An Introduction to hpxMP – A Modern OpenMP Implementation Leveraging HPX, An Asynchronous Many-Task System

Tianyi Zhang
tzhan18@lsu.edu
Center for Computation and Technology, LSU

Shahrzad Shirzad
sshirz1@lsu.edu
Center for Computation and Technology, LSU

Patrick Diehl
patrickdiehl@lsu.edu
Center for Computation and Technology, LSU

R. Tohid
mraste2@lsu.edu
Center for Computation and Technology, LSU

Weile Wei
wwei9@lsu.edu
Center for Computation and Technology, LSU

Hartmut Kaiser
hkaiser@cct.lsu.edu
Center for Computation and Technology, LSU

ABSTRACT

Asynchronous Many-task (AMT) runtime systems have gained increasing acceptance in the HPC community due to the performance improvements offered by fine-grained tasking runtime systems. At the same time, C++ standardization efforts are focused on creating higher-level interfaces able to replace OpenMP or OpenACC in modern C++ codes. These higher level functions have been adopted in standards conforming runtime systems such as HPX, giving users the ability to simply utilize fork-join parallelism in their own codes. Despite innovations in runtime systems and standardization efforts users face enormous challenges porting legacy applications. Not only must users port their own codes, but often users rely on highly optimized libraries such as BLAS and LAPACK which use OpenMP for parallization. Current efforts to create smooth migration paths have struggled with these challenges, especially as the threading systems of AMT libraries often compete with the treading system of OpenMP.

To overcome these issues, our team has developed hpxMP, an implementation of the OpenMP standard, which utilizes the underlying AMT system to schedule and manage tasks. This approach leverages the C++ interfaces exposed by HPX and allows users to execute their applications on an AMT system without changing their code.

In this work, we compare hpxMP with Clang's OpenMP library with four linear algebra benchmarks of the Blaze C++ library. While hpxMP is often not able to reach the same performance, we demonstrate viability for providing a smooth migration for applications but have to be extended to benefit from a more general task based programming model.

IWOCL'19, May 13–15,2019, Boston, MA, USA

ACM ISBN 978-1-4503-6230-6/19/05…$15.00
https://doi.org/10.1145/3318170.3318191

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**;

KEYWORDS

OpenMP, hpxMP, Asynchronous Many-task Systems, C++, clang, gcc, HPX

ACM Reference Format:
Tianyi Zhang, Shahrzad Shirzad, Patrick Diehl, R. Tohid, Weile Wei, and Hartmut Kaiser. 2019. An Introduction to hpxMP – A Modern OpenMP Implementation Leveraging HPX, An Asynchronous Many-Task System. In *International Workshop on OpenCL (IWOCL'19), May 13–15,2019, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3318170.3318191

1 INTRODUCTION

The Open Multi-Processing (OpenMP) [OpenMP Consortium 2018] standard is widely used for shared memory multiprocessing and is often coupled with the Message Passing Interface (MPI) as *MPI+X* [Bader 2016] for distributed programming. Here, MPI is used for the inter-node communication and *X*, in this case, OpenMP, for the intra-node parallelism. Nowadays, Asynchronous Many Task (AMT) run time systems are emerging as a new parallel programming paradigm. These systems are able to take advantage of fine grained tasks to better distribute work across a machine. The C++ standard library for concurrency and parallelism (HPX) [Heller et al. 2017] is one example of an AMT runtime system. The HPX API conforms to the concurrency abstractions introduced by the C++ 11 standard [C++ Standards Committee 2011] and to the parallel algorithms introduced by the C++ 17 standard [C++ Standards Committee 2017]. These algorithms are similar to the concepts exposed by OpenMP, *e.g.* `#pragma omp parallel for`.

AMT runtime systems are becoming increasing used for HPC applications as they have shown superior scalability and parallel efficiency for certain classes of applications (see [Heller et al. 2018]). At the same time, the C++ standardization efforts currently focus on creating higher-level interfaces usable to replace OpenMP (and other `#pragma`-based parallelization solutions like OpenACC) for modern C++ codes. This effort is driven by the lack of integration of `#pragma` based solutions into the C++ language, especially the language's type system.

Both trends call for a migration path which will allow existing applications that directly or indirectly use OpenMP to port potions of the code an AMT paradigm. This is especially critical for applications which use highly optimized OpenMP libraries where it is not feasible to re-implement all the provided functionalities into a new paradigm. Examples of these libraries are linear algebra libraries [Anderson et al. 1999; Blackford et al. 2002; Galassi et al. 2002; Guennebaud et al. 2010; Iglberger et al. 2012; Rupp et al. 2016; Sanderson and Curtin 2016; Wang et al. 2013], such as the Intel math kernel library or the Eigen library.

For these reasons, it is beneficial to combine both technologies, *AMT+OpenMP*, where the distributed communication is handled by the AMT runtime system and the intra-node parallelism is handled by OpenMP or even combine OpenMP and the parallel algorithms on a shared memory system. Currently, these two scenarios are not possible, since the light-weighted thread implementations usually present in AMTs interferes with the system threads utilized by the available OpenMP implementations.

To overcome this issue, hpxMP, an implementation of the OpenMP standard [OpenMP Consortium 2018] that utilizes HPX's lightweight threads is presented in this paper. The hpxMP library is compatible with the clang and gcc compiler and replaces their shared library implementations of OpenMP. hpxMP implements all of the OpenMP runtime functionalities using HPX's lightweight threads instead of system threads.

