International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 3, Number 1, pages 15–36, January 2013

A dataflow-like programming model for future hybrid clusters

Jens Breitbart
Research Group Programming Languages / Methodologies, University of Kassel
Wilhelmshöher Allee 73, 34121 Kassel, Germany

Email: jbreitbart@uni-kassel.de

Received: June 4, 2012
Revised: October 27, 2012
Accepted: December 5, 2012
Communicated by Akihiro Fujiwara

Abstract

It is expected that the first exascale supercomputer will be deployed within the next 10 years, however both its CPU architecture and programming model are not known yet. Multicore CPUs are not expected to scale to the required number of cores per node, but hybrid multicore CPUs consisting of different kinds of processing elements are expected to solve this issue. They come at the cost of increased software development complexity with e. g., missing cache coherency and on-chip NUMA effects. It is unclear whether MPI and OpenMP will scale to exascale systems and support easy development and scalable and efficient programs. One of the programming models considered as an alternative is the the so-called partitioned global address space (PGAS) model, which is targeted at easy development by providing one common memory address space across all cluster nodes. In this paper we first outline current and possible future hardware and introduce a new abstract hardware model able to describe hybrid clusters. We discuss how current shared memory, GPU and PGAS programming models can deal with the upcoming hardware challenges and describe how synchronization can generate unneeded inter- and intra-node transfers in case the memory consistency model is not optimal. As a major contribution, we introduce our variation of the PGAS model allowing implicit fine-grained pairwise synchronization among the nodes and the different kinds of processors. We furthermore offer easy deployment of RDMA transfers and provide communication algorithms commonly used in MPI collective operations, but lift the requirement of the operations to be collective. Our model is based on single assignment variables and uses a data-flow like synchronization mechanism. Reading uninitialized variables results in the reading thread to be blocked until data are made available by another thread. That way synchronization is done implicitly when data are read. Explicit tiling is used to reduce synchronization overhead and to increase cache and network utilization. Broadcast, scatter and gather are modeled based on data distribution among the nodes, whereas reduction and scan follow a combining PRAM approach of having multiple threads write to the same memory location. We discuss the Gauß-Seidel stencil, bitonic sort, FFT and a manual scan implementation in our model. We implemented a proof-of-concept library showing the usability and scalability of the model. With this library the Gauß-Seidel stencil scaled well in initial experiments on an 8-node machine and we show that it is easy to keep two GPUs and multiple cores busy when computing a scan.

Keywords: programming model, PGAS, data flow, distributed memory, hybrid multicore, GPU

1 Introduction

Each of the top ten supercomputers provides more than 1 petaflop of computing power[1] and consists of ten-thousands of nodes each with tenths of cores. Current predictions [9] state that the first exascale system will be deployed in 2018 and is expected to consist of hundreds of thousands or even millions of nodes each with thousands or ten-thousands of cores. These systems are expected to raise two major problems with regard to programming model design.

Multicore scaling may end The last decade saw a large increase in single thread performance being made available transparently, this trend came to an end. Vendors started putting multiple cores in one CPU to still be able to increase performance of the whole chip, yet the performance increase using this technique again seems to slow down. Manycore chips such as GPUs on the other hand continue to increase performance at a high rate and provide multiple times the performance of modern CPUs. Unfortunately, not all problems are feasible for manycore chips, and the rather slow connection between the multicore CPUs and manycore GPUs makes it hard to effectively have both types of cores closely cooperate on solving tasks.

A possible solution for future hardware are so-called hybrid multicore chips, which combine different kinds of cores on the same chip. This allows to increase overall chip performance and eases the communication between different kinds of cores. Lately AMD started shipping its CPU and GPU hybrid chip named Llano, which is currently the only hybrid chip in mainstream market. We expect upcoming hybrid multicores to impose new challenges:

- Hybrid CPUs consist of different kinds of cores, e. g., throughput and latency optimized cores. Having such components working together requires fine-grained pair-wise synchronization, as e. g., having a latency core wait until the throughput cores have completed their task results in poor load balancing for the latency core.
- It is unclear if future hybrid CPUs will be cache coherent.
- With chip size increasing and multiple memory controllers being added to one chip, NUMA effects may appear between the different kinds of cores.

MPI / OpenMP may not scale Current supercomputers are often programmed with a combination of MPI for inter-node communication and OpenMP or MPI for intra-node communication, it is unclear whether this communication mechanism will scale to the required order and allow effective usage of upcoming architectures. Furthermore, the currently often used OpenMP like fork-join parallelism and bulk synchronous processing like communication pattern is not expected to keep all processors of exascale systems active, but it increases idle time by requiring global synchronization [9].

A model considered as an alternative is the so-called partitioned global address space (PGAS) model. It assumes a global address space for all nodes in a system in the form that any node can read and write any memory location. Furthermore the address space is partitioned and each partition is local to a specific node, so the best performance is achieved when threads access local data. There are different approaches for synchronization within the PGAS model. Unified Parallel C (UPC) [6] for instance provides locks and barriers to synchronize threads within and among nodes. The benefit of the PGAS model is that it allows direct support for remote DMA transfers over network and rather easy programming of distributed memory systems, as one can read data without requiring cooperation with the local program threads. PGAS in general treats clusters as large non-cache coherent NUMA systems and is thereby also a good match for future hybrid multicore CPUs, but the existing models are not designed to effectively deal with such scenarios.

Our work involves contributions in three areas:

1. survey of hardware development and proposal for unified model

[1] http://www.top500.org/list/2011/11/100

2. new synchronization model for PGAS and
3. prototype implementation of this model.

In the hardware part, we first discuss current hardware and possible future hybrid multicore chips, and explain why fine-grained pair-wise synchronization and the abilities to deal with missing cache coherence and NUMA aspects are essential to achieve high performance on such systems. Based on that, we introduce a generic hardware model able to describe distributed memory systems using multicore, NUMA, GPU and hybrid multicore nodes. The model is based on modules, which consist of a set of uniform processors and their main memory and caches. For example, a NUMA system composed of 4 CPUs is modeled by 4 modules, whereas a hybrid system consisting of a multicore CPU and a GPU is modeled by 2 modules.

As a basis for the second contribution, we first discuss common shared memory, GPU and PGAS languages and their differences regarding synchronization, remote memory accesses and memory consistency. We furthermore detail how well they can deal with the anticipated hardware changes. Our own PGAS variation extends the generic PGAS model with a pair-wise fine-grained automatic synchronization mechanism, i. e., developers do not manually take care of synchronization. Memory shared by multiple nodes is single assignment and therefore has two states: it can contain data, or be unitialized. A thread reading unitialized memory is blocked until another thread writes to that memory. A thread writing data automatically wakes up all threads waiting for the written data. In the extreme, one could use this synchronization mechanism on bit level, however this would require adding a synchronization bit to every data bit, which is obviously not feasible. We therefore apply this mechanism on batches of data. Furthermore the model assures that the available processors are oversaturated with threads, so that in case a thread is waiting for data the system can continue to work. We expect this form of synchronization to be easy to use and debug, as one cannot have a race condition when reading data as data cannot be overwritten. The worst case is a deadlock of a thread reading data that is never written, however debugging such a case is mostly easier than identifying race conditions, as the threads are blocked clearly naming which data dependencies are not satisfied.

We allow distributing batches of single assignment data among all nodes, i. e., a matrix can be stored purely locally or its data batches can be distributed among the nodes. We furthermore introduce a special data distribution storing all data on every node for which writing acts as a broadcast. Gather and scatter are modeled based on assignment of data with different data distributions. Reductions with single assignment are inherently complex, so we introduce special reduction variables following the combining PRAM approach, that is, multiple threads write to the same variable and its value can be read as soon as all nodes have written a value. Scan is modeled similarly, except for being defined on an array that stores all results of the scan. We show that all communication algorithms used in MPI collective operations can be used in our model. To demonstrate the new programing model, we discuss a set of fundamental data structures like vector and matrix and use them to implement Gauß-Seidel stencil, and bitonic sort sorting and scan using multiple nodes.

