hpxMP, An Implementation of OpenMP Using HPX

By

Tianyi Zhang

Project Report

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the School of Engineering at
Louisiana State University, 2019

Baton Rouge, LA, USA

Acknowledgments

I would like to express my appreciation to my advisor Dr. Hartmut Kaiser, who has been a patient, inspirable and knowledgeable mentor for me. I joined STE||AR group 2 years ago, with limited knowledge of C++ and HPC. I would like to thank Dr. Kaiser to bring me into the world of high-performance computing. During the past 2 years, he helped me improve my skills in C++, encouraged my research project in hpxMP, and allowed me to explore the world. When I have challenges in my research, he always points things to the right spot and I can always learn a lot by solving those issues. His advice on my research and career are always valuable. I would also like to convey my appreciation to committee members for serving as my committees. Thank you for letting my defense be to an enjoyable moment, and I appreciate your comments and suggestions. I would thank my colleagues working together, who are passionate about what they are doing and always willing to help other people. I can always be inspired by their intellectual conversations and get hands-on help face to face which is especially important to my study. A special thanks to my family, thanks for their support. They always encourage me and stand at my side when I need to make choices or facing difficulties.

hpxMP[1], An OpenMP runtime implemented using HPX

Tianyi Zhang

Department of Computer Science
Louisiana State University
Baton Rouge, Louisiana
2019

ABSTRACT

OpenMP is a popular standard for single node parallelization which provides easy to use APIs for C, C++ and FORTRAN users to parallelize their code. Nowadays, Asynchronous Many-task (AMTs) systems are catching increasing attention in the field of high-performance computing (HPC). AMTs are offering promising performance improvements using fine-grained tasking runtime systems. Many applications are directly or indirectly using both OpenMP and AMTs in its backend, which causes performance issues due to the incompatibility between the OpenMP threading mechanism and AMTs. They compete for resources and cause performance disasters, which is an increasing concern as the trend to have both OpenMP and AMTs in the same application. This project, hpxMP, is trying to solve this conflict and provides better compatibility and performance output. hpxMP is a library that implements functionalities following the OpenMP standard while using the HPX system underneath to schedule and manage tasking and threading. This project focuses on the implementation of hpxMP libraries. Besides, performance comparison using fair benchmarks between OpenMP provided by different vendors is also performed. Based on the testing results, hpxMP does a great job in connecting OpenMP and AMTs while sacrificing acceptable performance.

[1]https://github.com/STEllAR-GROUP/hpxMP

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Asynchronous many-task (AMT) systems are gaining increasing attention as a new programming paradigm for high performance computing (HPC) [2, 3]. They have shown superior efficiency for certain classes of applications. The OpenMP standard [4], which has been around since 1997, is widely employed in parallel computing due to its user-friendly pragmas and efficiency. Lots of AMTs are taking advantage of libraries that use OpenMP underneath to handle multi-threading [5, 6]. However, AMTs are not very comfortable with OpenMP since the user level threads of AMTs and system threads of OpenMP are not cooperating efficiently. Meantime, the C++ standardization community is focusing on providing higher-level functions to replace OpenMP `#pragma` based implementations because of the lack of integration into the C++ language. The new trending requires applications using OpenMP to migrate portions of code to AMTs. To help with this migration, this project report demonstrated a possible solution for the migration process, hpxMP, which is an OpenMP runtime implemented with HPX, implements all the corresponding pragmas in the OpenMP 3.0 [7]. Also, It introduces task-based computation in OpenMP 4.0 [8] and is keeping up to date with OpenMP 5.0 [9]. hpxMP exposes exactly the same APIs as other OpenMP implementation. However, hpxMP uses HPX light-weight threads instead of OS-level threads. It is designed to work with Clang and GCC compiler however other compiler support is on the way. By using HPX threads instead of system-level threads in OpenMP implementation, the gap between OpenMP and AMTs are bridged. This project aims to develop and ver-

ify the OpenMP implementation, hpxMP. Furthermore, the performance comparison is performed based on classic benchmarks such as Barcelona OpenMP Task Suit.

The OpenMP 3.0 standard [10] first introduced task-based programming to OpenMP and is extended over the years. The OpenMP 3.1 standard [11] added more optimization to task construct. Dependent tasks and taskgroup introduced an easier and more efficient way to users to synchronize tasks, which was proposed by the OpenMP 4.0 standard [8]. The OpenMP 4.5 standard [3] 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 [9] supports task reduction.

The structure of this project report is as follows: Chapter 2 describes the related work. Chapter 3 emphasizes the implementation of hpxMP in detail and Chapter 4 gives the benchmarks comparing hpxMP, GCC's OpenMP and Clang's OpenMP implementation. Finally, we draw conclusions in Chapter 5.

Chapter 2

Related Work

Parallelism on multi-core systems is gaining attention these years as the exploding amount of data and increasingly intensive computing demands. The increase in single-core computation is not satisfactory. Fine-grained parallelism was enabled by the POSIX Threads [12]. Intel's threading Building Blocks (TBB) [13], Microsoft's Parallel Pattern Library (PPL) [14], and Kokkos [15] are higher-level libraries providing parallel functions and features. Open Multi-Processing (OpenMP) [16] is widely accepted by applications and library developers. It is a standard that exposes fork-join model through compiler directives.

HPX is an extension of the C++ standard library for concurrency and parallelism. It extends the C++ standard library into parallel and even distributed cases. HPX is an AMT system that provides a C++ Standard equivalent API. HPX has been discussed and proposed in many publications [17, 18, 19, 20, 21]. HPX is employed as the backend of hpxMP in the context of this project. HPX is using light-weight threading system on the user level, compared to the fork-join mechanism on system level used by OpenMP, it is more efficient and thus provides better performance yield. The concepts such as dataflow, fine-grained synchronization, etc. are employed in the design of HPX. The combination of these ideas and design principles makes HPX unique [18]. The focus of HPX is always to solve the problems of scalability, resiliency, power efficiency which has been a long-time concern of the community. HPX is a runtime system implementing the ParalleX execution module [22, 23] and give programmers the ability to write asyn-

chronous code by using HPX threads. This project, hpxMP, reflected the advantage of using HPX by showing a more readable and concise code.