Blaze, an open source, high performance C++ math library, [Iglberger et al. 2012] is selected as an example library to validate our implementation. Blazemark, the benchmark suite available with Blaze, is used to run some common benchmarks. The measured results are compared against the same benchmarks run on top of the compiler-supplied OpenMP runtime. This paper focuses on the implementation details of hpxMP as a proof of concept implementing OpenMP with an AMT runtime system. We use HPX as an exemplary AMT system that already exposes all the required functionalities.

The paper is structured as follows: Section 2 emphasizes the related work. Section 3 provides a brief introduction to HPX's concepts and Section 4 a brief introduction to OpenMP's concepts utilized in the implementation in Section 5. The benchmarks comparing hpxMP with clang's OpenMP implementation are shown in Section 6. Finally, we draw our conclusions in Section 7.

2 RELATED WORK

Exploiting parallelism on multi-core processors with shared memory has been extensively studied and many solutions have been implemented. The POSIX Threads [Alfieri 1994] execution model enables fine grain parallelism independent of any language. At higher levels, there are also library solutions like Intel's Threading Building Blocks (TBB) [Intel 2019] and Microsoft's Parallel Pattern Library (PPL) [Microsoft 2010]. TBB is a C++ template library for task parallelism while PPL provides features like task parallelism, as well as parallel algorithms and containers with an imperative programming model. There are also several language solutions. Chapel [Chamberlain et al. 2007] is a parallel programming language with parallel data and task abstractions. The Cilk family of languages [Leiserson 2009] are general-purpose programming languages which target multi-thread parallelism by extending C/C++ with parallel loop constructs and a fork-join model. Kokkos [Edwards et al. 2014] is a package which exposes multiple parallel programming models such as CUDA and pthreads through a common C++ interface. Open Multi-Processing (OpenMP) [Dagum and Menon 1998] is a widely accepted standard used by application and library developers. OpenMP exposes fork-join model through compiler directives and supports tasks.

The OpenMP 3.0 standard[1] introduced the concept of task-based programming. The OpenMP 3.1 standard[2] added task optimization within the tasking model. The OpenMP 4.0 standard[3] offers user a more graceful and efficient way to handle task synchronization by introducing depend tasks and task group. The OpenMP 4.5 standard[4] was released with its support for a new task-loop construct, which is providing a way to separate loops into tasks. The most recent, the OpenMP 5.0 standard[5] supports detached tasks.

There have also been efforts to integrate multi-thread parallelism with distributed programming models. Charm++ has integrated OpenMP into its programming model to improve load balance [PPL 2011]. However, most of the research in this area has focused on MPI+X [Bader 2016; Barrett et al. 2015] model.

3 C++ STANDARD LIBRARY FOR CONCURRENCY AND PARALLELISM (HPX)

This section briefly describes the features of the C++ Standard Library for Concurrency and Parallelism (HPX) [Heller et al. 2017] which are utilized in the implementation of hpxMP in the Section 5. HPX facilitates distributed parallel applications of any scale and uses fine-grain multi-threading and asynchronous communications [Khatami et al. 2016]. HPX exposes an API that strictly adheres the current ISO C++ standards [Heller et al. 2017]. This approach to standardization encourages programmers to write code that is high portability in heterogeneous systems [Copik and Kaiser 2017].

HPX is highly interoperable in distributed parallel applications, such that, it can be used on inter-node communication setting of a single machine as well as intra-node parallelization scenario of hundreds of thousands of nodes [Wagle et al. 2018]. The *future* functionality implemented in HPX permits threads to continually finish their computation without waiting for their previous steps to be completed which can achieve a maximum possible level of parallelization in time and space [Khatami et al. 2017].

3.1 HPX Threads

The HPX light-weight threading system provides user level threads, which enables fast context switching [Biddiscombe et al. 2017]. With lower overheads per thread, programs are able to create and schedule a large number of tasks with little penalty [Wagle et al. 2018]. The advantage of this threading system combined with the *future* functionality in HPX facilitates auto-parallelization in a highly efficient fashion as such combination allows the direct expression

[1] https://www.openmp.org/wp-content/uploads/spec30.pdf
[2] https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf
[3] https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdff
[4] https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
[5] https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

of the generated dependency graph as an execution tree generated at runtime [Grubel et al. 2015].

3.2 HPX Thread Scheduling and Policies

The adaptive thread scheduling system employed in HPX improves performance of parallel applications [Biddiscombe et al. 2017]. The built-in scheduling policies enable optimal task scheduling for a given application and/or algorithm [Heller et al. 2017]. The programmers can code efficiently as they can focus on algorithms or application development itself instead of manually scheduling CPU resources. Also, the built-in scheduling policies allow users to provide their own scheduling policies if they require more specific control on application-level.

The HPX runtime now supports eight different thread scheduling policies: *priority local scheduling (default option)*: this policy creates one queue per OS thread. The OS threads remove waiting tasks from the queue and start task execution accordingly. The number of high priority queues equal to the number of OS threads. *Static priority scheduling*: the static priority scheduling policy maintains one queue per OS thread from which each OS thread places its tasks. Round Robin model is used in this policy. *Local scheduling*: this policy maintains one queue per OS threads from which each OS thread removes waiting tasks from the queue and start task execution accordingly. *Global scheduling*: this policy maintains one shared queue from which all OS threads pull waiting tasks. *ABP scheduling*: this policy maintains a double ended lock-free queue per OS thread. Threads are inserted on the top of the queue and are stolen from the bottom of the queue during the work stealing. *Hierarchy scheduling policy*: this policy constructs a tree of task items, and each OS thread traverses through the tree to obtain new task item. *Periodic priority scheduling policy*: this policy arranges one queue of task items per OS thread, a couple of high priority queues and one low priority queue.

