<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper presented at *31st IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017, Orlando, United States, 29 May 2017 through 2 June 2017*.

Citation for the original published paper:

Akhmetova, D., Iakymchuk, R., Ekeberg, Ö., Laure, E. (2017)
Performance study of multithreaded MPI and Openmp tasking in a large scientific code
In: *Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2017*, 7965119 (pp. 756-765). Institute of Electrical and Electronics Engineers (IEEE)
<https://doi.org/10.1109/IPDPSW.2017.128>

N.B. When citing this work, cite the original published paper.

© 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-213532>

Performance study of multithreaded MPI and OpenMP tasking in a large scientific code

Dana Akhmetova, Roman Iakymchuk, Örjan Ekeberg, Erwin Laure

Department of Computational Science and Technology

KTH Royal Institute of Technology

Stockholm, Sweden

Email: {danaak, riakymch, ekeberg, erwinl}@kth.se

Abstract—With a large variety and complexity of existing HPC machines and uncertainty regarding exact future Exascale hardware, it is not clear whether existing parallel scientific codes will perform well on future Exascale systems: they can be largely modified or even completely rewritten from scratch. Therefore, now it is important to ensure that software is ready for Exascale computing and will utilize all Exascale resources well. Many parallel programming models try to take into account all possible hardware features and nuances. However, the HPC community does not yet have a precise answer whether, for Exascale computing, there should be a natural evolution of existing models interoperable with each other or it should be a disruptive approach. Here, we focus on the first option, particularly on a practical assessment of how some parallel programming models can coexist with each other.

This work describes two API combination scenarios on the example of iPIC3D [26], an implicit Particle-in-Cell code for space weather applications written in C++ and MPI plus OpenMP. The first scenario is to enable multiple OpenMP threads call MPI functions simultaneously, with no restrictions, using an MPI_THREAD_MULTIPLE thread safety level. The second scenario is to utilize the OpenMP tasking model on top of the first scenario. The paper reports a step-by-step methodology and experience with these API combinations in iPIC3D; provides the scaling tests for these implementations with up to 2048 physical cores; discusses occurred interoperability issues; and provides suggestions to programmers and scientists who may adopt these API combinations in their own codes.

Index Terms—API interoperability, programming models, MPI, OpenMP, tasks, multithreading, MPI_THREAD_MULTIPLE thread safety

I. INTRODUCTION

The classical technology scaling ended (processor clock rates are not growing anymore), and performance increase have to be gained from explicit parallelism. As many HPC experts predict [6], [31], [34] that future Exascale machines will be more complex and massively parallel, more heterogeneous and hierarchical, with diverse types of processors and memory. While hardware vendors are working on bringing the Exascale system to market, software used to program such Exascale architecture efficiently should be in time ready to map to the Exascale hardware. Currently there is no a holistic programming model as a unified notation for inter- and intra-node programming, and it seems that in the nearest future the optimal (and evolutionary) approach will be to use a mixture of various programming models at various architectural levels. Also it is necessary to maintain legacy code even if it is not

well mapped to Exascale computers. In this case, the most important issue will be interoperability between different APIs both in terms of specification and implementation.

The usual practice for parallel programming is a so-called classical or “pure” approach: to use a pure message passing model (like MPI API [19]) for distributed-memory systems and to use a pure shared-memory programming model (such as OpenMP API [12]) for shared-memory systems. Nowadays, usual HPC systems integrate both shared and distributed-memory architectures. For such hybrid architectures, one can do hybrid parallel programming by combining different parallel programming models that are used on different architectural levels within a single code. This can give much more parallelism in the program and higher performance in comparison with pure MPI or pure OpenMP implementations [23].

Parallel applications should scale automatically with the number of processing units, so they should be programmed to exploit parallelism in the most efficient way. This puts responsibility on the programmer [14]. One may use auto-parallelization, for example, by using parallel compilers. The advantage of this approach is simplicity to get parallelism, but the disadvantage is that the level of parallelism reached in this way is small. Another way is to develop the code to directly exploit parallelism. This requires extensive programming effort, but should give a better performance. Learning different parallel programming techniques and studying experience of others in implementing parallel applications will help in developing own parallel codes efficiently.

In this paper, our particular focus is on enabling MPI communication by multiple OpenMP threads in the classical hybrid MPI plus OpenMP approach and on using OpenMP tasking on top. As a case study, we choose the iPIC3D code for space weather simulations and cover two distinguished test cases. We also consider two different simulation regimes: field- and particle-dominated. We particularly stress the interoperability and work splitting among MPI and OpenMP in the widely used halo exchange and particle mover parts of iPIC3D.

