IEEE Copyright Notice

This work has been published in “2016 Fourth International Symposium on Computing and Networking (CANDAR)“, 22-25 Nov, 2016, Hiroshima, Japan.
DOI: 10.1109/CANDAR.2016.0071

http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7818641

PGASUS: A Framework for C++ Application Development on NUMA Architectures

Wieland Hagen, Max Plauth, Felix Eberhardt, Frank Feinbube and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam
Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

***Abstract*—For the implementation of data-intensive C++ applications for cache coherent Non-Uniform Memory Access (NUMA) systems, both massive parallelism and data locality have to be considered. While massive parallelism has been largely understood, the shared memory paradigm is still deeply entrenched in the mindset of many C++ software developers. Hence, data locality aspects of NUMA systems have been widely neglected thus far. At first sight, applying shared nothing approaches might seem like a viable workaround to address locality. However, we argue that developers should be enabled to address locality without having to surrender the advantages of the shared address space of cache coherent NUMA systems.**

Based on an extensive review of parallel programming languages and frameworks, we propose a programming model specialized for NUMA-aware C++ development that incorporates essential mechanisms for parallelism and data locality. We suggest that these mechanisms should be used to implement specialized data structures and algorithm templates which encapsulate locality, data distribution, and implicit data parallelism. We present an implementation of the proposed programming model in the form of a C++ framework. To demonstrate the applicability of our programming model, we implement a prototypical application on top of this framework and evaluate its performance.

I. Introduction

The prevalence of Non-Uniform Memory Access (NUMA) architectures in server systems has soared since Uniform Memory Access (UMA) multiprocessor systems have reached their scalability limits several years ago. Modern NUMA architectures support much higher processor counts and larger memory dimensions, all while maintaining a single cache coherent address space. This enables application developers to hold on to the shared memory paradigm that they are familiar with, even though the underlying hardware exhibits many characteristics of distributed architectures. The High-Performance Computing (HPC) community has yielded several programming models based on the shared nothing paradigm that addresses both massive parallelism and data locality. Ideally, however, combining the scalability of distributed systems with a familiar programming paradigm would be a perfect match for large-scale parallel applications; especially for those that are hard to implement on distributed architectures such as *de Novo genome assembly* [1].

In a preceding research project [2] which involved the parallelization of the Scale-Invariant Feature Transform (SIFT) [3] image detection algorithm on large NUMA-based systems, our experiments demonstrated that of the two most widely used frameworks to parallelize C++ applications, OpenMP and the Message Passing Interface (MPI), neither one is reasonably suited for the implementation of a large-scale parallel application on hierarchical NUMA machines. OpenMP lacks concepts of data locality and an adequate method to specify where to execute tasks. MPI relies on the shared nothing paradigm and thus abandons the benefits of a cache coherent address space. Furthermore, developers that are used to shared memory systems have to adapt to an unaccustomed paradigm, which substantially increases development time.

In an attempt to provide a remedy, we propose an interface that allows programmers to deal with the challenges introduced by large NUMA-systems more easily. Furthermore, we argue that these challenges should not be addressed orthogonally of each other, but tightly coupled inside specialized, NUMA-aware data structures and algorithm templates. Based on these findings, we present *PGASUS*[1], an early prototype of a C++11-based programming framework that implements central aspects of the proposed interface. Our prototype provides basic mechanisms for task-parallelism and memory placement and enables programmers to use these mechanisms to implement NUMA-aware data structures, which in turn can be used for application development.

This paper is structured as follows: Section II identifies recurring challenges associated with the development of scalable C++ applications for NUMA machines. Subsequently, Section III reviews the current state of the art of parallel and distributed programming languages and frameworks and identifies fundamental concepts employed by these approaches. Based on these concepts, a parallel programming interface for C++ NUMA applications is proposed in Section IV. In Section V we present the implementation strategies of our C++11-based programming framework prototype. Lastly, Section VI tests the scalability of the implemented framework before a conclusion is reached in Section VII.

II. Background: NUMA Challenges in C++

This section discusses the challenges faced during the development of parallel C++ applications for NUMA systems.

[1] https://github.com/wheeland/pgasus

A. Object placement

C++ has no concept of data locality and operates on a flat address space. There are five types of object allocation in C++.

1) Stack allocation: Variables local to a function call and temporary objects are created on the thread's stack.

