : N$ mapping of threads, where M is the number of user level threads and N the number of available Operating System (OS) threads. Due to the inherent characteristic of being implemented in user space, HPX threads are cooperative, meaning they need to voluntarily suspend execution. This is handled by the LCOs, which are discussed in more detail in Section 3.2.

HPX threads are first class objects in the global address space, which would even enable remote management. However, this is avoided due to high costs involved (migrating the whole thread context is potentially very expensive). Instead, migration of work is realized via continuation by sending a parcel (see Section 3.3).

Due to the very central role of thread management, the performance is of course crucial. One way the thread manager achieves high performance is by avoiding calls into the underlying OS kernel completely. This has the effect that the HPX application can run for the whole time slice provided by the OS scheduler, and that HPX threads avoid expensive calls into the OS kernel. HPX uses one thread queue per OS thread. This queue is implemented with a lock-free FIFO queue algorithm. Once one OS-thread local queue is empty, thread items from other OS-thread queues will be stolen.

Figure 3.2 shows a comparison of HPX to other user level thread libraries. Qthreads [28] is one of the possible lightweight user level threads implementations in Chapel [29]. TBB stands for Intel Thread Building Blocks [10], which offers a rich and complete approach to express parallelism in a C++ program. SWARM is a commercial ParalleX implementation [30]. The benchmark consists of running 500000 independent threads doing arbitrary work [27]. The graphs show that HPX is able to outperform the competing solutions once the number of the OS-threads exceeds 12. When using less then 12 threads Qthreads, TBB and SWARM show similar performance behavior while HPX is far behind. However, using up to 12 OS-threads, HPX is able to increase the number of threads executed per second, while the other solutions are massively derogating. Adding more OS-threads shows a decrease for all solutions, but HPX is still performing best. This performance allows HPX to fulfill the premise of the ParalleX execution model to allow for fine-grain parallelism on conventional systems.

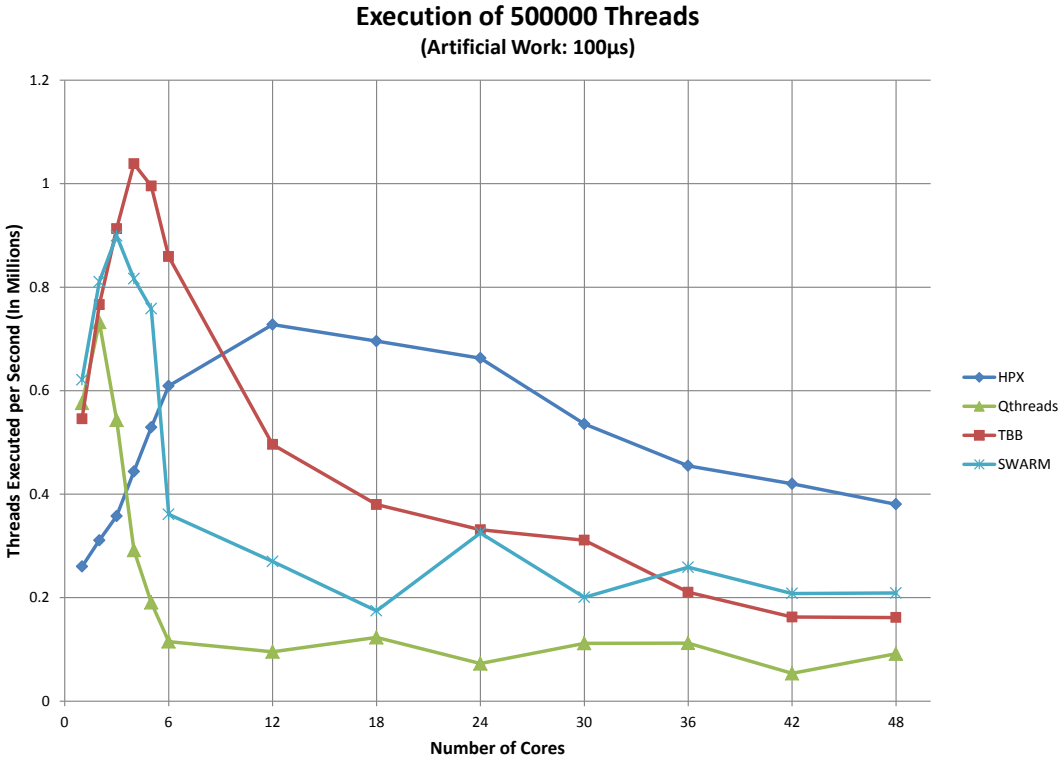


Figure 3.2.: The number of millions of threads executed per second. The graph shows that HPX is able to outperform the other investigated libraries once more than 12 cores are in use [27].

3.2. Local Control Objects

As mentioned in the previous section, the synchronization of HPX threads is done via LCOs. LCOs as implemented in HPX are user defined, first class objects living in the global address space. With modern object oriented paradigms, the four basic LCOs (see Section 2.3) can be composed and combined to form all varieties of different objects to control thread state. Chapter 4 will provide a deeper discussion of Data Flow as an LCO.

Listing 3.1 shows the base class definition of an LCO. The `base_lco` class is a `hpx::component`, which means it is a object managed through AGAS (see Section 3.4). A user-defined LCO merely has to overload the pure virtual `set_result` function (Line 4) and implement it.

```
1 template <typename Result>
2 struct base_lco : hpx::component<base_lco<Result> >
3 {
4     virtual void set_result(Result r) = 0;
5
6     HPX_COMPONENT_ACTION(base_lco, set_result, set_result_action);
7 };
```

Listing 3.1: Outline of the LCO base class

As example LCO, futures will be discussed in this section. Futures hold a promise to a value that will be computed by a (possibly remote) action. Figure 3.3 depicts a possible control flow of a future invocation. As soon as a future action is created, a corresponding work

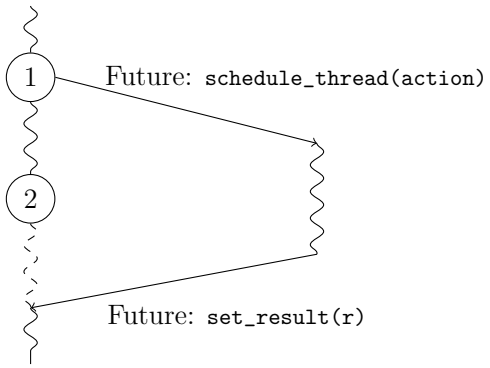


Figure 3.3.: Possible Control Flow for a future LCO. At point 1 the future is created, the thread continues to execute. Another thread gets created and eventually scheduled and run in the background. At 2, the calling thread is requesting the result. Since the thread invoking the action hasn't delivered the result yet, the thread gets suspended (dashed line) until the result is set.

item will be enqueued in the localities thread manager (in case of a remote action, a parcel is created; see Section 3.3 for more information on parcels). Eventually, the work item is dispatched and the action is executed. The call to the get function suspends the calling thread until the future value is eventually executed. Altogether, the future is a perfect example of message-driven computation, enabled by event-driven suspension and rescheduling of HPX threads. In the implementation of a future in terms of the `base_lco`, an action gets scheduled as a lightweight HPX-thread containing the `set_result_action` (Line 6 in Listing 3.1) as a continuation. Additionally, threads, which are waiting on the result are queued, until the `set_result_action` continuation is triggered. In the overloaded `set_result` function, the result is stored inside the future, and the waiting threads are woken up. This implementation forms the basis to realize constrained based synchronization (see Section 2.3) and asynchronous message driven computation (see Section 2.6).

3.3. Parcel Transport

Whenever a HPX application needs to pass messages between localities, parcels are involved (see Section 2.5).

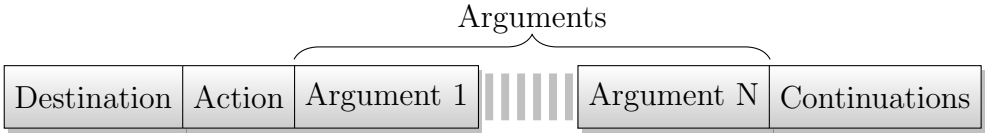


Figure 3.4.: The structure of a HPX Parcel

Figure 3.4 shows the structure of a parcel in HPX, they consist of 5 parts: the global identifier for the destination address (resolved by AGAS, see Section 3.4), the action to be performed, the arguments to pass to the invoked action and a list of continuations to trigger after the action is executed. A continuation is implemented as a LCO (see Section 3.2) which enables the multiple ways of chaining computations. The Data Flow implementation shown in Section 4.2 uses this extensively.

Parcels are executed whenever an action has to be invoked on a remote locality. Parcels enable distributed control flow. As such, they are the semantic equivalent to locally executed threads.

3.4. Active Global Address Space

The Active Global Address Space (AGAS) service currently implemented in HPX represents a 128-bit address space, which spans all involved compute nodes. The AGAS service consists of two naming layers. The primary naming service maps 128-bit unique, global identifiers (GIDs) to a meta-data that can be used to locate an object on a particular locality. The higher level symbolic namespace, maps symbolic names to GIDs. This is the basic foundation, which will enable data migration in the near future. The current AGAS implementation is a centralized service. Research in making it decentralized is being conducted and will be a key point in making the AGAS service scalable over thousands of compute nodes.

