Improving Parallel Applications Scalability for OP2 with HPX

Zahra Khatami[1,2], Hartmut Kaiser[1,2] and J. Ramanujam[1]
[1]Center for Computation and Technology, Louisiana State University
[2] The STE||AR Group, http://stellar-group.org

***Abstract*—Computer scientists and programmers face the difficultly in improving application scalability while using the conventional techniques. So, advanced runtime system is required to take full advantage of the available parallel resources in order to achieve to the highest parallelism level as possible. In this paper we present the capability of HPX in achieving a desired scalability and allowing asynchronous task execution.**

OP2 is an active library, which provides a framework of the parallel execution for unstructured grid applications on different multi-core/many-core hardware. OpenMP is used for a loop parallelization within an application code generated with OP2 for both single-threaded and multi-threaded machines. We use HPX for parallelizing an application generated with OP2 instead of using OpenMP. Our results shows the advantage of using asynchronous programming model implemented by HPX, which enables the better scalability due to the better latency hiding and using a fine-grain parallelism.

We compare the performance results of using HPX and OpenMP for a loop parallelization within an Airfoil application. The results shows an improvement in OP2 parallelization performance while using HPX. We present the results of the strong scalability test for an Airfoil application on one node with up to 64 threads. We observe on an average by about 27% scaling improvement in using `hpx::lcos::local::dataflow` compared to `#pragma omp parallel for`.

I. Introduction

Nowadays, a parallel application scalability is one of the major challenges when using the conventional techniques [1], [2]. To achieve an efficient scalability, it is needed to take full advantage of all available parallel resources. Avoiding resource starvation and hiding overheads can overcome this challenge and significantly improve a parallelism level. Parallelization is implemented with decomposing the domain space into several sub-domains and assigning each of them to the group of the processors. However, the overhead time due to the communication between processors inhibits the desired scalability. As a results, in addition to space, time should be considered as an another factor which helps to get a maximum possible parallelism level [3] [4]. So the parallelization should be done in both space and time domains.

HPX [5] overcomes this difficulty by significantly reducing the SLOW factors [6]. SLOW factors avoid the parallel scalability, which are explained as: Starvation, which is due to the poor utilization of resources; Latencies, which is the related delay for accessing to the remote resources; Overhead, which is the cost for managing parallel actions; Waiting, which is the related delay due to the shared resources [3].

HPX is a parallel C++ runtime system that aims to use the full parallelization capabilities of today's and tomorrow's hardware available for an application. HPX implements the concepts of the ParalleX execution model [7]–[9] on conventional systems including Windows, Macintosh, Linux clusters, XeonPhi, Bluegene/Q and the Android.

HPX enables asynchronous task execution which results in having a parallelization in both space and time domains. As a result, it removes a global barrier synchronization and improves a parallel performance. In this paper, HPX is used to improve a performance of OP2 for a parallel application scalability.

Op2 provides a framework of the parallel execution for unstructured grid applications [10]. OP2's design and development is presented in [11], [12]. With Op2, applications can be targeted to execute on different multi-core/many-core hardware [11], [13]. To achieve further performance gains with OP2 on modern multi-core/many-core hardware, some optimizations should be applied to improve performance for different parallel applications. This can be obtained by avoiding the SLOW factors as much as possible. OpenMP is used for a loop parallelization in OP2 on a single node and also on the distributed nodes using MPI. `#pragma omp parallel for` used for a loop parallelization in OP2 causes an implicit global barrier at the end of loop, which does not result in an efficient speedup. Using HPX parallelization methods instead of OpenMP helps eliminating these SLOW factors.

This paper describes the results from porting the parallel simulation backend of OP2 to utilize HPX. The performance of OP2 optimization with HPX is studied on a multi-core and many-core platforms through a standard unstructured mesh finite volume computational fluid dynamics (CFD) application, called Airfoil, which uses OP2 API and it is written in C. The experimental results are studied about an application scalability while using HPX in OP2 for a loop parallelization and these results are compared with OpenMP performance.

The remainder of this paper is structured as following: Section II explains briefly about OP2. An overview of the HPX is presented in Section III with the key features that distinguish it from the conventional techniques. Section IV presents the Airfoil application used in this research and gives details of using HPX for loop parallelization used in OP2. The experimental tests and the strong scaling are presented in Section V. Conclusion is provided in Section VI.