2) new: The new operator calls `malloc()` to allocate memory. It can be overwritten globally or class-specifically.

3) Placement new: The placement new operator lets the programmer call the object's constructor on a supplied pointer.

4) Member: If the object is a member of another data structure, the object's constructor is automatically called by the enclosing object's constructor. The location of the object implicitly follows the layout of the enclosing class.

5) malloc: C code usually uses the dynamic memory allocation functions provided by the C standard library.

To influence memory placement decisions, either the new operator has to be overwritten or memory has to be allocated manually. However, neither method considers that every class used in a context sensitive to object placement is allocated using the modified new operation. This cannot be guaranteed for classes that are defined outside of the program, e.g. by libraries. Also, many template-based container data structures rely on the default behavior of `placement new` and implement their own memory management based on `malloc()`. Lastly, overwriting `new` has no effect on data structures implemented in C libraries that use `malloc()`.

B. Object migration

Object migration faces problems very similar to those described for object placement. Using the page migration mechanisms provided by the operating system to move an object to a different NUMA node raises many pitfalls. Consider the programmer wants to move an instance of `std::vector<std::string>` to another node. In order to do so, the pages containing the `std::string` instances must be moved to the other node, but also all those pages that hold the string data, which are separately allocated on the heap for each `std::string` (see Figure 1). One way to bypass this issue would be that all associated objects have to be allocated in contiguous memory and occupy a private set of pages that is not shared with other objects.

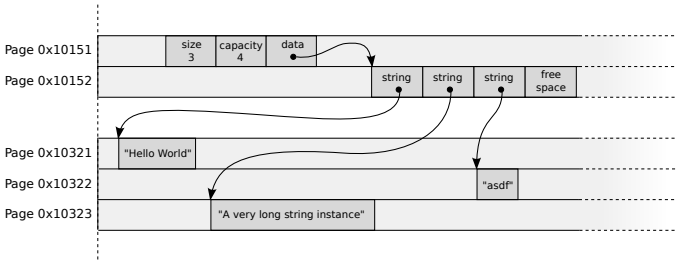

Fig. 1: An instance of `std::vector<std::string>` occupies many distinct pages, as the programmer has no influence on memory allocation placement using standard C++. Each pages may contain other, independent objects.

III. Related Work

In this Section, a number of programming languages, language extensions, frameworks and operating system mechanisms will be presented that exhibit promising features in the context of NUMA-aware C++ programming.

A. Programming Languages

1) X10: Being a popular representative of the Partitioned Global Address Space (PGAS) programming model, X10 [4] addresses many issues of large NUMA systems. It is based on the concepts *places*, *async*, *finish*, *atomic* and *at*. Variables reside in one place and are bound to that location. They can only be accessed directly from within that place. Accessing variables on remote places requires them to be exposed via global references and accessing them via place-shift.

2) Chapel: Chapel [5] introduces the construct of the *Locale*. Similar to *Places* in X10, a *Locale* consists of a memory region and one or more execution units. Each variable is associated with the locale it resides in. This data-to-locale mapping can be queried statically or dynamically at runtime.

3) Legion: Legion [6] is a programming system for heterogeneous distributed systems that defines logical regions. Logical regions may be partitioned into sub-regions and store data of arbitrary types and indexing schemes. The Legion runtime system maintains a task dependency graph and performs task scheduling based on this graph and on the logical region information stored with each task

B. Language Extensions

1) OpenMP: Since OpenMP lacks the concept of data locality, Muddukrishna et al. [7] presented an extension to OpenMP where the programmer has to specify which memory regions will be accessed for each task. ForestGOMP [8] [9] [10] is another extension to OpenMP based on maintaining a task-to-data association. Nikolopoulos et al. [11] presented an OpenMP extension that monitors page accesses via hardware instrumentation and, in regular intervals, migrates pages to those threads that access them most.

2) Unified Parallel C: Unified Parallel C (UPC) is an extension to C and employs the *SPMD* programming model [12]. UPC allows programmers to add the *shared* qualifier to variable declarations. While non-*shared* variables are private to each thread, the *shared* variables are exposed into a global address space and can be accessed by every other thread. UPC provides mechanisms to enforce a round-robin distribution of array data across all threads. It furthermore adds a *for_all* statement that allows the programmer to run a loop in parallel. Recently, the concepts of UPC have been adapted to C++, yielding UPC++ [13].