These policies can be categorized into three types: *thread local*: the thread local is currently set as the default scheduling option. This policy schedules one queue for each OS core, and the queues with high priority will be scheduled before any other works with lower priority; *static*: the static scheduling policy follows round robin principle and the thread stealing is not allowed in this policy; *hierarchical*: the hierarchical scheduling policy utilizes a tree structure of run queues. The OS threads need to traverse the tree to new task items.

4 INTEGRATION OF HPX IN OPENMP APPLICATIONS

This section describes the integration of HPX to the OpenMP 4.0[6] specification. Figure 1 illustrates how hpxMP fits in an OpenMP application. A user-defined application with OpenMP directives, library functions, and environment variable can be compiled with any compiler that supports OpenMP. The hpxMP shared library adds an additional layer, marked in gray, which carries out the parallel computation. Instead of calling the OpenMP functions and running them on OpenMP threads, the equivalent hpxMP functions are called redirecting the program to the corresponding functionality in HPX. HPX employs light-weight HPX threads following

[6]https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

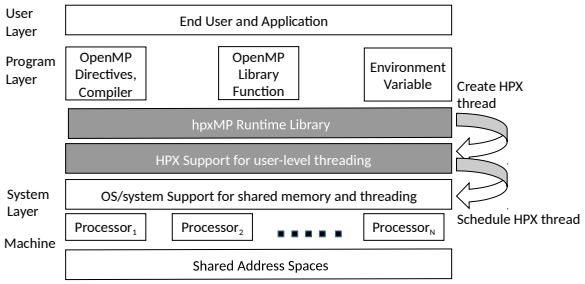


Figure 1: Layers of an hpxMP application. Applications are compiled with OpenMP flags and linked against the hpxMP runtime library where HPX threads perform the computation in parallel. Gray layers are implemented in hpxMP. This figure is adapted from [Mattson 2013].

thread scheduling policies for parallel computing. For more details see Section 3.

Table 1 provides the list of the directives that are currently supported by hpxMP. A list of runtime library functions implemented in hpxMP runtime library is available in Table 2. In the next section, the detailed implementation of these directives is discussed.

5 IMPLEMENTATION OF HPXMP

This section provides an overview of prominent OpenMP and OpenMP Performance Toolkit (OMPT) functionalities and elaborates on implementations of these functionalities in hpxMP.

The fundamental directives described in the OpenMP specification [de Supinski Michael Klemm 2017] are shown in Listing 1. It is important to note that implementation of OpenMP directives may differ based on the compiler. hpxMP is mapped onto LLVM-Clang as specified by LLVM OpenMP Runtime Library[7]. However, hpxMP also supports GCC[8] entry by mapping the function calls generated by the GCC compiler on to the Clang entry.

Listing 1: Fundamental OpenMP directives

```
#pragma omp parallel
#pragma omp for
#pragma omp task
```

5.1 Parallel Construct

Parallel construct initiates the parallel execution of the portion of the code annotated by the parallel directive. The directive `#pragma omp parallel` with its associated structured block is treated as a function call to `__kmpc_fork_call`, which preprocesses the arguments passed by the compiler and calls the function named `fork`, implemented in hpxMP runtime, see Figure 1. The implementation of `__kmpc_fork_call` is shown in Listing 2. HPX threads, as many as requested by the user, are created during the fork call which explicitly registers HPX threads, see Listing 3. Each HPX thread

[7]https://openmp.llvm.org/Reference.pdf
[8]https://gcc.gnu.org/onlinedocs/libgomp/

Table 1: Directives implemented in the program layer of hpxMP, see Figure 1. The corresponding functions are the main part of hpxMP runtime library.

Pragmas Implemented in hpxMP	
#pragma omp atomic	#pragma omp barrier
#pragma omp critical	#pragma omp for
#pragma omp master	#pragma omp ordered
#pragma omp parallel	#pragma omp section
#pragma omp single	#pragma omp task depend

Table 2: Runtime library functions in hpxMP's program layer, see Figure 1. The following functional APIs are provided to users.

Runtime Library Functions in hpxMP	
omp_get_dynamic	omp_get_max_threads
omp_get_num_procs	omp_get_num_threads
omp_get_thread_num	omp_get_wtick
omp_get_wtime	omp_in_parallel
omp_init_lock	omp_init_nest_lock
omp_set_dynamic	omp_set_lock
omp_set_nest_lock	omp_set_num_threads
omp_test_lock	omp_test_nest_lock
omp_unset_lock	omp_unset_nest_lock

Table 3: OMPT callbacks implemented in hpxMP runtime library, see Figure 1. First party performance analysis toolkit for users to develop higher level performance analysis policy.

OMPT callbacks
ompt_callback_thread_begin
ompt_callback_thread_end
ompt_callback_parallel_begin
ompt_callback_parallel_end
ompt_callback_task_create
ompt_callback_task_schedule
ompt_callback_implicit_task

follows HPX scheduling policies, see Section 3, performing its own work under the structured parallel block.