4. Data Flow

This chapter discusses the implementation of the Data Flow LCO for the ParalleX execution model. Data Flow was first described as a complete processor [31][32][33] to efficiently describe fine-grain, data-driven parallelism with the help of a tagged-token based system. Attempts on formulating an early processor designs have been done, but have been abandoned due to performance problems. The main driver application was signal processing, which is perfectly suited for a data driven application: The next processing step can only be started once every operation before has been completed.

With the ParalleX execution Model and the HPX implementation, a new possibility in implementing Data Flow semantics is given. In the remaining section, the main principle of a Data Flow processor is explained (Section 4.1), and the implementation with HPX is discussed in Section 4.2.

4.1. Basics

The elementary Data Flow processor as described in [31] needs an elementary system which is able to asynchronously propagate events and trigger certain actions and operators. The system is completely message driven and the different, possibly parallel, parts of a program only communicate with each other by returning a value, which is passed as a parameter to the next computation. This allows for a elegant functional formulation of given problems.

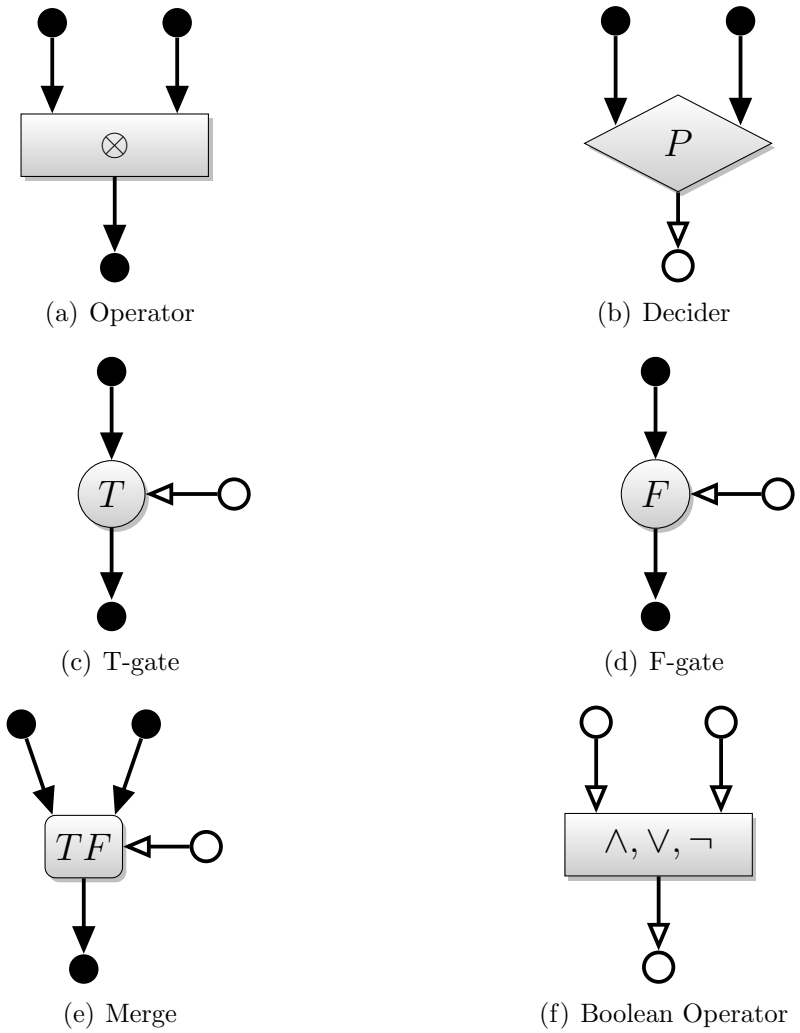
The Data Flow processor therefore is designed to have basic dataflow links and operations (see Figure 4.1 and Figure 4.2). The basic dataflow operations only prolong information:

(a) data link (b) control link

Figure 4.1.: Data Flow links. A distinction is made between data values and control values. Whenever a value arrives on the incoming edge, it will get distributed to the outcoming edges

Whenever a token arrives on the incoming edge, it will be carried on to the outgoing edges.

This can be either data or control values. In order to sufficiently describe the control flow of a generic program. The operations described in Figure 4.2. The solid dots represent data values



(a) Operator (b) Decider

(c) T-gate (d) F-gate

(e) Merge (f) Boolean Operator

Figure 4.2.: Elementary Data Flow Operations. The operation schematics in this figure show the building blocks of the static Data Flow processor

and the hollow circles represent data links. Figure 4.2(a) shows the schematics of an arbitrary operation. Once the input tokens arrive (solid dots), the operation is executed and the output token is triggered. Figure 4.2(b) produces, based on the input tokens and the predicate P, a boolean control token (hollow dots). Figure 4.2(c) and Figure 4.2(d) forward the input data token based on the input control token. The T-gate produces the output on a `true` token, and the F-gate when it gets triggered by a `false` token. The merge gate schematic shown in Figure 4.2(e) selects the input token based on the triggered control token. The boolean operator (Figure 4.2(f)) is used to compose control data values. The operations shown in this section can be used to build static Data Flow programs, which are equivalent in power and expressiveness as regular statement based languages.

4.2. Implementation

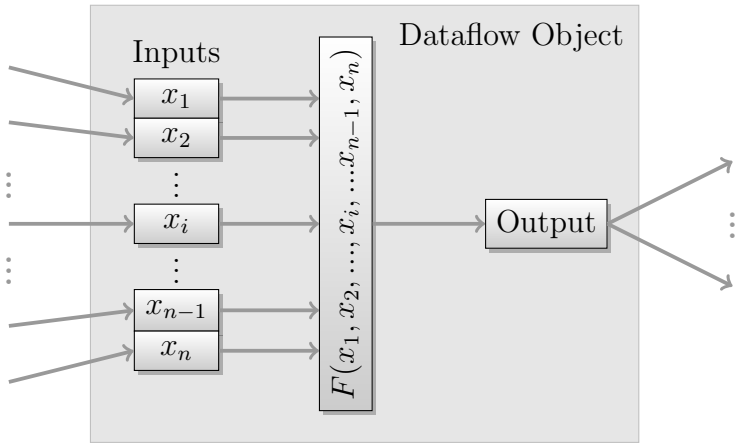


Figure 4.3.: Schematic of a Data Flow object. A Data Flow object encapsulates a (customizable) function $F(x_1, x_2, ..., x_N)$. This function takes N input arguments and produces an output result. The dataflow object is receiving each of the required input arguments separately from different data sources (i.e. other dataflow objects). As soon as the last input argument has been received, the function is scheduled for execution and, after the result has been calculated, it is sent to all objects connected to the output of the Data Flow object

The previous section covers static Data Flow. The HPX implementation in this thesis is a dynamic, at runtime created Data Flow processor. Figure 4.3 shows the schematics of a Data Flow object. It unifies the notion of the data link (see Figure 4.1(a)) and the Data Flow operator. The inputs and output resemble a Data Flow data link, and $F(x_1, x_2, ..., x_N)$ is a Data Flow operation. It acts as a proxy for a result initially not known, because the input arguments required for the computation of the result have not been computed yet. The Data Flow object synchronizes the computation of an output value by deferring its evaluation until all input values are available. Thus a Data Flow object is a very convenient means of constraint-based synchronization of the overall execution.The inherent meta information of the used data structures is leveraged (i.e. the structure of the computational grid) by creating a structurally matching network of Data Flow objects. For this reason the execution of the Data Flow network will always yield the correct result while the existing compute resources are optimally utilized as every particular calculation of any intermediate result will happen 'at its own pace'. The main advantage of such a Data Flow based execution is that it enables minimizing the total synchronization and scheduling overheads.

The actual implementation of the Data Flow LCO is made up of two parts. The first part consists of implementing the trigger for the required inputs and the second the actual execution of the Data Flow action itself. For the Data Flow input trigger two possible options arise. For the first, the similarity between a Data Flow object and a future (see Section 3.2) has to be noted: Both provide a promise to eventually deliver a computed result. With the ability to wait for results a naïve implementation of the trigger, which spawns a thread that immediately blocks until the input eventually is computed, is tempting. However, this approach turns out to be inapplicable since for every input a HPX-thread is created, even though HPX is able to handle millions of threads efficiently, it would create a huge overhead.

```
1  template <typename DataflowSource, typename DataflowObject, int N>
2  struct dataflow_trigger : base_lco<typename DataflowSource::result_type>
3  {
4      typedef typename DataflowSource::result_type result_type;
5
6      dataflow_trigger(DataflowSource source, DataflowObject object)
7          : dataflow_source(source)
8          , dataflow_object(object)
9      {
10         // Connecting the dataflow trigger to the input Data Flow Object.
11         // This way, the set_result function of the trigger will be
12         // executed in the set_result continuation of this input.
13         dataflow_source.connect(this->id());
14     }
15
16     void set_result(result_type r)
17     {
18         // When the set_result function for this input is triggered, the
19         // Data Flow object associated with this trigger will get notified.
20         dataflow_object.template set_input<N>(r);
21     }
22
23     DataflowSource dataflow_source;
24     DataflowObject dataflow_object;
25 };
```

Listing 4.1: Outline of the Data Flow Trigger Implementation

For the special purpose of handling Data Flow triggers a LCO is defined (see Listing 4.1). It is tightly coupled with the actual Data Flow object, for which an implementation outline can be seen in Listing 4.2. With this implementation, an efficient implementation has been achieved. In contrast to the naïve implementation idea, the created Data Flow Objects passively wait until all inputs are available. This automatically happens through the mechanisms described in Chapter 3. One of the important differences between Data Flow and future computation can now be observed: While with futures, the applications need to actively pull to receive the asynchronously computed results, Data Flow computation can happen completely automatically. Results are being pushed forward and literally flow through the dynamically created execution tree, allowing the HPX runtime system to perform further runtime optimizations.