3) High Performance Fortran (HPF): HPF [14] is an extension of Fortran-90 that introduces implicit data parallelism by offering a mechanism to specify the distribution of multi-dimensional arrays onto nodes while preserving a global memory view. The HPF compiler is responsible for inserting primitives for inter-node communication where necessary. Iterations in HPF can be annotated to be run in parallel.

C. Frameworks

1) Message Passing Interface (MPI): The MPI standard is the de-facto standard for programming large distributed applications for cluster systems [15], [16]. MPI defines an extensive list of message-based communication patterns, including point-to-point and group communication, reductions and one-sided communication through so-called *Memory Windows*.

2) Threading Building Blocks (TBB): The C++ template library [17] provides a framework for task and data parallelism. It implements a task parallelism infrastructure, synchronization primitives, atomic operations and concurrent data structures. The framework is extensible, enabling programmers to implement their own parallel algorithms.

3) TBB-NUMA: Building up on top of TBB, Majo [18] presented *TBB-NUMA*, which extends the TBB scheduler by introducing task affinities that can be set either manually or automatically by annotating parallel constructs with distribution templates. These task affinities are then considered by the scheduler. The library also offers page-level data distribution primitives based on the `move_pages()` system call.

4) Standard Template Adaptive Parallel Library (STAPL): STAPL [19] offers container data structures called *pContainers* that follow the same interface as the ISO C++ STL components, but which provide parallel semantics. The programmer operates on these *pContainers* through views, the equivalent of STL iterators. Views can offer sub-views. Through this hierarchical structure, the required degree of parallelism for operations on the data structure can be adjusted. STAPL defines *pAlgorithms*, the equivalent of STL algorithms, that operate on *pContainers* in parallel. These algorithms are represented by task dependency graphs, where each task operates on a view. The underlying runtime system automatically chooses an implementation and a parallelization strategy depending on the type of operation and data structure.

D. Operating System Facilities

1) Carrefour: Dashti et al. [20] presented an extension to the Linux kernel that tries to optimize task and page placement for running applications. Carrefour collects a set of global, per-application and per-page statistics over time, which are then used to reason about employing one or more of the mechanisms *page replication*, *page interleaving*, *page co-location* and *thread clustering*. The authors provide benchmarks made on a four-node machine that show decent performance benefits for some applications when compared to the Linux 3.6.0 kernel.

2) Next-touch policy: Another mechanism that has been proposed by multiple authors [10], [21] is the next-touch policy. The next-touch policy ensures that data about to be accessed by a thread does actually reside on the correct NUMA node. Before accessing a data region, the programmer calls a routine which marks the pages belonging to the data region to be migrated, if necessary, to the NUMA node from where they are read or written next. This mechanism can be implemented using a combination of system calls or by modifying the kernel and providing a specialized system call for that purpose.

IV. Proposed Programming Model

The goal of this section is to formulate a programming model that simplifies the implementation of parallel C++ applications on NUMA architectures. We try to assist application developers by providing basic building blocks for memory allocation, placement, and migration, location-aware task parallelism and synchronization.

A. Essential concepts

1) MemSource: MemSources (Listing 1) are logical memory regions that can be used to allocate objects of arbitrary size and type. *MemSources* serve memory allocations from a contiguous block of memory and try to prevent intra-page fragmentation. They make sure that the memory of all allocated objects resides on one NUMA node and offer a mechanism to migrate all pages to another node.

```
1 int initialSize = 1 << 24;   // 16 MiB
2 MemSource msource =
3     MemSource::create(targetNode, initialSize);
4
5 Foo *foo = msource->construct<Foo>(); // create object
6 void *buffer = msource->alloc(1024); // allocate memory
7
8 msource->migrate(newHomeNode); // migrate pages
```

Listing 1: *MemSources* encapsulate memory allocation and deallocation and bind them to a specific NUMA node.

2) MemGuard: MemGuards (Listing 2) provide a mechanism to influence the behavior of `new` and the underlying `malloc()` calls by overwriting the memory allocation policy.