5.2 Loop Scheduling Construct

Another common OpenMP construct is the loop construct which runs several iterations of a `for` loop in parallel where each instance runs on a different thread in the team. A team is defined as a set of one or more threads in the execution of a parallel region [de Supinski Michael Klemm 2017]. The loops are divided into chunks, and the scheduler determines how such chunks are distributed across the threads in the team. The default schedule type is `static` where the chunk size is determined by the threads and number of loops. Each thread gets approximately the same amount of loops and the structured block is executed in parallel.

For the static schedule, the directive `#pragma omp for` with its associated structured for loop block invoke the following sequence of function calls: `__kmpc_for_static_init`, `__kmpc_dispatch_next` and `__kmpc_dispatch_fini`. Chunks are distributed among threads in a round-robin fashion, see Listing 4.

5.3 Task Construct

Task Construct creates explicit tasks in hpxMP. When a thread sees this construct, a new HPX thread is created and scheduled based on HPX thread scheduling policies, see Section 3.

The directive `#pragma omp task` and its associated structured block initiate a series of function calls: `__kmpc_omp_task_alloc`, `__kmpc_fork_call`, see Listing 5.

A task object is allocated, initialized, and returned to the compiler by the task allocation function `__kmpc_omp_task_alloc`. A normal priority HPX thread is then created by the compiler generated function call `__kmpc_omp_task` and ready to execute the task allocated by prior functions.

5.4 OpenMP Performance Toolkit

OpenMP Performance Toolkit (OMPT) is an application programming interface (API) for first-party performance tools. It is integrated into the hpxMP runtime system and enables users to construct powerful and efficient custom performance tools. The implemented callback functions, see Table 3, make it possible for users to track the behavior of threads, parallel regions, and tasks.

The Implementation of the parallel begin callback is shown in Listing 6. This piece of code calls the user-defined callbacks and is hooked into the hpxMP runtime before the parallel region actually begins.

Listing 2: Implementation of __kmpc_fork_call in hpxMP

```
void __kmpc_fork_call(ident_t *loc, kmp_int32 argc, kmpc_micro microtask, ...) {
        vector<void*> argv(argc);
        va_list ap;
        va_start( ap, microtask );
        for( int i = 0; i < argc; i++ ){
                argv[i] = va_arg( ap, void * );
        }
        va_end( ap );
        void ** args = argv.data();
        hpx_backend->fork(__kmp_invoke_microtask, microtask, argc, args);
}
```

Listing 3: Implementation of hpx_runtime::fork in hpxMP

```
for( int i = 0; i < parent->threads_requested; i++ ) {
        hpx::applier::register_thread_nullary(
                std::bind( &thread_setup, kmp_invoke, thread_func, argc, argv, i, &team,
                    parent,
                boost::ref(barrier_mtx), boost::ref(cond), boost::ref(running_threads) ),
                "omp_implicit_task", hpx::threads::pending,
                true, hpx::threads::thread_priority_low, i );
        }
```

Listing 4: Implementation of __kmpc_for_static_init in hpxMP

```
void __kmpc_for_static_init( ident_t *loc, int32_t gtid, int32_t schedtype,
                int32_t *p_last_iter, int64_t *p_lower, int64_t *p_upper,
                int64_t *p_stride, int64_t incr, int64_t chunk ) {
        //code to determine each thread's lower and upper bound (*p_lower, *p_upper)
        //with the given thread id, schedule type and stride.
        }
```

Listing 5: Implementation of task scheduling in hpxMP

```
kmp_task_t* __kmpc_omp_task_alloc( ident_t *loc_ref, kmp_int32 gtid, kmp_int32 flags,
                        size_t sizeof_kmp_task_t, size_t sizeof_shareds,
                        kmp_routine_entry_t task_entry ){
        int task_size = sizeof_kmp_task_t + (-sizeof_kmp_task_t%8);
        kmp_task_t *task = (kmp_task_t*)new char[task_size + sizeof_shareds];
        task->routine = task_entry;
        return task;
}
int __kmpc_omp_task( ident_t *loc_ref, kmp_int32 gtid, kmp_task_t * new_task){
                //Create a normal priority HPX thread with the allocated task as argument.
                hpx::applier::register_thread_nullary(.....)
                return 1;
}
```

Listing 6: Implementation of thread callbacks in hpxMP

```
if (ompt_enabled.enabled) {
        if (ompt_enabled.ompt_callback_parallel_begin) {
                ompt_callbacks.ompt_callback(ompt_callback_parallel_begin)(
                        NULL, NULL,&team.parallel_data, team_size,
                        __builtin_return_address(0));
        }
}
#endif
```

5.5 GCC Support

LLVM OpenMP Runtime Library[9] provides the gcc compatibility shims. In order to achieve the GCC support in hpxMP, we exposes similar shims to map GCC generated entries to Clang. These mapping functions preprocess the arguments provided by the compiler and pass them directly to the hpxMP or call Clang supported entries. Therefore, programs compiled with GCC or Clang are supported by hpxMP.

5.6 Start HPX back end

HPX must be initialized before hpxMP can start execution. The HPX initialization can start both externally or internally. If HPX is started externally (by applications), hpxMP will initialize HPX internally before scheduling any work. The function designed to start hpx back-end properly is written in each function calls generated by the compiler make sure HPX is properly started before we call any `#pragma omp` related functions, see Listing 8.