The paper is organized as follows. Section II reviews these two programming models as well as pluses and minuses of combining them. Section III briefly introduces the iPIC3D code. Section IV presents our guidelines for mixing multithreaded MPI and OpenMP tasking. Section V provides performance results. Finally, we draw conclusions in Section VI.

II. PARALLEL PROGRAMMING MODELS OVERVIEW

A parallel programming model is an abstraction of the computer system architecture [27]. Although it is not attached to any particular computer type, but there are many different parallel programming models as processing units can be differently organized in a computing system. Here we will consider OpenMP and MPI for the development of parallel applications.

A. Shared-memory programming with OpenMP

OpenMP is the de facto standard for programming shared-memory systems. It consists of compiler directives, a runtime library, and environment variables [12]. In OpenMP, the switch between serial and parallel parts of the code is represented by the fork/join model [3]. The work in the OpenMP parallel region is replicated across all threads. Also, it is possible to share the work execution among threads (so called “worksharing”), like parallel iterations over the index set of for-loops.

Task-based programming in OpenMP was introduced in the OpenMP 3.0 API specifications [7], [5]. OpenMP 4.0 added the ability to specify task dependencies. Without this feature, calling MPI routines from inside OpenMP tasks did not have many realistic use cases.

Tasking or task-based approach refers to designing a program in terms of “tasks” - a logically discrete section of work to be done. Then these tasks are selected by a task scheduler, which dynamically assigns the tasks to threads that are idle and can execute them. The details of task scheduling to hardware are hidden from programmers; the task-scheduling runtime is responsible for efficient execution of a task-based program. Thus, developers just focus on implementing their algorithms in terms of tasks that decreases programming effort and gives portability across different hardware. Most likely, this separation of concerns is the main reason why task-based programming models are becoming more important. In addition, there are other performance-related benefits from using the task-based approach:

- the number of tasks is scaled with the size of the problem being solved;
- the task scheduler assigns tasks to queues of the available threads, and if a thread becomes idle, then it tries to steal a task from a “victim” thread (so called “work-stealing” scheduling) – this automatic load-balancing helps to utilize all CPU resources;
- instead of creating and deleting threads to execute new tasks, the same threads are used over and over again, thus minimizing scheduling overhead – scheduling a task is less expensive than creating and deleting a thread;
- task granularity plays an important role in application’s performance: too much coarse-grained tasks result in load imbalance at synchronization points, while too much fine-grained tasks increase runtime scheduling overhead (see Figure 1) [2].

It is beneficial to implement the code in terms of tasks where units of work are generated dynamically (in recursive/iterative

Fig. 1: Interplay between task granularity and scheduling overhead [2].

control structures or in while-loops). This helps to tackle dynamic data structures and graph algorithms [5].

B. Distributed-memory programming with message passing/MPI

Message passing is used to program a distributed-memory system where communication between processes is done by exchanging messages, as communication cannot be done by sharing variables. MPI is a specification for the message-passing programming model [19], [20], [29] and is currently the de facto standard for distributed-memory programming.

This model implies that processes run in parallel have separate memory address spaces, they communicate by sending messages to each other (copying parts of their address spaces into address spaces of each other). This communication is a two-sided model: it occurs only when one process sends a message and another process receives the matching message. Synchronization of processes is important to match messages. One-sided communication was allowed in the MPI 2.0 API specifications [17], [20]. Here, no sender-receiver matching is needed, this is asynchronous communication. It decouples data transfer from synchronization and allows remote memory access (RMA).

Another model for consideration is a combination of message passing and multithreading. Here, MPI processes are attached to one computing node or multi-core processor, and each MPI process itself consists of multiple threads. The interaction between MPI and user created threads in an MPI program is described in the MPI 2.1 API specifications [18]. MPI supports four “levels” of thread safety, which has to be explicitly stated by a programmer:

- `MPI_THREAD_SINGLE` The process has only one thread of execution. This represents a typical MPI-only application;
- `MPI_THREAD_FUNNELED` The process may be multi-threaded, but only the master thread will make calls to

the MPI library (all MPI calls are funneled to the main thread). The thread that calls `MPI_Init_thread()` is from now on the main thread;

- `MPI_THREAD_SERIALIZED` The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time;
- `MPI_THREAD_MULTIPLE` The process may be multithreaded, and multiple threads can call MPI functions simultaneously.