Following the *Resource Acquisition Is Initialization* (RAII) idiom [22], a *MemGuard* overwrites the behavior of `malloc()` until it exits scope or another *MemGuard* is specified. *MemGuards* can be configured to let `malloc()` draw memory from a specified node or *MemSource*, only.

```
1 // Make sure the instance is created on the target node
2 std::string* createRemoteString(Node targetNode) {
3     MemGuard guard(targetNode);
4     return new std::string("foo");
5 }
```

Listing 2: *MemGuards* configure the behavior of malloc() and assist programmers to specify object placement easily.

3) NUMA-aware Task-Parallelism: Tasks have priorities and can be created from functions or C++11 lambdas and may be bound to run on a particular NUMA node (Listing 3). Spawning a task returns a *Wait* object that can be used to wait for the task's completion and for retrieving its result. This task-parallelism interface is simple and versatile and can be used to implement other concepts of parallelism on top of it, like implicit parallelism or data parallelism.

```
1 auto task = async(targetNode, []() {
2     std::cout << "Executed on node "
3         << Node::current() << std:endl;
4     return 42;
5 }
6 // do something else in the meantime ...
7 auto result = task.wait(); // result = 42
```

Listing 3: Tasks can be spawned on a specified node using C++11 lambdas, enabling asynchronous and synchronous task parallelism.

4) Locality-aware Implicit Data Parallelism: Using the building blocks presented before, NUMA-aware concurrent data structures should automatically distribute their contents among the available NUMA nodes. The distribution strategy should be configurable by the programmer. Furthermore, these data structures should offer a method for executing a block of code for each element on the node that the element resides on. This is illustrated in Listing 4.

```
numa::HashTable<string,int> stringToIntMap;

// iterate over elements in parallel, ensuring locality
stringToIntMap.for_each([](std::pair<string,int> v) {
    std::cout << v.first << "-" << v.second << std::endl;
});
```

Listing 4: NUMA-aware concurrent data structures offer an interface for locality-aware implicit data parallelism.

B. C++ Data Type Properties

C++ types can be described by a number of properties based on their behavior and structure. We identify the following type properties to be relevant in the NUMA context:

1) Trivial Types: A *Trivial Type* is only defined by trivial operations (constructors, destructors, copy/move operators) and is non-virtual. Such a type can be safely allocated with malloc(), copied with memmove() and deleted with free(). Furthermore, *Trivial Types* are a true superset of the more well-known Plain Old Data (POD) types, which guarantee compatibility with C programs.

2) Resident Types: Resident Types store all data relevant to the functionality of the type, including all child objects, on the same *Home Node*. Virtual types store exactly one instance of their *vtable* in memory pages that are marked read-only, which eases caching of *vtables* on other nodes.

3) Migratable Types: In addition to the properties of *Resident Types*, *Migratable Types* implement a standardized mechanism that migrates all data to another node, thus changing their *Home Node*.

C. C++ Container Class Properties

Data structures storing elements of a given type can be characterized by a number of properties. We identified the following properties to be of relevance:

1) Indexing: The *Index Space* is the set of entities that can be used to index values in data structures. Three basic forms of *Index Spaces* can be identified: *Integral Ranges*, *Associative Indexing* and *Set Semantics*. Furthermore, advanced types of *Index Spaces* such as *Fuzzy Indexing* can be useful.

2) Concurrent Access: Data structures need to be designed in a way that allows highly concurrent access and if consistency can be guaranteed, concurrent modification. The actual implementation strategy strongly depends on the expected ratio between read, write, and possibly insert and delete operations.

3) Data Distribution: An important aspect of an NUMA-aware data structure is how items are distributed across the different nodes. We can identify a number of properties that may characterize this item mapping.

a) Node Distribution Ratio: Data structures distribute their items across NUMA nodes based on a ratio which may be explicitly specified by the programmer or derived implicitly from machine usage statistics such as average node utilization.

b) Item Placement Policy: Items are distributed across NUMA nodes according to value or index-based policies.

c) Distribution Balance: If the item distribution in a data structure diverges strongly from the specified *Node Distribution Ratio*, the data structure becomes *unbalanced*. Rebalancing operations might be triggered at regular intervals or at specific operations.

V. Implementation

This section presents *PGASUS*, a framework that implements the programming model outlined Section IV. The general architecture of *PGASUS* and the way it integrates into the software stack is illustrated in Figure 2. The framework is comprised of the following components:

1) topology: Provides information about the topology of the NUMA system by querying the `hwloc` library.