6 BENCHMARKS

In this paper, four benchmarks are used to compare the performance between Clang's implementation of OpenMP and our implementation of hpxMP, which are daxpy, dense vector addition, dense matrix addition, and dense matrix multiplication. We tested these benchmarks are tested on Marvin, a node in the Center of Computation and Technology (CCT)'s Rostam cluster at Louisiana State University. The hardware properties of Marvin are shown in Table 4 and the libraries and compiler used to build hpxMP and its dependencies can be found in Table 5.

The benchmark suite of Blaze[10] is used to analyze the performance. For each benchmark, a heat-map demonstrates the ratio, r, of the Mega Floating Point Operations Per Second (MFLOP/s) between hpxMP and OpenMP. To make the heat map plots easier to analyze, only a portion of the larger input sizes n is visualized. For the overall overview, three scaling graphs with thread number 4, 8, and 16 are plotted associated with each benchmark, showing the relation between MFLOP/s and size n. The size of the vectors and matrix in the benchmarks increases arithmetically from 1 to 10 million. We picked these three thread numbers as an example since the behavior looks similar for all other candidates. Blaze uses a set of thresholds for different operations to be executed in parallel. For each of the following benchmarks if the number of elements in the vector or matrix (depending on the benchmark) is smaller than the specified threshold for that operation, it would be executed single-threaded.

[9]https://openmp.llvm.org/Reference.pdf
[10]https://bitbucket.org/blaze-lib/blaze/wiki/Benchmarks

Table 4: System configuration of the marvin node. All benchmarks are run on this node.

Category	Property
Server Name	Rostam
CPU	2 x Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz
RAM	48 GB
Number of Cores	16

Table 5: Overview of the compilers, software, and operating system used to build hpxMP, HPX, Blaze and its dependencies.

Category	Property
OS	CentOS Linux release 7.6.1810 (Core)
Kernel	3.10.0-957.1.3.el7.x86_64
Compiler	clang 6.0.1
gperftools	2.7
boost	1.68.0
OpenMP	3.1
HPX[11]	140b878
Blaze[12]	3.4

6.1 Dense Vector Addition

Dense Vector Addition(dvecdvecadd) is a benchmark that adds two dense vectors a and b and stores the result in vector c, where $a, b \in \mathbb{R}^n$. The addition operation is $c[i] = a[i] + b[i]$. The parallelization threshold for daxpy benchmark is set to 38000. So we expect to see the effect of parallelization only when the vector size gets greater than or equal to 38000.

The ratio of performance r is shown in Figure2. For small vectors $\leq 103,258$ hpxMP scales less than OpenMP especially when the thread number is large, but gets closer OpenMP as the vector size is increasing. Compared to OpenMP, the best performance of hpxMP is achieved between vector size $431,318$ to $2,180,065$ and the threads number between 1 to 7. Except for some outliers, hpxMP is between 0% and 30% slower than the optimized OpenMP version for larger vector sizes . The scaling plots are shown in Figure6.

Listing 7: Implementation of gcc entry fork call in hpxMP

```
1 void
2 xexpand(KMP_API_NAME_GOMP_PARALLEL)(void (*task)(void *), void *data, unsigned num_threads,
      unsigned int flags) {
3         omp_task_data * my_data = hpx_backend->get_task_data();
4         my_data->set_threads_requested(num_threads);
5         __kmp_GOMP_fork_call(task,(microtask_t )__kmp_GOMP_microtask_wrapper, 2, task, data);
6 }
```

Listing 8: Implementation of starting HPX in hpxMP

```
1  hpx::start(f, desc_cmdline, argc, argv, cfg, std::bind(&wait_for_startup, boost::ref(
       startup_mtx), boost::ref(cond), boost::ref(running)));
```

For all different number of threads, we see that both implementations behave similar until the parallelization starts. For vector sizes between 10^5 and 10^6 hpxMP is slower. For larger input sizes the implementations are comparable again.

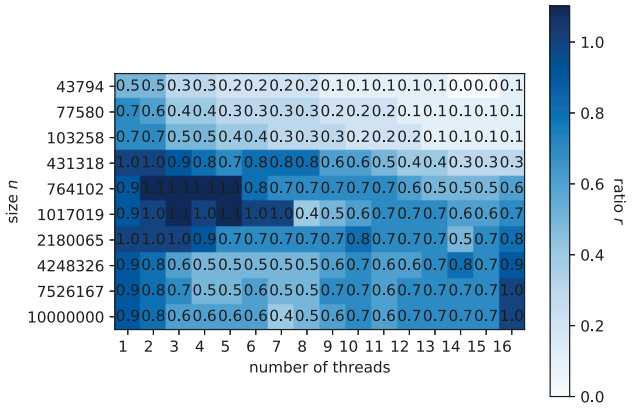


Figure 2: Performance Ratio using dvecdvecadd Benchmark (hpxMP/OpenMP)

6.2 Daxpy

Daxpy is a benchmark to multiply a number β with a dense vector a, add the result with a dense vector b, and store the result in same vector b, where $\beta \in \mathbb{R}$, and $a, b \in \mathbb{R}^n$. The operation used for this benchmark is $b[i] = b[i] + 3.0 * a[i]$. Same as dvecdvecadd benchmark, the parallelization threshold for daxpy benchmark is set to 38, 000. So we expect to see the effect of parallelization only when the vector size gets $\geq$ 38, 000.