As a simple example for a dynamically created Data Flow execution tree, Listing 4.3 shows an implementation of a Fibonacci number calculation using the Data Flow LCO. In order to overcome the missing Data Flow operations (decider, T-gate, F-gate, merge and boolean operator), the Data Flow graph is constructed dynamically using regular C++ control flow statements. Lines 1 to 5 define an identity function, which is used at the end of the recursion and Lines 7 to 11 define a function, which is used to add two numbers, which are recursively calculated. `dataflow_base` is a simple wrapper class holding the gid of the created Data Flow objects. The remaining code constructs the Data Flow graph: If the input number is less than `2`, the identity Data Flow function is created, otherwise the function is recursively called with `n-1` and `n-2`.

```
template <typename Action>
struct dataflow : base_lco<typename Action::result_type>
{
    typedef typename DataflowSource::result_type result_type;

    template <typename A...>
    dataflow(id_type id, A ... a)
        : action_id(id)
        // Expanding the received parameters and creating dataflow_trigger
        // objects managed by AGAS.
        , trigger(
            hpx::create<dataflow_trigger<A, dataflow, N> >(a, *this)...
        )
    {}

    template <int N, typename T>
    void set_input(T t)
    {
        // Saving the received inputs in order to forward them once all
        // inputs are available.
        if(all_triggers_completed<N>(t))
        {
            // After all inputs have been set. We asynchronously start the
            // requested action, passing the id of this object as the
            // continuation LCO to async_c. Thus, set_result will get called
            // whenever the action finished execution
            async_c<Action>(this->id(), action_id, inputs);
        }
    }

    void set_result(result_type r)
    {
        // When this function is executed, it means that the action finished
        // execution. The result is then forwarded to the connected targets.
        for(hpx::id_type target, targets)
        {
            async<base_lco<result_type>::set_result_action>(target, r);
        }
    }

    void connect(hpx::id_type target)
    {
        // Add the passed target to the vector of targets.
        targets.push_back(target);
    }
    HPX_COMPONENT_ACTION(dataflow, connect, connect_action);
    id_type action_id;
    std::vector<hpx::id_type> trigger;
    std::vector<hpx::id_type> targets;
    // ... remaining definitions for saved inputs and functions for checking
    // of all triggers have been completed
};
```

Listing 4.2: Outline of the Data Flow Object Implementation

Figure 4.4 illustrates the complete Data Flow graph in the semantics outlined in Section 4.1 for a call to `fib(5)`. The roots of this graph are the end of the recursion and the leaf represents the result of the computation. Once the Data Flow graph is constructed, the different dependencies are implicitly present in the graph and once the data input triggered, the operation can be executed. It is easy to see that the nodes on the same level can be processed embarrassingly parallel.

```
1  int identity(int i)
2  {
3      return i;
4  }
5  HPX_PLAIN_ACTION(identity, identity_action);
6
7  int add(int a, int b)
8  {
9      return a + b;
10 }
11 HPX_PLAIN_ACTION(add, add_action);
12
13 hpx::lcos::dataflow_base<int> fib(int n)
14 {
15     if(n < 2)
16     {
17         // return n;
18         return
19             hpx::lcos::dataflow<identity_action>(
20                 hpx::find_here()
21               , n
22             );
23     }
24
25     // return fib(n-1) + fib(n-2);
26     return
27         hpx::lcos::dataflow<add_action>(
28             hpx::find_here()
29           , fib(n-1)
30           , fib(n-2)
31         );
32 }
```

Listing 4.3: Data Flow Example: Fibonacci Number Calculation

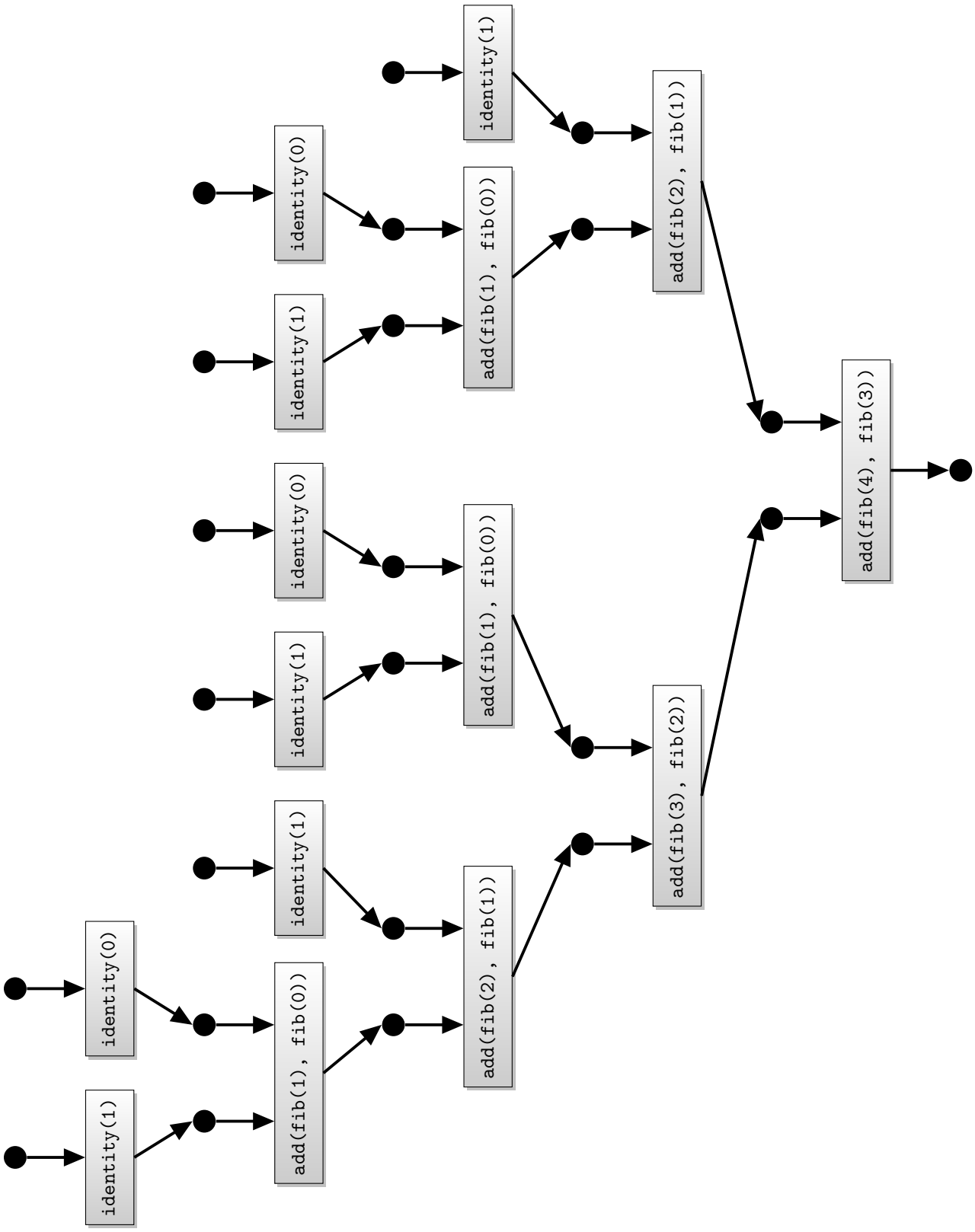


Figure 4.4.: Example Data Flow schematics for the 5th Fibonacci number

5. Application: Jacobi Method

The Jacobi Method is a numerical stationary iterative method. It is one example of a stencil-based iterative method and is used to solve a linear system of equations given by:

$$Ax = b,\ A \in \mathbb{R}^{NxN},\ x \text{ and } b \in \mathbb{R}^N \tag{5.1}$$

The Jacobi Method is a well-known numerical algorithm with well known parallelization techniques for OpenMP. OpenMP is the de-facto standard for implementing stencil-based codes on shared memory systems, and, for the algorithm discussed in this Chapter, exposes almost perfect strong and weak scaling behavior. Thus the performance of the Data Flow LCO can be evaluated and compared (see Chapter 4). Although the Jacobi Method is converging slower than other iterative methods, the easy parallelization is an advantage. The Jacobi Method is also used as a smoother and preconditioner for more advanced numerical algorithms like Multigrid or Krylov-Space methods. However, the Jacobi Method was not chosen based on its numeric properties but because of its well known performance characteristics.

In order to solve Equation 5.1 a fixed point iteration is applied in the following way:

$$\begin{aligned} Ax &= b, \text{ where } A = M - N && (5.2)\\ \Leftrightarrow Mx &= Nx + b && (5.3)\\ \Leftrightarrow x &= M^{-1}Nx + M^{-1}b && (5.4)\\ & && (5.5) \end{aligned}$$

So the algorithm can be formulate as follows:

$$\begin{cases} x^{(0)} \text{ given} \\ x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b \text{ else} \end{cases} \tag{5.6}$$

M is the diagonal part of the Matrix, and N everything but the diagonal. This fix point iteration converges as long as A is symmetric and positive definite [34].