The last part of our work describes a proof of concept library implementation of the model. Our implementation is based on CUDA and the active messaging implementation of the GASNet [5] library, also used for e.g. UPC and Chapel. Our matrix implementations are tile based, so that the synchronization is based on tiles. Using tiles is required for efficient network communication. Due to a limitation of GASNet we could not effectively oversaturate the available nodes with threads and therefore decided to test our implementation on problems with regular memory access patterns only. We decided to use GASNet as a starting point of our work despite this problem, as it is well known and widely adopted. Future work may choose another library or modify GASNet to solve this issue. We implemented the Gauß-Seidel stencil and our performance results show that it scales well for multiple multicore nodes. The techniques used in both the model and the implementation are mostly known and have been implemented previously, yet we expect the overall combination to be unique.

The paper is organized as follows. First, Sect. 2 surveys current hardware and predicts possible future hybrid multicore chips. The next section (Sect. 3) discusses our hardware model and how it describes the hardware introduced in the previous section. Sect. 4 describes current shared memory

and GPU programming models regarding our hardware predicts. Sect. 5 discusses they same aspects for current PGAS programming models. Our model is introduced in Sect. 6 and Sect. 7. We first introduce pair wise synchronization (Sect. 6), and then data distributions and optimized communication algorithms including broadcast and reduction (Sect. 7). Based on our model we introduce basic vector and matrix data structures in Sect. 8. In the next section (Sect. 9) we use our model to implement example algorithms. The next two sections detail our library implementation (Sect. 10) and discuss the Gauß-Seidel stencil implementation (Sect. 11). The paper finishes with an overview of related work and conclusions, in Sects. 12 and 13, respectively.

2 Current and future hardware

This section first gives an overview of hardware currently used in supercomputers and afterwards discusses hybrid multicore CPUs as we expect them to be the next step in hardware development.

Seemingly all current supercomputers are distributed memory systems consisting of multiple shared memory nodes. In its simplest form a node has only one multicore CPU, but a node may also incorporate multiple CPUs, typically as a NUMA system that allows to increase the total amount of memory and memory bandwidth of the node. A CPU may also be supported by a GPU to speed up data parallel computations. We detail these different hardware architectures next.

Multicore CPUs A multicore CPU consists of a set of cores, each of them optimized for single thread performance. We call these cores latency oriented. Latency oriented cores use coherent cache hierarchies to keep memory access latency down to a minimum. Communication between multiple cores often goes through a shared last level cache. On the current Intel Sandy Bridge architecture, for instance, the level 3 cache is shared and data used by multiple cores can be stored in that cache. Most current CPU cores support so called SIMD instructions, which allows the CPU to execute the same instruction on multiple values at the same time, so e. g., Sandy Bridge can add two single precision floating point vectors with a single instruction of the length 9 with one instruction. Use of the SIMD units increases the efficiency of instruction scheduling and the memory system.

NUMA A NUMA system consists of multiple CPUs each with its own local memory partition. A memory partition local to a different CPU is called a remote partition. Every CPU can access both its local and remote partitions, but accessing a remote partition has a higher latency. The difference in latency between accessing the local partition and a remote partition is measured by the so called NUMA factor. A NUMA factor of about 2 is common for e. g., small Intel Nehalem based NUMA systems and means that a remote access has twice the latency of a local access.

GPUs Modern GPUs are used as accelerators for data parallel workloads. GPUs have their own memory subsystem and their main memory is called *global memory*. The PCI Express bus (PCIe) is used to transfer data between (CPU) main memory and global memory, yet there is no mechanism to keep copies consistent. PCIe is optimized for transferring large chunks of data as the cost for initializing the transfer is rather high. GPUs are tile-based many-core systems, which bundle sets of processors together in tiles. NVIDIA calls a tile a streaming multiprocessor (SM). The processors of the same tile are SIMD units, so all processors of a tile execute the same instruction at a time. NVIDIAs GF100 (Fermi) has 448 cores organized into 14 SM with 32 cores each. A SM provides fast on-chip scratch pad memory and all processors of the SM can work together on it. A single processor does not provide high performance and is optimized for throughput rather than latency. The latest generation of GPUs provides caches, however they are mostly used to ease a restriction on efficient memory access patterns and not to expose temporal locality.

As discussed, the current CPU architecture landscape is dominated by two approaches: latency oriented cache coherent multicore CPUs and throughput oriented manycore GPUs. Multicore architectures seem to have reached a point at which further performance increase becomes rather

complicated [16, 1], especially maintaining cache coherency for all cores gets harder with each core. Performance of manycore architectures continues to increase and currently a GPU provides multiple times the performance of a multicore CPU, yet not all problems are suitable for manycore architectures. Furthermore, application specific processors can also be used to improve performance for specific applications.

Future processors can be built as so-called hybrid multicore CPUs, which consist of different kinds of cores each optimized for certain types of tasks, e. g.a latency optimized core, a throughput optimized core and an FPGA used as an application specific processor could put in the same CPU. Currently there are hardly any such systems on the market, however we give a brief overview next and discuss the principle possibilities and problems of such hardware after that. We refer to hybrid multicore CPUs as hybrid chips.

Probably the most well-known hybrid chip of the last years was the Cell Broadband Engine [7], which achieved high performance compared to other CPUs of its time. Future development of the Cell B.E. seems to be discontinued. A new hybrid chip is AMD's already mentioned Llano chip [4], which combines a quad-core CPU with a manycore GPU. The chip is focused on mainstream desktop use and not on high performance computing. Both modules share the memory interface, however not the same address space so data is not automatically shared [3]. We expect this limitation to be lifted with future hardware generations, but do not expect that current shared memory programming techniques are a good match for such systems due to the issues discussed next. Liu et al. [17] came to a similar finding when evaluating programming models for Intel's upcoming MIC architecture, previously known as Larrabee. MIC consists of multiple x86 based in-order CPUs with large vector units. We identified three major challenges crucial for a programming system for upcoming hybrid chips.

No cache coherence As stated before, we expect hybrid chips to offer shared memory for all cores. We furthermore assume future hybrid chips to contain caches, but not necessarily that all will be coherent. On non cache coherent systems, memory barriers for the whole CPU will become rather expensive, as not only values currently in registers must be written back to cache, but caches must be written back to main memory. Communication between the non-cache coherent cores should be minimized and well defined.

Efficient fine grained synchronization We expect hybrid systems to be most efficiently used when all cores are busy and supplied with tasks matching their architecture. The different kinds of cores follow different execution styles complicating synchronization: Latency cores produce single results fast, whereas throughput cores produce a batch of results at once with possible high latency between the batches. Synchronization must be at batch level to keep all cores busy as otherwise latency cores may be idle longer than necessary resulting in poor load balancing. For example, current systems use barriers for synchronization requirering the CPU to wait for the GPU to complete its work.

NUMA Another issue with current systems is that memory bandwidth is increasing rather slowly compared to processing power. Memory bandwidth can be increased by using multiple memory controllers on the same chip, as it is e. g., done by Intel's SCC [14], a manycore architecture without a shared address space. Having multiple memory controllers on the chip may lead to NUMA issues, i. e., accesses to a certain memory location have different latency for different cores. For simplicity we expect there to be one memory controller per module for the rest of this work, however the overall observations stay true even if the practical setup is changed.

In summary, the requirements stated above are similar to those of PGAS programming models for clusters, even though the concrete performance characteristics are different.

3 Abstract hardware model

In this section, we introduce our abstract hardware model, which is designed to describe current and future hardware in one unified model. We use the model later on to discuss how current programming

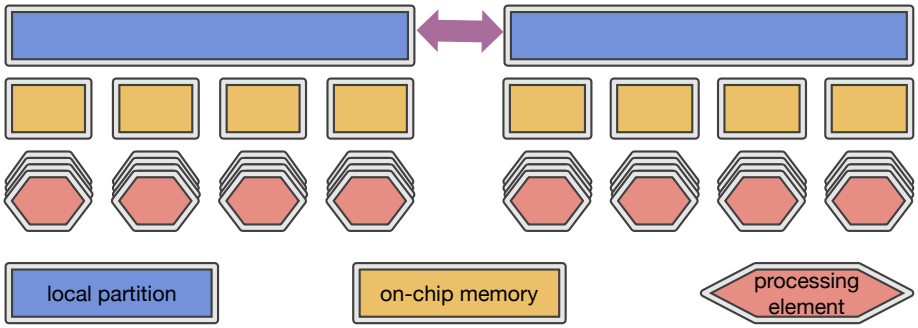

Figure 1: Two modules of our abstract hardware model.

models match the expected hardware requirements. Moreover, the hardware model can be slightly extended by a cost model to allow PRAM like algorithm design and analysis. This part of the model is out of scope for this work, though.

