A Massively Parallel Distributed N-Body Application Implemented with HPX

Zahra Khatami[1,3], Hartmut Kaiser[1,3], Patricia Grubel[2,3], Adrian Serio[3] and J. Ramanujam[1]
[1]Center for Computation and Technology, Louisiana State University
[2]New Mexico State University, Las Cruces, NM
[3]The STE||AR Group, http://stellar-group.org

***Abstract*—One of the major challenges in parallelization is the difficulty of improving application scalability with conventional techniques. HPX provides efficient scalable parallelism by significantly reducing node starvation and effective latencies while controlling the overheads. In this paper, we present a new highly scalable parallel distributed N-Body application using a *future*-based algorithm, which is implemented with HPX. The main difference between this algorithm and prior art is that a *future*-based request buffer is used between different nodes and along each spatial direction to send/receive data to/from the remote nodes, which helps removing synchronization barriers. HPX provides an asynchronous programming model which results in improving the parallel performance. The results of using HPX for parallelizing Octree construction on one node and the force computation on the distributed nodes show the scalability improvement on an average by about 45% compared to an equivalent OpenMP implementation and 28% compared to a hybrid implementation (MPI+OpenMP) [1] respectively for one billion particles running on up to 128 nodes with 20 cores per each.**

***Index Terms*—HPX, High Performance Computing, Parallel Runtime Systems, N-Body Application, Asynchronous Task Execution.**

I. Introduction

The N-Body problem is known to be communication intensive, significantly affecting the execution time, due to message latencies and overheads [2], [3]. Updating the information for each of the particles requires accessing to the information about all other particles. Various algorithms have been proposed to improve the parallel scalability of an N-Body application. The brute-force N-Body algorithm for computing new positions of all N particles has $O(N^2)$ time complexity. The Barnes-Hut and Fast Multipole Method (FMM) are well-known algorithms with $O(NlogN)$ and $O(log(1/\epsilon)N)$ time complexity respectively [4], [5]. These techniques reduce the number of the computations for each of the particles, resulting in an increase of the number of the particles that can be processed in the simulation [1]. In this research the Barnes-Hut algorithm is used as a basis for studying an N-Body problem.

The typical approach for parallelization is to decompose the domain space into several sub-domains and to assign each of those to one of the nodes. OpenMP is usually used for parallel processing within a node and MPI is usually used for communication between different nodes. However, local and global synchronization barriers imposed by the programming models of OpenMP and MPI impede an optimal parallelization for N-Body problems [6]. In each time step with this hybrid model, the node can continue its progress to the next iteration only after completing updating the information related to all of its particles, which makes it very difficult to achieve optimum scaling. So, full parallelization in both space and time has to be considered to achieve a maximum possible level parallelism [4]. The *future* concepts as implemented in HPX is one possible way of enabling full parallelization in time and space. HPX allows for the computation to continue its progress to the next time step instead of waiting for a node to receive all required information from the remote nodes and to complete the computations related to its all particles. In this paper, a *future*-based request buffer is used between different nodes that allows to continue the process without waiting for the previous step to be completed, which results in removing global barriers and thus improving the parallel performance. To our knowledge, we present a first attempt of implementing a distributed N-Body application with HPX and with using a *future*-based request buffer for communication between different nodes.

One of the challenges in updating the particle information is that the node should know to which of the remote nodes the information about a particle should be sent. For this purpose, data distribution across the nodes is designed based on assigning a unique id provided by HPX for each cube in an Octree and for each node. This algorithm helps the node to recognize all cubes ids in the remote nodes and confirms a destination of the particle. Also, it is shown that the algorithm minimizes idle nodes and maintains high utilization.

In this paper we show algorithmic and implementation improvements using HPX to hide communication latencies and to achieve the desired scalability for an N-body code by combining task-based parallelism, grain size control, and prevalently asynchronous execution. We observe similar performance running on one node using both HPX and hybrid model for the force computation and the Octree construction. We are however able to obtain a scalability improvement in using HPX for more number of threads and nodes.