2) msource: Implements logical memory regions (*MemSources*), which are responsible for memory allocation. They reside on specific nodes and can be migrated to other nodes.

3) memalloc: Replaces all memory allocation functions defined in the C standard. The behavior of these functions is configurable for each thread independently via *MemGuards*.

4) tasking: Implements user-mode cooperative multitasking. Tasks can be configured with a priority and a target NUMA node to run on. The tasks are scheduled and dispatched by groups of worker threads. The processor cores that are used for worker threads can be configured via environment variables.

5) HashTable: An example implementation of a NUMA-aware concurrent data structure.

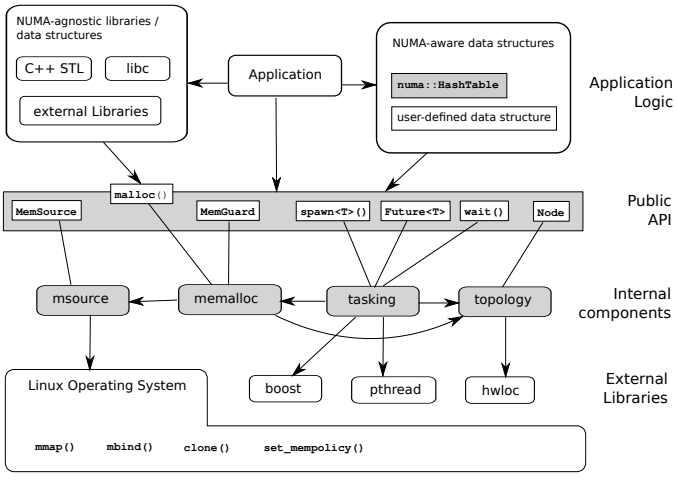

Fig. 2: Architecture of *PGASUS* and its integration in parallel applications. Rectangular boxes represent C++ interfaces, round-cornered boxes represent libraries or components. Gray entities are part of *PGASUS*. Arrows indicate usage.

A. `topology` component

The `topology` component exposes the machine model reported by `hwloc` using the `Node` and `NodeList` classes. It also considers node distances as reported by `/sys/devices/system/node/node[X]/distance`.

B. `msource` component

MemSources implement logical memory regions that provide memory allocation and deallocation and may be either *local* or *remote*. We decided to implement *MemSources* using the *dlmalloc* library, which employs a *Best-Fit* strategy [23] within the `mspace_t` constructs, which we will refer to as *Arenas*. Figure 3 illustrates a *Best-Fit* allocator after processing the operations in Listing 5.

```
1 int sizes[9] = {16,16,32,16,48,64,48,32,16};
2 void *p[9];
3 for (int i = 0; i < 9; i++) p[i] = malloc(sizes[i]);
4 free(p[0]); free(p[2]); free(p[6]);
```

Listing 5: A number of small allocations are made. Some of the allocated blocks are then deallocated.

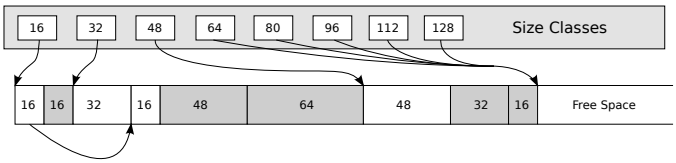

Fig. 3: Best-Fit allocator after running the code presented in Listing 5. Grey memory regions are allocated.

When a *MemSource* is created, it initializes a *Native Arena*. As soon as there is insufficient space available in any *Arena* within a *MemSource*, an additional *Arena* will be created. Smaller allocations are served through *Arenas*, whereas large allocations are created by the operating system via `mmap()`.

It is imperative that we can identify for each object the *MemSource* that it was allocated from, especially for location queries, migration, and de-allocation of objects. For that purpose, we store a *footer* element before each allocated chunk which contains a pointer to the *MemSource* and the *Arena* that the allocation was made from. For large `mmap()`-based allocations, the *footer* includes the block size and links to other mmap-allocated chunks, thus implementing a linked list.