Figure 1 gives a simple example of our model. The model itself does not limit the number of elements and allows to model systems with e.g. a different number of processing elements. In the model systems are divided in so-called *modules*. A module is a set of uniform processors and their memory subsystem, e. g., a cache and its main memory partition. In case different kinds of processors share one main memory partition they are considered as two modules. We call the main memory partition of a module *local partition*. We do not limit the number of processing elements per module and expect each to have its own set of registers. A module furthermore has caches or scratch pad memories, which we simply call *on-chip memory*. On-chip memory greatly benefits from working on batches of data.

All modules share a common address space and each module can access the local partition of all modules. We call the the local partition of a different module a *remote partition*. Accesses to remote partitions benefit from accessing blocks of data similar to accesses to the local partition. We expect the latency (l) of the different memories to fulfill

$$l(\text{registers }) \leq l(\text{on-chip memory}) \leq l(\text{local partition}) \leq l(\text{remote partition})$$

We do not require the NUMA factor to be identical throughout the machine.

We strictly follow the relaxed memory consistency model and the OpenMP naming conventions, i. e., we allow copies of data in different memories and require that the copies are explicitly made consistent. We call a set of copies *temporary view* and define a *flush* operation to make the temporary view and the original data consistent. Thus a flush implies the following actions:

- A modified temporary view is written back to the original location.
- An unmodified temporary view is marked invalid and the value must be reread when accessed again.

Temporary views have a lower latency than the original location. We allow one temporary view per data element in each memory and flushes between the temporary views in different memories. This allows sharing of temporary views among multiple processing elements, similar to e. g., shared caches. We expect support for atomic operations following a strict memory consistency model as it is common for atomic operations. The atomic operations must be system-wide and possible on all memories, including atomic operations on remote partitions.

The model has been designed to cover a wide variety of systems, so we hardly put any restriction on memory consistency covering a wide variety of possible hardware implementations. This choice is obviously motivated by the anticipated hardware changes discussed in the last section. We also do not put any direct requirements on how certain features are implemented. We do not distinguish between caches or scratch pad memory as their underlying concept of requiring locality is identical,

nor do we care if there is direct hardware support for accesses to remote memory or it requires support by a software layer. More requirements would complicate the model and restrict its flexibility.

In general, our hardware model treats various systems as non cache coherent NUMA systems, which allows us to model a wide variety of systems. Thus, Fig. 1 may represent:

NUMA The hardware could be a simple NUMA system with two multicore CPUs, each with its own local memory partition. Main memories are obviously identified with the local partitions and on-chip memory is matched to caches. Modern CPUs have more cache levels, yet the model does not represent the detailed on-chip memory hierarchy, but only that such memory is available and that it benefits from blocking.

CPU + GPU Figure 1 could also describe a multicore CPU with a GPU, in which case both modules have different kinds of processing elements, yet they match our model. We identify CPU main memory and GPU global memory each with a local partition, and CPU caches and GPU scratch pad memories with on-chip memory. Modern GPUs can directly access main memory, and CPUs can directly access global memory, however data is not kept coherent between the modules.

Hybrid multicore Of course, the figure also describes a hybrid chip, as we do not make any assumption on how communication between the different modules is handled, and thus this case corresponds to that of a CPU + GPU combination.

Cluster A fourth way to interpret Fig. 1 would be a cluster of nodes each with a multicore CPU. Communication must go over network and there is no hardware shared address space, yet it can be implemented in software.

The four architectures represented by the model differ only in their communication costs, so e. g., smaller batches of data can be transfered effectively between the modules of a hybrid chip than between cluster nodes. We expect application specific cores to follow the model as well, even though they may only be used for certain tasks.

4 Shared memory and GPU programming models

In this and the following section, we discuss whether existing programming models are suitable for hardware as represented by the model of Sect. 3. Among the most widespread today are shared memory, GPU and PGAS systems. The various PGAS systems are discussed in the next section, whereas in this section shared memory programming is exemplified by OpenMP and GPU programming by CUDA [18]. CUDA is almost identical to OpenCL [19] for the discussed aspects and we only refer to CUDA throughout the rest of this work. We chose OpenMP and CUDA as they are among the most widely used programming models.

OpenMP is a pragma based shared memory parallel programming system targeting multicore CPUs. OpenMP strictly follows a fork-join structure of creating teams of threads, so for example `#pragma omp parallel num_threads(4)` creates a team of 4 threads that execute the code block after the pragma. At the end of that code block the threads are joined and a single master thread continues execution until the next parallel region is found or the program ends. Synchronization of threads in OpenMP uses locks or so called criticals, which enforce mutual exclusion for code regions. With OpenMP 3.1 threads can also be synchronized by atomic operations, however in that case developers must manually enforce memory consistency. The OpenMP memory model allows temporary views of shared variables similar to our description in the last section. Furthermore, OpenMP defines the flush operation at which the temporary view of the calling threads is made consistent with memory, i. e., local changes are written back and unchanged values are invalidated. As discussed, such a definition obviously works well on cache coherent systems, on non-cache coherent systems the whole cache must be flushed creating high communication overhead. A flush is automatically included in all OpenMP constructs implying synchronization, except atomic operations. However synchronization with atomic operations normally requires explicit flushes, as otherwise threads hardly can exchange any data.

	UPC	X10	Chapel	XcalableMP	Our Model
execution model	PGAS	APGAS	APGAS	PGAS	PGAS
memory					
direct remote memory access	✓	✗	✓	✗	✓
private memory	✓	✓	✗	✓	✓
strict memory consistency	✓	✗	✓	✗	✓
relaxed memory consistency	✓	✓	✓	✓	✗
synchronization					
barrier	✓	✓	✗	✓	✗
lock / critical	✓	✓	✗	✗	✗
synchronization variables	✗	✗	✓	✗	✓
global communication algorithms	✓	✗	✗	✗	✓

Table 1: Characterization of PGAS languages.

OpenMP locks and criticals impose rather high overhead due to lock management and the implicit flush. Requiring locking for both reading and writing can easily result in contention on the lock even if data is unchanged, yet OpenMP does not allow reading shared variables without synchronization in most scenarios. Furthermore the lock itself must be kept consistent for all modules and accessing a lock in remote memory is therefore rather expensive. Overall, using criticals to synchronize on small batches of data is most likely not feasible on large systems and using OpenMP atomic operations is rather complicated.

OpenMP by itself does currently not support NUMA optimization, so it cannot solve any of the anticipated NUMA issues. OpenMP has been implemented for non cache coherent systems by e. g., Intel's Cluster OpenMP [15], yet it has not been successful. Overall, OpenMP and similar systems have not been designed with high costs for flush operations in mind, nor are their synchronization constructs easy to use when trying to achieve fine-grained pair-wise synchronization. These issues make OpenMP a bad match for our hardware model.

CUDA is NVIDIA's GPU programming system targeting mostly hybrid systems consisting of a CPU and GPU each with its own memory. In CUDA, developers specify a set of data parallel tasks called threadblocks that are automatically scheduled on the GPU. It is not possible to utilize possible data locality between tasks nor is synchronization between tasks allowed. Synchronization between the CPU and GPU always takes the form of a barrier at which the GPU must have completed a whole set of data parallel tasks. The barrier obviously enforces memory consistency, i. e., all writes done by the GPU tasks are visible to the CPU. Finer grain synchronization between CPU and GPU could be implemented using atomic operations and explicit memory fences, yet this is error prone and on most current systems not beneficial due to the PCIe communication link.

Data from a remote partition is normally read by making an explicit copy into the local partition and developers must make sure that the data stays up to date whenever they access it. CUDA has no explicit flush operation for such copies, and developers must re-copy the values to be sure they are up to date.