The remainder of this paper is structured as follows: in Section II, some related research and methods for parallelization are explained as well as introducing HPX with the key feature that separate it from the conventional techniques and parallel programming models; Section III gives an overview of an N-Body and its asynchronous parallelization with HPX;

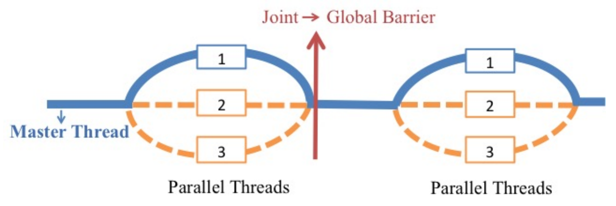


Figure 1: Fork-join model. The master thread creates a group of parallel tasks. Only when all threads complete their tasks the computation will continue with sequential tasks on the master thread, causing an implicit global barrier at each join-point.

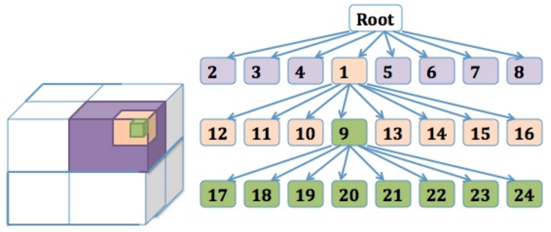


Figure 2: The Octree data structure used in the described N-Body application.

Section IV presents the experimental tests and the scaling speedup of the results. Then, we show the significant scalability improvement in HPX comparing to MPI and OpenMP. The conclusions can be found in Section V.

II. Related Work

Most of the parallelization methods for an N-Body are based on the fork-join model [4]–[9], which is demonstrated in Fig.1. In this model, the master thread creates a group of the parallel tasks in each time step and only when all the threads complete their tasks, the computation process can continue to the next time step, causing an implicit global barrier [10], [11]. The scalability and the efficiency of the parallel N-Body applications are constrained by conventional parallel programming when the barrier times dominate.

Distributed N-Body applications are usually parallelized with hybrid methods, which use OpenMP for parallel processing within a node and MPI for communication between different nodes. Local and global synchronization barriers, communication overhead and load balancing are the major challenges in these programs [6], [9], [12]. Communication between different nodes in MPI often creates large overheads that degrades the parallel performance [12]. In *blocking communication* applied using *"MPI_ Send()"* and *"MPI_ Recv()"*, each node must complete its current process before it can continue to the next step. On the other hand, in *non-blocking communication* applied using *"MPI_ Isend()"* and *"MPI_ Irecv()"*, communications are not blocked even if the current process is not yet completed, which allows the other works to proceed. However, *"MPI_ Wait()"* or *"MPI_ Probe()"* should be used in this method to confirm whether communication is completed, which avoids achieving an optimum scaling. Another problem in using MPI within an N-Body is related to *granularity*, which is defined as the amount of computation per each execution block in a task. Fine-grained tasks have small quantum of work and help in achieving a better load balancing [7], [13], while on the other hand, coarse-grained tasks have large amount of computations. In MPI, the coarse-grained tasks are often chose to obtain optimal efficiency [12], which avoids having a better load balancing [14].

Integrating OpenMP into MPI is expected to increase the parallel performance, however the overheads due to the loop scheduling and synchronization in OpenMP [15] in addition to the mentioned difficulties usually avoids having the desired speedup [6], [16]. Various optimization techniques have been proposed to improve hybrid programs performance [11], [16]–[18]. In [1], sorting methods are used to obtain optimal parallelization in an N-Body. Other works such as [9] implement space filling curves, which achieve better load balancing.

In this paper, HPX is used for both communication between different nodes and parallel processing within each node to develop a new parallel distributed N-Body application that meets these challenges. HPX is a parallel C++ runtime system that facilitates distributed operations and enables fine-grained task parallelism, which results in better load balancing and lower communication overheads. The *future* construct is used in HPX and the goal is to let every computation proceed as far as possible. Therefore, HPX allows the system to start working on the next time step instead of waiting for all computations to be completed, which mitigates global barriers and improves parallel performance.