Task has been an important concept among the history of the OpenMP standard. 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 users 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 [24]. However, most of the research in this area has focused on MPI+X [25, 26] model.

[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

Chapter 3

Implementation

Users' applications should be compiled with OpenMP flags using GCC or Clang in order to be able to use hpxMP. When running a user's program, hpxMP runtime instead of OpenMP runtime provided by other vendors is LD_PRELOAD to the program. Figure 3.1 shows the layers of hpxMP, which exposes exactly the same APIs as OpenMP on program layer, following the OpenMP standard. The OpenMP library functions, environment variables, and directives are passed to the hpxMP runtime where the user-level threading is supported by HPX. A list of pragma directives supported can be found at Table 3.1. A list of runtime library functions implemented in hpxMP is provided in Table 3.2, which is an essential part of hpxMP library, providing a way for programmers to interact with the underlying runtime.

Besides, OMPT(OpenMP Performance Toolkit) is also implemented as a toolkit provided to the users for the measurement of the user programs' performance. A list of implemented callbacks can be found in Table 3.3. Similar to llvm-OpenMP, hpxMP is

Table 3.1: Directives implemented in the program layer of hpxMP, see Figure 3.1. 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 3.2: Runtime library functions in hpxMP's program layer, see Figure 3.1. 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

following the convention of LLVM-Clang and provided a function layer mapping GCC generated function calls to the Clang ones.

3.1 Structure of Code

hpxMP have to provide the compiler required APIs in order to map the function calls to the back-end, the Intel and GCC OpenMP runtime calls generated by the compiler are implemented in intel_hpxMP and gcc_hpxMP. The bulk of the back-end support work is done in hpx_runtime, where threads and tasks are created and OpenMP APIs are implemented. The OMPT support is implemented in ompt_hpx_general, providing users a way to measure their programs' performance with little overhead. Cmake tests are under folder tests, containing both regression tests and unit tests.

3.2 Class Implementation

An instance of `omp_task_data` class is set to be associated with each HPX thread by calling `hpx::threads::set_thread_data`. Instances of `omp_task_data` are passed by a

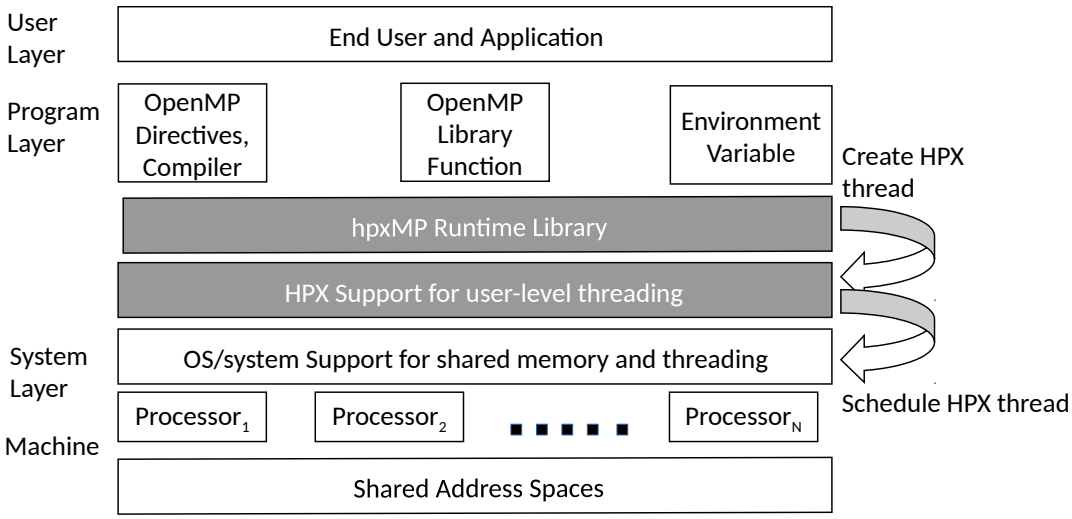


Fig. 3.1: Layers of an hpxMP application. Applications are compiled with OpenMP flags and linked against the hpxMP runtime library where HPX threads perform the computation in parallel. Gray layers are implemented in hpxMP. This figure is adapted from [1].

.

raw pointer which is `reinterpret_cast`ed to `size_t`. For better memory management, a smart pointer `boost::intrusive_ptr` is introduced to wrap around `omp_task_data`. The class `omp_task_data` consists the information describing a thread, such as a pointer to the current `team`, `taskLatch` for synchronization and if the `task` is in `taskgroup` . The `omp_task_data` can be retrieved by calling `hpx::threads::get_thread_data` when needed, which plays an important role in hpxMP runtime.

Another important class is `parallel_region`, containing information in a `team`, such as `teamTaskLatch` for task synchronization, number of threads requested under the parallel region, and the depth of the current team.

Listing 3.1: Implementation of __kmpc_fork_call in hpxMP

```
1  void __kmpc_fork_call(ident_t *loc, kmp_int32 argc,
       kmpc_micro microtask, ...)
2  {
3          vector<void*> argv(argc);
4          va_list ap;
5          va_start( ap, microtask );
6          for( int i = 0; i < argc; i++ )
7          {
8                  argv[i] = va_arg( ap, void * );
9          }
10         va_end( ap );
11         void ** args = argv.data();
12         hpx_backend->fork(__kmp_invoke_microtask, microtask,
               argc, args);
13 }
```

3.3 Parallel Construct

Parallelization is started by the directive `#pragma omp parallel`. Codes inside the parallel structure written by users are executed in parallel. The library function `__kmpc_fork_call` implemented by hpxMP is called by compiler generated argument when compilers see the `#pragma omp parallel`, see Listing 3.1. Team of HPX threads are created inside the function according to number of threads users requested. A team is defined as a set of one or more threads in the execution of a parallel region [27]. The fork function call explicitly registers the HPX threads by calling `hpx::applier::register_thread_nullary`, see Listing 3.2. Each thread is provided with a function routine to be executed. The synchronization of threads is achieved by passing HPX Latch to each threads, which will be discussed later in Section 3.6.