II. OP2

This paper presents an optimization study of the OP2 "active" library [10]. OP2 utilizes the source-to-source translation which targets a single application code to be written for different backend hardware platforms [11], [13], [14]. OP2 generates a code for single-threaded on a CPUs, multi-threaded using OpenMP for single SMP node of multi-core CPUs, using CUDA for a single NVIDIA GPU, using MPI and OpenMP for a cluster of CPUs and using MPI and CUDA for a cluster of NVIDIA GPUs [14].

OP2 API has been developed for an unstructured grids. So the algorithm includes four different parts: sets, data on sets, mapping connectivity between the sets and computation on the sets [11], [15]. Sets can be nodes, edges, faces or other elements. Data are values and parameters associated with these sets. Map is used to define a connectivity between sets. There is two different kinds of $loops$ defined in OP2: $indirect$ loopanddirect $loop$. The operation over a set is done within a loop in an application code. Arguments passed to loops in OP2 have an explicit indication that how each of them are accessed within a loop : `OP_READ` (read only), `OP_WRITE` (write) or `OP_INC` (increment - to avoid race conditions due to indirectly accessed) [10]. If data is accessed through a mapping, the loop is an $indirect$ loop. Otherwise, it is a $direct$ loop.

OP2's design is based on achieving a near-optimal performance for scaling on multi-core processors. In [13], [14], it is studied that OP2 API is able to produce a near-optimal performance in a parallel loops for different frameworks without the intervention of the application programmer. However, processors starvation, latencies and overhead communications in parallelization using conventional techniques usually inhibits a desired scalability. Most of the parallelization methods are based on the fork-join model. In the fork-join model, the computational process will be stopped if the results from the previous step are not completed yet. As a result, there is always a global barrier after each step.

`#pragma omp parallel for` is used in a code generated with OP2 for a single-threaded and also for a multi-threaded machine. There is an implicit global barrier using `#pragma omp parallel for`, which avoids extracting an optimal parallelism from a parallel application. In this research, HPX is used for loop parallelization for a single-threaded and a multi-threaded machine instead of using OpenMP. HPX allows automatically creating an execution tree of an application which represents a dependency graph. This enables HPX to have an asynchronous task execution. The performance of HPX is explained in more details in section III.

In this research Airfoil application is studied that is presented in [16]. This application includes both $direct$ and $indirect$ loops that all of them are parallelized with OpenMP. Optimization of Airfoil application with HPX for parallelizing loops used in a code generated with OP2 is discussed in more details in section IV.

The source-to-source code translator for OP2 is written in Matlab and Python [13]. Python source-to-source code translator is used in this research and some part of it is modified to automatically generate the parallel loops with HPX instead of OpenMP within an application problem.

III. HPX

In order to hide latencies even for very short operations, having a light-weight threads is needed which should have the extremely short context switching times, that results in becoming optimally executable within one cycle. HPX is a parallel C++ runtime system that facilitates distributed operations and enables fine-grained task parallelism. The fine-grained tasks result in better load balancing and lower communication overheads. HPX has been developed to overcome conventional limitations such as global barriers and poor latency hiding [6] by embracing a new way of coordinating parallel execution. It has been developed for different architectures, such as large Non Uniform Memory Access (NUMA) machines, SMP nodes and systems using Xeon Phi accelerators.

HPX's design focuses on parallelism than concurrency. Concurrency is defined to have several simultaneously computations and parallelism is simultaneous execution of tasks [17]. So it enables HPX to have both time and spatial parallelization [5] due to using $future$, which results in having asynchronous task execution. In HPX, asynchronous function execution is the fundamental basic of asynchronous parallelism.

A $future$ is a computational result that is initially unknown but becomes available at a later time [18]. The goal of using $future$ is to let every computation proceed as far as possible. Using $future$ enables the continuation of the process without waiting for the results of the previous step to be completed, which eliminates the global barriers at the end of the execution of the parallel loop. $future$ based parallelization provides rich semantics for exploiting the higher level parallelism available within each application that significantly improves scaling. Figure 1 shows the scheme of $future$ performance with 2 *localities*, where a *locality* is a collection of processing units (PUs) that have access to the same main memory.

Figure 1 shows that until returning the computed value, the other threads do not stop their process. Instead, they continue their process until they need the computation result from the previous step. Then, HPX threads requesting suspend until $future$ value is computed. Threads access the $future$ value by performing a $future.get()$ in Figure 1. When the result becomes available, the $future$ resumes all HPX suspended threads waiting for the value. It can be seen that this process eliminates the global barrier synchronizations at the end of application parallelization while only those threads that depend on the $future$ value are suspended. With this scheme, HPX allows asynchronous execution of threads.