CUDA obviously works well for non cache coherent memory spaces, yet pure barrier synchronization will hardly be able to keep all processors of an exascale system busy and the costs for barriers scales with the number of cores to synchronize. Current GPU programming models lack efficient synchronization constructs to be a good match for our hardware model.

5 PGAS programming models

In this section we first give an introduction to the PGAS model and discuss common variations of it. We use our abstract hardware model even though some PGAS variates have defined their own hardware model. The PGAS model itself defines a shared address space for all modules of the system, so that each thread on each module can access every data element, even without shared memory

at hardware level. The global address space is partitioned with one partition per module, which is the local memory of that module. Accessing data within the local partition is obviously faster than accessing data from remote partitions and developers should maximize local memory accesses. In general, data are created on one module and live there for their lifetime. Data structures may be local to one module or distributed among multiple modules, e. g., an array can be split in stripes and the stripes be distributed, however data may not be redistributed. Most systems require the user to make sure that its algorithm matches the data distribution in a way local data reads are maximized or performance may suffer greatly. PGAS variations are often classified into the standard model and the so called asynchronous PGAS (APGAS) model [22]. The standard model defines a SPMD execution model so every module executes the same program, whereas the APGAS models allows to dynamically spawn tasks/threads on specific modules. This difference, however, is not important for the rest of this work and not further discussed.

One well known realization of the standard PGAS model is the Unified Parallel C (UPC) programming language. In UPC memory is divided into thread local private memory and shared memory accessible by all threads. Reads and writes to shared memory are done with standard read-/write operations. Arrays can be distributed in various ways, e. g., cyclic, block-cyclic or blocked. Furthermore, UPC allows developers to specify the memory consistency model on a per memory access basis, that is, each individual memory access operation can be relaxed or strict. Strict operations act like a memory fence and previous operations must be finished before the strict operation is completed. For synchronization among threads, UPC provides locks and barriers, which use strict memory accesses.

IBM's X10 is an APGAS language that does not directly allow to access remote memory, but for its APGAS nature it is possible to spawn tasks with a data payload on a remote module and that way access remote memory. Synchronization in X10 uses the atomic statement which enforces module local synchronization similar to e. g., the OpenMP critical construct – that is, both read and write accesses to data shared by multiple tasks must be within an atomic statement. X10 memory consistency is therefore similar to that of shared memory programming models, yet it is not defined in the specification.

Cray's Chapel is another APGAS language, but allows to directly access remote memory. For synchronization, Chapel supports different versions of synchronization variables with a full/empty state including one with a single assignment syntax. Accessing such synchronization variables acts as a fence operation, similar to UPC's strict memory accesses. Furthermore Chapel plans to support distributed software transactional memory, which however is currently in an experimental stage [24].

XcalableMP is a rather new directive-based PGAS language extension for C and Fortran. Version 1.0 was released in November 2011. XcalableMP calls its execution model SPMD, but allows APGAS like spawning of tasks on specific modules – implemented similarly to OpenMP tasks. Following the PGAS model, memory is divided into private and shared memory with the latter being distributed across all modules based on so-called distributions. A module can only directly access data that is stored in its private or the local partition of shared memory. In the so called global programming model of XcalableMP one can access remote shared memory by explicitly marking the access as remote (`gmov`) or have the runtime system automatically copy it to that module. The copies are not kept consistent, that is one must manually make sure data are copied again if needed. XcalableMP also supports the so called local programming model, which resembles the coarray model of Coarray Fortran. Data of coarrays can be copied into a local array, which however must explicitly be synchronized by a `sync_memory` function. The only global synchronization primitive available is a barrier, which is the only point at which memory is consistent among all modules. Table 1 gives an overview of the different programming models.

All existing models handle memory consistency, remote memory accesses and global synchronization differently, however all three aspects are important for high performance in a PGAS environment. Chapel and UPC require the developers to manually use locks or synchronization variables and all memory is consistent at these synchronization points. Every time such a synchronization point is reached a flush must be done, that is all local data modifications must be written back to their correct memory location and furthermore all reads after the synchronization point must access the original location. Thus a remote value may be transferred over the network, again, even

if it is unchanged. It is currently not possible to prevent such communication overhead, except by manually caching data locally, which is cumbersome.

Systems disallowing direct remote memory accesses do not face these issues, but require developers to manually mark remote accesses. In X10 remote memory access operations must be moved into remote tasks, and use local synchronization on the remote module. In XcalableMP remote reads must be explicitly marked and a form of global synchronization is required to make sure to read an up-to-date value. Both X10 and XcalableMP do not directly take care of global memory consistency, but impose this task on the developer. Managing memory consistency can be rather complex and error-prone, depending on the concrete application.

As PGAS is expected to scale to exascale systems, fine grained synchronization among modules is required to reduce idle time of processors. Pure barrier like synchronization will most likely not scale. As discussed above, direct access to remote memory obviously complicates the memory consistency model, but allows easy employment of remote DMA transfers. RDMA supports high-bandwidth low-latency memory transfers and furthermore often removes the need of copying data into buffers. However, flushes with direct remote memory accesses can easily generate a high amount of network traffic not obvious to developers complicating program design.

The PGAS concept by itself is a good match for our hardware model, as it allows to easily control NUMA issues and is targeted at non-cache coherent distributed memory systems. However at the time of writing the upcoming systems lack an efficient memory consistency model and synchronization is often cumbersome. Furthermore the current PGAS variants have no direct way of increasing on-chip memory efficiency.

6 Our model - Pairwise synchronization

Our model is designed to allow efficient usage of hybrid clusters and must therefore solve the issues discussed in the last sections. For simplicity our model expects the same program running on all modules, however it can most likely be extended to an APGAS model. We divide memory in private and shared memory and allow direct remote shared memory accesses. Remote accesses are not explicitly marked as such, similar to UPC. We only discuss shared memory and expect private memory to be used as it is typically done for the respective module type, e. g., one can use OpenMP for a multicore module.

In contrast to the standard PGAS model we directly incorporate thread synchronization within our model. Synchronization is tied to reading/writing shared memory. Shared data are single assignment, so only one thread can write once to a specific shared memory location, whereas it can be read multiple times by all threads. As memory is single assignment, we can define two states for any memory location:

- memory is uninitialized or
- it contains data.

Synchronization is based on these states following two rules:

- If a thread tries to read from unitialized memory, it is blocked.
- A thread writing data will unblock all threads waiting for the written data.

Following these rules threads can only read memory that contains data and data cannot be overwritten. Therefore, it is impossible to have race conditions and there is no need for a complex memory consistency model. We expect this form of synchronization to simplify development. As already mentioned, we could use our form of synchronization on bit level, but that is not feasible and unnecessary for most problems. We call the batch of data to which synchronization is applied a *synchronization unit* and its size the *synchronization size*. We refer to this concept as *synchronization granularity*. For example, consider an algorithm using tiled matrices. The synchronization size can be identical to the tile size by which a tile becomes a synchronization unit. As a result, threads are

only able to read data of a completely written tile. Working on forms of data blocks is essential for most hardware architectures and our hardware model. We expect explicit blocking to be easier to use than an implicit mechanism as it is done for current multicore CPUs by rearranging memory accesses. Reusing our blocking concept for synchronization again emphasizes the importance of working on blocks of data. The synchronization size can differ for different data and the optimal synchronization size depends on the algorithm and hardware used. For example Cray's XMT architecture [8] supports full/empty bits for memory in hardware, which may most likely allow smaller tiles than on, e. g., x86 systems.

Furthermore our model provides for a way of hiding latency by oversaturating the modules with threads and suspending threads waiting for data. This technique follows a similar pattern as it is e.g. widely used by GPUs for hiding off-chip memory access latency. Oversaturating the system is required for problems with irregular memory access, as in these scenarios threads may be blocked for a rather long time waiting for a thread to write data. In this paper we concentrate on regular workloads and leave irregular workloads for future work.

In contrast to e. g., UPC, developers must no longer explicitly synchronize, and in contrast to X10, one must no longer explicitly move accesses into remote tasks. In Chapel one can achieve similar behavior by using synchronization variables, but Chapel itself does not support the concept of synchronization granularity nor does guarding chunks of memory with synchronization variables allow the system to cache the guarded memory without in detail code analysis, whereas single assignment memory of course allows us to have temporary views of data in all memory spaces of our hardware model. OpenMP also does not allow the system to cache values, as all must be reread after a flush. CUDA and OpenCL do not allow this kind of fine-grained synchronization.