C. Hybrid programming with MPI and OpenMP

Mixing the distributed-memory and shared-memory programming models in one program is not a new idea [33]. Most of the hybrid MPI plus OpenMP codes are based on a hierarchical model: MPI is used across distributed nodes, while OpenMP within each node. This allows to utilize large- and medium-grain parallelism at the MPI level, and fine-grain parallelism at the OpenMP level. The code is directly written as MPI processes, its serial parts contains OpenMP directives in order to benefit from multithreading. The programmer should firstly consider the nature of the code before porting it to this hybrid model, because it does not guaranty to be always efficient. This hybrid approach was studied in [30], [16], [32].

There are two different schemes to implement this hybrid approach, depending on the overlap between communication and computation [30]:

- No overlapping scheme means that there are no MPI calls overlapping with other application code in other threads, the non-communicating threads are idle. Here there are two possible options. The first one is when MPI is called only outside parallel regions and by the master thread. Thus, there is no message passing inside computer nodes, the master thread controls communication between nodes, and therefore there is no problem with the topology. The disadvantage can be low efficiency, because all other threads are sleeping while the master thread is doing communication. Also, the MPI library should support at least the `MPI_THREAD_FUNNELED` level of thread safety. The second option is when MPI is called outside the parallel regions of the code, but the MPI communication is done itself by several processing units. The thread parallelization of the MPI communication can be done automatically by the MPI library routines, or explicitly by the application, using a full thread-safe MPI library.
- Overlapping communication and computation implies the situation when non-communicating threads are doing computation, while the master thread (or several threads) is doing communication. In this case, either only the master thread calls MPI routines (all communications are funneled to the master), or each thread is responsible for its own communication needs (communication is funneled to more than one thread). One of minuses of this strategy is that one cannot use worksharing directives (the major OpenMP feature), because the communication/computation is done via threads' ID rank in OpenMP.

Furthermore, the communication load of the threads is inherently unbalanced.

After choosing the most suited hybrid programming scheme for a particular application, the strategy should be to implement it with that scheme from scratch. However, usually developers do hybrid programming on top of an existing pure MPI or pure OpenMP code. The following suggestions should be considered in order to efficiently port a pure HPC code to the MPI plus OpenMP version [13]:

- Porting a pure MPI code to MPI plus OpenMP: this should not be a problem, as MPI is already explicitly managing the program state synchronization. The speedup of the new hybrid version depends on the amount of OpenMP parallelized work (usually, this is loop-level parallelism). This porting is favourable for communication-bound applications, as it uses less MPI processes for communication. However, CPU processor use on each node becomes an issue to study during the porting process.
- Porting a pure OpenMP code to MPI plus OpenMP: this is harder to implement than the previous option, because the program state must be explicitly handled by MPI. The programmer has to carefully examine how each MPI process will communicate with other processes. Sometimes it can lead to the complete redesign of the parallelization structure. However, this approach gives better performance and scaling.

Many works [10], [15], [21], [32], [11] have described cases when pure MPI programs outperform their hybrid versions despite the underlying architectures. On the other hand, there are also many examples, when such hybrid programming gave considerable speedup of the code execution [9], [8], [25], [24], [28], [4], [22].

III. OVERVIEW OF IPIC3D

iPIC3D is a C++ and parallel (MPI plus OpenMP worksharing) Particle-in-Cell code to simulate space and fusion plasmas during the interaction between the solar wind and the Earth magnetic field (see Figure 2).

Fig. 2: Interaction between the solar wind and the Earth magnetic field [1].

The magnetosphere is a large system with many complex physical processes, requiring realistic domain sizes and billions of computational particles. The numerical discretization of Maxwell's equations and particle equations of motion is based on the implicit moment method that allows simulations with large time steps and grid spacing still retaining the numerical stability. Plasma particles from the solar wind are mimicked by computational particles. At each computational cycle, the velocity and the location of each particle are updated, the current and charge densities are interpolated to the mesh grid, and Maxwell's equations are solved. Figure 3 depicts these computational steps in iPIC3D.

Fig. 3: Structure of the iPIC3D code.

The computational load is divided among processes. Each process computes a subdomain of the simulation box. Extensive communication between processes are necessary for the halo exchange of field values defined on the mesh and for communicating particles exiting a subdomain. In the particle mover part hundreds of particles per cell are constantly moved, resulting in billions of particles in large-scale simulations. All these particles are completely independent from each other, which ensures very high scalability. MPI communication at this stage is only required to transfer some of the particles from one cell or a subdomain to its neighbor.