This chapter deals with two applications of this algorithm. Section 5.1 discusses the algorithm for solving a uniformly discretized diffusion equation in 2D, a implementation for OpenMP and HPX based on the Data Flow LCO is presented. Section 5.2 outlines the application of the Jacobi Method to a highly irregular sparse grid obtained through a Finite-Element analysis. Again, implementations for OpenMP and HPX are presented.

5.1. Uniform Workload

The first example application of the Jacobi Method in this thesis is derived from solving the diffusion equation for a scalar function $u(x, y)$. The first problem is a uniform grid, which is derived by discretizing the diffusion equation for a scalar function $u(x, y)$

$$\triangle u = f$$

on a rectangular grid with Dirichlet boundary conditions. After discretizing the differential operator, the five-point stencil can be derived (without the loss of genericity, this thesis is restricted to 2D in this thesis and a right-hand-side of constant zero) to be used for updating one point:

$$u_{new}(x, y) = \frac{u_{old}(x + 1, y) + u_{old}(x - 1, y)}{4} + \frac{u_{old}(x, y + 1) + u_{old}(x, y - 1)}{4}$$

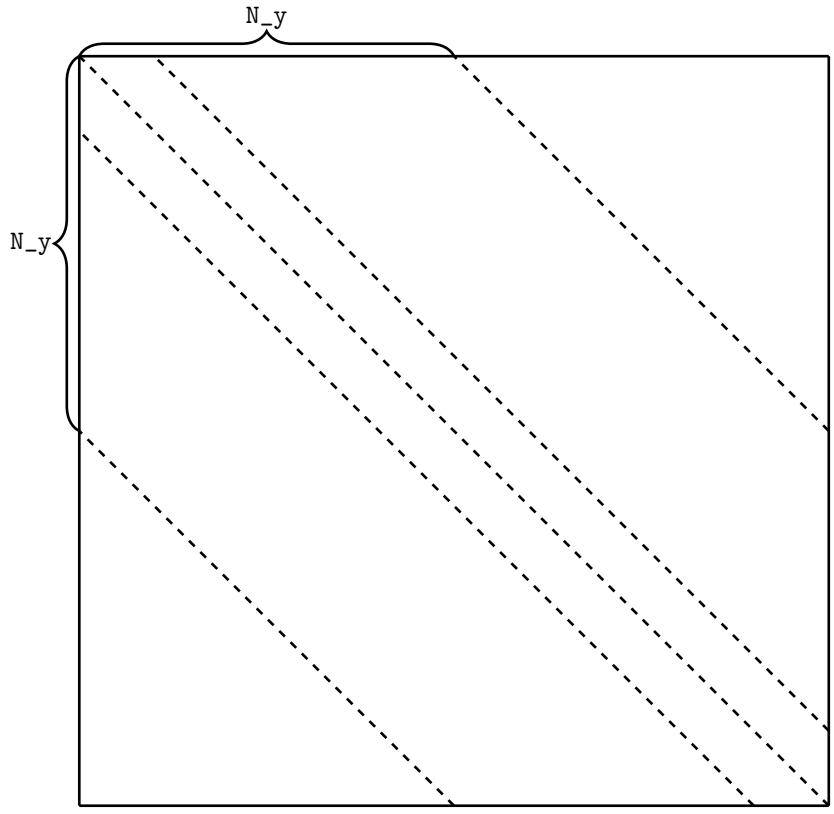


Figure 5.1.: Uniform symmetric Matrix. The structure of the Matrix originating from discretizing the two-dimensional diffusion equation. The entries being non-zero are the diagonal, one of the diagonal below and above and the diagonal starting `N_y` off the diagonal

Figure 5.1 shows the structure of the matrix, which can be derived from that stencil. Due to the uniform nature of that stencil, the matrix never gets explicitly formulated. Listing 5.1 shows an example implementation in C++ for that particular update, conforming to Equa-

tion 5.6. The type `grid<double>` is a user defined type to represent the discretized solution. It is a straightforward implementation of a linearized two-dimensional data structure. The implementation can be found in Appendix A.1 Listing A.1.

```
typedef pair<size_t, size_t> range_t;
void jacobi_kernel(
    vector<grid<double> > & u
  , range_t x, range_t y
  , size_t dst, size_t src)
{
  for(size_t j = y.first; j < y.second; ++j)
  {
    for(size_t i = x.first; i < x.second; ++i)
    {
        u[dst](i,j)
         = (u[src](i+1,j) + u[src](i-1,j)
            + u[src](i,j+1) + u[src](i,j-1))
           * 0.25
    }
  }
}
```

Listing 5.1: Example Kernel for the Uniform Jacobi Method

This implementation of the kernel counts 4 double precision floating point operations (three additions and one multiplication). It can be assumed that the cache is large enough, such that three rows of the grid are resident in the cache. Thus, one memory transfer to write the result back into main memory, and two for loading the cache-lines for the top and bottom elements have to be done. Furthermore the destination grid element, and the elements in the x direction are cache resident and can be read. According to that assumption, three memory transfers have to be done for every iteration. This implies that this specific algorithm is memory bound. These considerations act as a baseline for the performance considerations in Section 6.1. In this example implementation of the kernel Low-Level optimizations are not taken into account. Those optimizations are heavily discussed in the literature [35], and have no effect on the outcome of this comparison. Additionally, no convergence test is performed in order to have a fixed number of iterations for both implementations discussed.

5.1.1. OpenMP Implementation

OpenMP is the dominant shared-memory programming standard today. Developed since 1997 as a joint effort of compiler vendors, OpenMP has been used even prior to the multicore era as a platform independent parallelization tool.

Listing 5.2 shows the OpenMP-parallel implementation of the Jacobi solver introduced in this section. Initially, the two grids are initialized with `1` according to the Dirichlet boundary conditions. The iterate over the grid is done in a block-wise fashion in order to use the cache efficiently. In every inner loop, the function outlined in Listing 5.1 is executed on each single block.

This algorithm uses standard C++ facilities almost exclusively and hints the compiler to create a parallel section with `#pragma omp for`. Due to the nature of the Jacobi method, this algorithm does not contain any data races. It is worth noting that with the directive `#pragma`

`omp for` an implicit global barrier is introduced. This barrier is required to complete the calculation in every thread so that the grids can be swapped before the next iteration starts. However, as the workload of the calculation is equally distributed among each thread, this barrier should have little to no effect. For better comparison, a version with dynamic work scheduling has been compiled for this algorithm.

Since the Jacobi method is a memory-bound algorithm, performance of this implementation is expected to scale well until the maximum memory bandwidth of one CPU is exhausted.

```
1  vector< grid<double> >
2    u( 2, grid<double>(N_x, N_y, 1) );
3  size_t src = 0;
4  size_t dst = 1;
5  for(size_t iter = 0; iter < max_iter; ++iter)
6  {
7  #pragma omp for shared(u) schedule(static)
8    for(size_t y=1; y < N_y-1; y+=block_size)
9    {
10     for(size_t x=1; x < N_x-1; x+=block_size)
11     {
12       jacobi_kernel(
13           u
14         , range_t(x, min(x+block_size, N_x))
15         , range_t(y, min(y+block_size, N_y))
16         , dst, src);
17     }
18   }
19   swap(src, dst);
20 }
```

Listing 5.2: OpenMP - Uniform Jacobi Method

In case of applications that are memory-bound, locality and contention problems can occur on NUMA systems if no special care is taken for the memory initialization. Memory-bound code must be designed to employ proper page placement, for instance by first touch [35]. In order to accommodate both use-cases, an interleaved memory policy is used. That means that pages will be allocated using round robin on the NUMA nodes. This decreases the overall performance for this application, but is the most practical way of placing memory for the non-uniform workload application (see Section 5.2). Additionally, thread migration was prevented by an explicit pinning of all active threads via the Likwid tool suite [36]. Without applying these techniques, OpenMP would fail to scale properly on NUMA architectures.

5.1.2. HPX Implementation

The HPX implementation of the uniform workload eliminates the implicit global barriers introduced by parallelizing the iteration loops using OpenMP. This is achieved by using LCOs as introduced by ParalleX (see Chapter 2) and implemented in HPX (see Chapter 3). In particular the Data Flow LCO (see Chapter 4) is used to ensure a fluid flow of computation.

The Jacobi Method can then be formulated with the help of Data Flow by explicitly specifying the dependencies. In the case of the algorithm outlined above (See Listing 5.1), the constraints are that the elements of the old grid have been computed so that data races can be neglected. In the case of the OpenMP Listing, this has been ensured by the implicit global

barrier (See Listing 5.2). The HPX implementation solves this by explicitly calculating the block-wise dependencies. Figure 5.2 illustrates the dependencies of one grid element.

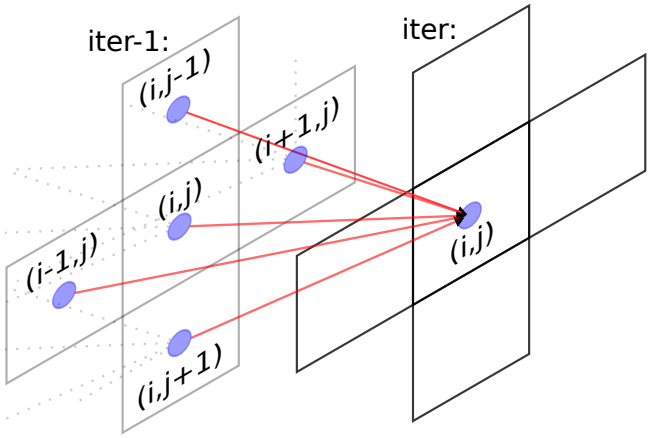


Figure 5.2.: Jacobi Iteration dependencies. As can be seen, a grid element depends upon the neighboring elements and that the previous element computation has been completed.

