Practical Distributed Programming in C++

Maurizio Drocco
IBM TJ Watson
Yorktown Heights, NY, USA
maurizio.drocco@ibm.com

Vito Giovanni Castellana
Pacific Northwest National Laboratory
Richland, WA, USA
vitogiovanni.castellana@pnnl.gov

Marco Minutoli
Pacific Northwest National Laboratory
Richland, WA, USA
marco.minutoli@pnnl.gov

ABSTRACT

The need for coupling high performance with productivity is steering the recent evolution of the C++ language where low-level aspects of parallel and distributed computing are now part of the standard or under discussion for inclusion. The Standard Template Library (STL) includes containers and algorithms as primary notions, coupled with execution policies that allow exploiting parallel platforms (e.g., multi-cores) on top of a well-defined operational semantics. In this work, we discuss the design of a stack for STL-compliant containers, iterators, algorithms, and execution policies targeting distributed-memory systems. Finally, we evaluate the proposed approach by analyzing the performance of our proof-of-concept implementation over a set of STL algorithms.

ACM Reference format:
Maurizio Drocco, Vito Giovanni Castellana, and Marco Minutoli. 2020. Practical Distributed Programming in C++. In *Proceedings of Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, Stockholm, Sweden, June 23–26, 2020 (HPDC '20),* 5 pages.
https://doi.org/10.1145/3369583.3392680

1 INTRODUCTION

The parallel programming community has been investigating how to express parallelism in mainstream languages with core language features and libraries. For instance, memory consistency models for parallel programs have been introduced for Java [15] and C++ [4]. In particular, the support for parallel programming in native C++, considering both the core language and the Standard Template Library (STL), is undergoing a rapid evolution. Since the introduction of the C++11 specification, *low-level* parallelism can be expressed for *shared-memory* platforms by means of threads, atomic memory operations, and futures, whereas other features are undergoing standardization [12]. The Networking Technical Specification (TS) provides low-level features for parallelism on *distributed-memory* platforms, such as sockets and communication protocols.

At a *higher level* of abstraction, since the introduction of the C++17 specification, shared-memory parallelism is targeted by the so-called *Parallel STL*, consisting of *execution policies* for STL algorithms on STL containers [13]. For instance, Listing 1 shows the STL parallelization of the Black-Scholes application. However, the language specification is still lacking support for distributed-memory platforms at this level of abstraction. Although several libraries have been proposed, based on Application Programming Interfaces (APIs) sharing some similarities with the STL (cf. Sect. 2), none of them expresses distributed-memory parallelism in *native* C++.

```
price_t seq(std::array<option_t, n> &a) {
  auto m = std::numeric_limits<price_t>::min();
  for (auto it = a.begin(); it != a.end(); ++it)
    m = std::max(res, black_scholes(*it));
  return m;
}

price_t par(std::array<option_t, n> &a) {
  std::array<price_t, n_options> p;
  std::transform(std::execution::par,
    a.begin(), a.end(), p.begin(), black_scholes);
  return *std::max_element(p.begin(), p.end());
}

price_t dpar(dstd::array<option_t, n> &a) {
  dstd::array<price_t, n> p;
  dstd::transform(dstd::execution::par,
    a.begin(), a.end(), p.begin(), black_scholes);
  return *dstd::max_element(dstd::execution::par,
    p.begin(), p.end());
}
```

Listing 1: Parallel Black-Scholes [3] with the distributed STL APIs.

This work tackles this challenge by providing a direct path from single-thread to parallel for distributed systems, while keeping the syntax and semantic of the C++ language and STL. In summary, we make the following contributions:

- design and implementation of a stack for distributed programming in native C++, with STL-compliant distributed containers, iterators, and algorithms (Sect. 3);
- a pattern-based formulation for several distributed STL algorithms, with sequential and parallel semantics expressed in terms of execution policies, endowed with optimizations for various categories of algorithms (Section 4);
- a performance study on a preliminary implementation of the approach, backed by experimental evaluation (Sect. 5).