7 Our model - Data distribution and global communication algorithms

In this section we discuss possible data distributions and optimized global communication algorithms. Our concept allows to use algorithms from MPI collective operations with only small constant overhead. This section again considers a partitioned global address space.

Data distribution by itself is fairly simple, a synchronization unit can be in the local module or in a remote module. A data structure can combine both types of placement e. g., the synchronization units of an array can be distributed among all modules. Regardless of its location, a synchronization unit can be read by all modules. In case a module has successfully read data from a remote module it can keep a temporary view in all its memory spaces without ever having to reread the value. Based on this, we can also define data that is in all partitions, that is the data are local to all modules. We call this distribution *everywhere* and it is expected to be implemented by having writes broadcast the synchronization unit to all modules. The implementation can of course use the broadcast algorithms used in MPI, however in contrast to MPI our broadcast implementation is not collective. Only one module writes a synchronization unit and the other modules do not directly participate, but internally e. g., a RDMA transfer can write the data to the correct memory location. In general, MPI global communication algorithms rely on the fact, that every module calls a collective operation. We can circumvent this requirement by using so called active messages. Active messages automatically trigger code execution when a message is received at a module, which can be used to continue distribution of data. We discuss active messages in detail in our Sect. 10.

Scatter is modeled by copying local data to a distributed data structure e. g., a local array is assigned to one with stripe-wise distribution. This is a straightforward approach, yet still allows us to use global communication algorithms. Gather is modeled the other way round, that is by assigning a distributed array to a local one. The optimal communication algorithm obviously depends on the concrete distribution of data, but in all cases the MPI algorithms can be used.

To allow efficient reduction, we extend our model by an approach similar to combining PRAMs. We add so called *reduction variables* to our model. Reduction variables closely follow the concept of our single assignment memory, except that all modules must have written to a synchronization unit before the value can be read. The reduction is performed while the data are written and the

reduction operation is defined per reduction variable, so e. g., one variable can be used to compute a sum and another computes a minimum. One can also define an array of reduction variables, in which multiple reduction variables are in the same synchronization unit. Such an array is reduced by reducing whole synchronization units that is as soon as every module has written to all variables in one synchronization unit the variables are reduced.

Scan is modeled similarly, except that it is not be defined on a single element, but an array. A scan array consist of one synchronization unit per module. The values of synchronization unit i can be read after values have been written to all synchronization units with indexes $\leq i$.

Data distribution is independent from the scan and reduce behavior, so synchronization units of e. g., reduction variables can be distributed among the system or use the everywhere distribution, in which case MPI all-reduce is accomplished.

8 Basic data structures

In this section we describe a set of basic data structures realized using our model. We refer to the data structures throughout the rest of the paper and have implemented them in our prototype library. The model itself is not limited to these data structures.

First, for a tile based *local matrix* all data is stored on one module only. The data is stored in tiles, which are the synchronization units. The data of such a local matrix can be accessed by all threads knowing their address, which however is not automatically known to all threads. We use a vector with the everywhere distribution called shared vector (see below) to give all threads the address of the matrix. We could change our model and provide a common namespace for all variables, however this would complicate the implementation of the model as a C++ library by a great deal and hide possible important communication costs. Future work may decide to lift this limitation.

The *shared vector* is a vector using the everywhere distribution. The creation of a shared vector must be done by all modules collectively and results in communication, in contrast to the creation of a local matrix. To give other threads access to a local matrix the creator thread must write its address into a previously created shared vector.

The third data structure is called *distributed matrix*, and it is essentially a combination of both previous data structures. The distributed matrix distributes its data on all modules, by using a shared vector and local (sub-)matrices on every module. The vector stores the addresses of all local matrices, which enables every thread to read all data. The data structure supports arbitrary data distributions, for which one must only define a mapping of the global matrix indexes to sub-matrix indexes.

As an optimization, all data structures cache remote data locally.

We have defined our model without a notion of groups of modules. Future work may introduce module-groups into our model, which is obviously a requirement for exascale systems. Other PGAS programming models do not support groups at the time of writing either, nor do OpenMP or CUDA.

As an example one can use our data structures to parallelise I/O and computation in matrix multiply, i. e., one module is used for I/O and multiple modules for computation. The two input matrices can be stored as distributed matrices that equally distribute the data among the compute modules and no data on the I/O module, whereas the result matrix is local to the I/O module. The I/O module reads the input matrices from file and writes them into the input data structures. Storing the data of the input matrices on the compute modules allows the input modules to use remote writes, which can easily be implemented as fire-and-forget operations and are therefore more efficient than remote reads. As soon as the input module has read the input matrices, the module can start writing the result matrix to file. The compute modules will start computing by reading from the input matrices and writing the results into the result matrix. Synchronization is implicit and compute modules are blocked in case the required tile is not yet read from file, whereas the I/O module is blocked in case the required result is not yet computed. Storing the result matrix on the I/O module again allows for remote writes. Developers may choose to use a module with throughput cores for their computation.

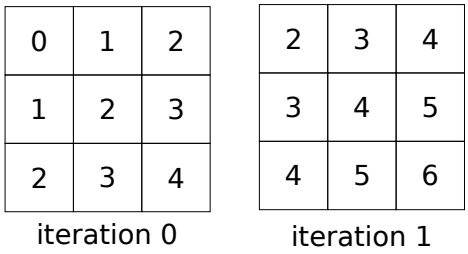

Figure 2: Wavefront pattern for multiple dimensions. The elements with the same number can be computed in parallel.

A common issue with single assignment memory is its high memory requirements since memory cannot be reused. Many scientific algorithms use double buffering, that is after every iteration input and output are swapped and memory is reused. It is possible to allocate new memory for every iteration in our model, however depending on the overhead of memory allocation this may not result in reasonable performance. To prevent unneeded reallocation, we allow modules to mark data structures as unused. If all modules have marked a data structure as unused, it can be reused. If a thread tries to reuse a data structure before all modules have marked it as unused, it gets blocked. This technique allows direct use of double buffering and gives all modules the ability to clear their local caches, since they must actively participate in reuse. This feature has been put in place to allow easy implementation of the model and may be removed at a later date. For example efficient memory pools may minimize the overhead of memory allocation so reallocating memory may be feasible. When a data structure is in its unused form it is possible to change the data distribution. Implementing this feature is rather complex and it is therefore not yet available. The ability to change the distribution of an existing data structure is considered future work and evaluated together with techniques to solve the high memory requirements, however we discuss its potential usage in the next section.

9 Example Algorithms

In this section we discuss four algorithms:

1. Gauß-Seidel stencil
2. Scan (all prefix sums)
3. Bitonic sort
4. FFT

Gauß-Seidel stencil The Gauß-Seidel stencil is well known and requires a non-trivial parallel solution. The calculations of the stencil are applied on a two-dimensional data matrix V of the size N^2 with the borders having fixed values. The non-border element with the coordinates (x, y) in the kth iteration is calculated by

$$V_{x,y}^k = \frac{V_{x-1,y}^k + V_{x+1,y}^{k-1} + V_{x,y-1}^k + V_{x,y+1}^{k-1}}{4}.$$

The computation is done row-wise starting at the top left non-border element $(1, 1)$ and is repeated for a fixed number of iterations or until it converges. We discuss the version running I iterations. We furthermore expect $I > N$ for our algorithm analysis. It is important to notice that the upper $(x, y - 1)$ and the left $(x - 1, y)$ values are from the current iteration, whereas the right $(x + 1, y)$ and bottom $(x, y + 1)$ value are from the previous iterations. These data dependencies result in a non-trivial parallel solution. It is possible to compute