Listing 3.2: Implementation of hpx_runtime::fork in hpxMP

```
1  int running_threads = parent->threads_requested;
2  hpxmp_latch threadLatch(running_threads+1);
3  for( int i = 0; i < parent->threads_requested; i++ )
4  {
5          hpx::applier::register_thread_nullary(...,
             boost::ref(threadLatch),...);
6  }
7  threadLatch.count_down_and_wait();
8  // wait for all the tasks in the team to finish
9  team.teamTaskLatch.wait();
```

Listing 3.3: Implementation of __kmpc_for_static_init in hpxMP

```
1  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 ) {
2  //code to determine each thread's lower and upper
     bound (*p_lower, *p_upper)
3  //with the given thread id, schedule type and stride
     .
4  }
```

3.4 Loop Construct

`For` loop parallelization is often a big concern of OpenMP. OpenMP defined an easy to use user interface which runs several iterations of the `For` loop in different threads. The iteration scheduling policy is determined by specifying the dynamic or static following the directive. The default type of scheduling is static, where chunk size is determined by the number of threads and iterations. Each thread take roughly the same amount of work and join at the end of the loop. The function call `__kmpc_for_static_init`, see Listing 3.3, `__kmpc_dispatch_next` and `__kmpc_dispatch_fini` are invoked by the compiled `#pragma omp for` with the associated structured for loop block. The Round-robin role is employed in properly distribute the chunks among threads.

3.5 Tasking Mechanism

Tasking is an important feature of OpenMP. Explicit tasks are created using `#pragma omp task` in hpxMP. HPX threads are created with the task directives and tasks are running on these HPX threads created. `_kmpc_omp_task_alloc` is allocating, initializing tasks and then return the generated tasks to the runtime. `__kmpc_omp_task` is called with the generated `task` parameter and passed to the `hpx_runtime::create_task`. The tasks are then running as a normal priority HPX thread by calling function `hpx::applier::register_thread_nullary`, see Listing 3.4. Synchronization in tasking implementation of hpxMP are handled with HPX latch, which will be discussed later in Section 3.6.

Task dependency was introduced with OpenMP 4.0. The depend clause is `#pragma omp task depend(in: x)depend( out: y)depend(inout: z)`. Certain dependency should be satisfied among tasks specified by users. In the implementation, `future` in HPX is employed. The functionality called `future` allows for the separation of the initiation of an operation and the waiting for the result. A list of tasks current task depend on are stored in a `vector<shared_future<void>>` and `hpx::when_all(dep_futures)` are called to inform the current task when it is ready to run.

In November 2018, OpenMP 5.0 was released and added great extension to the tasking structure in OpenMP. `task_reduction` along with `in_reduction` gives users a way to tell the compiler reduction relations among tasks and specify the tasks in taskgroup which are participating the reduction. The implementation of taskgroup can be found in Listing 3.5.

Listing 3.4: Implementation of task scheduling in hpxMP

```
1  kmp_task_t* __kmpc_omp_task_alloc(... )
2  {
3          kmp_task_t *task = (kmp_task_t*)new char[task_size +
               sizeof_shareds];
4          //lots of initilization goes on here
5          return task;
6  }
7  void hpx_runtime::create_task( kmp_routine_entry_t task_func
       , int gtid, intrusive_ptr<kmp_task_t> kmp_task_ptr){
8       auto current_task_ptr = get_task_data();
9          //this is waited in taskwait, wait for all tasks
             before taskwait created to be done
10         // create_task function is not supposed to wait
             anything
11          current_task_ptr->taskLatch.count_up(1);
12         //count up number of tasks in this team
13         current_task_ptr->team->teamTaskLatch.count_up(1);
14         //count up number of task in taskgroup if we are
             under taskgroup construct
15         if(current_task_ptr->in_taskgroup)
16         current_task_ptr->taskgroupLatch->count_up(1);
17         //Create a normal priority HPX thread with the
             allocated task as argument.
18         hpx::applier::register_thread_nullary(.....)
19         return 1;
20 }
```

Listing 3.5: Implementation of __kmpc_taskgroup and __kmpc_end_taskgroup in hpxMP

```
1  void __kmpc_taskgroup( ident_t* loc, int gtid )
2  {
3          auto task = get_task_data();
4          intrusive_ptr<kmp_taskgroup_t> tg_new(new
               kmp_taskgroup_t());
5          tg_new->reduce_num_data = 0;
6          task->td_taskgroup = tg_new;
7          task->in_taskgroup = true;
8          task->taskgroupLatch.reset(new latch(1));
9  }
10 void __kmpc_end_taskgroup( ident_t* loc, int gtid )
11 {
12         auto task = get_task_data();
13         task->taskgroupLatch->count_down_and_wait();
14         task->in_taskgroup = false;
15         auto taskgroup = task->td_taskgroup;
16         __kmp_task_reduction_fini(nullptr,taskgroup);
17
18 }
```

3.6 Thread and Task Synchronization

Exponential backoff, manually dealing with locks and condition variables were used for thread synchronization which is hard to manage, inefficient,difficult to reason the correctness of the implementation logic. HPX latch, see Listing 3.6 provides an easier to use and more efficient way to manage thread and task synchronization originally proposed in the draft C++ library Concurrency Technical specification[1]. Latch in HPX is implemented with mutex, condition variable, and locks however is well-designed and higher level. An internal counter is initialized in a latch to keep track of a calling thread needs to be blocked. The latch blocks one or more threads from executing until the counter reaches 0. Several member functions such as `wait()`, `count_up()`, `count_down()`, `count_down_and_wait()` of the Latch class is provided. The difference

