HPX-Improving Parallel Applications Scalability for OP2

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 an 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 instead of OpenMP by porting the parallel simulation backend of OP2 to utilize HPX. We compare the performance results of different parallelization methods using HPX and OpenMP for a loop parallelization within an Airfoil application. The results of the strong scalability and the weak scaling tests for an Airfoil application on one node with up to 64 threads are presented, which show an improvement in OP2 parallelization performance while using HPX.

Without changing OP2 API, we observe on an average by about 5% scaling improvement in using `hpx::async` with `hpx::parallel::for_each(par(task))` compared to `#pragma omp parallel for`. To fully exploit the potentials of the emerging technology, we modify OP2 API to get a $future$ based model. We observe on an average by about 21% scaling improvement in using `hpx::lcos::local::dataflow` compared to `#pragma omp parallel for`. 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.

***Index Terms*—High Performance Computing, HPX, OP2, Asynchronous Task Execution.**

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, Xeon-Phi, 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 and extracting an optimal parallelism from a parallel application.

This paper describes the results from porting the parallel simulation backend of OP2 to utilize HPX. We study the parallelization performance of OP2 with HPX with both changing OP2 API and without changing OP2 API. The parallelization performance is studied on 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 of an application scalability are studied 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. The operation over a set is done with a user's defined kernel function within a loop. 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]. There is two different kinds of *loops* defined in OP2: *indirect* loop and *direct* loop. If data is accessed through a mapping, the loop is an *indirect* loop. Otherwise, it is a *direct* loop.

In this research Airfoil application is studied that is presented in [16]. We demonstrate OP2 loops through the C++ code shown in Figure 1, which includes `op_par_loop_save_soln` and `op_par_loop_adt_calc` loops from an Airfoil application. `op_par_loop_save_soln` loop is a *direct* loop which applies `save_soln` over cells with accessing data arguments `p_q` and `p_qold` passed to the loop. `op_par_loop_adt_calc` is an *indirect* loop which applies `op_par_loop_adt_calc` over cells with accessing data arguments `p_x`, `p_q` and `p_adt` passed to the loop. Function `op_arg_dat` creates an argument based on the information passed to it. For example, in `op_arg_dat(p_x,0,pcell,2,"double",OP_READ)`, an argument is created from its inputs, where `p_x` is a data, `0` shows that a data is accessed indirectly, `pcell` is a map which a data is mapped on, 2 is a dimension of data, `double` shows a type of data and `OP_READ` shows that this data is a read_only data.

OP2's design is based on achieving a near-optimal performance for scaling on multi-core processors. In [13], [14], it is

```
op_par_loop_save_soln("save_soln", cells,
  op_arg_dat(p_q,-1,OP_ID,4,"double",OP_READ),
  op_arg_dat(p_qold,-1,OP_ID,4,"double",OP_WRITE));

op_par_loop_adt_calc("adt_calc",cells,
  op_arg_dat(p_x,0,pcell,2,"double",OP_READ),
  op_arg_dat(p_x,1,pcell,2,"double",OP_READ),
  op_arg_dat(p_x,2,pcell,2,"double",OP_READ),
  op_arg_dat(p_x,3,pcell,2,"double",OP_READ),
  op_arg_dat(p_q,-1,OP_ID,4,"double",OP_READ),
  op_arg_dat(p_adt,-1,OP_ID,1,"double",OP_WRITE));
```

Figure 1: `op_par_loop_save_soln` and `op_par_loop_adt_calc` loops from an Airfoil application. `op_par_loop_save_soln` loop is a *direct* loop and `op_par_loop_adt_calc` is an *indirect* loop.

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 conventional 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 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.

An Airfoil 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 2 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.