III. Studied Model

Here, we briefly describe the steps of the N-Body simulation: Octree construction, Interaction list creation, and Force computation. Then, we explain how N-Body is simulated and modeled with HPX.

A. N-Body Problem Overview

1) Octree construction: Fig.2 shows the Octree data structure used in an N-Body [5], [19]. The parameter $N_{threshold}$ is an upper bound for the number of the particles within each cube. Each cube is subdivided into eight equally sized subcubes if it has more than $N_{threshold}$ particles and in each step, each particle is reassigned to one of the newly created subcubes. In this research, the adaptive Octree is created, which is able to be modified automatically if the positions of the particles are changed and it grows to more cubes if the number of the particles exceeds $N_{threshold}$. The center of mass for each cube is computed with Eq. 1 based on the positions of the particles in that cube [7],

$$CM = mass_{total} \times \sum_{i=1}^{n} r_{p_i} \times mass_{p_i} \tag{1}$$

where CM is the center of mass of the cube and $mass_{total}$ is the sum of those particles masses.

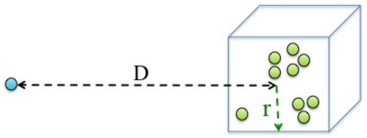


Figure 3: The gravitational potential of a distant group of particles is approximated as the potential of a single particle located in the center of mass of all particles in the cube.

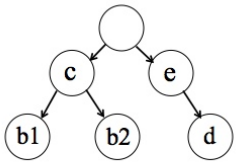


Figure 4: The interaction list scheme used by the fast multi-pole method (FMM).

Parallelizing an Octree construction is possible if it is constructed from the root to the leaf, layer by layer. Here, breadth first search is used for traversing the group of the cubes in parallel. id assignment eases traversing and constructing the Octree in parallel. A unique id is assigned to each cube by considering its parent's id as follows: if p is the parent's id, each subcube has $p + 8 \times i$ as its id, where i varies from 1 to 9, the sequence of the parent's subcubes. For example, in Fig.2, a cube with id as 1, has eight subcubes with id from 9 to 16. With this method, each cube can also confirm the id of its neighbors by knowing its parent's id.

2) Interaction list creation: Once the particles are partitioned into an Octree, the interaction list for each particle is created by a traversal of the Octree by each sub-Octree piece. The interaction list of the particle holds the list of the particles that are near objects to that particle and the cubes that are modeled as faraway objects to that particle. Fig.3 shows that the remote cube of the particles is treated as a single particle only if D is greater than r/θ; where D is the distance between the cube and the particle, r is the radius of the cluster of the particles, and the parameter θ controls the error of the approximation. A smaller ratio produces more accurate results but increases the execution time. On the other hand, a larger ratio produces less accurate results but decreases the execution time.

In the Barnes-Hut method, only the particle-to-particle interactions list are computed [20], while in the FMM method, interactions between internal cubes are computed [21], [22]. So in the FMM method, it is not required to study the effect of all particles in one cube on those in another one, when those two cubes are considered as being well-separated. As a result, the interaction list of all particles in a cube is approximated as a single entry in the interaction list and it is reduced to have one traversal per subcube. In this paper, Barnes-Hut algorithm is modified based on this technique of the FMM algorithm. For example in Fig.4, if c is the parent of b_1 and b_2 and e is the parent of d, instead of d sending its information directly to b_1 and b_2 separately, d sends its information to c and c provides it to them. Therefore, the time complexity of this modified model is $O(log(1/\epsilon)N)$, which is less than that of Barnes-Hut algorithm, $O(NlogN)$.