[1] https://github.com/cplusplus/concurrency-ts/blob/master/latch_barrier.html

Listing 3.6: Defination of Latch Class in HPX

```
1  class Latch
2  {
3  public:
4          void count_down_and_wait();
5          void count_down(std::ptrdiff_t n);
6          bool is_ready() const noexcept;
7          void wait() const;
8          void count_up(std::ptrdiff_t n);
9          void reset(std::ptrdiff_t n);
10 protected:
11         mutable util::cache_line_data<mutex_type> mtx_;
12         mutable util::cache_line_data<local::detail::
              condition_variable> cond_;
13         std::atomic<std::ptrdiff_t> counter_;
14         bool notified_;
15 }
```

between `count_down()` and `count_down_and_wait()` is if the thread will be blocked if the data member inside Latch is not equal to 0 after decreasing the counter by 1. In the parallel region, when one thread is spawning a team of threads, an HPX latch called threadLatch will be initialized to `threads_requested+1` and member function `threadLatch.count_down_and_wait()` is called by the parent thread after threads are spawned, making parent threads wait for child threads to finish their work, see Listing 3.2. The Latch is passed as a reference to each child thread and the member function `threadLatch.count_down()` is called by each child thread when their works are done. When all the child threads have called the member function, the internal counter of `threadLatch` will be reduced to 0 and the thread will be released. For task synchronization, the implementation is trickier and needs to be carefully designed. In Listing 3.4, three Latches `taskLatch`, `teamTaskLatch`, and `taskgroupLatch` are `count_up(1)` when a task is created. Based on the definition of OpenMP standard, tasks are not necessarily synchronized unless a `#pragma omp taskwait` or `#pragma omp barrier` is called either explicitly or implicitly, see Listing 3.7. The member function of Latch `count_down(1)` is

Listing 3.7: Implementation of taskwait and barrier wait

```
1  void hpx_runtime::task_wait()
2  {
3          auto task = get_task_data();
4          intrusive_ptr<omp_task_data> task_ptr(task);
5          task_ptr->taskLatch.wait();
6  }
7  void hpx_runtime::barrier_wait()
8  {
9          auto *team = get_team();
10         task_wait();
11         //wait for all child tasks to be done
12         team->teamTaskLatch.wait();
13 }
```

called when a task is done with its work. `TaskLatch` only matters when `#pragma omp taskwait` is specified, where `taskLatch.wait()` is called, making sure the current task is suspended until all child tasks that it generated before the taskwait region complete execution. The `teamTaskLatch` is used to synchronize all the tasks under a team, including all child tasks this thread created and all of their descendant tasks. An implicit barrier is always triggered at the end of parallel regions, where `team->teamTaskLatch.wait()` is called and the current task can be suspended. Taskgroup implementation in hpxMP is similar to a barrier, see Listing 3.5. All tasks under the same taskgroup are blocked until the `taskgroupLatch->count_down_and_wait()` function inside `kmpc_end_taskgroup` is called by all child tasks and their descend tasks.

3.7 GCC support

The hpxMP library was initially implemented following the Clang API convention which is different from the GCC translate convention. In order to achieve the support for programs compiled with GCC compiler, we added a layer that maps GCC function entry

Listing 3.8: Implementation of API mapping from GCC to Clang

```
1 #define KMP_API_NAME_GOMP_PARALLEL
                              GOMP_parallel
2 extern "C" void
3 xexpand(KMP_API_NAME_GOMP_PARALLEL)(void (*task)(
    void *), void *data, unsigned num_threads,
    unsigned int flags);
```

to Clang ones, see Listing 3.8. The arguments provided by the compiler are processed by those mapping function and passed to the hpxMP runtime functions directly or call the equivalent Clang supported entries. Those programs compiled with GCC can use the underneath hpxMP library without any difference.

3.8 OpenMP Performance Toolkit

First party OpenMP performance toolkit (OMPT) is defined as part of the standard and provided with many venders. It gives users the ability to analyze the performance and debug their OpenMP programs without introducing too much overhead. The routine of hpxMP such as creating threads, entering the parallel region or creating tasks can be registered as callbacks with OMPT. A list of callbacks hpxMP have implemented can be found in table 3.3.

Table 3.3: OMPT callbacks implemented in hpxMP runtime library, see Figure 3.1. 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

Chapter 4

Benchmark

Benchmarks are available for testing the performance of OpenMP. This project employed a daxpy benchmark written by the author and popular and widely used benchmarks from Barcelona OpenMP Task Suite[1]. Three implementations of the OpenMP standard, llvm-OpenMP, GOMP, and hpxMP are compared using those benchmarks. Under the measurement, threads are pinned for both llvm-OpenMP and GOMP. The benchmarks are tested on Marvin (2 x Intel® Xeon® CPU E5-2450 0 @ 2.10GHz and 48 GB RAM), a node having 16 physical cores in two NUMA domains. The software dependencies are shown in Table 4.1. Also, the performance of OpenBLAS benchmark was also tested on medusa(2 x Intel® Xeon® Gold 6148 CPU @ 2.40GHz and 98 GB RAM), a node having 40 physical cores in two NUMA domains for cross-reference with the same software settings.

Table 4.1: Overview of the compilers, software, and operating system used to build hpxMP, HPX, Blaze and its dependencies. The OpenMP implementation comes with the compiler are used in this measurement.

Category	Property	Category	Property
OS	CentOS Linux release 7.6.1810 (Core)	Kernel	3.10
Clang Compiler	8.0.0	GCC Compiler	9.1.0
LLVM OpenMP	4.5	GOMP	4.5
HPX[2]	commit id: 1000bb0	Blaze[3]	3.4
gperftools	2.7	boost	1.70.0
hpxMP[4]	commit id: 2c04f9d		

[1] https://github.com/bsc-pm/botsand

4.1 Daxpy Benchmark

In order to compare the performance of `#pragma omp parallel for`, which is a fundamental pragma in OpenMP, Daxpy benchmark is used in this measurement. Daxpy is a benchmark that measures the multiplication of a float number c with a dense vector a consists 32 bit floating numbers, then add the result with another dense vector b (32 bit float), the result is stored in the same vector b, where $c \in FLOAT$ and $a, b \in FLOAT^n$.

We wrote this toy application, See Listing 4.1 to compare the performance calculated in Mega Floating Point Operations Per Second (MFLOP/s). We determine the speedup of the application by scaling our results to the single-threaded run of the benchmark using hpxMP.