IV. Airfoil code with HPX

For our evaluation we chose the Airfoil application presented in [16]. This model uses an unstructured grid and it consists of five parallel loops: `save_soln`, `adt_calc`,

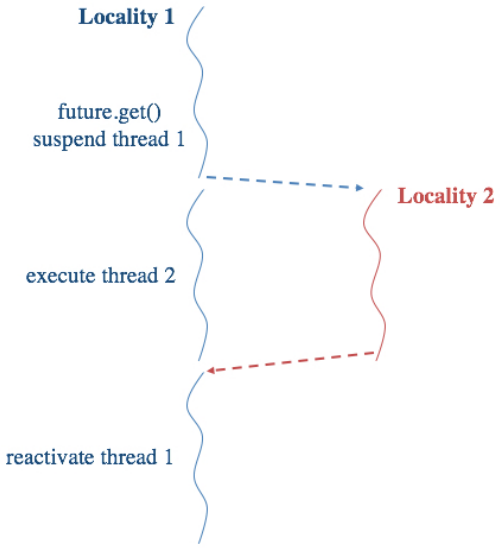

Figure 1: The principle of operation of a *future* in HPX

res_calc, bres_calc, update, which save_soln, and update loops are *direct* loops and the others are *indirect* loops. Figure 2 shows the sequential loops used in Airfoil application within airfoil.cpp. Saving old data values, applying the computation on each data value and updating them are implemented with mentioned five loops. Each loop iterates over a specified set and the operations that is performed with a user's kernels are defined in a header file for each loop: save_soln.h, adt_calc.h, res_calc.h, bres_calc.h and update.h. OP2 API provides a parallel loop function allowing the computation over sets through op_par_loop for each loops in Figure 2.

```
op_par_loop(save_soln,"save_soln", cells,
  op_arg_dat_0,...,op_arg_dat_n1);

op_par_loop(adt_calc,"adt_calc",cells,
  op_arg_dat_0,...,op_arg_dat_n2);

op_par_loop(res_calc,"res_calc",edges,
  op_arg_dat_0,...,op_arg_dat_n3);

op_par_loop(bres_calc,"bres_calc",bedges,
  op_arg_dat_0,...,op_arg_dat_n4);

op_par_loop(update,"update",cells,
  op_arg_dat_0,...,op_arg_dat_n5);
```

Figure 2: Five loops are used in airfoil.cpp for saving old data values, applying the computation on each data value and updating them. save_soln and update loops are *direct* loops and the others are *indirect* one.

Figure 3 shows the loop of adt_calc function from Figure 2 parsed with op_par_loop, which illustrates how each cell updates its data value with using the data values passed to adt_calc: blockId, offset_b and nelem. The value of blockId is defined based on the value of blockIdx captured from OP2 API. offset_b and nelem are defined based on the value of blockId. The arguments are passed to the adt_calc user kernel subroutine, which does the computation for each iteration in the inner loop from offset_b to offset_b+nelem of each iteration of the outer loop from 0 to nblocks. More details about the Airfoil application and its computation process can be found in [16].

```
#pragma omp parallel for
for(int blockIdx=0; blockIdx<nblocks; blockIdx++){
  int blockId  = //defined with blockIdx in OP2 API
  int nelem    = //defined based on blockId
  int offset_b = //defined based on blockId

  for ( int n=offset_b; n<offset_b+nelem; n++ ){
   .
   .
   .

   adt_calc(...);
  }
}
```

Figure 3: #pragma omp parallel for is used for loop parallelization in OP2 for Airfoil application to obtain the loop parallelization on one node and also on the distributed nodes using MPI.

As shown in Figure 3, #pragma omp parallel for is used for each loops passed with op_par_loop in OP2 for loop parallelization, on one node and also on distributed nodes. However, scalability is limited due to the sequential time caused by the implicit barrier in the fork-join model [19] between the parallel loops as described by Amdahl's Law. HPX parallelization methods are used here instead of OpenMP to achieve an optimal parallelization for parallelizing the loops parsed with OP2.

In this research we use two HPX methods for loop parallelization: I) using hpx::parallel::for_each with hpx::parallel::par as an execution policy and II) using hpx::lcos::local::dataflow with hpx::parallel::for_each. First method is described in more details in section IV-A and second method is explained in section IV-B. The comparison results of these two methods and OpenMP can be found in section V.