C. `memalloc` component

The `memalloc` component implements the memory allocation functions defined in the C standard on top of the memory allocation mechanisms provided by *MemSources*. Every thread maintains a stack of places (*MemSources* or NUMA nodes) that can be used for allocations. For the latter case, each thread has a thread-local *MemSource* that is used whenever the node it resides on is specified at the top of the stack, or if the stack contains no places at all. This guarantees good scalability for regular local allocations. On every node we create one *remote MemSource* for each remote node, resulting in $n \times (n-1)$ *remote MemSources*. These places are used when a remote node is at the top of the stack.

D. `tasking` component

The `tasking` component implements a task parallelism environment for NUMA systems using user-mode cooperative multitasking. As illustrated in Figure 4, tasks are in one of the five states *Ready*, *Running*, *Waiting*, *Suspended* or *Completed* at any point in time.

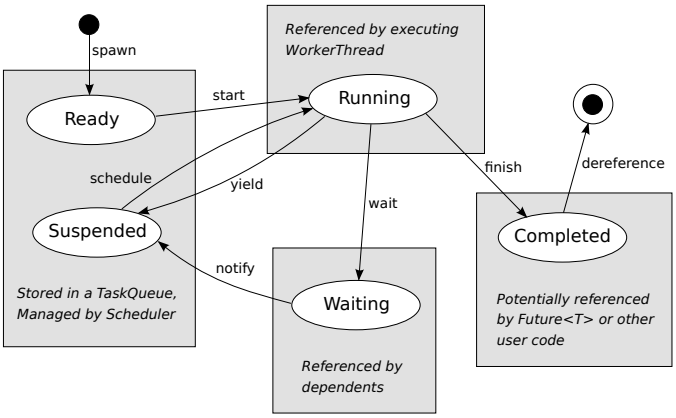

Fig. 4: Possible task states. Tasks, which are reference-counted, are referenced by the entities marked in the grey boxes. Task ownership changes on most state transitions.

Tasks are created with the following properties:

- *Priority*: High priorities always exceed low priorities.
- *Target NUMA node*: The task is run on the specified NUMA node. If the target node is not specified, the task is executed at an arbitrary idle node.
- *Return value type*: Tasks may return a value or `void`. Tasks that return a value return a Future object that can be used to obtain the result value.
- *Task Function*: Defined as a `std::function<T>`, where T is the return value type (including lambdas).

1) Synchronization: We provide a base interface that provides basic synchronization primitives. Listing 6 shows simplified pseudocode of this functionality. Resources that may be waited upon inherit from `Triggerable` and implement `must_wait()` according to the object logic.

```
1  class Triggerable {
2      list<Synchronizer*> clients; // clients waiting on this
3      virtual bool must_wait() = 0; // impl. by subclasses
4      void trigger(int count); // signal N clients
5      bool register_wait(Synchronizer *client) {
6          bool wait = must_wait();
7          if (wait) clients.push_back(client);
8          return wait; // return true, if client must wait
9      }}
10
11 class Synchronizer {
12     list<Triggerable*> dependencies; // waiting on deps
13     virtual void notify() = 0; // implemented by subclasses
14     void signal(Triggerable *dep) {
15         dependencies.remove(dep);
16         if (dependencies.empty()) notify();
17     }
18     bool synchronize(Triggerable *dep) {
19         if (dep->register(this)) dependencies.push_back(dep);
20         return !dependencies.empty(); // true, if must wait
21     }}
```

Listing 6: Base interface for synchronization primitives.

2) Task Scheduling: Task Queues are organized inside `TaskCollections`, which group tasks with the same priority (see Listing 7). `TaskCollections` associate *Suspended* tasks with the worker thread that executed them before they were interrupted. A `SchedulingDomain` associates one `TaskCollection` to each priority level for which schedulable tasks exist. `SchedulingDomains` always return one of the highest-priority tasks therein, trying to preserve thread locality, if possible.

`TaskSchedulers` are associated with a particular NUMA node and are responsible for launching and managing a set of worker threads on their NUMA node. Each `TaskScheduler` manages its tasks through a private `SchedulingDomain`.

```
class TaskCollection {
    TaskQueue unbound; // 'Ready' tasks
    TaskQueue bound[]; // 'Suspended' tasks
    void registerThread(int id);
    void putTask(Task *task, int threadId = -1);
    Task* getTask(int threadId) {
       Task *ret = bound[threadId].tryGet();
       if (ret == NULL) ret = unbound.tryGet();
       if (ret == NULL) ret = stealFromRandomThread();
       return ret;
    }}
```

Listing 7: Task collections group tasks of the same priority.