Figure 4.1 shows the speedup ratio with different numbers of threads. Our first experiment compared the performance of the OpenMP implementations when the vector size was set to 10^3, see Figure 4.1d. llvm-OpenMP runs the fastest while following with GOMP and hpxMP Figure 4.1c shows that with a vector size of 10^4, GOMP and llvm-OpenMP are still able to exhibit some scaling while hpxMP struggles to scale past 4 threads. For very large vector sizes of 10^6, the three implementations perform almost identically. hpxMP is able to scale in these scenarios because there is sufficient work in each task in order to amortize the cost of the task management overheads.

4.2 Barcelona OpenMP Task Suit

Two benchmarks, SORT and FFT, from Barcelona OpenMP Task Suit, are tested in this project. The fast parallel sorting, a variation of the ordinary mergesort [28] and One-dimensional Fast Fourier Transform of an n sized vector using the Cooley Tukey algorithm is used to test the task-based performance of our implementation.

Listing 4.1: Daxpy benchmark

```
1  int main(int argc, char* argv[]){
2          int64_t vector_size = 0;
3          if(argc > 1)
4                  vector_size = atoi(argv[1]);
5          std::vector<float> a(vector_size,2.0f);
6          std::vector<float> b(vector_size,3.0f);
7          std::vector<float> c(vector_size,0);
8          int count = 0;
9          std::chrono::duration<double> elapsed_seconds;
10         //properly start runtime before actually measuring
11         #pragma omp parallel
12                 sleep(1);
13         while(1)
14         {
15                 count++;
16                 auto start = std::chrono::system_clock::now
                      ();
17         #pragma omp parallel for
18                 for (int64_t i = 0; i < vector_size; i++){
19                         c[i] = 6.0*a[i]+b[i];
20                 }
21                 auto end = std::chrono::system_clock::now();
22
23                 if(count==1)
24                         elapsed_seconds=end-start;
25                 else
26                         elapsed_seconds+=end-start;
27
28                 if((elapsed_seconds.count()>=30 && count >=
                      5) || elapsed_seconds.count()>=60)
29                         break;
30         }
31         double elapsed_secs = elapsed_seconds.count();
32         float operations = vector_size*3/elapsed_secs
              /1000000*count;
33         for (int64_t i = 0; i < vector_size; i++){
34                 if(c[i] != 15)
35                         throw std::invalid_argument("wrong␣
                              result");
36         }
37         std::cout<<","<<operations<<",\""<<std::endl;
38         return 0;
39 }
```

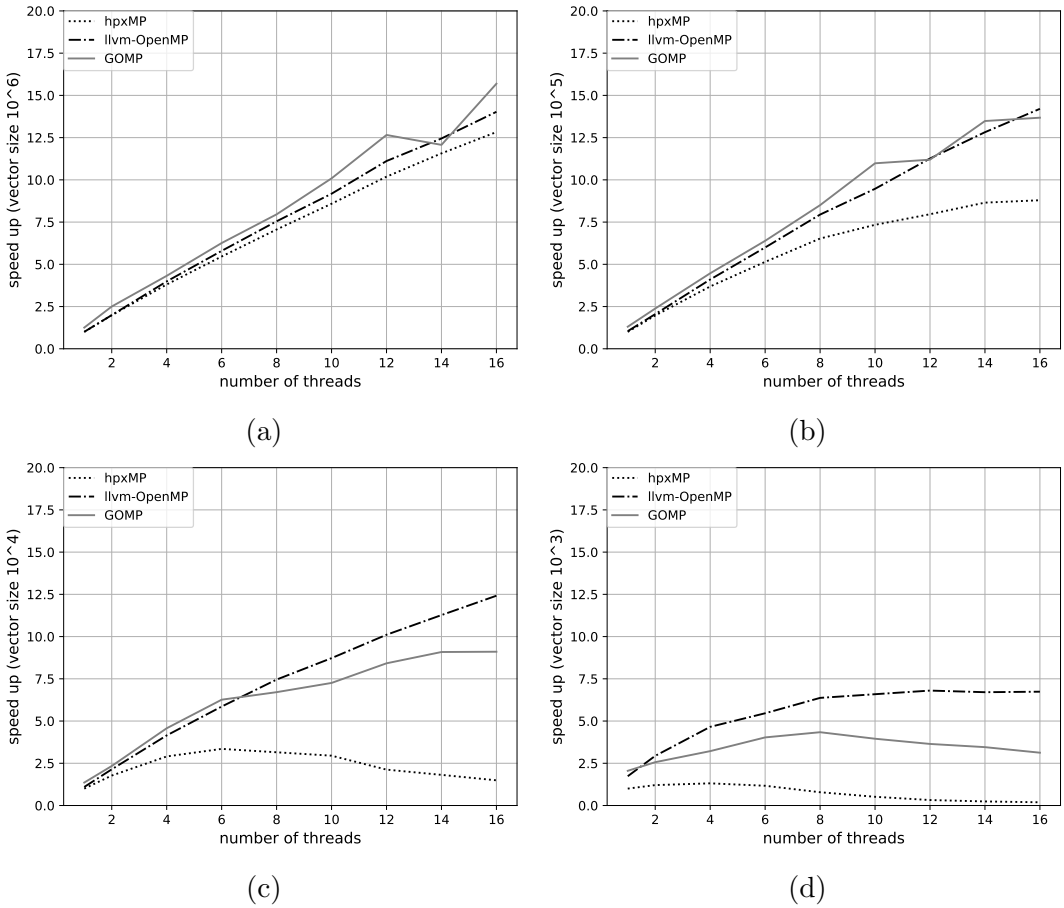


Fig. 4.1: Scaling plots for the Daxpy benchmarks running with different vector sizes: (a) 10^6, (b) 10^5, (c) 10^4, and (d) 10^3. The graphs demonstrate the relationship between speedup and the number of threads when using hpxMP, llvm-OpenMP, and GOMP. Larger vector sizes mean larger tasks are created. The speedup is calculated by scaling the execution time of a run by the execution time of the single threaded run of hpxMP. A larger speedup factor means a smaller execution time of the sample.