A. *hpx::parallel::for_each*

In this method, for the loops exist within Airfoil kernel functions, we implement one of the execution policies from HPX to make the loops to execute in parallel. The list of the execution policies can be found in [20].

hpx::parallel::par as an execution policy is used while implementing hpx::parallel::for_each. We were able to modify OP2 source-to-source translator with Python to automatically produce hpx::parallel::for_each instead of using #pragma omp parallel for for a loop parallelization. Figure 4 shows the loop of adt_calc function parsed with op_par_loop. In this method airfoil.cpp remains the same as Figure 2.

```
auto r=boost::irange(0, nblocks);
hpx::parallel::for_each(hpx::parallel::par,
  r.begin(), r.end(),[&](std::size_t blockIdx){

  int blockId  = //defined with blockIdx in OP2 API
  int nelem    = //defined based on blockId
  int offset_b = //defined based on blockId

  for ( int n=offset_b; n<offset_b+nelem; n++ ){
   .
   .
   .

   adt_calc(...);
  }
});
```

Figure 4: Implementing `hpx::parallel::for_each` for loop parallelization in OP2 for Airfoil application. HPX is able to control a grain size in this method. As a result, it helps in reducing processor starvation caused by the fork-join barrier at the end of the execution of the parallel loop.

```
hpx::parallel::dynamic_chunk_size dcs(SIZE);
auto r=boost::irange(0, nblocks);
hpx::parallel::for_each(hpx::parallel::par.with(dcs
    ), r.begin(), r.end(),[&](std::size_t blockIdx)
    {

  int blockId  = //defined with blockIdx in OP2 API
  int nelem    = //defined based on blockId
  int offset_b = //defined based on blockId

  for ( int n=offset_b; n<offset_b+nelem; n++ ){
   .
   .
   .

   adt_calc(...);
  }
});
```

Figure 5: Implementing `hpx::parallel::for_each` for loop parallelization in OP2 for Airfoil application. HPX is able to avoid degrading the scalability for small loops with defining a static grain size with `hpx::parallel::dynamic_chunk_size dcs(SIZE)` before the parallel loop execution.

This example exposes the same disadvantage as OpenMP, which is the representation of fork-join parallelism that introduces the global barriers at the end of the loop. But the difference of this method with OpenMP is that with using `hpx::parallel::for_each`, HPX is able to automatically control a grain size during a runtime. Grain size is the amount of works per threads. As discussed in section III, HPX enables fine-grained task parallelism and determines the grain size as small as possible to give task to all available threads. Grain size, which is named as `chunk_size` within HPX, is determined from auto-partitioner algorithm in HPX estimated at runtime while sequentially executing 1% of an application. So `hpx::parallel::for_each` helps creating sufficient amount of parallelism that how many iterations will run on the same thread, which helps in reducing processor starvation caused by the fork-join barrier at the end of the execution of the parallel loop.

However, it should be considered that if the computational time of a loop is not large enough compared to an application execution time, using an auto-partitioner algorithm within HPX will not be efficient. Since for the small loops, 1% of an application execution time used for determining a grain size will affect the application's scalability, so HPX provides another way to avoid degrading the scalability while using `hpx::parallel::for_each`. Grain size can be determined as a static grain size with `hpx::parallel::for_each(par.with(dcs))` before executing a loop, which dcs is defined with `hpx::parallel::dynamic_chunk_size dcs(SIZE)` as a static size. Figure 6 shows the part of code related to `adt_calc` function parsed with `op_par_loop` with determining a static grain size for a loop with `hpx::parallel::dynamic_chunk_size dcs(SIZE)`.

The experimental result for an Airfoil application is discussed in section V for both dynamic and static `chunk_size`.

B. *`hpx::lcos::local::dataflow`* with *`hpx::parallel::for_each`*

Here, we implement `hpx::lcos::local::dataflow` with `hpx::parallel::for_each` for loop parallelization that makes the invocation of the loop asynchronous. Asynchronous task execution means that a new HPX-thread will be scheduled. The call to `hpx::lcos::local::dataflow` provides a new $future$ instance which represents the result of the function execution.

In Figure 6, `hpx::lcos::local::dataflow` is used which returns a $future$ representing the result of a function and allows the asynchronization for the executed loops. Also, `hpx::parallel::for_each` as a previous method (see section IV) is used for a loop parallelization within each kernels. The only difference is that here all arguments passed to each kernel functions are $future$ except the name of a function. In using `hpx::lcos::local::dataflow`, if an argument is a future, then the invocation of a function will be delayed. Non-future arguments are passed through. For an instance, in Figure 6, `set,arg0,...,argn2` are expressed at the last line of the code, which invokes a function only once all of these futures get ready. `unwrapped` in Figure 6 is a helper function created in HPX, which unwraps the futures for a function and passes along the actual results.