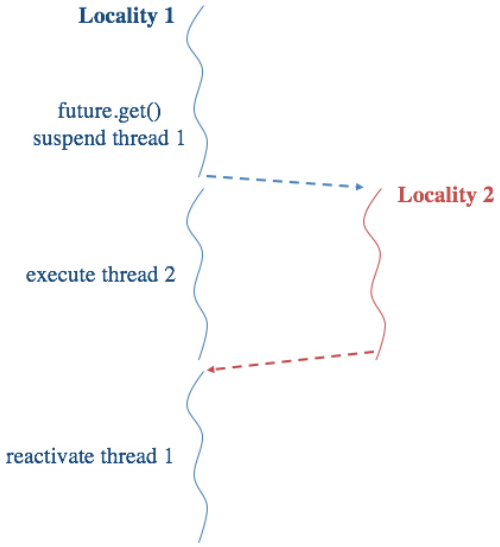

Figure 2: The principle of operation of a *future* in HPX

Figure 2 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 2. 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: `op_par_loop_save_soln`, `op_par_loop_adt_calc`, `op_par_loop_res_calc`, `op_par_loop_bres_calc`, `op_par_loop_update`, which `op_par_loop_save_soln`, and `op_par_loop_update` loops are *direct* loops and the others are *indirect* loops. Figure 3 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 these five loops. Each loop iterates over a specified set and the operations that is performed with a user's kernels 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 3.

```
op_par_loop_save_soln("save_soln", cells,
  op_arg_dat(data_a0,...),...,
  op_arg_dat(data_an);

op_par_loop_adt_calc("adt_calc",cells,
  op_arg_dat(data_b0,...,
  op_arg_dat(data_bn);

op_par_loop_res_calc("res_calc",edges,
  op_arg_dat(data_c0,...,
  op_arg_dat(data_cn);

op_par_loop_bres_calc("bres_calc",bedges,
  op_arg_dat(data_d0,...,
  op_arg_dat(data_dn);

op_par_loop_update("update",cells,
  op_arg_dat(data_e0,...,
  op_arg_dat(data_en);
```

Figure 3: 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 4 shows the loop of `op_par_loop_adt_calc` function from Figure 3 parsed with OP2, which illustrates how each cell updates its data value with using `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 computed 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].

As shown in Figure 4, `#pragma omp parallel for` is used for each loops passed with `op_par_loop` in OP2 for parallelizing the loops on one node and also on distributed nodes. However, scalability is limited due to the sequential

```
#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 4: `#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.

time caused by an 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 different HPX parallelization methods without changing OP2 API and with changing OP2 API. Airfoil parallelization performance with HPX and without changing OP2 API is studied in Section IV-A and with the modified OP2 API is studied in Section IV-B. The comparison results of these two methods and OpenMP can be found in Section V.

A. *HPX without changing OP2 API*

In this Section, we study two different HPX parallelization methods on an Airfoil application without changing the OP2 API. In Section IV-A1, `parallel::for_each` with `hpx::parallel::par` as an execution policy is used for parallelizing both *direct* and *indirect* loops. In Section IV-A2, `hpx::async` and `hpx::parallel::for_each` with `hpx::parallel::par` is used for parallelizing the *direct* loops, and for the *indirect* loops, `hpx::parallel::for_each` with `hpx::parallel::par(task)` as an execution policy is implemented .

1) ***`parallel::for_each`****:* In this method, we implement one of the execution policies of HPX to make the loops shown in Figure 3 executing 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. In this method `airfoil.cpp` remains the same as Figure 3 and also OP2 API is not changed too.

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. Figure 5 shows the loop of `op_par_loop_adt_calc` function parsed with OP2 with

```
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 5: 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 6: 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.

dynamic chunk_size.

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 12 shows the loop of op_par_loop_adt_calc function with static chunk_size implemented 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.

2) ***async and parallel::for_each:*** Here, we implement two different parallelization methods for the loops based on their types. For the *direct* loops, hpx::async and hpx::parallel::for_each with hpx::parallel::par as an execution policy is used. For the *indirect* loops, hpx::parallel::for_each with hpx::parallel::par(task) as an execution policy is implemented. The call to hpx::async and also hpx::parallel::par(task) provide a new *future* instance, which represents the result of the function execution that make the invocation of the loop asynchronous. Asynchronous task execution means that a new HPX-thread will be scheduled. As a result it avoids a global barrier synchronization forced with using hpx::parallel::for_each from Section IV-A1.

In Figure 7, hpx::async and hpx::parallel::for_each with hpx::parallel::par is used for op_par_loop_save_soln, which is a *direct* loop and returns a *future* representing the result of a function. In Figure 8, hpx::parallel::for_each with hpx::parallel::par(task) is used for op_par_loop_adt_calc, which is an *indirect* loop and it also returns a *future* representing the result of a function. The *future* returned from all *direct* and *indirect* loops allow the asynchronization for the executed loops.