3) Context Switching: Cooperative user-mode multitasking avoids the overhead associated with operating system scheduling mechanisms. To implement this user-mode context switching we use the `Boost::context` library. When a *running* task enters the *Waiting* or *Suspended* state, the current context is saved within the task's data structure. When this task is re-scheduled to a worker thread, this context is jumped to.

E. Prototypical Data Structure: Distributed Hash-Table

To demonstrate the applicability of *PGASUS*, we implemented a *Hash Table* as a proof-of-concept NUMA-aware data structure. Our *Hash Table* allows an arbitrary amount of concurrent insert, update, read and delete operations. The programmer has to make sure though to only perform delete operations on objects that are not currently read, modified or iterated upon.

The *Hash Table* is divided into 2^N buckets, each of which is responsible for a part of the index space and resides on a specific NUMA node. The last N bits of a key's hash value are used to identify the bucket responsible for storing that key. Each bucket is furthermore subdivided into 2^M bins. Bins are linked lists that store an arbitrary, but usually a very small amount of key-value pairs.

Synchronization is applied at a very fine-grained level via *Reader-Writer* locks. Bin entries are reference-counted to relax synchronization constraints. The *Hash Table* features a number of hierarchical iterators, allowing the iteration space to be divided into sub-iterations for each node. Automatic parallel iteration over the data structure is implemented by collecting all bucket iterators for each node. Each worker thread then iterates over iterators from that node. If there are iterators for remote nodes left after all local ones have been processed, workers start stealing from this remote work pool.

VI. Evaluation

In this section, we provide a premature evaluation of the performance and scalability of *PGASUS*. For that purpose, we developed a simple *WordCount* benchmark, resembling a server application that requires a large memory footprint and which involves requests that have to access non-contiguous data scattered across many objects. We compared the *PGASUS*-based implementation outlined in Listing 8 to a baseline implementation based on OpenMP.

```
numa::HashTable<string,LoadedFile*> files;

int count(string word) {
    atomic_int sum = 0;

    numa::for_each_distributed(
      files, [word](pair<string,TextFile*> value) {
        sum += value.second->count(word);
    }).wait();

    return sum;
}

void wordList(list<string> fileNames) {
    numa::TaskList allTasks;
    for (string fileName : fileNames) {
       allTasks.push_back( numa::async( [fileName] () {
          files[fileName]->wordList();
       }));
    }
    allTasks.wait();
}
```

Listing 8: Parallelization of the *WordCount* application using *PGASUS*. The NUMA-aware hash table presented in section V-E is used to automatically ensure locality.

Figure 5 demonstrates the performance of the *WordCount* micro-benchmark. In this scenario, the *PGASUS* implementation outperforms the OpenMP-based implementation by a factor 2 for large node counts. However, a thorough examination of *PGASUS* is required in order to corroborate the performance improvements detected compared to OpenMP. Furthermore, additional instrumentation will help to identify remaining bottlenecks and to make reliable statements about the performance capabilities for real-world use cases.

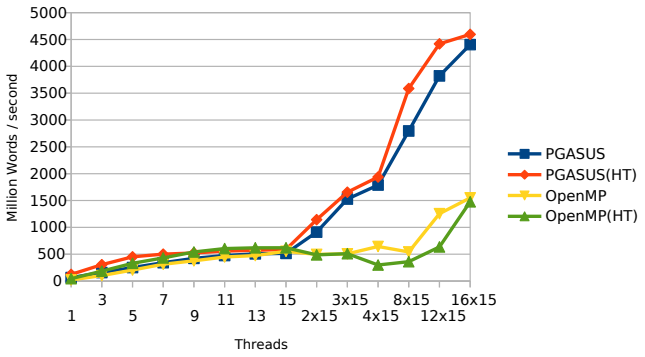

Fig. 5: Throughput of the *WordCount* benchmark executed on a SGI UV300H [24]. The *PGASUS*-based implementation (see Listing 8) achieves a 2x speed-up on large node counts.

VII. Conclusion

In this paper, we presented a review of parallel programming languages and frameworks. Recurring ideas and concepts of parallelism, data locality, and data distribution were identified within these approaches. Major key insights from these concepts were adapted to the context of C++ application development on NUMA systems.