2 RELATED WORK

Parallel programming in native C++ is being targeted by several efforts towards implementing the requirements posed by recent C++ standards (i.e., the so-called parallel STL). Among them, the standard libraries shipped with most common compilers (e.g., `gcc`, `llvm`, `MSVC`) provide some experimental, yet incomplete support

HPDC '20, June 23–26, 2020, Stockholm, Sweden

ACM ISBN 978-1-4503-7052-3/20/06…$15.00
https://doi.org/10.1145/3369583.3392680

for the parallel STL. Moreover, several proof-of-concept implementations are being developed as external libraries, including *Intel Parallel STL* [11], *Microsoft CodePlex Parallel STL*, *KhronosGroup SyclParallelSTL* [10], and *STellar Group HPX* [14]. However, with the exception of HPX, none of them targets distributed programming.

Among distributed programming tools, STAPL [2] is a template-based library, with an API based on STL-like notions such as distributed containers, ranges, and algorithms. Beside not providing an STL-compliant API, STAPL is based on Single Program Multiple Data (SPMD), thus imposing on the user the paradigm shift from regular C++ that we discussed in Section 1. DASH [8] provides a template-based library of distributed data structures with an API that exposes some control on data locality over hierarchical memory systems. DASH sacrifices STL compliance in favor of allowing lower-level optimizations and is also based on SPMD.

HPX drops the SPMD model and provides a prototypical implementation of several distributed STL algorithms. However, it does not cover distributed associative containers. Furthermore, with respect to HPX, the stack we propose is more flexible at both the runtime support level and the library level, where new algorithms can be introduced as instances of implementation patterns (cf. Sect. 4).

Our approach shares some similarities with the Thrust [6] framework, that aims at unifying CPU and GPU programming under an STL-like API. Finally, our approach to implement distributed STL algorithms based on patterns is the core design principle in the GrPPI [7] library, that provides a C++ API with common parallel patterns (e.g., map, reduce) with different implementation back-ends. However, neither of both targets distributed platforms.

3 DISTRIBUTED CONTAINERS

This work proposes a *stack* whose bottom layer provides *distributed tasking* and serves as lightweight support for the *partitioned data structures* layer. Both layers are included in the Scalable High-Performance Algorithms and Data-structures (SHAD) library [5], that we exploit as foundation. We define a *distributed STL container* to have the same API as a regular STL container, and therefore a C++ program using STL can be ported to distributed STL by mere namespace substitution.[1] We currently provide contiguous[2] (`array`) and associative containers (`unordered_set` and `unordered_map`). Although implementing the whole STL is out of the scope of this work, the proposed design virtually covers all the STL containers.

Distributed Iterators. Iterators are a core building block in STL, providing common abstractions to access, manipulate, and apply algorithms to containers. A major effort in our work regarded the design and implementation of *distributed iterators*, that represent iterators over distributed containers. Distributed iterators have `std::iterator` type and preserve the semantics of STL iterators. STL algorithms operate on *ranges* (i.e., sub-collections), either ordered or not. A range is specified as a pair (b, e) of iterators,[3] denoting the range $r = [b, e)$. We refer to r as a *distributed range* if b and e are distributed iterators. Hence, in Listing 1, replacing `std::array` with `dstd::array` yields a distributed implementation in which all the iterators and ranges are distributed. The *distributed-iterator traits* serve as central abstraction for working with distributed ranges, providing functions that allow to exploit code factorization while leveraging simple optimizations. For instance, when processing a range, it is possible to exclude those *localities* (i.e., memory partitions in SHAD) that do not own any data. Dereferencing a distributed iterator produces a *distributed reference*, whose exact type depends on the underlying container. In most cases, a distributed reference has the same semantics as a regular reference, as it is cast to the referenced value-type whenever it is read (e.g., appearing on the right side of an assignment). However, we could not mirror the semantics of the dot-based syntax, since, as from a long-standing issue that is currently debated in the C++ community, it is not possible to overload the dot operator [17].