In this method OP2 API is not changed but airfoil.cpp is changed as shown Figure 9. 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. So, new_data.get() is used to get all *futures* ready before the next steps. The place of using new_data.get() depends on an application and a programmer should put them manually in a right place by considering the data dependency between loops. In the next Section we address this problem. OP2 source-to-source translator

```
return async(hpx::launch::async,[adt_calc,set,arg0
    ,...,argn1](){

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

    int start  = //defined based on number of
        threads;
    int finish =//defined based on number of
        threads;

    for ( int n=start; n<finish; n++ ){
     save_soln(...);
    }
  });
```

Figure 7: Implementing hpx::async and hpx::parallel::for_each with hpx::parallel::par for a *direct* loop parallelization in OP2 for Airfoil application. The returned *future* representing the result of a function.

```
hpx::parallel::dynamic_chunk_size dcs(SIZE);
auto r=boost::irange(0, nblocks);
hpx::future<void> new_data;
new_data=hpx::parallel::for_each(hpx::parallel::par
    (hpx::parallel::task).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 8: Implementing hpx::parallel::for_each with hpx::parallel::par(task) for an *indirect* loop parallelization in OP2 for Airfoil application. The returned *future* representing the result of a function.

with Python is modified and it automatically produces hpx::async and hpx::parallel::for_each with hpx::parallel::par for each direct loops and hpx::parallel::for_each with hpx::parallel::par(task) for each *indirect* loops within Airfoil application. The experimental results of this Section can be found in Section V.

B. HPX with the modified OP2 API

To fully exploit the potentials of the emerging technology, we modify OP2 API to get a *future* based model. In Figure 1, op_arg_dat creates an argument that is passed to a kernel function through op_par_loop. The modified OP2 API passes the argument as a

```
new_data1=op_par_loop_save_soln("save_soln",
   cells,op_arg_dat_0,...,op_arg_dat_n1);

new_data2=op_par_loop_adt_calc_("adt_calc",
   cells,op_arg_dat_0,...,op_arg_dat_n2);

new_data3=op_par_loop_res_calc("res_calc",
   edges,op_arg_dat_0,...,op_arg_dat_n3);

new_data4=op_par_loop_bres_calc("bres_calc",
   bedges,op_arg_dat_0,...,op_arg_dat_n4);

new_data1.get();
new_data2.get();

new_data5=op_par_loop_update("update",
   cells,op_arg_dat_0,...,op_arg_dat_n5);

new_data3.get();
new_data4.get();
new_data5.get();
```

Figure 9: `airfoil.cpp` is changed while using `hpx::async` and `hpx::parallel::par(task)` for loop parallelization in OP2. `new_data` is returned from each kernel function after calling `op_par_loop` and `new_data.get()` is used to get *futures* ready before the next steps.

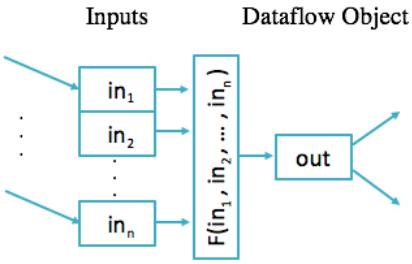

Figure 10: A `data flow` object encapsulates a function $F(in_1, in_2, ..., in_n)$ with n inputs from different data resources. As soon as the last input argument has been received, the function F is scheduled for an execution [3].

future created with the modified `op_arg_dat`, which uses `hpx::lcos::local::dataflow`. 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. Figure 10 shows the schematic of a `data flow` object. A `data flow` object encapsulates a function $F(in_1, in_2, ..., in_n)$ with n inputs from different data resources. As soon as the last input argument has been received, the function F is scheduled for an execution [3]. The main advantage of a data-flow based execution is minimizing the total synchronization by scheduling overheads.

Figure 11 shows the modified `op_arg_dat` and `dat` expressed at the last line of the code invokes a function only once it gets ready. `unwrapped` is a helper function created in HPX, which unwraps the futures for a function and passes

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

return dataflow(unwrapped([&](op_dat dat){
      .
      . //same as op_arg_dat in an original OP2 API
      .

    return arg;
    }
  }),dat);
```

Figure 11: `op_arg_dat` is modified to create an argument as a *future*, which is passed to a function through `op_par_loop`.

along the actual results. All these *future* arguments are passed to the kernel functions through `op_par_loop`.

`hpx::lcos::local::dataflow` with `hpx::parallel::for_each` is implemented for a loop parallelization that makes the invocation of the loop asynchronous. In Figure 12, `hpx::lcos::local::dataflow` returns a *future* of `arg.dat` representing the result of a function and allows the asynchronization for the executed loops. All arguments passed to each kernel functions are *future* except the name of a function and `op_set` passed to a loop. `hpx::parallel::for_each` as a previous method (see Section IV) is used for a loop parallelization within each kernels.