In this method `airfoil.cpp` is changed as shown Figure 7. Each kernel function within `op_par_loop` returns a $future$ stored in a `new_data`. Each future depends on a future in a previous step. The $futures$ returned represent the results as a dependency tree, which represents the execution graph that is automatically created by using `hpx::lcos::local::dataflow`. `hpx::lcos::local::dataflow` gives an ability of having an asynchronous task execution.

```
using hpx::lcos::local::dataflow;
using hpx::util::unwrapped;

return dataflow(unwrapped([&adt_calc](op_set set,
    op_arg arg0, ... , op_arg argn2){

    hpx::parallel::dynamic_chunk_size dcs(SIZE);
    auto r=boost::irange(0, nblocks);
    hpx::parallel::for_each(hpx::parallel::par.with
        (dcs), r.begin(), r.end(),[&](std::size_t
        blockIdx){

    int blockId  = //defined with blockIdx in OP2
    int nelem    = //defined based on blockId
    int offset_b = //defined based on blockId

    for ( int n=offset_b; n<offset_b+nelem; n++ ){
     .
     .
     .

     adt_calc(...);
    }
  }),set,arg0,...,argn2);
```

Figure 6: Implementing `hpx::parallel::for_each` within `hpx::lcos::local::dataflow` for loop parallelization in OP2 for Airfoil application. It makes the invocation of the loop asynchronous and return *future*, which is is stored in `new_data`. `hpx::lcos::local::dataflow` allows automatically creating the execution graph which represents a dependency tree.

```
std::vector<hpx::future<void>> new_data1;

new_data1.push_back(
op_par_loop(save_soln,"save_soln",
        hpx::make_ready_future{cells},
        hpx::make_ready_future{op_arg_dat_0},...,
        hpx::make_ready_future{op_arg_dat_n1});

new_data1.push_back(
op_par_loop(adt_calc,"adt_calc",
        hpx::make_ready_future{cells},
        hpx::make_ready_future{op_arg_dat_0},...,
        hpx::make_ready_future{op_arg_dat_n2});

new_data1.push_back(
op_par_loop(res_calc,"res_calc",
        hpx::make_ready_future{edges},
        hpx::make_ready_future{op_arg_dat_0},...,
        hpx::make_ready_future{op_arg_dat_n3});

new_data1.push_back(
op_par_loop(bres_calc,"bres_calc",
       hpx::make_ready_future{bedges},
       hpx::make_ready_future{op_arg_dat_0},...,
       hpx::make_ready_future{op_arg_dat_n4});

new_data1.push_back(
op_par_loop(update,"update",
       hpx::make_ready_future{cells},
       hpx::make_ready_future{op_arg_dat_0},...,
       hpx::make_ready_future{op_arg_dat_n5});

hpx::when_all(new_data1).get();
```

Figure 7: `airfoil.cpp` is changed for using `hpx::async` for loop parallelization in OP2. `new_data` is returned from each kernel function after calling `op_par_loop` and `hpx::wait_all` is used after each loop till completing the results related to the previous loops.

Moreover, `hpx::lcos::local::dataflow` provides a way of interleaving execution of *indirect* loops. Interleaving execution of *direct* loops can be done during a compile-time, however it is almost difficult to interleave *indirect* loops during a compile-time. Using *future* based techniques in HPX such as `hpx::lcos::local::dataflow` enables having an *indirect* loop interleaving during a run-time.

In Figure 7, all of the arguments except the name of kernel function are passed as a future within each `op_par_loop`. `hpx::make_ready_future` is used to change them to a *future* one. Then, `hpx::when_all(new_data).get()` is used which waits for a `new_data` computation to be completed. It should be noted that the function represented in this section is the same as a function in section IV-A but asynchronous. The caller `hpx::when_all` for the function provides the result of the function execution. So, synchronously task execution suspends the current thread executing the process and the thread will be rescheduled once the function is completely executed.

The goal of using *future* is to let the computation within a loop as far as possible and avoid a global barrier synchronization forced with using `hpx::parallel::for_each` and `#pragma omp parallel for`. As a result, using *futures* allows the continuation of the current computations without waiting for the computations of the previous step, if their results are not needed in the current step. Removing the global barrier synchronizations improves the parallelization performance.