The iPIC3D code is used in production by several research groups in European universities, and it is a pilot application in different large European Exascale projects.

IV. IMPLEMENTATION GUIDELINES

A. Enabling calls to MPI by multiple OpenMP threads

As the first step to allow multiple OpenMP threads call MPI functions simultaneously, with no restrictions, in the original version of the iPIC3D code, the MPI initialization routine has been changed: in `utility/MPIdata.cpp` the `MPI_Init()` function was replaced by `MPI_Init_thread()`, and now it looks as `MPI_Init_thread(0, 0, MPI_THREAD_MULTIPLE, &provided)`.

This allowed to initialize the MPI execution environment. `MPI_Init_thread()` has two additional parameters for the level of thread support required, and for the level of thread support provided by the MPI library implementation. After the initialization, the programmer may add OpenMP

directives and runtime calls as long as he/she sticks to the level of thread safety that was specified in the call to `MPI_Init_thread()`.

In order to get the support from MPI that allows any thread to call the MPI library at any time, one should have `MPI_THREAD_MULTIPLE` for the `required` variable, and the value of the `provided` variable has to be equal to 3.

With the `MPI_THREAD_MULTIPLE` option, any thread within any process is eligible to call MPI functions, and the MPI library is responsible for thread safety within that library and for libraries that it uses. Figure 4 illustrates two cases of the MPI and OpenMP programming models interoperability:

- the standard MPI and OpenMP interoperability within one code (on the left side of the figure);
- when multithreaded MPI is used in an MPI plus OpenMP code (on the right side of the figure).

Fig. 4: MPI and OpenMP interoperability.

In order to select a thread support level higher than `MPI_THREAD_SINGLE`, a programmer should set the environment variable `MPICH_MAX_THREAD_SAFETY`, otherwise provided will return `MPI_THREAD_SINGLE`. For the case of `MPI_THREAD_MULTIPLE`, it should be as `export MPICH_MAX_THREAD_SAFETY=multiple`.

In iPIC3D, the halo exchange consists of three communication phases: between faces, between edges, and between corners. The first phase (communication between faces) includes six sequential calls to `MPI_Irecv()` and six sequential calls to `MPI_Isend()`. With the usage of `MPI_THREAD_MULTIPLE`, it is possible to parallelize each set of these calls. As there are only six independent serial calls to the non-blocking MPI receive functions and six to the non-blocking MPI send functions, the maximal possible level of parallelism here is six. And therefore, taking into account that the number of CPUs in a node of a supercomputer is

a power of two, there are several possible strategies for the parallelization of the region (with two, four and eight threads). Table I describes these approaches.

TABLE I: Possible parallelization strategies for the phase of communication between faces.

No thread support	6 serial calls to <code>MPI_Irecv()</code> functions and 6 <code>MPI_Isend()</code> functions
2 OpenMP thrs. per 1 MPI proc.	Within 1 MPI process, 2 OpenMP threads call <code>MPI_Irecv()</code> functions in parallel and then <code>MPI_Isend()</code> functions in parallel
4 OpenMP thrs. per 1 MPI proc.	Within 1 MPI process, 4 OpenMP threads call <code>MPI_Irecv()</code> functions in parallel and then <code>MPI_Isend()</code> functions in parallel, and thereafter 2 OpenMP threads do the same with <code>MPI_Isend()</code> and <code>MPI_Irecv()</code> functions
8 OpenMP thrs. per 1 MPI proc.	Within 1 MPI process, every OpenMP thread (except 2) make calls to <code>MPI_Irecv()</code> in parallel and then to <code>MPI_Isend()</code> in parallel

Listing 1 shows a pseudocode of the halo exchange in iPIC3D for the phase of communication between faces, without `MPI_THREAD_MULTIPLE`, so threads cannot call an MPI function, and therefore all MPI function calls have to be sequential.