3) Force computation: In the three dimensional N-Body problem, there are N particle masses m_i, moving under the influence of gravitational attraction. Each particle m_i has an initial position r_i with initial velocity v_i. We use Newton's law of gravity, Eq. 2, for computing the gravitational force felt by the mass m_i as it interacts with a single mass m_j. The gravitational force exerted on each particle is computed by considering all particles and subcubes included in the interaction list of that particle.

$$F_{ij} = \frac{G\ m_j\ m_i\ (r_i - r_j)}{\|\ r_j - r_i\ \|^3} \tag{2}$$

The velocity and the new position of each particle are computed in Eq.3 and Eq.4 respectively,

$$v_i = v_i + \frac{F_{ij} \times t}{m_i} \tag{3}$$

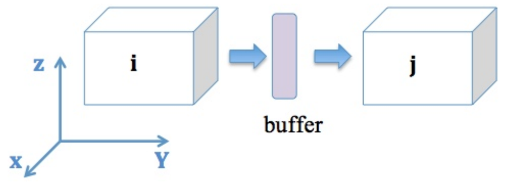
$$r_i = r_i + v_i \times t + \frac{0.5 \times t^2 \times F_{ij}}{m_i} \tag{4}$$

where t is a time step. All given pairwise interactions are computed based on the positions of the related particles.

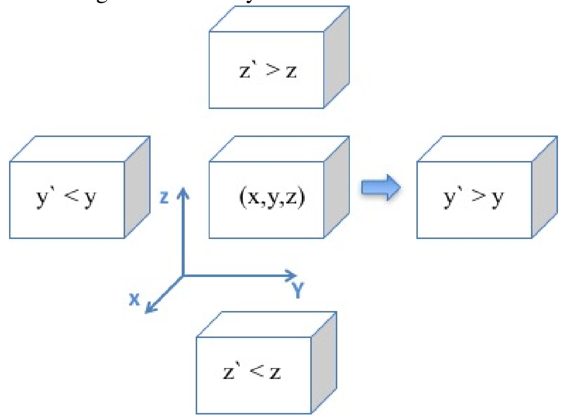
B. Algorithm Implemented with HPX

In each time step, the particle may need to move to another cube in the remote node based on the force on it. Sending one large amount of particles is more efficient than sending several small ones in parallel. Therefore, it is important to aggregate all particles needed to be sent to a direction i from all cubes within a *locality* and send all of them together. Hence, we create a request message containing particle information and store this massage in a sending-buffer that is destined for that remote *locality*. Each *locality* has six sending-buffers associated with its six remote neighbor localities, defined as *hpx::lcos::local::receive-buffer* that contain data shared between two corresponding localities. Sending and receiving-buffers are the same. For example, in Fig.5a, when *locality* i sends information to *locality* j, it stores this information in the buffer ij and *locality* j gets it from this buffer. After receiving a new particle from the remote localities, each *locality* confirms whether this particle belongs to it or not. If so, this particle traverses the Octree within that *locality* to find its new parent. But if the particle does not belong to that *locality*, it will be sent to the buffer of the remote localities in the same direction. We use *hpx::component* for accessing data from the remote *localities*, which is a C++ object that can be created remotely and all its member functions can be remotely invoked [23].

The key implementation technique used in an N-Body distributed model with HPX, which seperates it from the previous



(a) The scheme of sending a particle to a remote *locality*. When *locality* i sends information to *locality* j, it stores this information in the buffer ij and *locality* j receives that information from this buffer. HPX uses *future* based algorithms, which allows the *locality* to continue its tasks even if it has not received all needed information from its neighbors or has not yet sent all the information the remote neighbors require, which avoids a global barrier synchronization.



(b) The destination *locality* is chosen based on the particle's new position $r_{i,new}(x_{i,new}, y_{i,new}, z_{i,new})$: if $x_{i,new} > x_{i,old}$, the particle will be sent to the *locality* in front of, else to the *locality* to the back; if $y_{i,new} > y_{i,old}$, the particle will be sent to the *locality* to the right, else to the *locality* to the left; if $z_{i,new} > z_{i,old}$, the particle will be sent to the *locality* above, else to the *locality* below.

Figure 5: Sending/Receiving data to and from remote localities.