4.2.1 SORT

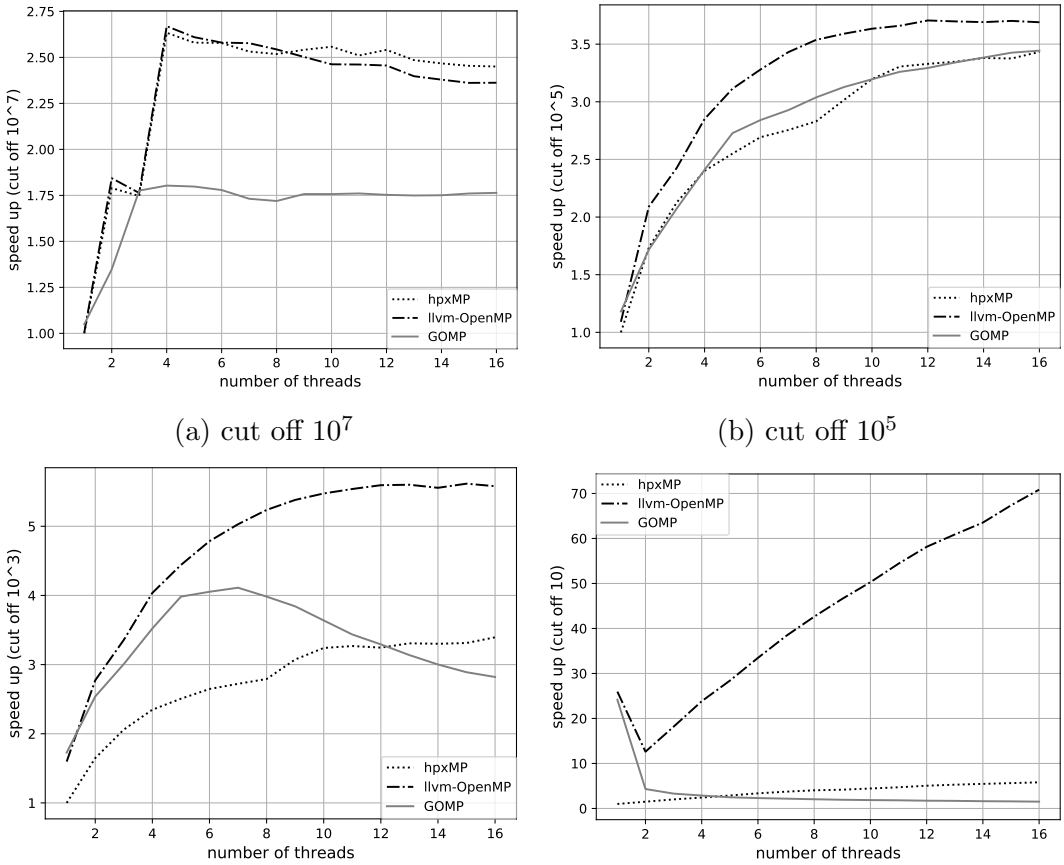
In SORT benchmark, a random array with 10^7 32-bit numbers is sorted with cut off values from 10 to 10^7. The cut off value determines when to perform serial quicksort instead of dividing the array into 4 portions recursively where tasks are created. Higher cut off values create larger tasks and, therefore, fewer tasks are created. In order to simplify the experiment, the parallel merge is disabled and the threshold for insertion sort is set to 1 in this benchmark. For each cut off value, the execution time of hpxMP using 1 thread is selected as the base point to calculate speedup values. Figure 4.2 shows the speedup ratio when using different numbers of threads.

For the cut off value 10^7 (Figure 4.2a), the array is divided into four sections and four tasks in total are created. The speedup curve rapidly increases when moving from 2 threads to 4, but no significant speedup is achieved when using more than 4 threads in all three implementations. hpxMP and llvm-OpenMP show comparable performance while GOMP is slower.

The cut off value of 10^5 (Figure 4.2b) increases the number of tasks generated. In this case, llvm-OpenMP has a performance advantage while hpxMP and GOMP show comparable performance.

For the cut off value 10^3 (Figure 4.2c), llvm-OpenMP shows a distinct performance advantage over hpxMP and GOMP. Nevertheless, hpxMP still scales across all the threads while GOMP has ceased to scale past 8 threads.

For a cut off value of 10 (Figure 4.2d), a significant number of tasks are created and the work for each task is considerably small. Here, hpxMP does not scale due to the large number of overheads associated with the creation of many user tasks. Because each task performs little work, the overhead that they create is not amortized by the increase in concurrency.



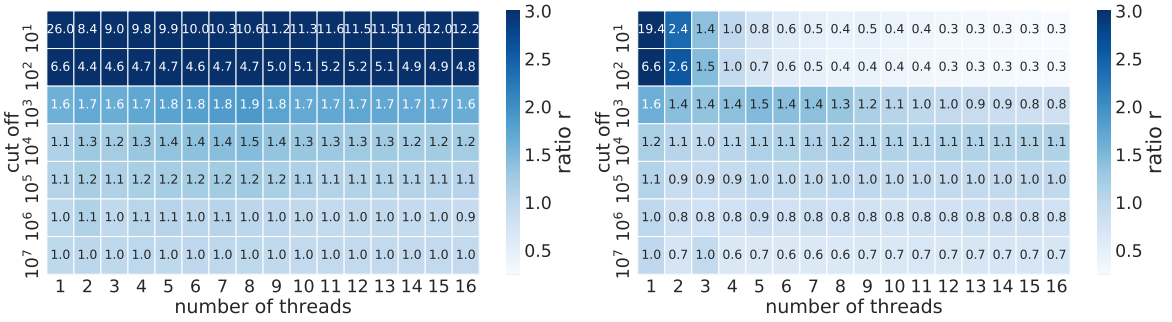
(a) cut off 10^7

(b) cut off 10^5

(c) cut off 10^3

(d) cut off 10

Fig. 4.2: Scaling plots for the Barcelona OpenMP Task suit's Sort Benchmarks ran with the cut off values of: (a) 10^7, (b) 10^5, (c) 10^3, and (d) 10. The relation between speedup and the number of threads using hpxMP, llvm-OpenMP, and GOMP are plotted. Higher cut off values means that a smaller number of larger tasks are created. The speedup is calculated by scaling the execution time of a run by the execution time of the single-threaded run of hpxMP. A larger speedup factor means a smaller execution time of the sample.