OP2 source-to-source translator with Python is modified here and `hpx::lcos::local::dataflow` with `hpx::parallel::for_each` is automatically produced for each loop within Airfoil application instead of `#pragma omp parallel for`. The experimental result is discussed in more details in section V.

V. Experimental Results

The experiments in this research have been executed on a 32 core system with 2 sockets. The main OS used by the shared memory system is 64 bit Linux Mint 17.2. The OpenMP linking is done through the version of OpenMP primitives available in the GNU g++ compilers version 5.1.0. The HPX version 0.9.10 [20] is used here.

For evaluating HPX performance for loop parallelization produced with OP2, we perform the strong scaling experiments, which the problem size is kept the same as the number of threads increases. Figure 8 shows an execution time for three loop parallelization methods: `#pragma omp parallel for` and `hpx::parallel::for_each` with dynamic and static `chunk_size` for an Airfoil application explained in section IV-A and Figure 9 shows the strong scaling for these three loop parallelization methods. As discussed in section IV-A, HPX allows controlling a grain size at runtime while using

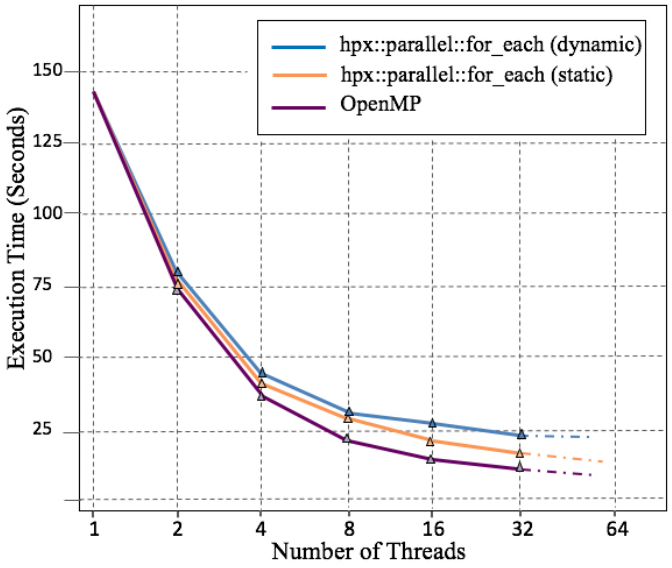

Figure 8: Execution time of an Airfoil application with `#pragma omp parallel for` and `hpx::parallel::for_each` (dynamic and static `chunk_size`) used for an Airfoil application with up to 64 threads.

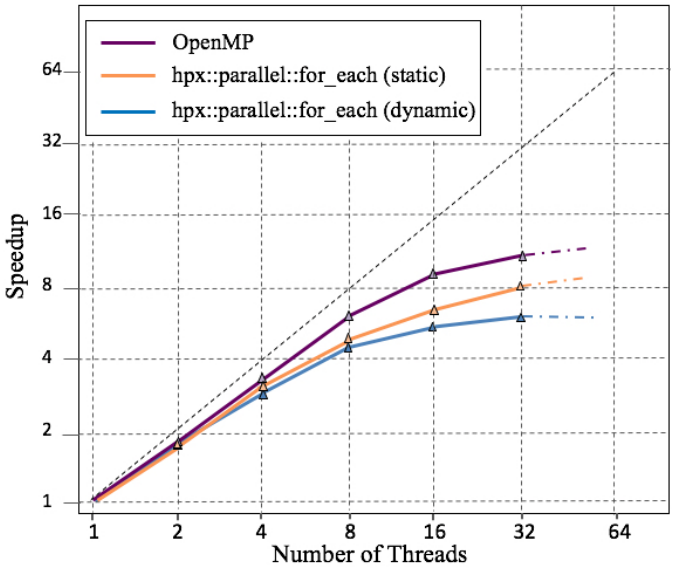

Figure 9: Comparison results of strong scaling between `#pragma omp parallel for` and `hpx::parallel::for_each` with dynamic and static `chunk_size` used for an Airfoil application with up to 64 threads. HPX allows controlling a grain size while using `hpx::parallel::for_each` to improve scalability. The results illustrate a better performance for `hpx::parallel::for_each` with the static `chunk_size` compared to the dynamic `chunk_size` for small loops. However, OpenMP performs better than HPX in this example.