In Listing 5.3 the block-wise dependency calculation is shown:

- The current item has to be calculated in order to be able to continue (Line 8)
- The left item, iff index is not on the left boundary (Line 10)
- The right item, iff index is not on the right boundary (Line 12)
- The top item, iff index is not on the top boundary (Line 14)
- The bottom item, iff index is not on the bottom boundary (Line 16)

```
1  typedef hpx::lcos::dataflow<void> dataflow_t;
2  vector<dataflow_t> get_deps(
3    grid<dataflow_t> const & deps
4  , size_t x, size_t y
5  , size_t dst, size_t src)
6  {
7    vector<dataflow_t> r;
8    r.push_back(deps(x,y));
9    if(x>0)
10     r.push_back(deps[src](x-1,y));
11   if(x+1<deps.N_x)
12     r.push_back(deps[src](x+1,y));
13   if(y>0)
14     r.push_back(deps[src](x,y-1));
15   if(y+1<deps.N_y)
16     r.push_back(deps[src](x,y+1));
17
18   return r;
19 }
```

Listing 5.3: HPX - Uniform Dependency Calculation

Listing 5.4 shows the full implementation of the Jacobi Method with HPX. It is still 100% C++ Standard compliant, but implements a complete novel approach to parallelism. Instead of hinting the compiler to parallelize the loop, the parallelism is hidden inside the Data Flow LCO. This Data Flow LCO is responsible for activating a new thread once all dependencies have been met (see Listing 5.3). On Line 5 a new object is created and registered with AGAS (see Chapter 3) and wrapped inside a special datatype. The iterations look exactly like in Listing 5.2 but are now used to create the Data Flow tree. At first, of the current iteration (Line 22) need to be determined. The call to `u.apply` (Line 24) creates the Data Flow object. With `hpx::bind` (Line 25) a function object is created that will be executed once the passed dependencies (Line 31) are met. This complies to the "moving work to data" paradigm imposed by the ParalleX execution model (see Chapter 2). Interestingly, whether the object lives on the current system or on a remote location is completely irrelevant for this code to work.

```
1  typedef dataflow_base<void> dataflow_t;
2  typedef vector<grid<double> > vector_t;
3
4  hpx::components::dataflow_object<vector_t>
5    u = new_<vector_t>(hpx::find_here(), 2, grid<double>(N_x, N_y, 1));
6
7  size_t N_x_block = N_x/block_size;
8  size_t N_y_block = N_y/block_size;
9
10 vector<grid<dataflow_t> > deps(2, grid<dataflow_t>(N_x_block, N_y_block));
11
12 size_t dst = 0;
13 size_t src = 1;
14
15 for(size_t iter = 0; iter < max_iter; ++iter)
16 {
17   for(size_t y = 1, y_block = 0; y < N_y-1; y += block_size, ++y_block)
18   {
19     for(size_t x = 1, x_block = 0; x < N_x-1; x += block_size, ++x_block)
20     {
21       vector<dataflow_t>
22         iteration_deps = get_deps(deps, x_block, y_block, dst, src);
23       deps(x_block, y_block) =
24         u.apply(
25             hpx::bind(jacobi_kernel,
26                 hpx::placeholder::_1
27               , range_t(x, min(x+block_size, N_x))
28               , range_t(y, min(y+block_size, N_y))
29               , dst, src
30             )
31           , iteration_deps
32         );
33     }
34   }
35   swap(dst, src);
36 }
37 hpx::wait(deps);
```

Listing 5.4: HPX - Uniform Jacobi Method

It is important to note that after the loops are finished, the system will have already started working in the background. By making these changes the removal of the implicit global barrier imposed by OpenMP has been achieved. It was replaced with constraint based LCOs, which control, with the help of the runtime system, all the introduced parallelism.

Just like the OpenMP implementation, the HPX implementation is also memory bound. Except for pinning the operating system threads to the according cores, no other precautions to accommodate NUMA architectures have been done. Due to the aggressive thread stealing of the HPX thread-manager (see Chapter 3) an optimal NUMA memory placement strategy cannot be found. Although, this thread stealing also implies the possibility to hide latencies induced by inter-node memory traffic, it cannot compensate the intrinsic bandwidth limits of current computing platforms.

5.2. Non-Uniform Workload

The second application of the Jacobi Method is chosen to show how different workloads on an element update influence our implementations. A matrix obtained from a Finite Element Method is chosen. The matrix is obtained from a structural problem discretizing a gas reservoir with tetrahedral Finite Elements [37][38]. The matrix is saved in a compressed row storage format to efficiently traverse it in the Jacobi Method Kernel. The implementation of the sparse matrix data structure can be found. An implementation of the sparse matrix data structure can be found in Appendix A.2 Listing A.2. As the uniform workload application, this problem is memory bound. The resulting stencil is dynamic, and dependent on the number of entries in each row. The chosen matrix has a minimum of one entry per row, and a maximum of 228 entries. The mean are 23.69 entries, with at least one element on the diagonal. The structure of the matrix can be seen in Figure 5.3.

```
1  typedef pair<size_t, size_t> range_t;
2  void jacobi_kernel_nonuniform(crs_matrix<double> const & A
3    , vector<vector<double> > & x, vector<double> const & b
4    , pair<size_t, size_t> range, size_t dst, size_t src
5  )
6  {
7      for(size_t i = range.first; i < range.second; ++i)
8      {
9          double result = b[i];
10         double div = 1.0;
11         const size_t begin = A.row_begin(i);
12         const size_t end = A.row_end(i);
13         for(size_t j = begin; j < end; ++j)
14         {
15             if(j == i) div = div/A.values[j];
16             else result -= A.values[j] * x[src][A.indices[j]];
17         }
18         x[dst][i] = result * div;
19     }
20 }
```

Listing 5.5: Example Kernel for the Non-Uniform Jacobi Method

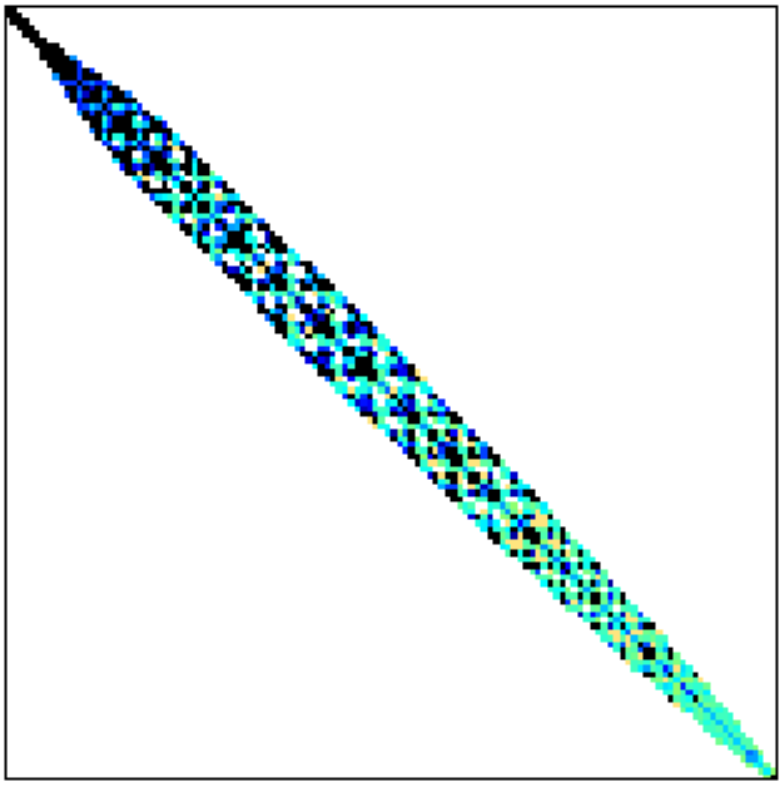


Figure 5.3.: The structure of the Serena Matrix. This figure shows a spy plot of the the Serena Matrix [37]

Based on these properties, it can be assumed that the diagonal element of the matrix, and the element of the source grid are cache resident. Based on these numbers, the mean memory transfers for this problem are 47.38 (2 for the destination grid and the right hand side, 22.69 for the matrix elements and 22.69 for the source grid). These numbers are the baseline for the performance discussion in Section 6.2.

In Listing 5.5 the implementation for the non-uniform jacobi kernel is shown. Instead of the fixed stencil in Listing 5.1, the stencil is now given by the non-zero entries of the current row in the given matrix. This is what determines the non-uniform workload in the different block-wise updates.