The ratio of performance r is shown in Figure3. For small vectors $\leq$ 103, 258 hpxMP scales less than OpenMP especially when the thread size is large but gets closer OpenMP as the vector size is increasing. Compared to OpenMP, the best performance of hpxMP is achieved between vector size 431, 318 to 1, 017, 019 and the threads number between 1 to 8. Except for some exceptions, hpxMP is between 0% and 40% slower than the optimized OpenMP version for larger vector sizes The scaling plot is shown in Figure7 shows that for all different number of threads, we see that both implementations behave similarly until the parallelization starts. For vector sizes between 10^5 and 10^6 hpxMP is slower but for larger input sizes the implementations are comparable again.

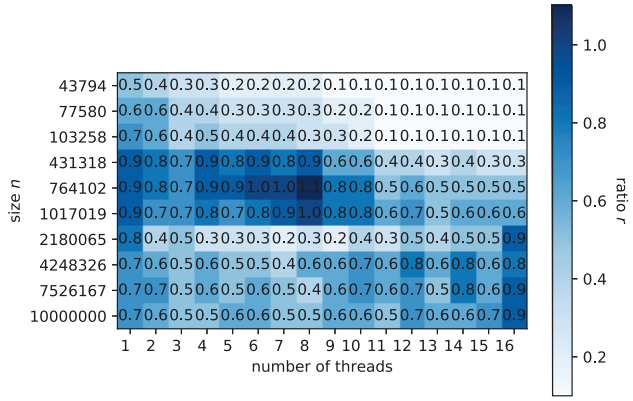


Figure 3: Performance Ratio using daxpy Benchmark (hpxMP/OpenMP)

6.3 Dense Matrix Addition

Dense Matrix Addition(dmatdmatadd) is a benchmark to add two dense matrix$A$ and B and stores the result in matrix C, where $A, B \in \mathbb{R}^{n \times n}$. The matrix addition operation is $C[i, j] = A[i, j] + B[i, j]$. The ratio of performance r is shown in Figure4. The scaling plot is shown in Figure4. For the dmatdmatadd benchmark, the parllelization threshold set by Blaze is 36, 100. Whenever the target matrix has more than or equal to 36, 100 elements (corresponding to matrix size 190 by 190), this operation is executed in parallel.

Figure4 shows that OpenMP performs better especially when the matrix size is small and the number of thread is large. For a larger number of threads, hpxMP gets closer to OpenMP for larger matrix sizes. Except for some exceptions, hpxMP is between 0% and 40% slower than the optimized OpenMP version. Figure8 shows that for all different number of threads, we see that both implementations

behave similar until the parallelization starts. For matrix sizes between 230 and 455 hpxMP is slower but the implementations are comparable again as the input size is increasing.

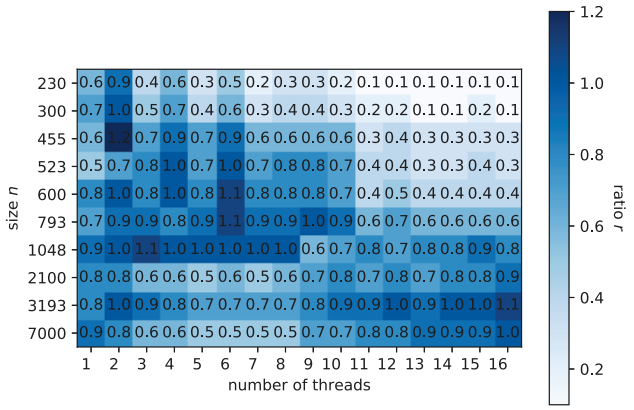


Figure 4: Performance Ratio using dmatdmatadd Benchmark (hpxMP/OpenMP)

6.4 Dense Matrix Multiplication

Dense Matrix Multiplication(dmatdmatmult) is a benchmark that multiplies two dense matrix A and B and stores the result in matrix C, where $A, B \in \mathbb{R}^{n \times n}$. The matrix addition operation is $C = A * B$. The ratio of performance r is shown in Fig.5. The scaling plot is shown in Fig.9. For the dmatdmatmult benchmark, the parallelization threshold set by Blaze is 3,025. Whenever the target matrix has more than or equal to 36, 100 elements (corresponding to matrix size 55 by 55), this operation is executed in parallel.

Fig.5 shows that OpenMP outperforms hpxMP only when the matrix size is between 230 and 300 and the number of thread is between 12 to 16. hpxMP gets as fast as OpenMP for other vector sizes. Fig.9 shows that for all different number of threads, we see that both implementations behave similar until the parallelization starts. For matrix sizes between 74 and 113 hpxMP is slower. For larger input sizes, the implementations are comparable again.

7 CONCLUSION AND OUTLOOK

This paper presents the design and architecture of an OpenMP runtime library built on top of HPX that supports most of the OpenMP V3 specification. We have demonstrated its full functionality by running various linear algebra benchmarks of the Blaze C++ library that for the tested functionalities relies on OpenMP for its parallelization needs. By replacing the compiler-supplied OpenMP runtime with our own, we were able to compare the performance of the two implementations. In general, our implementation is not able to reach the same performance compared to the native OpenMP solution yet. This is in part caused by Blaze being optimized for the compiler-supplied OpenMP implementations. We will work on optimizing the performance of hpxMP in the future. We have however demonstrated the viability of our solution for providing a smooth migration for applications that either directly or indirectly

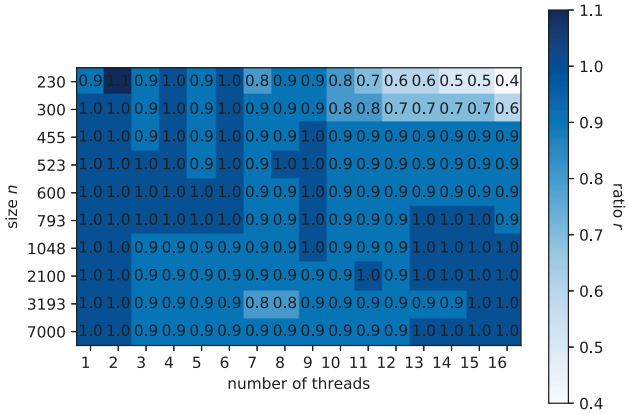


Figure 5: Performance Ratio using dmatdmatmult Benchmark (hpxMP/OpenMP)

depend on OpenMP but have to be extended to benefit from a more general task based programming model.