Listing 1: Pseudocode of communication between faces in the halo exchange in iPIC3D without enabling `MPI_THREAD_MULTIPLE`.

```

if(there is a neighbor on the left on the
    X axis – left_neighborX)
{
    MPI_Irecv(one yzFace–type element
        from left_neighborX into
        &vector[0][1][1] with XR tag);
    communicationCnt[0]=1;
}
if(there is a neighbor on the right on the
    X axis – right_neighborX)
{
    MPI_Irecv(one yzFace–type element
        from right_neighborX into
        &vector[nx–1][1][1] with XL tag);
    communicationCnt[1]=1;
}
...
if(communicationCnt[0]==1)
{
    MPI_Isend(one yzFace–type element
        from &vector[1+offset][1][1] to
        left_neighborX with XL tag);
}
if(communicationCnt[1]==1)
{
    MPI_Isend(one yzFace–type element
        from &vector[nx–2–offset][1][1] to
        right_neighborX with XR tag);
}

```

Listing 2 illustrates the same case in iPIC3D, but with enabling OpenMP thread support for MPI for the case with

four OpenMP threads using `MPI_THREAD_MULTIPLE` four threads call MPI functions in parallel. As there are six possible parallel calls to the `MPI_Irecv()` functions and another six to the `MPI_Isend()` functions, four calls to `MPI_Irecv()` are executed by the threads in parallel, and then the remain two calls to `MPI_Irecv()` are performed in parallel. Same strategy is applied to `MPI_Isend()`.

Listing 2: Pseudocode of communication between faces in the halo exchange in iPIC3D with `MPI_THREAD_MULTIPLE` using four OpenMP threads per one MPI process.

```

#pragma omp parallel default(shared)
    private(id_thread, nthreads)
{
    id_thread=omp_get_thread_num();
    nthreads=omp_get_num_threads();
    if (nthreads==4)
    {
        if(id_thread==0)
            if(there is a neighbor on the left
                on the X axis – left_neighborX)
            {
                MPI_Irecv(one yzFace–type element
                    from left_neighborX into
                    &vector[0][1][1] with XR tag);
                communicationCnt[0]=1;
            }
        if(id_thread==1)
            if(there is a neighbor on the right
                on the X axis – right_neighborX)
            {
                MPI_Irecv(one yzFace–type element
                    from right_neighborX into
                    &vector[nx–1][1][1] with XL tag);
                communicationCnt[1]=1;
            }
        ...
        #pragma omp barrier
        if(id_thread==0)
            if(communicationCnt[0]==1)
            {
                MPI_Isend(one yzFace–type element
                    from &vector[1+offset][1][1]
                    to left_neighborX with XL tag);
            }
        if(id_thread==1)
            if(communicationCnt[1]==1)
            {
                MPI_Isend(one yzFace–type element
                    from &vector[nx–2–offset][1][1]
                    to right_neighborX with XR tag);
            }
    }
}

```

The phase of communication between edges also consists of the number of calls to the `MPI_Irecv()` functions and to the `MPI_Isend()` functions. The phase of communication between corners uses two calls to `MPI_Irecv()` and two calls to `MPI_Isend()`. In order to parallelize these two phases, exactly the same approach has been used as for the phase of communication between faces.

B. Adding OpenMP tasking on top of the previous hybrid version

In order to use OpenMP tasking in the new hybrid version of iPIC3D based on `MPI_THREAD_MULTIPLE`, `#pragma omp task` directives were added in the existing for-loops of the particle mover part of iPIC3D.

In general, there are two options of how to change for-loops to enable tasking:

- single task producer (Listing 3);
- parallel task producer (Listing 4).

Listing 3: Single task producer.

```
void foo(float* x, float* y, float a, int n) {
    #pragma omp parallel {
        #pragma omp single {
            for (int i = 0; i < n; i++)
                #pragma omp task {
                    y[i] = y[i] + a*x[i];
                }
        }
    }
}
```

The first approach is a common pattern, but it is not NUMA friendly. Here the `#pragma omp single` directive allows only one thread to execute the for-loop, while the rest of the threads in the team wait at the implicit barrier at the end of the `single` construct. At the each iteration, a task is created.

Listing 4: Parallel task producer.

```
void bar(float* x, float* y, float a, int n) {
    #pragma omp parallel {
        #pragma omp for {
            for (int i = 0; i < n; i++)
                #pragma omp task {
                    y[i] = y[i] + a*x[i];
                }
        }
    }
}
```

The second approach (depicted in Listing 4) is more NUMA friendly, and it was used for iPIC3D. Here the task creation loop is shared among the threads in the team.