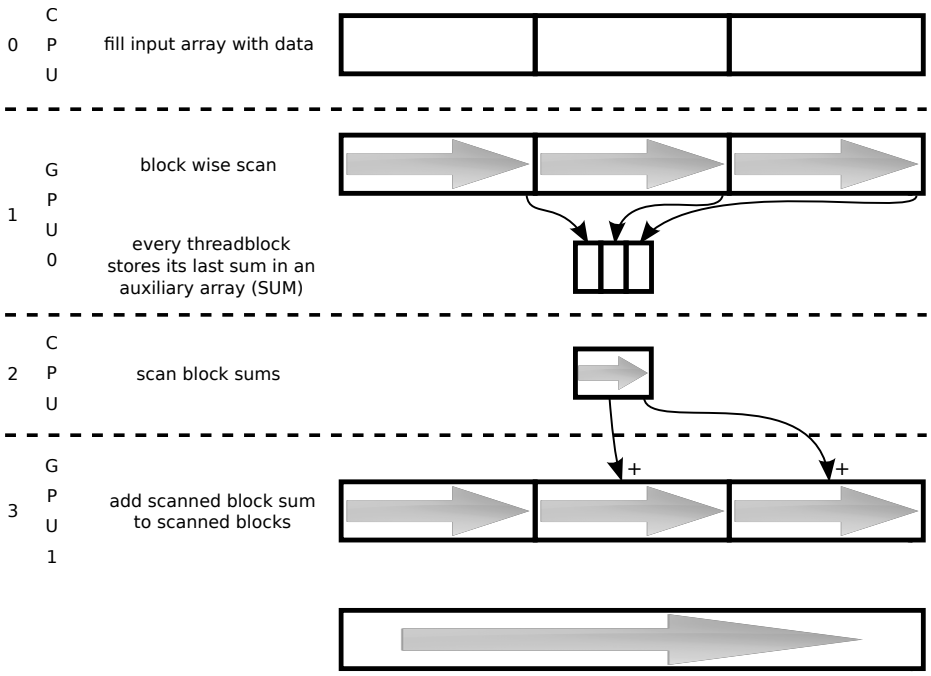

Figure 3: Overview of the scan algorithm.

multiple elements in one iteration in parallel and to compute multiple iterations in parallel. Both dimensions of the parallel solution follow the wavefront pattern [21]. See Fig. 2 for a visualization of which elements can be calculated in parallel. In one iteration only the elements in the same diagonal can be processed in parallel. Computing multiple iterations in parallel follows the same pattern, but the next iteration is always two diagonals behind the previous one. We use both dimensions in our solution.

Our solution uses one distributed matrix for every iteration. We expect the program to run at P modules. The matrices are distributed by giving each module a batch of rows, that is the first module stores the top $N/P = m$ rows, whereas the next module stores the next m rows. For simplicity we expect m threads to be available at each module. Every module will compute its local m rows for all iterations by using one thread per row, for less threads the m rows can be split equally among the available threads. Synchronization among the threads on different modules is done automatically, as threads reading matrix tiles not yet calculated will be blocked until the data are made available. We use the same synchronization among local and remote threads as the whole matrices are stored in shared memory. This form of pair-wise synchronization between the threads will not result in a strict wavefront pattern, but threads may calculate ahead. Overall this algorithm requires $O(I * \frac{N^2}{m})$ network transfers – the first and last rows of a module must be transfered to the neighbor module for every module except the first and last one. Furthermore it requires up to $O(N^2 * m)$ memory, when using a form of multi-buffering with m matrices. When using m matrices at the same time a thread does not need to wait for memory to be available to be filled, yet the memory requirement is rather high. It is possible to further limit the number of of matrices, however this will not only reduce the total memory requirement but also limit parallelism. As the algorithm works on matrix tiles it can easily utilize on-chip memory.

Scan All prefix sums – also known as scan – is an important parallel building block used in a wide variety of algorithms. Scan takes an input array $[x_0, x_1, ..., x_{n-1}]$ and a binary associative operator $+$ with the identity I as input and returns $[I, x_0, (x_0 + x_1), ..., (x_0 + x_1 + ...x_{n-2})]$. Harris et al. [13] suggested a work-efficient implementation for CUDA, which is split in three steps. In the first step the whole array is subdivided into blocks and a local scan is performed on

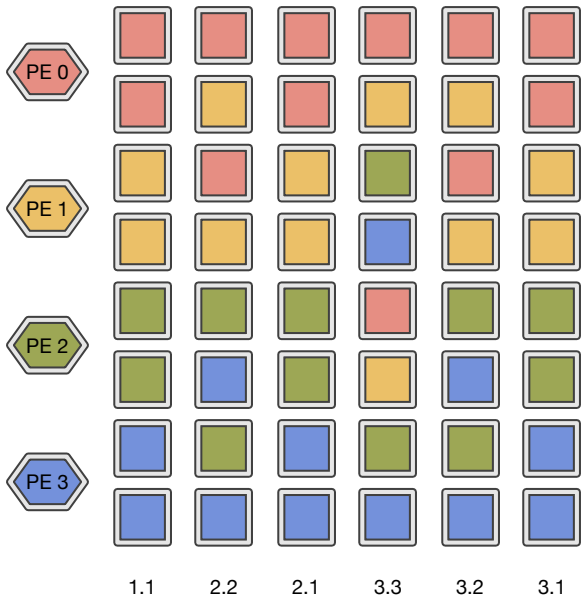

Figure 4: Bitonic sort. The color shows both the data distribution and which module reads which data elements, that is data read is always local but may be written to a remote partition.

every block. In this step the block sums of every block are written to an auxiliary array (SUM). In step 2, SUM is scanned and in a third step, the result of the scan are used to update the original array, so it contains the final result. Figure 3 gives a basic overview of the algorithm in which we added the filling of the input array as a step 0. We discuss a manual scan implementation in our model despite the fact that we have defined a global communication algorithm simply because it is a well known algorithm allowing demonstrate use of our model.

We can parallelize scan by using e.g. one vector and one module per step. Synchronization is done automatically. As soon as module 0 has generated the first batch of data in step 0, module 1 can start to compute the block wise scan. Module 2 can start as soon as the first block sum is available and module 3 starts computing after the first block sums are scanned. Thereby all four steps form a pipeline and can be executed in parallel. Scan is often used as a final processing step, so that step 0 takes the most time and we could use multiple modules to compute step 0.

Bitonic sort Bitonic sort by itself has rather high communication costs and is therefore most likely not a good choice for large non cache coherent systems. Nevertheless we consider it a good example to illustrate use of our model due to its non-trivial data dependencies shown in Fig. 4. We start by using one vector per iteration and discuss optimization by multi-buffering afterwards. We ignore the start of the algorithm, which merges single elements without the need of communication, and begin our discussion when two synchronization units are to be merged. We expect there to be N elements to sort and P modules. Furthermore we expect P to be a power of two and that the array consists of $P * 2$ synchronization units. In step 1.1 of the algorithm module p merges the synchronization units $2 * p$ and $2 * p + 1$. Step 2.2 then merges elements further away from each other, whereas step 2.1 is essentially identical to 1.1. This pattern continues with 3.2 being identical to 2.2 and 3.1 to 2.1 and 1.1. The access pattern changes over time and we choose a distribution allowing for remote writes and local reads. The distribution follows the colors in Fig. 4, that is, in the vector of step 1.1 the first and second blocks are local to module 0, whereas in step 2.2 the first and third blocks are local to module 0.

We can apply a form of multi-buffering by allowing the distribution to be changed whenever the vector is unused. We consider changing the distribution to be simpler than modifying the

access pattern of the algorithm, which obviously is also a possible solution to that problem. Overall the algorithm has $O(log^2 N)$ steps and thereby $O(N * log^2 N)$ communication costs. Note that in contrast to most bitonic sort implementations we use pair-wise synchronization and no barriers.

FFT FFT has the so called butterfly data dependency graph, which is similar to the one of bitonic sort. One can create the butterfly graph from Fig. 4 by removing steps 2.1, 3.2, 3.1. We can therefore reuse our approach for computing an FFT as well.

10 Implementation

Our implementation is currently only a proof of concept and has not been optimized for maximum performance and consists of two branches one for hybrid CPU/GPU systems and one for distributed memory systems. We first describe the generic approach for both hybrid and distributed memory systems and afterwards detail the concrete implementations.