(a) llvm-OpenMP/hpxMP (b) GOMP/hpxMP

Fig. 4.3: Speedup Ratio of Barcelona OpenMP Task suit's Sort Benchmark ran over several threads and cut off values using the hpxMP, llvm-OpenMP, and GOMP implementations. Values greater than 1 mean that the OpenMP implementation achieved better performance when compared to hpxMP. Values below 1 indicate that hpxMP outperformed the OpenMP implementation.

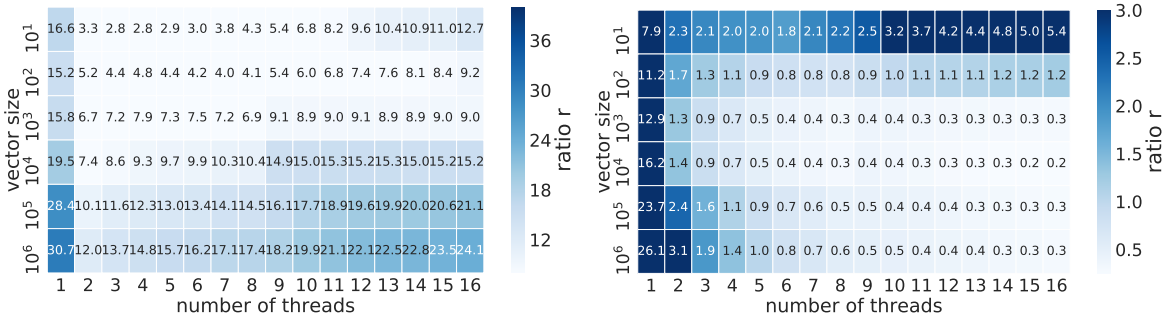
For a global view, the speedup ratio r is shown in Figure 4.3, where the larger the heatmap value is, the better performance OpenMP has achieved in comparison to hpxMP. Values below 1 mean that hpxMP outperforms the OpenMP implementation. As shown in the heatmap, llvm-OpenMP works best when the task granularity is small and the number of tasks created is high. GOMP is slower than both implementations in most cases. For large task sizes, hpxMP is comparable with llvm-OpenMP (Figure 4.3a). This result demonstrates that when the grainsize of the task is chosen well hpxMP will not incur a performance penalty.

4.2.2 FFT

FFT computes one-dimensional Fast Fourier Transform of a n sized vector using the Cooley Tukey [29] algorithm. Multiple tasks are created when the algorithm recursively divide Discrete Fourier Transform (DFT) into multiple smaller DFT's. The vector size is measured by the power of 10 from 10 to 10^6

The speedup ratio r is shown in Figure 4.4, from which we can tell llvm-OpenMP works

better among the three implementations. In this benchmark , the number of task created are proportional to the vector size and the task granularity are always the same and relatively small. The results are consistent with the results produced by another task benchmark, SORT. With higher number of tasks created and low task granularity, the speedup of llvm-OpenMP is the highest among the three implementations while hpxMP ranks the middle except for lower number of threads.



(a) llvm-OpenMP/hpxMP (b) GOMP/hpxMP

Fig. 4.4: Speedup Ratio of Barcelona OpenMP Task suit's FFT Benchmark ran over several threads and vector size values using the hpxMP, llvm-OpenMP, and GOMP implementations. Values greater than 1 mean that the OpenMP implementation achieved better performance when compared to hpxMP. Values below 1 indicate that hpxMP outperformed the OpenMP implementation.

Figure 4.5 shows the speedup ratio with different number of threads. For very large vector size, large number of tasks are created with relatively small task granularity. llvm-OpenMP and GOMP both has a speedup drop when the number of threads increases form 1 to 2, despite that, speedup increases for both OpenMP and hpxMP while decreases for GOMP with growing number of threads. The graph looks similar until the vector size decreases to very small, say 100. For very small vector size, all implementations has decreasing speedup with increasing number of thread.