It should be noted that the function represented in this Section is the same as a function in Section IV-A1 but asynchronous. The *future*s returned represent the results as a dependency tree, which represents the execution graph that is automatically created. As a result, a modified OP2 API and `hpx::lcos::local::dataflow` give an ability of having an asynchronous task execution.

Each kernel function returns a data of an output argument as a *future* stored in `data[t]`, where t is a time step. Each future depends on a future in a previous step, which is `data[t-1]`. In this method `airfoil.cpp` is changed. Figure 13 shows only `op_par_loop_adt_calc` and `op_par_loop_res_calc` from `airfoil.cpp`. It shows that the data of each arguments depends on data from a previous step produced with `op_par_loop` in that time step. We can see that the problem of manually putting `new_data.get()` addressed in Section IV-A2 is solved here. Moreover, `hpx::lcos::local::dataflow` provides a way of interleaving execution of *indirect* loops and *direct* loops together. 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.

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`

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

return dataflow(unwrapped([&adt_calc,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(...);
    }
    return arg5.dat;
  }),arg0,...,argn2);
```

Figure 12: 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.

```
p_adt[t]=op_par_loop_adt_calc("adt_calc",cells,
 op_arg_dat1(p_x[t-1],0,pcell,2,"double",OP_READ),
 op_arg_dat1(p_x[t-1],1,pcell,2,"double",OP_READ),
 op_arg_dat1(p_x[t-1],2,pcell,2,"double",OP_READ),
 op_arg_dat1(p_x[t-1],3,pcell,2,"double",OP_READ),
 op_arg_dat1(p_q[t-1],-1,OP_ID,4,"double",OP_READ),
op_arg_dat1(p_adt[t-1],-1,OP_ID,1"double"OP_WRITE))