5.2.1. OpenMP Implementation

Listing 5.6 shows the OpenMP implementation of the Jacobi solver for the non-uniform problem. It can be observed, that it is almost the same as before (see Listing 5.2). The differences are the explicitly saved matrix and the call to the different kernel. In order for OpenMP to better cope with the dynamic workload, a binary with with dynamic work has been compiled.

```
vector<vector<double> > u(2, x);

high_resolution_timer t;
size_t src = 0;
size_t dst = 1;
t.restart();
for(size_t iter = 0; iter < max_iterations; ++iter)
{
/**
 * Alternatively, dynamic scheduling can be enabled by commenting out
 * the following line and comment the other pragma.
#pragma omp parallel for shared(u) schedule(dynamic)
**/
#pragma omp parallel for shared(u) schedule(static)
    for(int i = 0; i < x.size(); i += block_size)
    {
        jacobi_kernel_nonuniform(
            A
          , u
          , b_
          , pair<size_t, size_t>(i, min(i + block_size, x.size()))
          , dst
          , src
        );
    }
    swap(old, new_);
}
```

Listing 5.6: OpenMP - Non-Uniform Jacobi-Method

5.2.2. HPX Implementation

The HPX implementation of the non-uniform Jacobi Method in Listing 5.9 is subject to more changes. Some of the differences are the explicitly saved matrix and the dataflow application of the nonuniform kernel (see Listing 5.5) instead of the one outlined in Listing 5.1. Due to these changes, the iteration and creation of the Data Flow objects changes.

```
struct lse_data
{
    lse_data(
        crs_matrix<double> const & A, vector<double> const & x
      , vector<double> const & b)
        : A(A), x(2, x), b(b) {}
    crs_matrix<double> A; vector<vector<double> > x;
    vector<double> b;
};
void jacobi_kernel_nonuniform_lse(lse_data & lse, range_t range,
    size_t dst, size_t src)
{
    jacobi_kernel_nonuniform(lse.A, lse.x, lse.b, range, dst, src);
}
```

Listing 5.7: HPX - LSE Data Structure

Another significant difference is the data structure, which holds the matrix, left hand side, and right hand side information (see Listing 5.7). This is needed in order to efficiently store the necessary information in the global address space. A wrapper function is created that forwards the call to the right kernel and unpacks the LSE elements.

The dependency calculation in Listing 5.8 now determines which of the neighboring blocks contain the depending elements.

```
1
2 vector<vector<size_t> > calc_dependencies(vector<range_t> block_ranges)
3 {
4    vector<vector<size_t> > dependencies(block_ranges.size());
5    for(size_t b = 0; b < block_ranges.size(); ++b)
6    {
7       for(size_t i = block_ranges[b].first; i < block_ranges[b].second; ++i)
8       {
9          size_t begin = A.row_begin(i);
10         size_t end = A.row_end(i);
11         for(size_t ii = begin; ii < end; ++ii)
12         {
13            size_t idx = A.indices[ii];
14            for(size_t j = 0; j < block_ranges.size(); ++j)
15            {
16               if(block_ranges[j].first <= idx &&
17                  idx < block_ranges[j].second)
18               {
19                  if(find(dependencies[b].begin(), dependencies[b].end(), j)
20                     == dependencies[b].end())
21                  {
22                     dependencies[b].push_back(j);
23                  }
24                  break;
25               }
26            }
27         }
28      }
29   }
30   return dependencies;
31 }
```

Listing 5.8: HPX - Non-Uniform Dependency Calculation

The algorithm in Listing 5.9 then computes the Jacobi iterations by looping over the calculated block ranges (Line 2 to 8), extracting the correct dependencies for the current block and applying the appropriate Data Flow object.

```
1
2 vector<range_type> block_ranges;
3 for(size_t i = 0; i < x.size(); i += block_size)
4 {
5     block_ranges.push_back(
6         range_type(i, min(x.size(), i + block_size))
7     );
8 }
9 vector<vector<size_t> > dependencies(calc_dependencies(block_ranges));
10
```

```
11 typedef dataflow_object<lse_data> object_type;
12 object_type lse(new_<lse_data>(find_here(), A, x, b_).get());
13
14 size_t src = 0;
15 size_t dst = 1;
16 typedef dataflow_base<void> dataflow_t;
17 typedef vector<dataflow_t> dataflow_grid_t;
18 vector<dataflow_t> deps(2, promise_grid_type(dependencies.size()));
19
20 for(size_t iter = 0; iter < max_iterations; ++iter)
21 {
22     for(size_t b = 0; b < block_ranges.size(); ++b)
23     {
24         vector<dataflow_t> iteration_deps(dependencies[b].size());
25         for(size_t p = 0; p < dependencies[b].size(); ++p)
26         {
27             iteration_deps[p] = deps[src][dependencies[b][p]];
28         }
29         deps[dst][b]
30             = lse.apply(
31                 hpx::bind(jacobi_kernel_nonuniform_hpx,
32                     hpx::placeholder::_1
33                   , block_ranges[b]
34                   , dst, src
35                 )
36               , iteration_deps
37             );
38     }
39     swap(src, dst);
40 }
41 hpx::wait(deps);
```

Listing 5.9: HPX - Non-Uniform Jacobi Method

6. Performance Evaluation

This chapter evaluates the performance of the implementations discussed in Chapter 5. The benchmarks were run on a multi-socket AMD Istanbul platform. The nominal specifications of this machine are:

- 8 Sockets, one AMD Opteron 8431 each socket 6 Cores per socket (48 Cores in total):
  - Base speed: 2.4 GHz
  - L1 Cache Size: 128 KB (per Core)
  - L2 Cache Size: 512 KB (per Core)
  - L3 Cache Size: 6144 KB (shared)
- 2 GB RAM per Core (96 GB RAM in total)
- Maximum 8.5 GB/s Memory Bandwidth per socket.
- Theoretical double precision peak performance: 230 GFLOPS/s

Apart from the nominal numbers, the STREAM benchmark [39] was run to measure the maximum memory bandwidth. Figure 6.1 shows the result of the STREAM benchmark Copy test. Due to memory bound nature of the discussed algorithm (see Section 5). These experimental results are used as the basis of the subsequent considerations. Deriving from theses numbers, the maximum achievable speedup is 4.88 and scaling beyond four cores cannot be expected. Additionally, the implications of non-optimal memory placement can be observed by considering the difference between the theoretical bandwidth and the measured. By placing the allocated memory pages in a round-robin fashion (interleaved) a good estimate on how much performance can be expected in our benchmarks was acquired.

6.1. Uniform Workload

For the uniform problem (Section 5.1), weak scaling and strong scaling tests have been performed. For weak scaling, the number of points on the y-axis were kept constant at 100000, and for every core 1000 grid points on the x-axis were added. For strong scaling, a problem size of 10000x100000 (`N_x = 10000` and `N_y = 100000`) has been chosen. The `block_size` has been chosen to be 1000 such that one tile calculation fits easily into the level-2 cache of our platform. Additionally, the problem size has been chosen big enough such that enough work for all processors can be assumed. The algorithm has been run for 100 iterations.

The performance is measured in Mega Lattice Site Updates per second (MLUPS). One lattice site is one element of the grid. Based on the performance characteristics of the benchmark platform, the maximal achievable performance can be calculated. The update of one lattice in