Local Iterators. Distributed iterators are inherently more complex than regular STL iterators, as they abstract pointers over multiple address spaces. From the tasking perspective, where multiple tasks are executed by multiple localities, distributed iterators must support dereferencing from any locality, so that they can be copied and passed around between different localities. Accessing a distributed iterator through direct dereferencing, either for reading or writing the referenced value, involves a *translation logic* (e.g., to identify the locality to which the value is physically mapped), which overhead stacks up with the cost of actually accessing the value. For this reason, we introduce the concept of *local iterators*, which are valid only on the corresponding locality and provide direct access to the underlying data, with no need for any translation. Like their distributed counterpart, local iterators are STL-compliant and therefore they can be used as arguments to any STL algorithm, facilitating the implementation of distributed STL algorithms.

Lazy references. Distributed references allow accessing the referenced value in a *lazy* manner. For instance, when a distributed reference appears as left side operand of an assignment, a write-access is triggered to the referenced location; conversely, if is being casted to its referenced value (e.g., it appears as right-side operand), a read-access is triggered instead. Upon lazy access, remote operations may be triggered in form of (synchronous) remote tasks, that is obviously the most relevant source of performance degradation associated with direct access to distributed iterators. Nonetheless, we adopt a few optimizations to reduce the amount of network communication. For instance, we exploit *value caching* in iterators over associative containers, that is safe since only constant iterators are exposed by such containers and modifying an associative container while iterating is not supported by the C++ standard.

Iterators Arithmetic. On contiguous containers (e.g., arrays), iterator arithmetic can be resolved locally, without triggering remote tasks. Since the data distribution across the localities is fixed for the underlying data structure, it is possible to identify the locality where the data resides as a function of the position referenced by the iterator. Therefore, the arithmetic can be performed by updating the locality information and the local iterator. Conversely, for associative containers (e.g., maps), the underlying SHAD data structure adopts a storage based on linked-lists, whose structure may change over time. For this reason, incrementing an iterator may trigger a

[1] We implement our STL-equivalent API in the `dstd` namespace.
[2] STL arrays are required to be *physically* contiguous in memory. However, this constraint cannot be honored in a distributed-memory setting.
[3] For simplicity, we do not discuss the alternative range format, consisting in an iterator b and a length $n \in \mathbb{N}$, denoting the range $[b_i, b_{i+n})$.

remote task, directed to the locality on which the next[4] item resides, to retrieve the information needed to update the iterator fields.

4 DISTRIBUTED ALGORITHMS LIBRARY

Distributed iterators are compliant with the STL specifications, therefore STL algorithms can seamlessly process distributed ranges. However, naively passing distributed ranges to STL algorithms would result in performance penalties due to fine-grained (remote) read-write dereferencing (cf. Section 3). At a high level, we target the challenge of efficiently exploiting the parallelism provided by distributed platforms, without losing STL compliance. In C++17, this problem has been addressed for shared-memory platforms through the introduction of *execution policies*. For instance, the `reduce` algorithm has well-defined parallel semantics, and its parallel implementation is triggered by a policy-typed function argument. Conversely, `accumulate` is strictly sequential: its combining function has heterogeneous typing that prevents parallel execution, and therefore no policy is defined for this algorithm.

We mirror the C++ approach by introducing two execution policies: *distributed-sequential* and *distributed-parallel*. Listing 1 shows the code for the distributed-parallel version of the Black-Scholes running example. The `dstd::execution::seq` policy denotes the *distributed-sequential* execution model, that provides *sequential semantics* on distributed executions. Our design follows three principles: (1) the computation is partitioned across the localities, so that each portion of the input range is processed by the owning locality; (2) STL algorithms are directly used to process each portion; (3) sequential ordering among processed items is preserved. Conversely, the `dstd::execution::par` policy denotes the *distributed-parallel* execution model, that exploits parallelism between localities with the same semantics as the parallel STL on single-node systems. In particular, distributed-parallel executions exploit *data parallelism* by treating parallel STL algorithms as *list homomorphisms* [9], according to a *map-reduce* pattern[5].