We have implemented our model in a set of data structures described in Sect. 8 available as a C++ library. The library has been tested on a x86 based cluster and on NVIDIA GPUs. As the current target architectures do not have direct support for our synchronization mechanism, we implemented it by adding a flag, to each synchronization unit. The flag is stored directly in front of the synchronization unit. We put a memory barrier before setting the flag to make sure all data is written before another thread can read the flag as available. The flag is set using volatile variables, so the flag itself is directly made available. The flag need not be set atomically, as in a correct program it is not changed by multiple threads, however our implementation uses atomic operations to try to detect possible developer errors. In case two threads try to change the same flag, an assertion is triggered and the program is stopped. It is only a best effort approach, as there is no atomic operation for both CPUs and GPUs in current hardware.

Our flag based implementation is rather lightweight. The costs for writing a synchronization unit are increased by the cost of setting the flag. There is no contention on modifying the flag nor will there be any false sharing with reasonable large synchronization sizes. In case the flag is not yet set, reading a synchronization unit will spinloop until the flag is set. Note that after the loop reads may be cached, as data is single assignment.

In our library, data placement is managed by distributions, which essential follow the concept described in Sect. 7. Distributions must be supplied whenever a data structure is created. They are used as both an allocator and a random access iterator, that is the distribution allocates all data required for the data structures and manages all accesses to it. Data access is managed by three basic functions:

- `get()` manages reads of the synchronization unit and blocks in case data is not yet written.
- `get_unitialized()` returns a pointer to the location data can be written to.
- `set()` marks the synchronization unit to after all writes have finished.

Data writes are in general done in three steps.

1. `get_unitialized()` is called. The functions returns a pointer to a synchronization unit.
2. Data is written to the pointer returned in step 1.
3. `set()` is called marking the data as available.

The optimal distribution most likely depends on the specific task, so we made distributions easy to create. We supply low level allocators and pointer like random access iterators, which take care of memory allocation and synchronization, as well as data access itself. When the low level constructs are used, a distribution must only implement a mapping of a linear address space to the allocated memory blocks.

As an example, a minimal distribution maps a linear address space to a single block of CPU memory, but the memory will still be accessible by all modules. A distribution could also allocate memory on multiple modules and distribute the data stripe-wise.

We use NVIDIA's CUDA to program hybrid systems and use its so-called *unified virtual address space*. By using the virtual address space, CPU main memory and the global memories of all GPUs are mapped to the same address space. In general each device can access every memory location:

- GPUs can access CPU main memory, if allocated as page locked memory, also known as pinned memory. Developers can allocate memory to be cache coherent with the CPU, or allocate memory as write combined, which prevents the CPU from caching it. Accessing write combined memory can result in a performance increase of up to 40% [18] for the GPU at the cost of increased CPU access latency.
- The CPU can access GPU memory by explicitly making a partial copy of GPU memory into main memory using a CUDA specific `memcpy` function.
- If both GPUs are Tesla cards they can directly read and write each others global memory.

Memory accesses from a GPU to main memory or from a CPU to global memory are DMA transfers over PCIe, which has a rather high latency and a maximum bandwidth of 8 GB/s. Remote memory accesses are therefore obviously more expensive than for hybrid multicore systems.

The concrete implementation using CUDA follows the concept outlined before and we can define distributions that e. g., store data on the CPU or distribute data among the global memories of multiple GPUs. Our hybrid implementation does not yet support optimized communication algorithms as described in Sect. 7.

Our implementation for clusters uses so-called active messages, a message based communication primitive for communication in distributed memory systems. In contrast to, e. g., MPI messages active messages do not only contain a data payload, but automatically trigger code execution when received. We use the active message implementation of the GASNet library, which works as follows:

1. Module A sends a request to module B. The request contains both a handler-id and a possible data payload.
2. When the request is received by module B, it automatically executes code defined by the handler-id to process the request. The code is executed in background and will not directly influence the threads of the main program written by the developer. The handler code can send a reply to module A. The reply again consists of a handler-id and possible data payload.
3. A reply at module A is handled identical to the initial request, but must not send any active message.

Memory reads from remote partitions are implemented by sending a request to the remote module including the memory address to be read. The remote module checks if the memory contains data. In case data are available, a reply with the data is sent. In case memory is unitialized, a thread waits until the data are available and afterwards sends them. Due to a GASNet limitation the thread cannot be awakened by a signal, but a spinlock must be used. As described before, we use flags to identify availability of data. To allow for high network utilization, we always send whole synchronization units over the network.

The optimized communication algorithms are implemented using the matching MPI functions. This is done by using an active message that triggers the MPI collective function call on all remote modules. Unfortunately this implementation using both the latest version of GASNet and Open-MPI is unstable and regularly crashes. We therefore also implemented inefficient trivial fall-back functions. Future work may directly implement more efficient algorithms.

Not all features of both of our branches work well with each other at the time of writing, for example the PGAS implementation cannot access memory of remote GPUs. Communication with remote GPUs is only possible by using main memory that can be accessed by the GPU as well. Future work will unify both branches and offer uniform memory accesses.

Problem size (in 256 element blocks)	50	100	200	400	500	1000
Time (ms)	5.833	11.595	23.537	46.528	58.126	116.958
Input (ms)	5.806	11.553	23.468	46.404	57.974	116.668
Pure computation (ms)	0.027	0.042	0.069	0.124	0.152	0.29

Table 2: Runtime for a single scan computation with different input sizes.

11 Experiments

We implemented the scan algorithm using the GPU based branch and the Gauß-Seidel stencil using the distributed memory branch.

Scan was tested on a system consisting of an Intel Core i7 920 with both a NVIDIA GeForce GTX 480 and a NVIDIA Tesla C2050:

- Core i7 920 is a quadcore CPU based on Intel's Nehalem architecture running at 2.66 GHz.
- GeForce GTX 480 is a GPU based on NVIDIA's Fermi architecture and consists of 15 so-called multiprocessors each with 32 cores resulting in a total of 480 cores. The GPU has access to 1.5 GB global memory.
- Tesla C2050 is a GPU based on NVIDIA's Fermi architecture as well, but only consists of 14 multiprocessors resulting in a total of 448 cores. The GPU has access to 3 GB of global memory.

Our scan implementation uses the CPU for filling the input array and the scan of the block sums (Fig. 3: steps 0 and 2), whereas one GPU performs the block wise scan and the other produces the final results. This way the system automatically forms a pipeline. Our scan code uses 5 vectors, all storing the data in pinned main memory:

- The input array filled in step 0 uses a synchronization size of 256, which is reasonable for a block local scan on the GPU.
- The scanned blocks are stored in another vector, again with a synchronization size of 256.
- The block sums are stored in a vector with a synchronization size of 1, so the CPU can compute the scan as soon as possible.
- The scanned block sums are again stored with a synchronization size of 1.
- The final result vector uses a synchronization size of 256.

Whereas this distribution of work does not provide high performance on the used system due to the slow communication link, it demonstrates that using multiple modules in parallel must hardly increase overall programming complexity. As a downside, the algorithm uses a rather high amount of memory. Previous dataflow architectures solved such issues by treating all but the final result as transient memory, which was automatically removed when no longer needed. Implementing such a feature in software may turn out to be rather complex and future work is required to identify if, e. g., a reference counting approach offers sufficient results. Experiments with reference counting in the PGAS model seem to provide a reasonable usability. The current implementation requires the program to free the memory when it is no longer used, e. g., in the scan algorithm the input data can be deleted as soon as the block sum scan is complete.

Scan is normally computed on intermediate values to get the final result. To simulate such a computation we slowed down the generation of input by doing 50,000 additions per generated tile. We measured performance of our implementation with several input sizes (Tab. 2). The table shows:

Time The total running time including generation of input data.