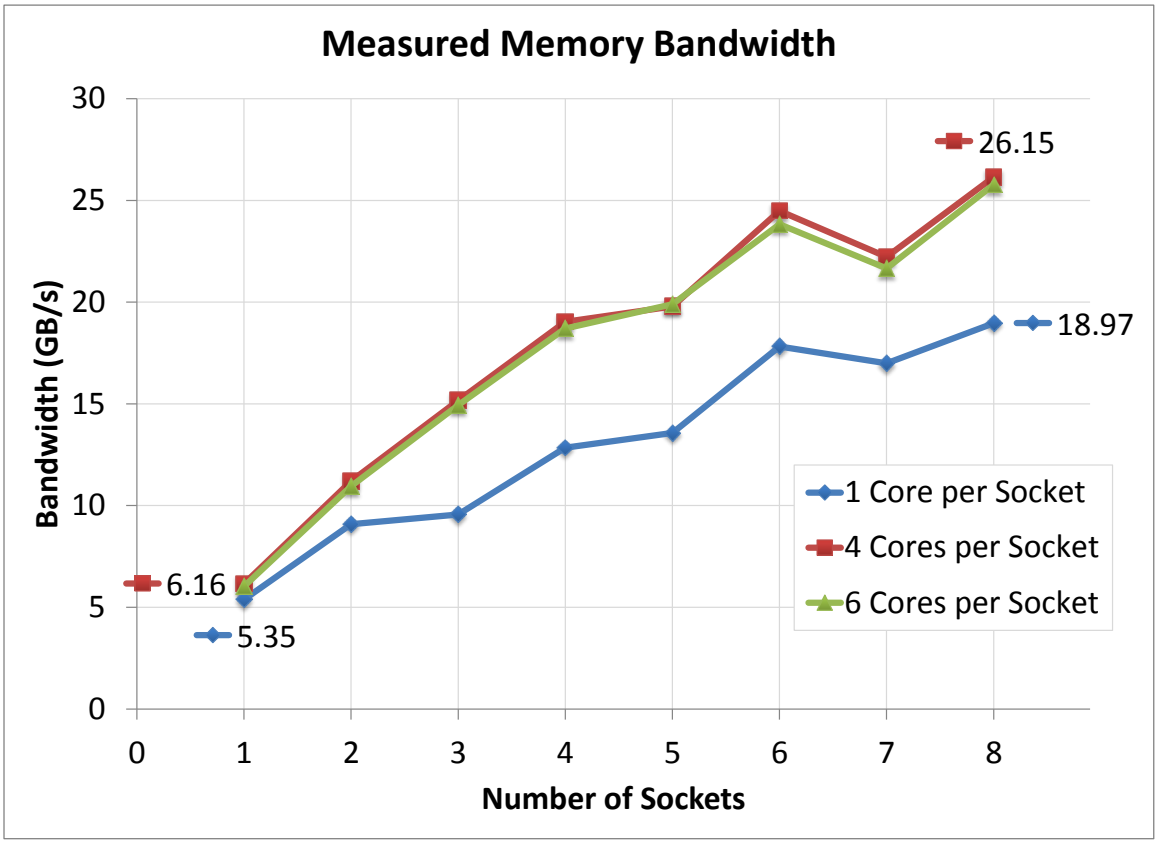


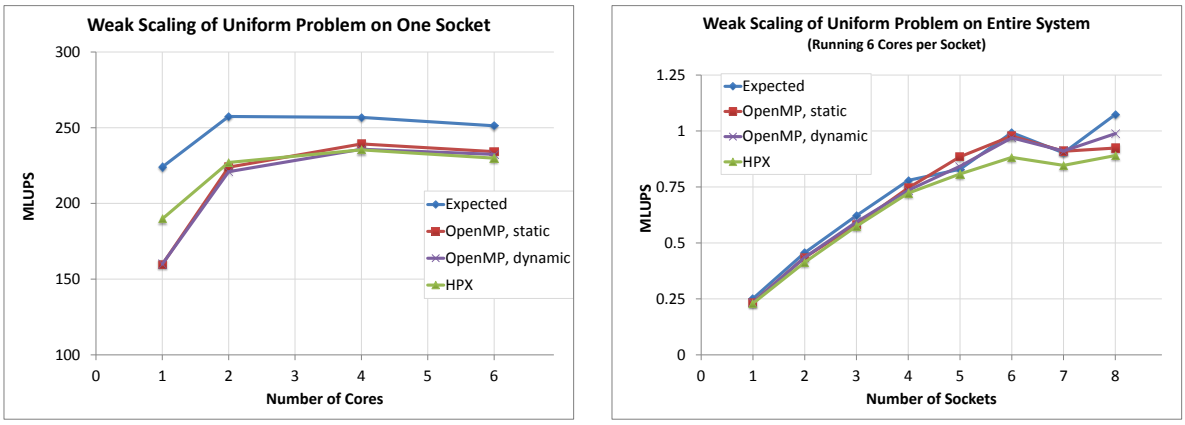
Figure 6.1.: Measured memory bandwidth on the benchmark platform. The shown data allows to derive that the best possible speedup when scaling from 1 to 48 cores on this particular system is 4.88 ($\frac{26.15}{5.35}$) Further, this figure shows that increasing the number of utilized cores per socket beyond four does not yield any performance benefit (the scaling numbers for six cores per socket are smaller than those for four cores).

the uniform grid requires three memory transfers, thus 24 bytes are transferred. Dividing the measured peak memory bandwidth by those numbers, leads to an expected maximum performance (shown as blue lines in all figures). For the OpenMP implementations, the performance has been measured with static and dynamic work scheduling.

All of the benchmarks show the same weak scaling behavior for the uniform testcase. It can be observed that the HPX implementation is able to perform and scale as good as the OpenMP implementation (Figure 6.2(a)).

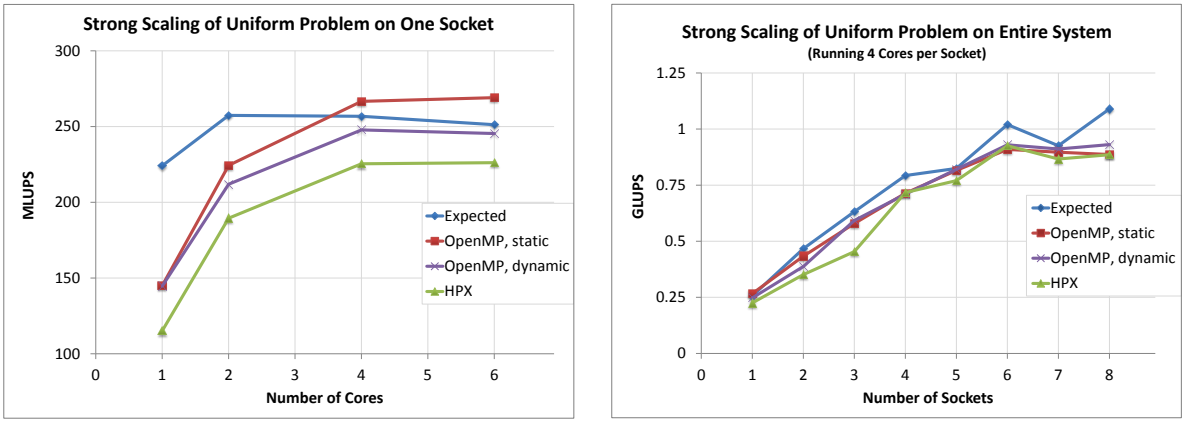
Weak scaling on the entire system (Figure 6.2(b)) is similar to scaling on one core. When utilizing the complete machine, the presented implementations are able to reach the expected performance almost everywhere. HPX performance starts to decline after using more than 5 sockets. This is due to the currently non-optimized NUMA placement (a maximum of 9% slower).

For strong scaling, OpenMP is able to perfectly use the resources of the system, while HPX is around 15% slower (Figure 6.3(a)) when using only one socket. This behavior is expected and can be explained with runtime overheads of the HPX runtime system, which cannot be amortized by such a highly regular workload. The scaling behavior however is consistent for



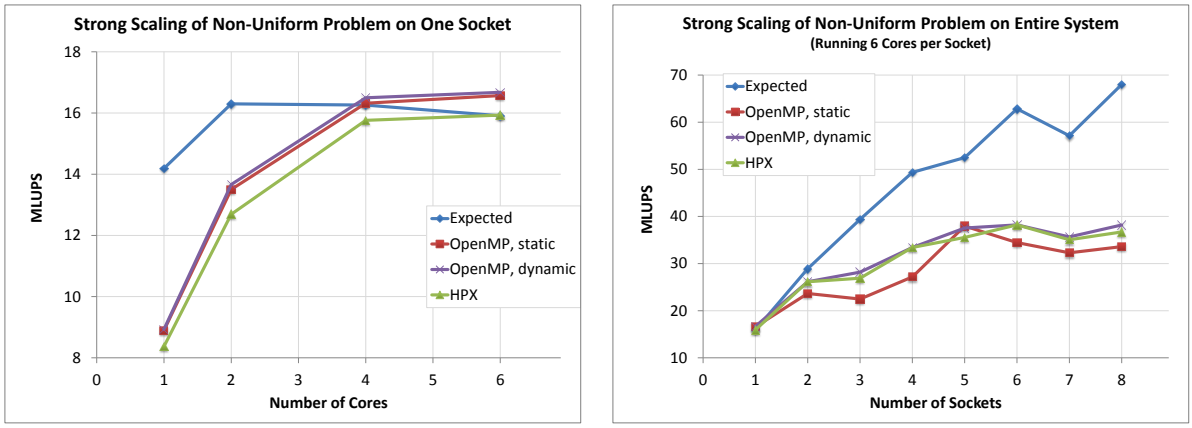
(a) One Socket (b) Entire Machine

Figure 6.2.: Weak scaling of the uniform problem. The measured results for the OpenMP and HPX applications are compared with the measured maximum bandwidth (as derived from Figure 6.1). Both, the OpenMP and HPX applications have similar qualitative scaling behavior. All three implementations show good weak scaling.



(a) One Socket (b) Entire Machine

Figure 6.3.: Strong scaling of the uniform problem. The measured results for the OpenMP and HPX applications are compared with the measured maximum bandwidth (as derived from Figure 6.1). On one socket HPX is constantly slower, this behavior is consistent with our expectations. The OpenMP application can leverage almost the full memory bandwidth. When adding more and more parallel resources, the HPX implementation is able perform as good as the OpenMP one.



(a) One Socket

(b) Entire Machine

Figure 6.4.: Strong Scaling of Non-Uniform workload. These figures show that both implementations have similar qualitative scaling behavior. When running on one socket, the HPX application runs consistently slower than the OpenMP application. However, the difference is much smaller than in the uniform case. On the entire machine, it can be seen that the HPX application is able to gain ground with an increasing number of parallel resources utilized for the runs.

both implementations.

Figure 6.3(b) shows the performance of both implementations for the entire system. For both runs, interleaved memory placement is used. It can be seen that the OpenMP implementations are able to almost reach the expected performance. The HPX implementation shows the same scaling behavior. However, the MLUPS for the HPX can not reach the performance of the OpenMP implementations. Overall it is marginally slower. Changing the scheduling of the OpenMP work chunks, only influences the results minimally.

6.2. Non-Uniform Workload

The Jacobi Solver for the irregular grid discussed in Section 5.2 was run for 1000 iterations and a `block_size` of 50000 has been chosen such that the level-2 cache is employed appropriately.

The expected MLUPS for the non-uniform application were derived from the mean number of entries in one row in the sparse matrix (see Section 5.2). Thus a mean number of memory transfers of 47.38, which means that 379.04 bytes are transferred for one lattice update. Again, dividing the measured peak memory bandwidth by those numbers, leads to an expected maximum performance. For the OpenMP implementations the performance has been measured with static and dynamic work scheduling.