Most algorithms access a single range, to either read the input elements (e.g., `reduce`) or modify them in-place (e.g., `generate`). In this case, under both execution policies, *perfect partitioning* can be exploited to guarantee that each locality l works on a portion r_l of the input range that is located at l, avoiding any remote access. However, multi-range algorithms do not guarantee that all the sub-ranges accessed by a locality l during the computation are co-located in l. We introduce the following optimizations to tackle this issue: (1) Local-Assign to avoid the overhead associated to accessing distributed iterators over node-local sub-portions; (2) Async-Remote-Assign to hide the latency associated with remote output operations over physically contiguous memory blocks (e.g., arrays), through asynchronous, block-wise remote memory copying; (3) Async-Insert, based on asynchronous insertions through *buffered* insert-iterators, to mitigate latency even when no assumptions can be made about the data layout (e.g., associative maps).

[4]Notice that, for unordered containers, the selection of the *next* element is arbitrarily defined by the implementation.

[5]With map-reduce, we denote the higher-order function from functional programming, rather that the Google's MapReduce paradigm, which semantics is slightly different [1].

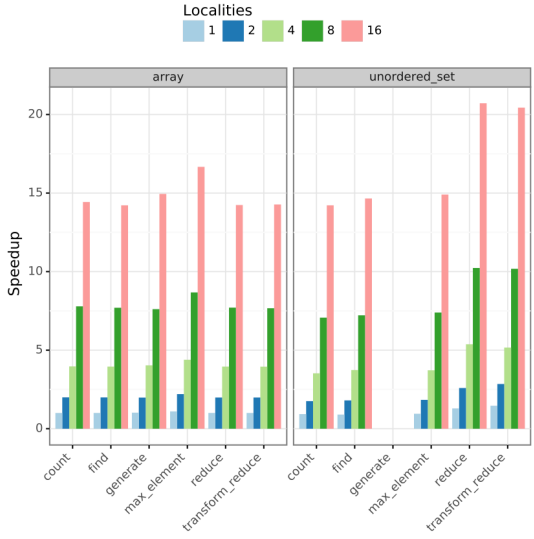

Figure 1: Performance of single-range distributed algorithms over containers with 10^9 items, with respect to `std` executions. Note that the assignment-based `generate` algorithm on sets is missing, as prescribed by the STL specification.

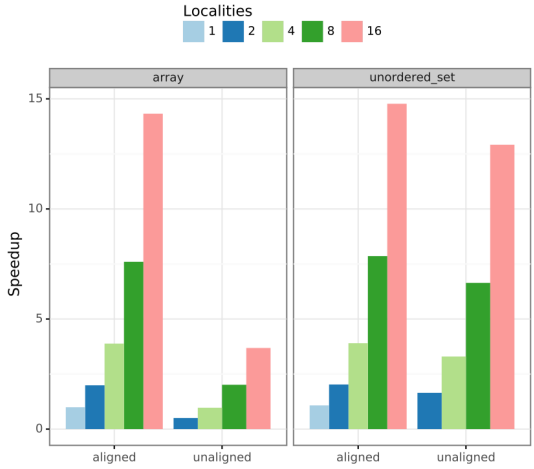

Figure 2: Optimized distributed-parallel `transform`: Local-Assign (aligned array ranges), Async-Remote-Assign (unaligned array ranges), and Async-Insert (set ranges)

5 EXPERIMENTAL RESULTS

We built a proof-of-concept implementation of the stack described in Sections 3 and 4 as a header-only C++ library. The API components are in the `dstd` namespace, including distributed containers, distributed iterators and their buffered extensions, execution policies, and a library of STL algorithms. For the evaluation, only `array` and `unordered_set` are considered, as SHAD maps and sets are analogous in both data layout and performance.