Based on these concepts we developed a C++ programming interface which is based on a number of essential mechanisms for NUMA-aware data parallelism, the specification of data distribution and data locality. These mechanisms enable programmers to solve common problems related to NUMA-aware C++ development in a straightforward way, using a simple but flexible interface. We discussed properties of C++ container classes and data structures and argue that a programming framework should provide specialized, NUMA-aware data structures and algorithm templates. These building blocks automatically enforce a data distribution pattern specified by the programmer and offer an interface for implicit locality-aware data parallelism.

We presented *PGASUS*, a C++ framework implementing the programming model outlined before, as well as an NUMA-aware *Hash Table* implementation based on *PGASUS*. Architectural decisions and implementation details were discussed for both the framework and the prototypical data structure. A premature performance evaluation based on the *WordCount* micro-benchmark yielded a 2x performance increase when using *PGASUS* instead of OpenMP. Although further efforts are necessary to make meaningful statements about the performance and scalability of the framework for real-world workloads, our initial micro-benchmark is indicative of the potential that lies within *PGASUS*.

All features and central components of the programming framework are working and usable, thus enabling programmers to solve NUMA-related programming problems. Still, the framework is a prototype and some architectural shortcuts had to be taken. In order to transform the framework into a robust, stable and versatile programming environment, further steps have to be taken.

Acknowledgement

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

Disclaimer

This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] T. P. Morgan, "Shared memory pushes wheat genomics to boost crop yields," *The Next Platform*, May 2016, http://www.nextplatform.com/2016/05/10/shared-memory-pushes-wheat-genomics-boost-crop-yields/.

[2] M. Plauth, W. Hagen, F. Feinbube, F. Eberhardt, L. Feinbube, and A. Polze, "Parallel Implementation Strategies for Hierarchical Non-Uniform Memory Access Systems by Example of the Scale-Invariant Feature Transform Algorithm," in *Proceedings of the 30th IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW 2016)*, May 2016.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[4] K. Ebcioglu, V. Saraswat, and V. Sarkar, "X10: Programming for hierarchical parallelism and non-uniform data access," in *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, vol. 30. Citeseer, 2004.

[5] B. L. Chamberlain, "A brief overview of chapel," 2013.

[6] M. E. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Stanford University, 2014.

[7] A. Muddukrishna, P. A. Jonsson, and M. Brorsson, "Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors," *Scientific Programming*, vol. 2015, 2015.

[8] S. Thibault, F. Broquedis, B. Goglin, R. Namyst, and P.-A. Wacrenier, "An efficient openmp runtime system for hierarchical architectures," in *A Practical Programming Model for the Multi-Core Era*. Springer, 2007, pp. 161–172.

[9] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic task and data placement over numa architectures: an openmp runtime perspective," in *Evolving OpenMP in an Age of Extreme Parallelism*. Springer, 2009, pp. 79–92.

[10] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "Forestgomp: an efficient openmp environment for numa architectures," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 418–439, 2010.

[11] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "A transparent runtime data distribution engine for openmp," *Scientific Programming*, vol. 8, no. 3, pp. 143–162, 2000.

[12] "UPC Language Specifications, Version 1.3," 2013.

[13] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS Extension for C++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, may 2014, pp. 1105–1114.

[14] H. Richardson, "High Performance Fortran: history, o verview and current developments," 1996.

[15] D. Puppin, N. Tonellotto, and D. Laforenza, "Using web services to run distributed numerical applications," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2004, pp. 207–214.

[16] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[17] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[18] Z. Majó, "Modeling memory system performance of numa multicore-multiprocessors," Ph.D. dissertation, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 22006, 2014, 2014.

[19] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "Stapl: An adaptive, generic parallel c++ library," in *Languages and Compilers for Parallel Computing*. Springer, 2001, pp. 193–208.

[20] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: a holistic approach to memory placement on numa systems," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 381–394, 2013.

[21] S. Lankes, B. Bierbaum, and T. Bemmerl, "Affinity-on-next-touch: an extension to the linux kernel for numa architectures," in *Parallel Processing and Applied Mathematics*. Springer, 2009, pp. 576–585.

[22] B. Stroustrup, *The Design and Evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994.

[23] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*. Springer, 1995, pp. 1–116.

[24] S. G. I. Corp, "SGI UV 300H for SAP HANA," Tech. Rep., 2015.