`hpx::parallel::for_each` to improve scalability. Figure 9 illustrates that `hpx::parallel::for_each` with the static `chunk_size` has a better performance compared to the dynamic `chunk_size`, which is due to the small execution time of the loops compared to the total execution time of an Airfoil application as shown in Figure 8. Also it is illustrated that OpenMP still performs better than HPX in this example.

Figure 10 shows an execution time for `#pragma omp parallel for` and `hpx::lcos::local::dataflow` and Figure 11 shows the strong scaling comparison results for these methods. `hpx::parallel::for_each` is used for the loop parallelization `hpx::lcos::local::dataflow` as discussed in Section IV-B. Figure 11 illustrates a better performance for `hpx::lcos::local::dataflow` which is due to the asynchronous task execution. `hpx::lcos::local::dataflow` automatically generated an execution tree, which represents a dependency graph and allows to execute a function asynchronously. Asynchronous task execution removes a global barrier synchronization after each loop and as a results improves a scalability for a parallel applications.

By considering the above results, we can see the improvement in the performance over the OP2 (initial) version. For 32 threads in Figure 11, `hpx::lcos::local::dataflow` improves a scalability by about 27% compared to `#pragma omp parallel for`. These results show a good scalability achieved by HPX and indicates that it has the potential to continue to scale on more number of threads. Also, when the problem size is large enough, there will be enough work for all threads, which hides the communication latencies behind useful work. So for the larger problem size, the more

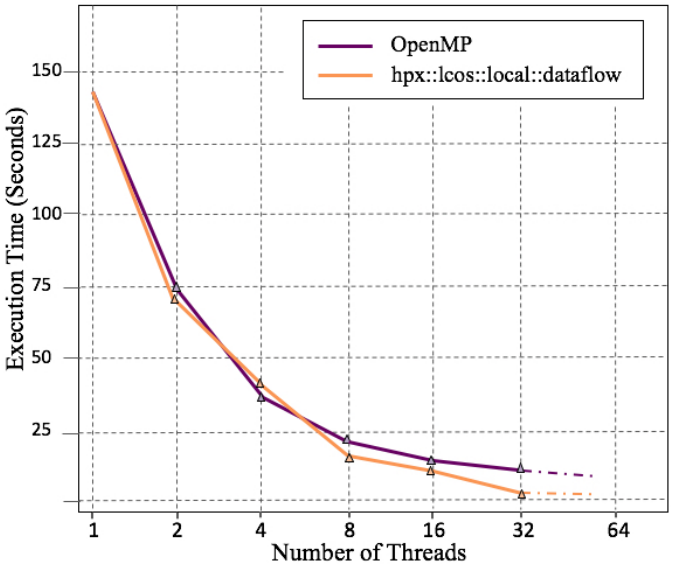

Figure 10: Execution time of an Airfoil application with `#pragma omp parallel for` and `hpx::lcos::local::dataflow` with `hpx::parallel::for_each` used for an Airfoil application with up to 64 threads.

parallelism can be extracted from the application which results to the better parallel efficiency.

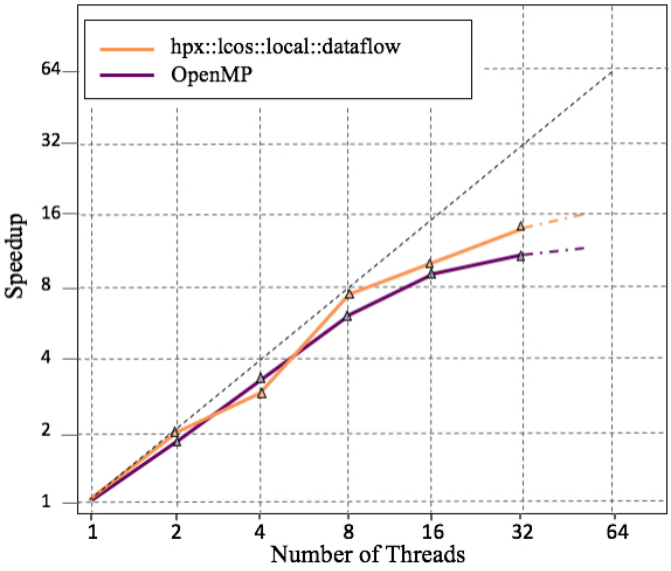

Figure 11: Comparison results of strong scaling between `hpx::lcos::local::dataflow` with `hpx::parallel::for_each` and `#pragma omp parallel for` used for Airfoil application with up to 64 threads. The results illustrate a better performance for `hpx::lcos::local::dataflow` for the larger number of threads, which is due to the asynchronous task execution. `hpx::lcos::local::dataflow` automatically generated an execution tree, which represents a dependency graph and allows to execute a function asynchronously.