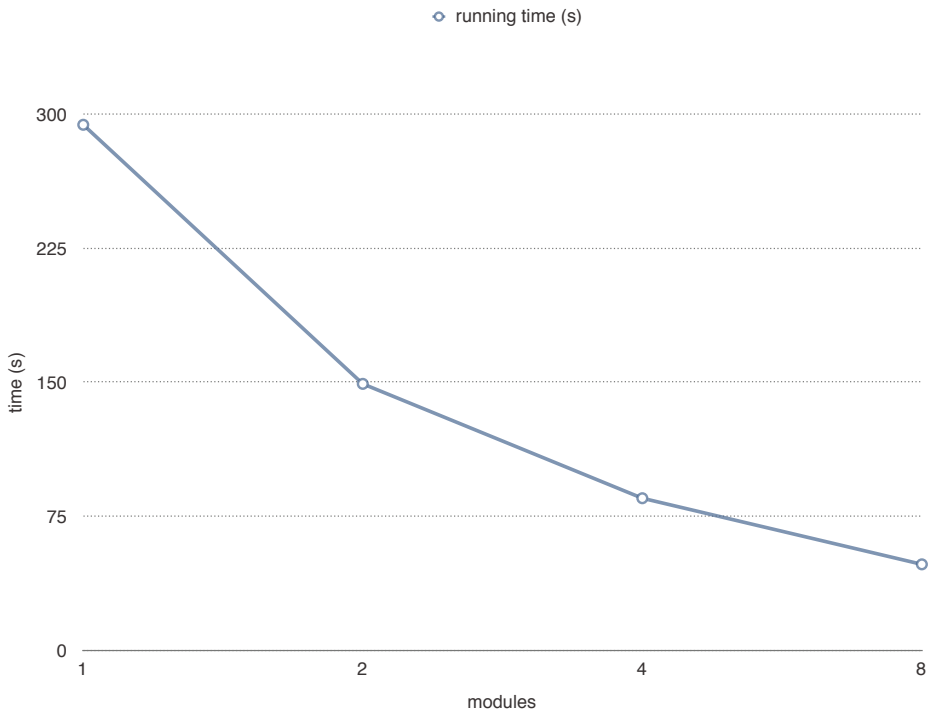

Figure 5: Gauß-Seidel performance for a matrix size of $32768 * 32768$

Input The time required to generate the input data.

Pure computation The time required to complete the computation after the input was generated.

We can see that the time required for input generation increases with larger problem sizes, but pure computation time remains constant. This gives a clear indication that the pipeline works as expected and almost all computations are done while the input data is generated.

The Gauß-Seidel stencil implementation in the distributed memory branch follows the schema outlined before. We used multiple distributed matrices to store the different iterations and reused them when no longer in use. Marking matrices as unused has been implemented with a shared counter that was increased every time a thread marked a matrix as unused. As soon as every module had increased the counter, the distributed matrix was reused.

We tested our implementation on a cluster connected by gigabit Ethernet. Each module is a NUMA system consisting of two six core AMD Opteron 2427 CPUs with 32 GB of RAM. The GASNet active messages are sent using an MPI back-end. This back-end is available for portability and does not offer the same level of performance as highly tuned back-ends such as the Infiniband back-end. Using Infiniband is expected to reduce network transfer time by a great deal, as e. g., Infinibands RDMA hardware support is a good match to the active message semantics[2].

Figure 5 shows the overall performance of the Gauß-Seidel stencil running at up to 8 systems. We used one CPU for simplicity and limited the number of threads to 4, as the PGAS system regularly spawns additional threads in background, which may have to spin-loop until data gets available. Synchronization of threads on the same module uses the same interface as on different modules, however internally no active message handler is called. Figure 5 shows the speedup and total runtime of our implementation. We can see that our Gauß-Seidel implementation scales well for up to 8 modules. As already said, our library is a proof-of-concept implementation and has not been tuned for absolute performance. The implementation on one module is rather efficient as it is lock and barrier free, however communication between different modules can be improved and

[2]http://gasnet.cs.berkeley.edu/performance/

we have therefore not tested the performance on larger systems. Future work, however, will try to increase performance and measure it on larger systems.

12 Related work

In addition to the programming models discussed before, the topic of this paper is related to a large spectrum of research. Dataflow hardware has been an active area of research in the 1970s and 1980s, as exemplified by the Manchester dataflow architecture [12] and the Goodyear STARAN [11]. From a current viewpoint these approaches were unable to scale effectively and were replaced by other architectures. However, the techniques partially returned in, e. g., out of order architectures, although at a smaller scale than originally planned. Single assignment memory is not used in mainstream languages, but of course in functional programming. For example, SISAL (Streams and Iteration in a Single Assignment Language) [10] is a functional programming language using implicit parallelism extracted from a dataflow graph. SISAL relies on compilers to generate dataflow graphs and targets SMP systems. Haskell uses so-called MVars [20], which are atomically filled communication channels between threads that block the reading thread in a similar fashion to our single assignment memory, however this mechanism is not targeted at distributed memory systems either and MVars may be refilled. One of the first hardware architectures supporting synchronization bits was the Denelcor HEP [23], which marked values as unavailable after they have been read. After working on the HEP, Smith has been following the concept of tying synchronization to memory accesses for a longer time [2], yet the concept has not been used in any major HPC programming system. A modern architecture supporting synchronization bits is Cray's XMT [8]. We are not aware of any previous work combining global communication algorithms with synchronization variables. MPI 3 will feature improved one-sided communication, but the specification is not yet available.

13 Conclusion

This paper consists of three major contributions. We described a new abstract hardware model capturing on current predictions for future hardware. The model emphasizes the problems arising from the memory subsystem getting more complex, possibly missing cache coherency and NUMA effects. We used our model to judge how well current programming models match the anticipated future hardware. We furthermore described a novel approach of integrating synchronization in PGAS languages that allows efficient communication and synchronization for various hardware platforms. Our programming model is designed to be able to cope with upcoming hardware restrictions and to be easy to use. We use a global partitioned address space to cope with NUMA effects and tie synchronization directly to accessing single assignment memory. Threads are suspended when they try to read data that has not yet been made available. This approach removes the need for a complex memory consistency model and still allows caching of remote data on non cache coherent systems. We furthermore provide global communication algorithms as they are used in MPI collective operations. The model has been implemented in a proof-of-concept library and we have shown that it yields reasonable speedups. While the library is only a prototype, we believe our model to be an easy to use approach for programming non-trivial data parallel applications.

Future work has been outlined throughout the paper. Since the model is rather new, future research must address a wide variety of topics, for example test our model on hybrid multicore chips, tune the implementation, and test its performance on larger systems.

References

[1] Dennis Abts, Steve Scott, and David J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[2] Gail Alverson, Robert Alverson, et al. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the 6th International Conference on Supercomputing*, pages 188–197.

[3] AMD. AMD Accelerated Parallel Processing OpenCL., 2011.

[4] AMD. AMD Fusion Family of APUs, 2011.

[5] Dan Bonachea. GASNet Specification, v1.1. Technical report, 2002.

[6] William W. Carlson, Jesse M. Draper, and David E. Culler. Introduction to UPC and Language Specification, 1999.

[7] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51:559–572, 2007.

[8] Cray Inc. Introducing the Cray XMT Supercomputer, 2010.

[9] Jack Dongarra. Impact of Architecture and Technology for Extreme Scale on Software and Algorithm Design, August 2010. Euro-Par 2010 keynote.

[10] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, 10:349–366, 1990.

[11] Goodyear Aerospace Cooperation. STARAN APPLE Programming Manual, 1972.

[12] J. R Gurd, C. C Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Commun. ACM*, 28:34–52, January 1985.

[13] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[14] Jim Held. "Single-chip cloud computer", an IA tera-scale research processor. In *Euro-Par 2010 Proceedings of the 2010 Conference on Parallel Processing*, pages 85–85, 2011.

[15] Intel Corporation. Cluster OpenMP User's Guide, 2006.

[16] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the 32nd annual international Symposium on Computer Architecture*, pages 408–419, 2005.

[17] Wei Liu, Brian Lewis, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Sai Luo, and Bratin Saha. A balanced programming model for emerging heterogeneous. multicore systems. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, 2010.

[18] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. Version 4.1, 2012.

[19] The OpenCL Specification. Version 1, revision 43, May 2009.

[20] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[21] Gregory F. Pfister. *In search of clusters (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[22] Vijay Saraswat, Gheorghe Almasi, et al. The Asynchronous Partitioned Global Address Space Model. In *1st ACM SIGPLAN Workshop on Advances in Message Passing*, 2010.

[23] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Society of Photo-Optical Instrumentation Engineers Conference Series*, pages 241–+, 1981.

[24] S. Sridharan, J.S. Vetter, et al. A Scalable Implementation of Language-Based Software Transactional Memory for Distributed Memory Systems. Technical report, 2011.