V. EXPERIMENTAL RESULTS

In this section, we discuss the performance results from the weak scaling tests. We evaluated the performance of four different iPIC3D implementations:

- 1) original version based on pure MPI (compiled without an OpenMP flag);
- 2) original version that combines MPI (common `MPI_Init()`) and OpenMP (this version is used as a baseline for the comparison reason);
- 3) our implementation when enabling multithreaded MPI (`MPI_THREAD_MULTIPLE`) in the original hybrid version;
- 4) our implementation when adding OpenMP tasking on top of our previous implementation.

To illustrate the scalability of iPIC3D, each of these implementations executes two standard iPIC3D simulation cases:

- GEM3D: a 3D magnetotail reconnection simulation with respect to the Geospace Environment Modeling (GEM) reconnection challenge;
- Magnetosphere3D: a 3D Earth radiation belts simulation.

In addition, we use two different input data sizes (regimes) for both simulation cases:

- *field* solver dominated regime: a relatively small number of particles (27 per cell), so that the most computationally expensive part of the iPIC3D code results in the Maxwell field solver;
- *particle* mover dominated regime: it is characterized by a large number of particles (1,000 per cell), which, therefore, stresses the particle mover.

Hence, there are four different test cases (two simulation types with two regimes). For simplicity we refer to them as GEM3D-field, GEM3D-particle, Magnetosphere3D-field, Magnetosphere3D-particle. Thus, each figure in this section corresponds to one of these test cases.

These tests were performed on the Beskow supercomputer at the PDC Center for High Performance Computing at KTH. It is a Cray XC40 system, based on Intel Haswell processors and Cray Aries interconnect technology. Compute memory per node is 64 GB. Each node has 32 cores divided between 2 sockets, with 16 cores each, where the cores are Intel Xeon E5-2698v3. Figure 5 illustrates a Beskow computing node. All iPIC3D versions are compiled with Cray CC v.8.3 compiler and Cray MPICH v.7.0.4 library.

Fig. 5: Scheme of Beskow node architecture.

Three-dimensional decomposition of MPI processes on X-, Y- and Z-axes is used, resulting in different topologies of MPI processes, each having two, four, and then eight threads in addition. Table II explains how the number of MPI processes varies (at the intersection of columns and rows) when the number of threads is changed in our experiments. We turn off hyperthreading and also disable threading for the original MPI version.

In order to ensure a fair comparison, the number of iterations in the linear solver was fixed to 20, although in a real simulation the number of iterations depends on the speed of convergence. The total number of particles and cells in a simulation are calculated from numbers of

Fig. 6: Weak scaling tests of the GEM3D simulation with the field solver dominated regime for the four different iPIC3D implementations.

Fig. 7: Weak scaling tests of the GEM3D simulation with the particle mover dominated regime for the four different iPIC3D implementations.

cells and numbers of particles per cell in the X-, Y- and Z- directions: $n_{xc} \times n_{yc} \times n_{zc} \times n_{pcelx} \times n_{pcely} \times n_{pcelz}$ and $n_{xc} \times n_{yc} \times n_{zc}$, respectively. Thus, for example, in the particle mover dominated Magnetosphere3D simulation on 32 physical cores (2x2x2 MPI processes x 4 OpenMP threads), 27×10^6 particles and $30 \times 30 \times 30$ cells were used, and the simulation size increased proportionally to the number of processes.

TABLE II: Number of MPI processes with respect of the number of OpenMP threads employed.

# physical cores	no thr.	2 thr.	4 thr.	8 thr.
32	32	16	8	4
64	64	32	16	8
128	128	64	32	16
256	256	128	64	32
512	512	256	128	64
1024	1024	512	256	128
2048	2048	1024	512	256

Figures 6 to 9 correspond to the GEM3D-field, GEM3D-particle, Magnetosphere3D-field, and Magnetosphere3D-particle cases, respectively. For clarity reasons, here we provide results for the iPIC3D executions with two and eight threads; the original MPI implementations uses the number of MPI processes equal to the number of physical cores. The design of these plots is uniform:

- the X-axis represents the number of total physical cores (see Table II regarding the number of MPI processes);
- the Y-axis is the speedup of each iPIC3D implementation normalized with respect to the original hybrid version; all hybrid versions run with the same number of threads as other three versions;
- in each plot, the red bars correspond to the original version based on pure MPI; the yellow bars stands for the original iPIC3D based on MPI plus OpenMP; the green bars – to our implementation when enabling multithreaded MPI in the original hybrid version; and the blue bars – to our implementation with OpenMP tasking on top of the previous implementation;
- each bar is calculated as the mean simulation time with values compiled from multiple test runs.