We remark that, in analogy with the parallel STL on shared-memory systems, the distributed API does not expose to the user any way to control the level of concurrency in parallel algorithms, leaving these low-level details to the SHAD backend. For the reported experiments, the backend is the MPI-based tasking layer from the Global Memory and Threading (GMT) library [16], in which the number of localities corresponds to the number of MPI ranks.

All the experiments have been conducted on 8 nodes of a homogeneous cluster, each equipped with two Intel Xeon E5-2680 v2 CPUs working at 2.8 GHz (hyper-threading disabled) and 768GB of memory. Our experimental setup binds a locality (i.e., a MPI rank) to each socket in the allocated nodes (i.e., two localities per node), to minimize the effects of Non-Uniform Memory Access (NUMA) across sockets.

5.1 Single-Range Algorithms

Perfect partitioning provides a simple performance model, resulting in theoretical linear speedup with respect to the number of localities and constant overhead with respect to the size of the input range. We evaluate distributed single-range algorithms by comparing their execution time with the corresponding STL algorithm over containers with 10^9 elements. Our experimental results show that the observed performance is consistent with the model: distributed-sequential execution times are comparable with STL executions and distributed-parallel executions scale linearly with the number of localities (Fig. 1). In some cases, STL algorithms are slower than their distributed one-locality counterpart (e.g., `reduce` on sets), yielding super-linear speedup in the distributed-parallel policy. This is explained observing that, although executing a plain STL algorithm, the distributed execution accesses a different container underneath (i.e., a SHAD data structure), which in those cases performs better than its STL counterpart.

5.2 Multi-Range Algorithms

The performance model for multi-range algorithms is more complex, therefore we articulate the evaluation focusing only on the `transform` algorithm, measuring first the impact of the proposed optimizations and then the scaling sustained by the optimized executions. Since the performance depends on the physical distribution of the output range over the involved localities, we consider two boundary conditions. In the *aligned* case, all the output operations are directed to the same locality that originates the operation, thus no task is executed remotely. Conversely, in the *unaligned* case, all the output operations are directed to a different locality, turning into remote tasks. Tab. 1 shows the configurations for the two cases, according to the current layout of SHAD data structures: sequential distribution and modulo-based mapping for, respectively, contiguous and associative containers.

5.2.1 Optimization Effects. The impact of the proposed optimizations is evaluated comparing the execution times of `transform` on small (i.e., 10^5 items) distributed ranges, as the non-optimized cases were prohibitive on larger ranges. We measured the gain provided by each optimization over the corresponding non-optimized executions. Tab. 2 shows the measured gains. As expected, larger gains can be observed when optimizing against remote task executions in the unaligned cases, whereas the improvement fades out on small containers when increasing parallelism. Nonetheless, the optimizations better support parallelism on large containers, as we show in the following section.

Table 1: Configurations inducing different physical data layouts for the `dstd::transform`.

	aligned	**unaligned**
array	input/output ranges span two arrays of the same size	output range is the second half of an array with double size of the array spanned by input range
set	mapping: $x \mapsto x$	mapping: $x \mapsto x + 1$

Table 2: Absolute speedup (over non-optimized executions) provided by multi-range optimizations: Local-Assign (aligned array ranges), Async-Remote-Assign (unaligned array ranges), and Async-Insert (set ranges).

	Sequential		Parallel	
Localitites	aligned	unaligned	aligned	unaligned
		`array`		
1	27.23	-	48.55	-
2	6.14	4038.64	3.02	4925.33
4	2.42	1761.46	1.55	1716.99
8	1.28	655.51	1.14	381.20
16	0.80	333.18	0.99	186.27
		`unordered_set`		
1	0.55	-	0.53	-
2	1093.52	11212.22	2824.16	9586.49
4	1328.20	5939.44	1952.63	5428.82
8	959.86	2604.85	1929.86	3771.65
16	713.17	1349.40	944.30	1546.61