p_res[t]=op_par_loop_res_calc("res_calc",edges,
 op_arg_dat1(p_x[t-1],0,pedge,2,"double",OP_READ),
 op_arg_dat1(p_x[t-1],1,pedge,2,"double",OP_READ),
 op_arg_dat1(p_q[t-1],0,pecell,4,"double",OP_READ),
 op_arg_dat1(p_q[t-1],1,pecell,4,"double",OP_READ),
 op_arg_dat1(p_adt[t],0,pecell,1,"double",OP_READ),
 op_arg_dat1(p_adt[t],1,pecell,1,"double",OP_READ),
 op_arg_dat1(p_res[t-1],0,pecell,4,"double",OP_INC)
 op_arg_dat1(p_res[t-1],1,pecell,4"double"OP_INC));
```

Figure 13: `airfoil.cpp` is changed while using `hpx::lcos::local::dataflow` for loop parallelization in OP2. `data[t]` is returned from each kernel function after calling `op_par_loop` using `data[t-1]`.

`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 cores 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 0.9.11 [20] is used here.

For evaluating HPX performance for the loop parallelization generated with OP2, we perform the strong scaling and weak scaling experiments. For studying speedup, we use strong scaling, for which the problem size is kept the same as the number of threads increases. Figure 14 shows the strong scaling for these three loop parallelization methods: `#pragma omp parallel for` and `hpx::parallel::for_each(par)` with dynamic and static `chunk_size` for an Airfoil application explained in Section IV-A1. As discussed in Section IV-A1, HPX allows controlling a grain size at runtime while using `hpx::parallel::for_each` to improve scalability. Figure 14 illustrates that `hpx::parallel::for_each(par)` 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. But, it can be seen that OpenMP still performs better than HPX in this example.

Figure 15 shows the strong scaling comparison results for `#pragma omp parallel for` and `hpx::async` with `hpx::parallel::for_each(par(task))` from Section IV-A2. It shows a better performance for `hpx::async` with `hpx::parallel::for_each(par(task))`, which is a result of asynchronous execution of loops provided with the returned $futures$.

Figure 17 shows the strong scaling comparison results for `#pragma omp parallel for` and `hpx::lcos::local::dataflow`. `hpx::parallel::for_each(par)` is used for the loop parallelization `hpx::lcos::local::dataflow` as discussed in Section IV-B. Figure 17 illustrates a better performance for `hpx::lcos::local::dataflow` which is due to the asynchronous task execution provided with the returned $futures$. `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 the unnecessary global barrier synchronization and as a results improves a scalability for a parallel applications.

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(par)` and `#pragma omp parallel for`. As a result, using $futures$ allows the continuation of the current computations without waiting

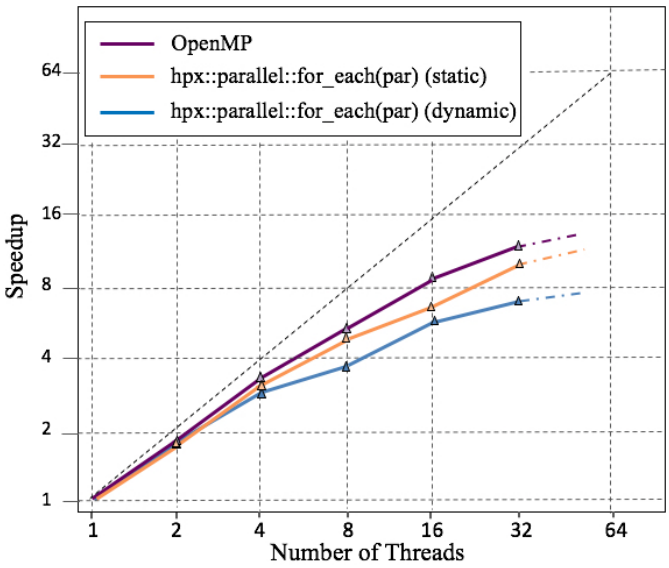

Figure 14: Comparison results of strong scaling between `#pragma omp parallel for` and `hpx::parallel::for_each(par)` with dynamic and static `chunk_size` used for an Airfoil application with up to 64 threads. HPX allows controlling a grain size while using `hpx::parallel::for_each` to improve scalability. 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.

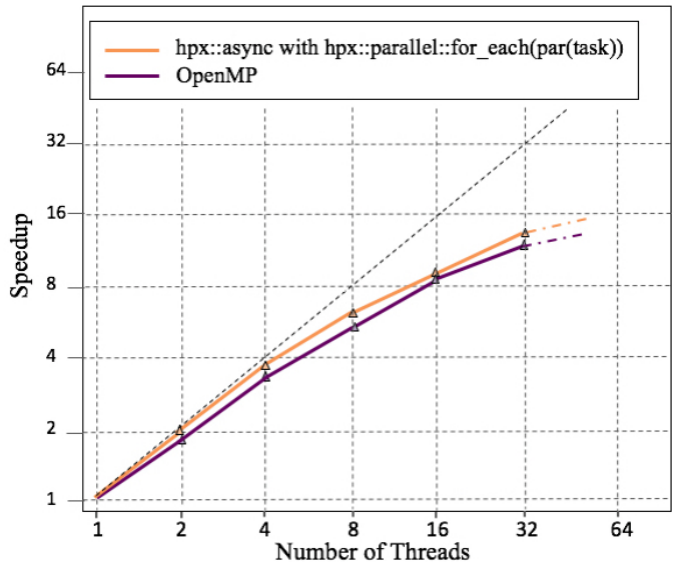

Figure 15: Comparison results of strong scaling between `#pragma omp parallel for` and `hpx::async` with `hpx::parallel::for_each(par(task))` used for Airfoil application with up to 64 threads. The results illustrate a better performance for `hpx::async`, which is due to the asynchronous task execution provided with a returned $future$.

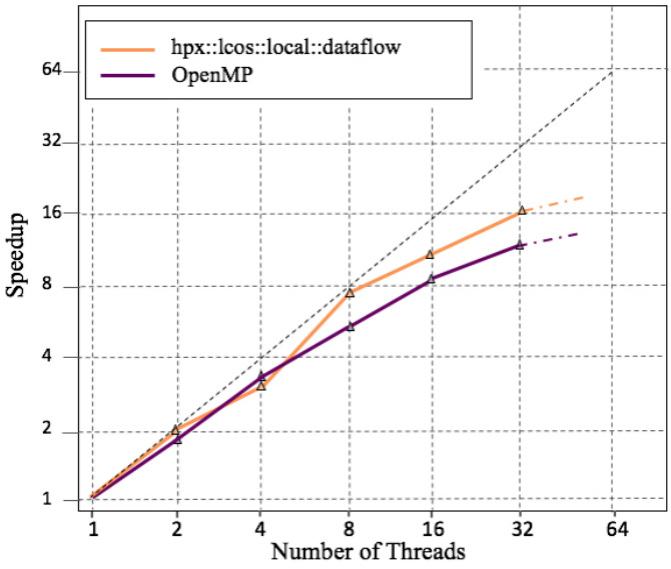