For both GEM3D and Magnetosphere3D simulation cases, the results of the particle mover dominated regime and the field solver dominated regime are distinguishable. For the former regime, the simulation time is quite stable (see Figure 7 and Figure 9) as there is huge amount of particles to be calculated and a good opportunity to overlap communication with computation. Additionally, we deal with the weak scaling tests – input data grows proportionally to the number of processes, so each process always has the same amount of workload – the speedup does not change much with the increase in the number of physical cores. For the field solver regime, numbers fluctuate more (see Figure 6 and Figure 8):

there is much less computation, therefore communication is the driven factor in the execution time.

In most of the simulation cases, the original pure MPI version of iPIC3D (the red bars) is 30% on average slower than the original hybrid version. The performance gap is higher on larger number of physical cores. To clarify, we compare the original pure MPI version running on N MPI processes (where N is the number of physical cores) and the hybrid version running on d OpenMP threads x n MPI processes ($d \times n = N$ physical cores). Hence, the number of MPI processes is not the same.

In Figure 6, with the higher number of OpenMP threads per process and on larger number of physical cores, both new implementations outperform the baseline. For the case with eight threads on 2048 physical cores (see Figure 6b) the OpenMP task-based version (the blue bar) is roughly 15% faster than the baseline version (the yellow bar); the multithreaded MPI-based version (the green bar) even more than 32% faster. For the case with four threads (not presented here) the speedup were 41% and 53% correspondingly.

Figure 7 shows that our OpenMP task-based version is on average 20% slower than the reference version. With the large number of cores, its performance increases. The multithreaded MPI-based version on large number of physical cores (512, 1024 and 2048) is 3-6% faster than the reference. Figure 9 illustrates similar performance results: the multithreaded MPI version outperforms the original hybrid iPIC3D on large cores. The OpenMP task-based performance results are slightly better than in Figure 7.

In Figure 8, the larger number of threads – the better speedup for both new implementations. The OpenMP tasking version outperforms the original hybrid version starting from 512 cores for the case with two threads (Figure 8a) and from 128 cores with eight threads (Figure 8b). On 2048 physical cores with eight OpenMP threads, this version is 13% faster than the reference. In addition, the multithreaded MPI-based implementation often shows the best performance: in particular, on 2048 physical cores with eight threads, it has the speedup of 27% compared to the referenced version.

VI. CONCLUSIONS

The goal of this work was to study interoperability and performance of different parallel programming models in a large scientific real-world application and to share the gained experience in the form of a best practice programming guide. As a case study, we used the iPIC3D code that supports the pure MPI and the conventional hybrid MPI plus OpenMP version. Our focus was on the latter, where we employed two API combination scenarios: the first was to enable multiple OpenMP threads call MPI functions simultaneously; the second was to utilize the OpenMP tasking model on top of the first scenario. Both API interoperability scenarios showed promising results for codes that often occur in scientific simulations. In the field solver dominated regime, where there is a little overlap of communication and computation, on

Fig. 8: Weak scaling tests of the Magnetosphere3D simulation with the field solver dominated regime for the four different iPIC3D implementations.

Fig. 9: Weak scaling tests of the Magnetosphere3D simulation with the particle mover dominated regime for the four different iPIC3D implementations.

the large number of cores (512 and larger) both our new implementations were faster than the original hybrid version.

Our future study is focused on extending these preliminary results as well as studying interoperability of other API combinations. Our next research step is to evaluate the use of MPI endpoints and OpenMP threading in the iPIC3D code. MPI endpoints aim to address some problems with the multithreaded mode, remove threading restrictions in MPI, and facilitate high performance communication between multithreaded OS processes. Additionally, we are targeting to experiment with asynchronous programming models – such as GASPI (a PGAS-based programming model) – and other shared-memory runtime systems – OmpSs and StarPU. At the end, we will contribute to training materials, best practice guides and possible recommendations to standards bodies via INTERTWinE – the EC-funded project on programming-model design and implementation for the Exascale.

ACKNOWLEDGMENT

This work was funded by the European Union's Horizon 2020 Research and Innovation programme through the INTERTWinE project under Grant Agreement no. 671602 (www.intertwine-project.eu). The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC Centre for High Performance Computing (PDC-HPC).