VI. Conclusion

The work presented in this paper shows how the HPX runtime system can be used to implement C++ application frameworks. We changed the OP2 python source-to-source translator without changing OP2 API to automatically use HPX for loop parallelization within a code generated by OP2. Airfoil simulation written in OP2 is used to compare the HPX performance with OpenMP that is used in OP2 parallel loops. We were able to obtain 27% scalability improvement in using `hpx::lcos::local::dataflow` for loop parallelization compared with OpenMP.

With using `hpx::parallel::for_each`, HPX is able to control a grain size at runtime and as a result the resource starvation is reduced as well. However, using `hpx::parallel::for_each` introduces the global barriers at the end of the loop same as OpenMP, which inhibits having a desired scalability. On the other hand, `hpx::lcos::local::dataflow` allows the computation within a loop to be processed as far as possible, which avoids a global barrier synchronization. `hpx::lcos::local::dataflow` gives a capability of automatically interleaving consecutive loops during a runtime, which they do not need to wait for the previous loops to be completed. `hpx::when_all` is used at the end of all these loops and as a result, the other unnecessary global barrier between each consecutive loops will be removed as well.

References

[1] R. Rabenseifner, G. Hager, G. Jost, and R. Keller, "Hybrid MPI and OpenMP parallel programming," in *PVM/MPI*, 2006, p. 11.

[2] A. Rane and D. Stanzione, "Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems," in *Proc. of 10th LCI Int'l Conference on High-Performance Clustered Computing*, 2009.

[3] T. Heller, H. Kaiser, and K. Iglberger, "Application of the ParalleX execution model to stencil-based problems," *Computer Science-Research and Development*, vol. 28, no. 2-3, pp. 253–261, 2013.

[4] P. Grubel, H. Kaiser, J. Cook, and A. Serio, "The Performance Implication of Task Size for Applications on the HPX Runtime System," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 682–689.

[5] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[6] T. Heller, H. Kaiser, A. Schäfer, and D. Fey, "Using HPX and LibGeoDecomp for scaling HPC applications on heterogeneous supercomputers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 2013, p. 1.

[7] M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling, "An application driven analysis of the parallex execution model," *arXiv preprint arXiv:1109.5201*, 2011.

[8] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling, "Preliminary design examination of the parallex system from a software and hardware perspective," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 81–87, 2011.

[9] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*. IEEE, 2009, pp. 394–401.

[10] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–12.

[11] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly, "Performance analysis and optimization of the op2 framework on many-core architectures," *The Computer Journal*, p. bxr062, 2011.

[12] G. R. Mudalige, M. B. Giles, J. Thiyagalingam, I. Reguly, C. Bertolli, P. H. Kelly, and A. E. Trefethen, "Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems," *Parallel Computing*, vol. 39, no. 11, pp. 669–692, 2013.

[13] C. Bertolli, A. Betts, G. Mudalige, M. Giles, and P. Kelly, "Design and performance of the op2 library for unstructured mesh applications," in *Euro-Par 2011: Parallel Processing Workshops*. Springer, 2011, pp. 191–200.

[14] G. Mudalige, M. Giles, B. Spencer, C. Bertolli, and I. Reguly, "Designing op2 for gpu architectures," *Journal of Parallel and Distributed Computing*, 2012.

[15] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly, "Performance analysis of the op2 framework on many-core architectures," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 9–15, 2011.

[16] M. Giles, D. Ghate, and M. Duta, "Using automatic differentiation for adjoint cfd code development," 2005.

[17] L. Smith, "Mixed mode MPI/OpenMP programming," *UK High-End Computing Technology Report*, pp. 1–25, 2000.

[18] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[19] J. M. Bull, "Measuring synchronization and scheduling overheads in OpenMP," in *Proceedings of First European Workshop on OpenMP*, vol. 8, 1999, p. 49.

[20] H. Kaiser, T. Heller, A. Berge, and B. Adelstein-Lelbach, "HPX V0.9.11: A general purpose C++ runtime system for parallel and distributed applications of any scale," 2015, http://github.com/STEllAR-GROUP/hpx. [Online]. Available: http://dx.doi.org/10.5281/zenodo.33656