Figure 16: Comparison results of strong scaling between `hpx::lcos::local::dataflow` and `#pragma omp parallel for` used for Airfoil application with up to 64 threads. The results illustrate a better performance for `hpx::lcos::local::dataflow` for the larger number of threads, which is due to the asynchronous task execution. `hpx::lcos::local::dataflow` automatically generated an execution tree, which represents a dependency graph and allows to execute a function asynchronously.

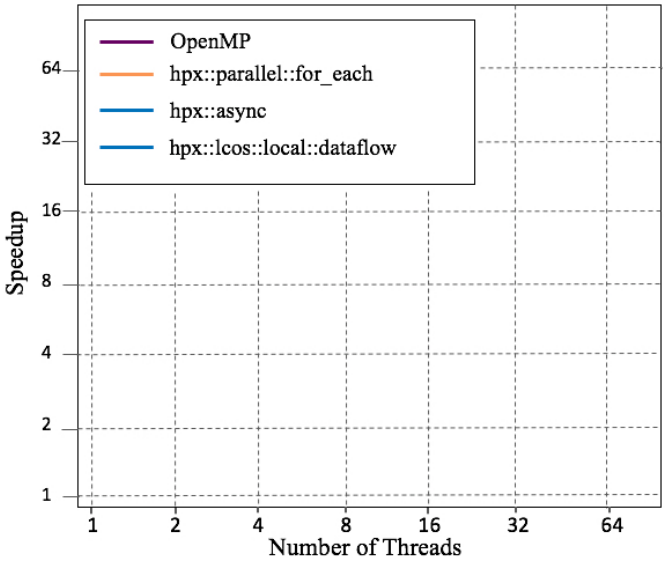

Figure 17: Comparison results of a weak scaling between `#pragma omp parallel for`, `hpx::parallel::for_each`, `hpx::async` and `hpx::lcos::local::dataflow` used for Airfoil application with up to 64 threads. The results illustrate a better performance for `hpx::lcos::local::dataflow`, which shows the perfect overlap of computation with communication enabled by HPX.

for the computations of the previous step, if their results are not needed in the current step. Figures 15 and 17 show that removing the global barrier synchronizations improves the parallelization performance.

By considering the above results, we can see the improvement in the performance over the OP2 (initial) version. For 32 threads in Figure 15, `hpx::async` with `hpx::parallel::for_each(par(task))` improves a scalability by about 5% and in Figure 17, `hpx::lcos::local::dataflow` improves a scalability by about 21% and 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.

To study the effects of communication latencies, we perform weak scaling experiments, where the problem size is increased in proportion to the increase of the number of cores. Figure shows a weak scaling of using `#pragma omp parallel for`, `hpx::parallel::for_each(par)`, `hpx::async` with `hpx::parallel::for_each(par(task))` and `hpx::lcos::local::dataflow` for a loop parallelization. `hpx::lcos::local::dataflow` with the modified OP2 API has a better parallel performance, which shows the perfect overlap of computation with computation enabled by HPX. Also, it can be seen 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 parallelism can be extracted from the application which results to the better parallel efficiency.

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 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 5% scalability improvement in using `hpx::async` and 21% scalability improvement in using `hpx::lcos::local::dataflow` for loop parallelization compared with OpenMP.

HPX is able to control a grain size at runtime by using `hpx::parallel::for_each` and as a result the resource starvation is reduced as well. However using `hpx::parallel::par` as an execution policy introduces the global barriers at the end of the loop same as OpenMP, which inhibits having a desired scalability. It was shown that the global barrier synchronization was removed by a $future$ based model used in `hpx::parallel::for_each(par(task))`, `hpx::async` and `hpx::lcos::local::dataflow`, which results in significantly improving a parallelism level. $future$ allows the computation within a loop to be processed as far as possible. Also, for using `hpx::lcos::local::dataflow`, OP2 API is modified to take full advantage of all available parallel resources through HPX. It was shown that `hpx::lcos::local::dataflow` implemented in the modified OP2 API gives a capability of automatically interleaving consecutive $direct$ and $indirect$ loops together during a runtime. As a result, it enables seamless overlap of communication with communication and helps in extracting the desired parallelism level from an application.

Acknowledgements

This works was supported by NSF awards 1447831 and .....

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