The performance behavior of the non-uniform problem on one socket (Figure 6.4(a)) shows similar properties than the regular problem. Due to the workload imbalance, the constrained based parallelization technique is able to gain ground. The HPX implementation is now only 6% slower than the OpenMP one.

In Figure 6.4(b) the benefits of the constrained based implementation with HPX can be

seen very clearly. The more parallel resource are added to the system, the better the HPX implementation is able to perform. It can be observed that a maximum of 17% performance gain (24 cores) over the OpenMP implementation with static scheduling can be reached. This can be explained by the fact that different timing behaviors due to the irregular mesh lead to imbalanced workloads on the different threads. While HPX is able to dynamically schedule work items, the implicit global barrier of OpenMP needs to wait until all blocks are finished. However, when using the dynamic work scheduler of OpenMP, both parallelization paradigms are mostly on par. The OpenMP implementation is still able to slightly outperform HPX. While HPX offers a solution that leads to acceptable performance for equally distributed and highly irregular workload, we need to manually adapt the OpenMP pragma and compile two different versions of our program to get the maximal achievable performance.

7. Conclusion

This thesis describes the ParalleX execution model with its implementation, the High Performance ParalleX runtime system. A Dataflow LCO has been developed and applied to a well known algorithm. The performance and scaling characteristics of two different applications of a Jacobi solver for a linear system of equations have been evaluated. One application is using a very regular, uniform grid, whereas the other solves the linear system of equations on a highly irregular grid. Both applications have been implemented using OpenMP and HPX. The results presented confirm that the data driven, task-queue based, fine-grain parallelism and the constraint based synchronization methods as implemented in HPX show no significant loss of performance for this class of algorithms. It can be seen that both approaches, using the OpenMP paradigm and the ParalleX model, lead to comparable results. While OpenMP is able to achieve better results than HPX in the general case, it should be noted that OpenMP has been in development and production for almost 15 years and as such was subject of various optimization. HPX on the other hand did not yet leave the experimental stage. Version 0.8, a public pre-beta has been released on March, 23rd, 2012.

While this thesis didn't discuss distributed memory systems it should be noted that formulating an algorithm with HPX that is able to run on clusters is easy to implement. The exchange of ghost zones can be formulated in terms of Data flow and the communication will be automatically overlapped with computation. Such an approach will lead to highly complex implementations if implemented with conventional methods such as MPI and OpenMP.

A. Additional Implementations

A.1. Grid Data Structure

```
template <typename T>
struct grid
{
  grid(size_t N_x, size_t N_y)
    : data(N_x * N_y)
    , N_x(N_x), N_y(N_y)
  {}
  T & operator()(size_t x, size_t y)
  {
      return data[x + N_x * y];
  }
  vector<T> data;
  size_t N_x;
  size_t N_y;
};
```

Listing A.1: Grid Data Structure

A.2. Sparse Matrix Data Structure

```
template <typename T>
struct crs_matrix
{
    typedef vector<T> values_type;
    typedef typename values_type::reference reference;
    typedef typename values_type::const_reference const_reference;
    typedef vector<size_t> indices_type;

    size_t row_begin(size_t i) const
    {
        return rows[i];
    }

    size_t row_end(size_t i) const
    {
        return rows[i+1];
    }

    size_t col(size_t j)
    {
        return indices[j];
```

```
22      }
23
24      T & operator[](size_t j)
25      {
26          return values[j];
27      }
28
29      T const & operator[](size_t j) const
30      {
31          return values[j];
32      }
33
34      vector<T>           values;
35      vector<size_t> indices;
36      vector<size_t> rows;
37  };
```

Listing A.2: Sparse Matrix Data Structure

B. Bibliography

[1] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *J. ACM*, vol. 31, no. 3, pp. 560–599, 1984.

[2] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX: An advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops*, (Los Alamitos, CA, USA), pp. 394–401, IEEE Computer Society, 2009.

[3] "The OpenMP API specification for parallel programming." `http://openmp.org/wp/openmp-specifications/`, 2011.

[4] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling, "Preliminary design examination of the ParalleX system from a software and hardware perspective," *SIGMETRICS Performance Evaluation Review*, vol. 38, p. 4, 2011.

[5] H. Kaiser, "Is the Free Lunch Over? Really?." `http://stellar.cct.lsu.edu/2012/01/is-the-free-lunch-over-really/`, 2012.

[6] InfiniBand Trade Association, "Introduction to infiniband for end users." `http://members.infinibandta.org/kwspub/Intro_to_IB_for_End_Users.pdf`, 2010.

[7] R. van der Pas, "Memory Hierarchy in Cache-Based Systems," Tech. Rep. 817-0742-10, Sun Microsystems, Sant a Clara, California, U.S.A., 2002.

[8] "NodeJS." `http://nodejs.org`, 2012.

[9] T. Q. Viet and T. Yoshinaga, "Improving linpack performance on smp clusters with asynchronous mpi programming," *IPSJ Digital Courier*, vol. 2, pp. 598–606, 2006.

[10] "Intel Threading Building Blocks for Open Source." `http://threadingbuildingblocks.org/`, 2012.

[11] "Parallel Patterns Library (PPL)." `http://msdn.microsoft.com/en-us/library/dd492418.aspx`, 2012.

[12] "Intel Cilk Plus." `http://software.intel.com/en-us/articles/intel-cilk-plus/`, 2012.

[13] C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. L. Sterling, "Improving the scalability of parallel n-body applications with an event driven constraint based execution model," *CoRR*, vol. abs/1109.5190, 2011.

[14] H. C. Baker and C. Hewitt, "The incremental garbage collection of processes," in *SIGART Bull.*, (New York, NY, USA), pp. 55–59, ACM, 1977.

[15] D. P. Friedman and D. S. Wise, "Cons should not evaluate its arguments," in *ICALP*, pp. 257–284, 1976.

[16] G. Papadopoulos and D. Culler, "Monsoon: An explicit token-store architecture," in *17th International Symposium on Computer Architecture*, no. 18(2) in ACM SIGARCH Computer Architecture News, (Seattle, Washington, May 28–31), pp. 82–91, ACM Digital Library, 1990.

[17] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, 2005.

[19] F. Cantonnet, T. A. El-Ghazawi, P. Lorenz, and J. Gaber, "Fast address translation techniques for distributed shared memory compilers," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, (Washington, DC, USA), pp. 52.2–, IEEE Computer Society, 2005.

[20] R. W. Numrich and J. Reid, "Co-arrays in the next fortran standard," *SIGPLAN Fortran Forum*, vol. 24, no. 2, pp. 4–17, 2005.

[21] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, pp. 97–107, Feb. 1992.

[22] D. W. Wall, "Messages as active agents," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, (New York, NY, USA), pp. 34–39, ACM, 1982.

[23] P. Miller, A. Becker, and L. Kalé, "Using shared arrays in message-driven parallel programs," *Parallel Computing*, vol. 38, pp. 66 – 74, 2012.

[24] STE||AR Group, "Systems Technologies, Emerging Parallelism, and Algorithms Reseach." `http://stellar.cct.lsu.edu`, 2012.

[25] M. Anderson, M. Brodowicz, H. Kaiser, and T. L. Sterling, "An Application Driven Analysis of the ParalleX Execution Model," *CoRR*, vol. abs/1109.5201, 2011.

[26] "Boost: a collection of free peer-reviewed portable C++ source libraries." `http://www.boost.org/`, 2012.

[27] B. Adelstein-Lelbach, "Benchmarking User Level Threads." `http://stellar.cct.lsu.edu/2012/03/benchmarking-user-level-threads/`, 2012.

[28] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *IPDPS*, pp. 1–8, 2008.

[29] K. B. Wheeler, R. C. Murphy, D. Stark, and B. L. Chamberlain, "The Chapel Tasking Layer Over Qthreads," in *Cray User Group 2011*, 2011.

[30] "ET International Announces First Commercial Implementation of ParalleX Execution Model Using SWARM Technology for Many-Core." http://www.etinternational.com/index.php/news-and-events/press-releases/eti-swarm-first-commercial-implementation-of-parallex/, 2011.

[31] J. B. Dennis, "First version of a data flow procedure language," in *Symposium on Programming*, pp. 362–376, 1974.

[32] J. B. Dennis and D. Misunas, "A primlinary architecture for a basic data-flow processor," in *25 Years ISCA: Retrospectives and Reprints*, pp. 125–131, 1998.

[33] Arvind and R. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," in *PARLE '87, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages* (J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, eds.), Berlin, DE: Springer-Verlag, 1987.

[34] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[35] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 2010.

[36] J. Triebig, "Likwid: Linux tools to support programmers in developing high performance multi-threaded programs." http://code.google.com/p/likwid/, 2012.

[37] C. Janna and M. Ferronato, "Janna/Serena sparse matrix." http://www.cise.ufl.edu/research/sparse/matrices/Janna/Serena.html, 2011.

[38] M. Ferronato, G. Gambolati, C. Janna, and P. Teatini, "Geomechanical issues of anthropogenic co2 sequestration in exploited gas fields," *Energy Conversion and Management*, vol. 51, no. 10, pp. 1918 – 1928, 2010.

[39] J. D. McCalpin, "STREAM: Sustainable memory bandwidth in high performance computers," a continually updated technical report, University of Virginia, Charlottesville, VA, 2007.