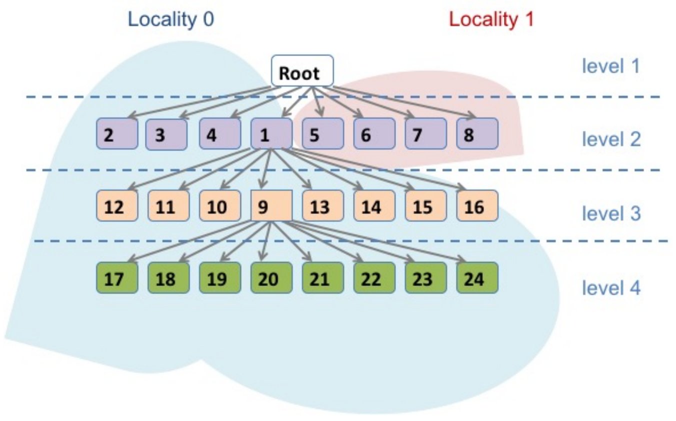


Figure 6: Data distribution using Eq.5 on 2 localities starting from level 2 of an Octree.

works such as [6] and [24], is the scheme of the sending-receiving requests that uses *future*. Each *locality* continues its tasks even if it does not receive all needed information from its neighbors or has not yet sent all the information the remote neighbors require. In each *locality*, HPX threads access the result value of the future by performing a *future.get()* and they remain suspended until the *future* value is computed while the other threads continue their process. In other words, the system has already started working on the next time step instead of waiting for a *locality* to complete all computations of its particles. This scheme allows asynchronous execution of HPX threads, also referred to as asynchronous task execution. What we achieve here is blurring the global barrier and improving parallel performance.

The communication cost is decreased by sending a particle to only one *locality* instead of broadcasting it to all remote *localities*. So here, the destination *locality* is chosen based on the particle's new position $r_{i,new}(x_{i,new}, y_{i,new}, z_{i,new})$: if $x_{i,new} > x_{i,old}$, the particle will be sent to the *locality* in front, else to the *locality* in back; if $y_{i,new} > y_{i,old}$, the particle will be sent to the *locality* at right, else to the *locality* at left; if $z_{i,new} > z_{i,old}$, the particle will be sent to the *locality* above, else to the *locality* down. Fig.5b shows the general scheme of sending a particle to the remote *locality*. One of the challenges in this figure is that the neighbor *localities* of the cubes must be known. In other words, if the particle doesn't belong to its current *locality* anymore, the *locality* should know in which direction it should send the particle to another remote *localities*. Data distribution in this paper is designed based on the cubes' ids that results in recognizing all cubes' ids in the remote *locality*. In this algorithm a unique id is assigned for each *locality* that can be retrieved using *hpx::find_here()*, and id for each remote *localities* that can be queried using *hpx::find_all_localities()*. A *locality* in HPX has a unique global identifier (GID) assigned by AGAS and can be addressed throughout the global address space; *hpx::id_type* [23]. The group of the cubes with $id_{cube\,i}$ are located on the *locality* with $l_i = id_{locality}$, which satisfies the following equation:

$$id_{cube} = i + \frac{(8 \times l_i)}{nl} \tag{5}$$

where nl is the total number of the *localities* and i varies from 1 to ol/nl. ol is initialized based on the value of nl and the level of the Octree. For example, Fig.6 illustrates data distribution on 2 localities, in which the Octree has 4 levels; level 1, level 2, level 3 and level 4. If ol is 8 and ol/nl is 4, then the range of i is $[1, 4]$. So with Eq.5, the cubes with id as 1, 2, 3 and 4 will be located at the locality with $l_i = 0$ and the cubes with id as 5, 6, 7 and 8 will be located at the locality with $l_i = 1$. Also, all of each cube's descendants will be located on the same *locality* as that cube. With this approach, each locality can ascertain the id of the cubes distributed on the other *localities* with Eq.5 and collaborate with its neighbors whenever it wants to send the particle to them. In addition, since data distribution is applied by considering the number of the nodes, each available node will be used efficiently, which results in significantly reducing nodes starvation. HPX

Table I: Various HPX API functions used in the described N-Body application

hpx::id_type	A global address (GID)
hpx::find_here	Retrieving this' *locality's* id
hpx::find_all_localities	Finding all *localities*
hpx::register_with_basename	Registering a unique GID

API functions that are used in this data distribution model are described in Table.I.