5.2.2 Optimization Scaling. Fig. 2 shows the *absolute strong scaling* of the optimized `transform` on ranges with 10^9 elements, with respect to STL executions. The optimizations amortize the latency from distributed iterators in the aligned cases. On set ranges, Async-Insert prevents the latency induced by remote accesses to grow with the number of localities and the result holds also in the unaligned case. On array ranges, STL executions are basically sequential traversals of memory chunks, therefore an unavoidable overhead is associated with communication in the distributed case. Nevertheless, Async-Remote-Assign effectively reduces the slowdown (although not shown here for lack of space, we measured a $4x$ slowdown on all-remote accesses). The scaling provided by parallel executions is almost ideal in all cases covered by Local-Assign and Async-Insert, and still proportional to the number of localities with Async-Remote-Assign. Remarkably, the execution time of an all-remote computation over 2 localities is comparable with the corresponding STL execution (i.e., speedup ~ 1).

REFERENCES

[1] Marco Aldinucci, Maurizio Drocco, Claudia Misale, and Guy Tremblay. 2018. *Languages for Big Data analysis.* Springer International Publishing, Cham, 1–12.

[2] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Timmie G. Smith, Gabriel Tanase, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. 2001. STAPL: An Adaptive, Generic Parallel C++ Library. In *Proc. of Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Vol. 2624. Springer, Berlin, Heidelberg, 193–208.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of the 17th Intl. Conference on Parallel Architectures and Compilation Techniques (PACT'08).* ACM, New York, NY, USA, 72–81.

[4] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI'08).* ACM, New York, NY, USA, 68–78.

[5] Vito Giovanni Castellana and Marco Minutoli. 2018. SHAD: The Scalable High-Performance Algorithms and Data-Structures Library. In *Proc. of the IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing (CCGRID'18).* IEEE Computer Society, New York, NY, USA, 442–451.

[6] NVIDIA Corporation. 2014. Thrust. http://thrust.github.io. (2014). Accessed: January 17, 2019.

[7] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. 2017. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience* 29, 24 (2017), e4175.

[8] Karl Fürlinger, Tobias Fuchs, and Roger Kowalewski. 2016. DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. In *Proc. of IEEE Intl. Conference on High Performance Computing and Communications (HPCC'16).* IEEE Computer Society, New York, NY, USA, 983–990.

[9] Sergei Gorlatch. 1996. Systematic efficient parallelization of scan and other list homomorphisms. In *Euro-Par'96 Parallel Processing*. Springer, Berlin, Heidelberg, 401–408.

[10] Khronos OpenCL Working Group. 2009. SYCL. https://www.khronos.org/sycl. (2009). Accessed: January 17, 2019.

[11] Intel. 2019. Parallel STL. https://github.com/intel/parallelstl. (2019). Accessed: April 23, 2019.

[12] ISO. 2016. *ISO/IEC TS 19571:2016 — Programming Languages — Technical specification for C++ extensions for concurrency* (first ed.). ISO, Geneva, Switzerland. 19 pages.

[13] ISO. 2017. *ISO/IEC 14882:2017 Information technology — Programming languages — C++* (fifth ed.). ISO, Geneva, Switzerland. 1605 pages.

[14] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proc. of the Intl. Conference on Partitioned Global Address Space Programming Models (PGAS'14).* ACM, New York, NY, USA, Article 6, 11 pages.

[15] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL'05).* ACM, New York, NY, USA, 378–391.

[16] Alessandro Morari, Antonino Tumeo, Daniel G. Chavarría-Miranda, Oreste Villa, and Mateo Valero. 2014. Scaling Irregular Applications through Data Aggregation and Software Multithreading. In *Proc. of the IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'14).* IEEE Computer Society, New York, NY, USA, 1126–1135.

[17] Bjarne Stroustrup and Gabriel Dos Reis. 2014. ISO/IEC/JTC1/SC22/WG21/N4173 — Operator Dot. WG paper. (2014).