ACKNOWLEDGMENTS

We thank Jeremy Kemp for providing the initial implementation of hpxMP[13] which was extended by the authors. The work on hpxMP is funded by the National Science Foundation (award 1737785).

A SOURCE CODE

The source code of hpxMP is available on github[14] released under the BSL 1.0.

REFERENCES

Robert A. Alfieri. 1994. An efficient kernel-based implementation of POSIX threads. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1 (USTC'94)*. USENIX Association, Berkeley, CA, USA, 5–5. http://portal.acm.org/citation.cfm?id=1267257.1267262

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

David A. Bader. 2016. Evolving MPI+X Toward Exascale. *Computer* 49, 8 (2016), 10. https://doi.org/doi.ieeecomputersociety.org/10.1109/MC.2016.232

Richard F Barrett, Dylan T Stark, Courtenay T Vaughan, Ryan E Grant, Stephen L Olivier, and Kevin T Pedretti. 2015. Toward an evolutionary task parallel integrated MPI+ X programming model. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 30–39.

John Biddiscombe, Anton Bikineev, Thomas Heller, and Hartmut Kaiser. 2017. ZERO COPY SERIALIZATION USING RMA IN THE HPX DISTRIBUTED TASK-BASED RUNTIME. (2017).

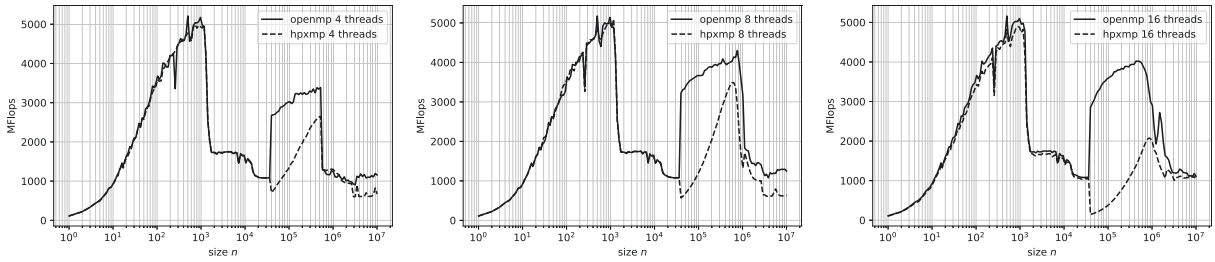
L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

C++ Standards Committee. 2011. *ISO/IEC 14882:2011, Standard for Programming Language C++ (C++11)*. Technical Report. ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee). https://wg21.link/N3337, last publicly available draft.

C++ Standards Committee. 2017. *ISO/IEC DIS 14882, Standard for Programming Language C++ (C++17)*. Technical Report. ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee). https://wg21.link/N4659, last publicly available draft.

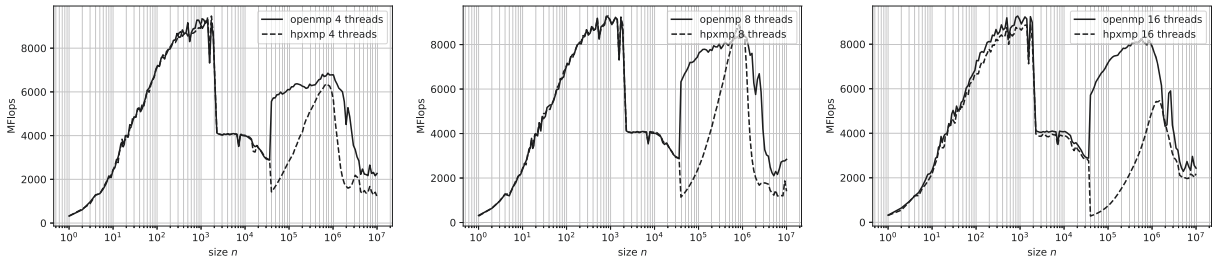
Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *International Journal of High Performance*

[13] https://github.com/kempj/hpxMP
[14] https://github.com/STEllAR-GROUP/hpxMP



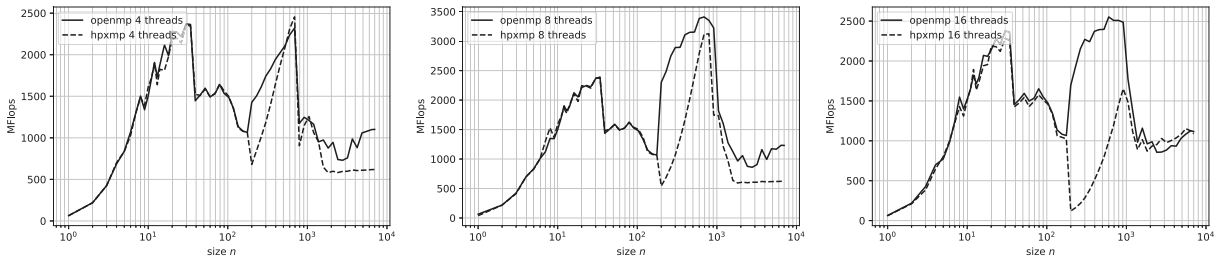
(a) **(b)** **(c)**

Figure 6: Scaling plots for dvecdvecadd Benchmarks for different number of threads: (a) 4, (b) 8, and (c) 16