Fig.7 shows an overview of the N-Body simulated model, which has been discussed in this Section. According to this algorithm, after an Octree construction, a unique id is assigned to each cube in an Octree by considering its parent's id. id of each locality and the remote localities are found with *hpx::find_here()* and *hpx::find_all_localities* respectively. Cubes and all their descendants are assigned to the locality using Eq.5. After creating a shared request buffer between each two remote localities, the gravitational force and the position of each particle are computed with Eq.2 and Eq.4 respectively. Then, if a particle does not belong to its current locality, the remote localities will check if the particle has joined them. If so, the particle information will be propagated to the old locality as well as to the new locality. Otherwise, they will send the particle to the corresponded buffer in the same direction.

```
Initialization Steps:
 1) Constructing an Octree
 2) Assigning $id$ to each cube
 3) Finding each locality's $id$ with hpx::find-here()
 4) Finding remote localities' $id$ with hpx::find-all-localities()
 5) Locating each cube and all its descendants with Eq.5
 6) Discovering the neighbors of each locality
 7) Creating shared buffers between two neighbor localities

For each time step Do:
 8) Computing the force on each particle with Eq.2
 9) Computing the new position of each particle with Eq.4
 10) Storing $p_i$ needed to be sent, in the:
     left buffer, if $x_{p_i} > x_{parent}$
     right buffer, if $x_{p_i} < x_{parent}$
     above buffer, if $y_{p_i} > y_{parent}$
     down buffer, if $y_{p_i} < y_{parent}$
     back buffer, if $z_{p_i} > z_{parent}$
     front buffer, if $z_{p_i} < z_{parent}$
 11) Receiving $p_i$ that doesn't belong to $l_i$, from
     left, then storing it in the right buffer
     right, then storing it in the left buffer
     above, then storing it in the down buffer
     down,then storing it in the above buffer
     back, then storing it in the front buffer
     front, then storing it in the back buffer
 12) Sending data in buffer $i$ in the direction $i$,
      $i \in$ { left, right, up, down, back, front }
Loop
```

Figure 7: Algorithm for the parallel distributed N-Body simulation used for the implementation based on HPX

IV. Experimental Results

Here, we analyze the scalability of our model described in the previous section in two parts: HPX scaling on one node and HPX scaling on multiple nodes. The presented results have been acquired on LSUs' SuperMIC machine. It has a total of 382 nodes, two 10-core 2.8 GHz Intel Ivy Bridge-EP nodes with 64 GB of memory and 500 GB of local HDD storage each. We have used the version of OpenMP provided by the GNU g++ compilers version 5.1.0. The results are based on HPX version 0.9.11 [25].

A. HPX Scaling on one Node

In this section, we study the parallel processing performance within each node by comparing the execution time and the strong scaling of the parallel Octree construction for 10^6 particles with HPX versus OpenMP. *"#pragma omp parallel"* and *"#pragma omp for schedule(dynamic)"* are used for parallelizing the Octree construction with OpenMP, in which a new chunk is assigned to a thread as soon as it becomes available. Using *"omp for"* spawns several worker threads from the main thread, which are synchronized with at the end of the parallel block. This (implicit) synchronization effectively imposes a global barrier onto the computation.

Two different methods are used for parallelizing the Octree construction with HPX. In the first method, *hpx::parallel::for_each* is used to distribute the tasks in a parallel loop and construct subcubes from them recursively. By using *hpx::parallel::for_each*, HPX is able to automatically control the grain size in the runtime by sequentially executing 1% of the loop, which creates sufficient parallelism by determining the number of the iterations to run on each thread. However, this method exposes the same disadvantage as OpenMP, which is the representation of the global barriers at the end of the loop. In the second method, the subtasks are explicitly spawned inside the time iteration using *hpx::async*. The calls to *hpx::async* provides a new *future* instance representing the result of the function execution, which makes the invocation of the loop asynchronous and eliminates the global barrier synchronization.