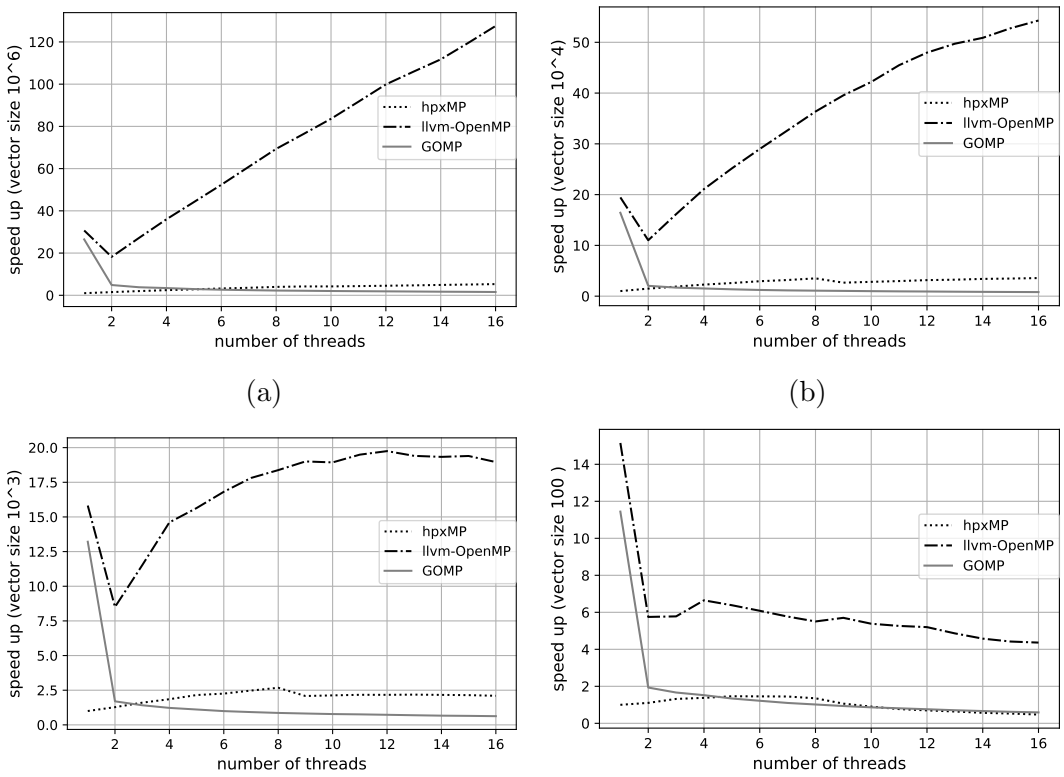


Fig. 4.5: Scaling plots for FFT Benchmarks for different vector size: (a) 10^6, (b) 10^4, (c) 10^3, and (d) 10^2. The relation between speedup and the number of threads using hpxMP, llvm-OpenMP, and GOMP are plotted. Higher vector size means that larger number of tasks with the same task granularity are created. The speedup is calculated by scaling the execution time of a run by the execution time of the single-threaded run of hpxMP. A larger speedup factor means a smaller execution time of the sample.

Chapter 5

Conclusion and Outlook

This project report introduced hpxMP as an implementation of the OpenMP standard utilizing the light-weight user-level HPX thread and describes the implementation of hpxMP in detail. hpxMP provides a way to close the compatibility gap between OpenMP and AMT systems as it demonstrates the technique as an example that enables AMT systems to leverage widely-used OpenMP libraries without sacrificing too much performance .

For the benchmarks described above, hpxMP exhibited similar performance when compared to other OpenMP runtimes for large task sizes. However, it faces performance difficulty in small grain sizes. This performance decrement arises from the more general-purpose threads created in HPX. These results show that hpxMP provides a way for bridging the compatibility gap between OpenMP and AMTs with acceptable performance for larger input sizes or larger task sizes.

In the future, improving performance for smaller input sizes by adding non-suspending threads to HPX, which do not require a stack, and thus reducing the overhead of thread creation and management are necessary.

Bibliography

[1] Tim Mattson. A "hands-on" introduction to openmp, 2013. v, 7

[2] Emmanuel Agullo, Olivier Aumage, Bérenger Bramas, Olivier Coulaud, and Samuel Pitoiset. Bridging the gap between openmp and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2794–2807, 2017. 1

[3] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, November 2015. 1, 2

[4] OpenMP Consortium. OpenMP Specification Version 5.0. Technical report, OpenMP Consortium, 2018. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. 1

[5] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications (IJHPCA)*, 21(3):291–312, 2007. https://dx.doi.org/10.1177/1094342007078442. 1

[6] Thomas Heller, Hartmut Kaiser, Andreas Schäfer, and Dietmar Fey. Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers. In *Proceedings of the ACM/IEEE Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA, SC Workshop)*, art. id 1, 2013. https://stellar.cct.lsu.edu/pubs/scala13.pdf. 1

[7] Charles E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM. 1

[8] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013. 1, 2

[9] OpenMP Architecture Review Board. OpenMP application program interface version 5.0, November 2018. 1, 2

[10] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008. 2

[11] OpenMP Architecture Review Board. OpenMP application program interface version 3.1, July 2011. 2

[12] Robert A. Alfieri. 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, pages 5–5, Berkeley, CA, USA, 1994. USENIX Association. 3

[13] Intel. Intel Thread Building Blocks, 2019. http://www.threadingbuildingblocks.org. 3

[14] Microsoft. Microsoft Parallel Pattern Library, 2010. http://msdn.microsoft.com/en-us/library/dd492418.aspx. 3

[15] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. 3

[16] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. 3

[17] Thomas Heller, Hartmut Kaiser, and Klaus Iglberger. Application of the ParalleX Execution Model to Stencil-Based Problems. *Computer Science - Research and Development*, 28(2-3):253–261, 2012. https://stellar.cct.lsu.edu/pubs/isc2012.pdf. 3

[18] Hartmut Kaiser, Thomas Heller, Bryce Adelstein Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the International Conference on Partitioned Global Address Space Programming Models (PGAS)*, art. id 6, 2014. https://stellar.cct.lsu.edu/pubs/pgas14.pdf. 3

[19] Hartmut Kaiser, Thomas Heller, Daniel Bourgeois, and Dietmar Fey. Higher-level Parallelization for Local and Distributed Asynchronous Task-based Programming. In *Proceedings of the ACM/IEEE International Workshop on Extreme Scale Programming Models and Middleware (ESPM, SC Workshop)*, pages 29–37, 2015. https://stellar.cct.lsu.edu/pubs/executors_espm2_2015.pdf. 3

[20] Hartmut Kaiser, Bryce Adelstein Lelbach aka wash, Thomas Heller, AgustÃn BergÃ©, John Biddiscombe, and Mikael Simberg et.al. STEllAR-GROUP/hpx: HPX V1.2.0: The C++ Standards Library for Parallelism and Concurrency, November 2018. 3

[21] Thomas Heller, Hartmut Kaiser, Patrick Diehl, Dietmar Fey, and Marc Alexander Schweitzer. Closing the performance gap with modern c++. In Michaela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing*, volume

9945 of *Lecture Notes in Computer Science*, pages 18–31. Springer International Publishing, 2016. 3

[22] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops*, pages 394–401, Los Alamitos, CA, USA, 2009. IEEE Computer Society. 3

[23] Thomas Sterling. ParalleX Execution Model V3.1. 2013. 3

[24] PPL. PPL - Parallel Programming Laboratory, 2011. http://charm.cs.uiuc.edu/. 4

[25] David A. Bader. Evolving mpi+x toward exascale. *Computer*, 49(8):10, 2016. 4

[26] Richard F Barrett, Dylan T Stark, Courtenay T Vaughan, Ryan E Grant, Stephen L Olivier, and Kevin T Pedretti. 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*, pages 30–39. ACM, 2015. 4

[27] Bronis R. de Supinski Michael Klemm. Openmp technical report 6:version 5.0 preview 2. Technical report, OpenMP Architecture Review Board, 2017. 8

[28] Selim G Akl and Nicola Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, 100(11):1367–1369, 1987. 18

[29] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. 23