REFERENCES

- [1] Interaction between solar wind and Earth magnetic field. <https://goo.gl/RrPQZV>, 2016. [Online].
- [2] Dana Akhmetova, Gokcen Kestor, Roberto Gioiosa, Stefano Markidis, and Erwin Laure. On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 428–437. IEEE, 2015.
- [3] Greg R Andrews. *Foundations of parallel and distributed programming*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [4] Rocco Aversa, Beniamino Di Martino, Massimiliano Rak, Salvatore Venticinque, and Umberto Villano. Performance prediction through simulation of a hybrid mpi/openmp application. *Parallel Computing*, 31(10):1013–1033, 2005.
- [5] Eduard Ayguadé, Nawal Copt, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.
- [6] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep.*, 15, 2008.
- [7] OpenMP Architecture Review Board. Openmp 3.0 api specifications. <http://www.openmp.org/wp-content/uploads/spec30.pdf>, 2017. [Online].
- [8] Steve Bova, Clay Breshears, Rudolf Eigenmann, Henry Gabb, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini, and Veer Vatsa. Combining message-passing and directives in parallel applications. *SIAM News*, 32(9):10–14, 1999.
- [9] IJ Bush, CJ Noble, and RJ Allan. Mixed openmp and mpi for parallel fortran applications. In *European Workshop on OpenMP (EWOMP2000)*, 2000.
- [10] Franck Cappello and Daniel Etienne. Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 12–12. IEEE, 2000.
- [11] Edmond Chow and David Hysom. Assessing performance of hybrid mpi/openmp programs on smp clusters. *Lawrence Livermore National Laboratory, Tech. Rep. UCRL-JC-143957*, 2001.
- [12] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.
- [13] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.
- [14] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of parallel computing*, volume 3003. Morgan Kaufmann Publishers San Francisco, 2003.
- [15] Jake Duthie, Mark Bull, Arthur Trew, and Lorna Smith. Mixed mode applications on hpcx. Technical report, HPCx Consortium, 2004.
- [16] B Estrade. Hybrid programming with mpi and openmp. In *High Performance Computing Workshop*, 2009.
- [17] MPI Forum. Mpi 2.0 api specifications. <http://mpi-forum.org/docs/mpi-2.0/mpi2-report.pdf>, 2017. [Online].
- [18] MPI Forum. Mpi 2.1 api specifications. <http://mpi-forum.org/docs/mpi-2.1/mpi21-report.pdf>, 2017. [Online].
- [19] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.
- [20] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.
- [21] David S Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 10. IEEE Computer Society, 2000.
- [22] Ilya Ivanov, Jing Gong, Dana Akhmetova, Ivy Bo Peng, Stefano Markidis, Erwin Laure, Rui Machado, Mirko Rahn, Valeria Bartsch, Alistair Hart, and Paul Fischer. Evaluation of parallel communication models in nekbone, a nek5000 mini-application. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing, CLUSTER '15*, pages 760–767, Washington, DC, USA, 2015. IEEE Computer Society.
- [23] Kushal Kedia. Hybrid programming with openmp and mpi. Technical report, Technical Report 18.337 J, Massachusetts Institute of Technology, 2009.
- [24] Richard D Loft, Stephen J Thomas, and John M Dennis. Terascale spectral element dynamical core for atmospheric general circulation models. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 26–26. IEEE, 2001.
- [25] Phu Luong, Clay P Breshears, and Le N Ly. Coastal ocean modeling of the us west coast with multiblock grid and dual-level parallelism. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 9–9. ACM, 2001.
- [26] Stefano Markidis, Giovanni Lapenta, et al. Multi-scale simulations of plasma with ipic3d. *Mathematics and Computers in Simulation*, 80(7):1509–1519, 2010.
- [27] Timothy G Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Pearson Education, 2004.
- [28] Kengo Nakajima. Parallel iterative solvers for finite-element methods using an openmp/mpi hybrid programming model on the earth simulator. *Parallel Computing*, 31(10):1048–1065, 2005.
- [29] Peter S Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.
- [30] Rolf Rabenseifner. Hybrid parallel programming on hpc platforms. In *proceedings of the Fifth European Workshop on OpenMP, EWOMP*, volume 3, pages 185–194, 2003.
- [31] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2010.
- [32] Lorna Smith and Mark Bull. Development of mixed mode mpi/openmp applications. *Scientific Programming*, 9(2-3):83–98, 2001.
- [33] Thomas Lawrence Sterling, Paul C Messina, and Paul H Smith. *Enabling technologies for petaflops computing*. MIT press, 1995.
- [34] ASCAC Subcommittee. Top ten exascale research challenges. *US Department Of Energy Report*, 2014.