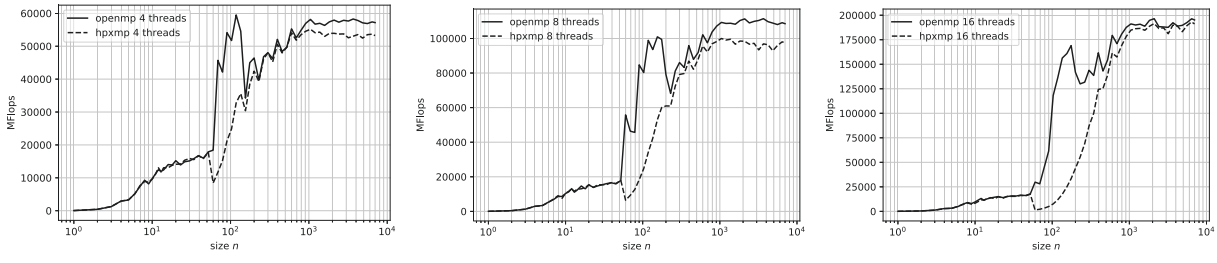
(a) **(b)** **(c)**

Figure 7: Scaling plots for daxpy Benchmarks for different number of threads: (a) 4, (b) 8, and (c) 16



(a) **(b)** **(c)**

Figure 8: Scaling plots for dmatdmatadd Benchmarks for different number of threads: (a) 4, (b) 8, and (c) 16



(a) **(b)** **(c)**

Figure 9: Scaling plots for dmatdmatmult Benchmarks for different number of threads: (a) 4, (b) 8, and (c) 16

Computing Applications (IJHPCA) 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442 https://dx.doi.org/10.1177/1094342007078442.

Marcin Copik and Hartmut Kaiser. 2017. Using SYCL as an Implementation Framework for HPX. Compute. In *Proceedings of the 5th International Workshop on OpenCL*. ACM, 30.

Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55. https://doi.org/10.1109/99.660313

Bronis R. de Supinski Michael Klemm. 2017. *OpenMP Technical Report 6:Version 5.0 Preview 2.* Technical Report. OpenMP Architecture Review Board.

H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Patrick Alken, Michael Booth, and Fabrice Rossi. 2002. GNU scientific library. *Network Theory Ltd* 3 (2002).

Patricia Grubel, Hartmut Kaiser, Jeanine Cook, and Adrian Serio. 2015. The performance implication of task size for applications on the hpx runtime system. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 682–689.

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. 2017. HPX–An open source C++ Standard Library for Parallelism and Concurrency.

Thomas Heller, Bryce Lelbach, Kevin Huck, John Biddiscombe, Patricia Grubel, Alice Koniges, Matthias Kretz, Dominic Marcello, David Pfander, Adrian Serio, Juhan Frank, Geoffrey Clayton, Dirk PflÃijger, David Eder, and Hartmut Kaiser. 2018. Harnessing Billions of Tasks for a Scalable Portable Hydrodynamic Simulation of the Merger of Two Stars. *International Journal of High Performance Computing Applications (IJHPCA)* (2018).

Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. 2012. High performance smart expression template math libraries. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 367–373.

Intel. 2019. Intel Thread Building Blocks. http://www.threadingbuildingblocks.org/ http://www.threadingbuildingblocks.org.

Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J Ramanujam. 2016. A massively parallel distributed n-body application implemented with hpx. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE, 57–64.

Zahra Khatami, Hartmut Kaiser, and J Ramanujam. 2017. Redesigning op2 compiler to use hpx runtime asynchronous techniques. *arXiv preprint arXiv:1703.09264* (2017).

Charles E. Leiserson. 2009. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*. ACM, New York, NY, USA, 522–527. https://doi.org/10.1145/1629911.1630048

Tim Mattson. 2013. A "Hands-on" Introduction to OpenMP.

Microsoft. 2010. Microsoft Parallel Pattern Library. http://msdn.microsoft.com/en-us/library/dd492418.aspx http://msdn.microsoft.com/en-us/library/dd492418.aspx.

OpenMP Consortium. 2018. *OpenMP Specification Version 5.0.* Technical Report. OpenMP Consortium. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

PPL. 2011. PPL - Parallel Programming Laboratory. http://charm.cs.uiuc.edu/ http://charm.cs.uiuc.edu/.

Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jüngel, and Siegfried Selberherr. 2016. ViennaCL—Linear Algebra Library for Multi-and Many-Core Architectures. *SIAM Journal on Scientific Computing* 38, 5 (2016), S412–S439.

Conrad Sanderson and Ryan Curtin. 2016. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software* (2016).

Bibek Wagle, Samuel Kellar, Adrian Serio, and Hartmut Kaiser. 2018. Methodology for Adaptive Active Message Coalescing in Task Based Runtime Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1133–1140.

Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 1–12.