Fig.8 shows the execution time of the Octree construction using HPX and OpenMP on one node. It is illustrated that HPX and OpenMP has by an average the same performance on 1 thread. Fig.9 shows the strong scaling, for which the problem size is kept the same as the number of threads increases. It is illustrated that both OpenMP and *hpx::parallel::for_each* inhibit the desired scalability due to imposing an implicit barrier. On the other hand, *hpx::async* has significantly better speedup, since it allows threads to continue to execute without barriers. We are able to obtain a speedup of about 19x on one *locality* with 20 cores using *hpx::async*.

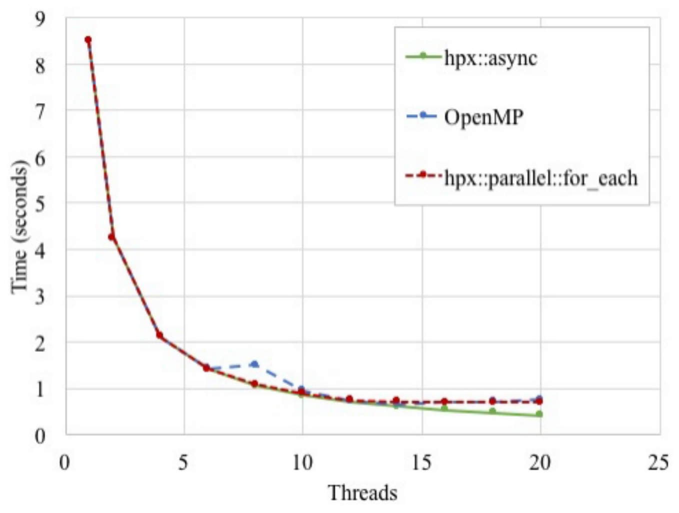


Figure 8: Comparison results of the runtimes for the Octree construction of 10^6 particles implemented with two HPX parallelization methods and OpenMP. The HPX and OpenMP application have comparable sequential execution times, the HPX application however shows better scalability for larger number of cores.

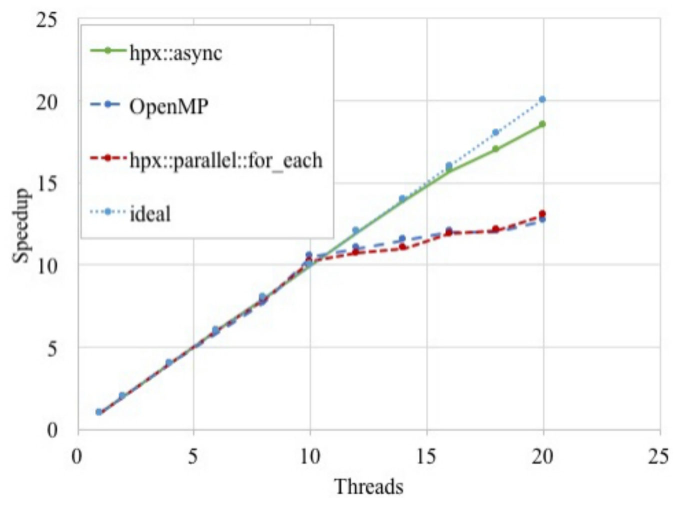


Figure 9: Comparing results of the strong scaling for the Octree construction of 10^6 particles implemented with two HPX parallelization methods and OpenMP. *hpx::async* exposes significantly better speedup, since it allows threads to continue to execute without unnecessary barriers.

B. HPX Scaling on multiple Nodes

In this section, we study the parallel performance of the distributed N-Body application for a different number of particles - 10^5, 10^6, 10^7, 10^8, and 10^9 - on a different number of the nodes running the HPX and the hybrid model (MPI with OpenMP) applications. We use a non-uniform particle distribution as the initial state of the system. Fig.10 shows the execution times for the force computations for 10^7 and 10^8 particles for 10^3 time steps. Since the communication latencies between different nodes increase with an increasing

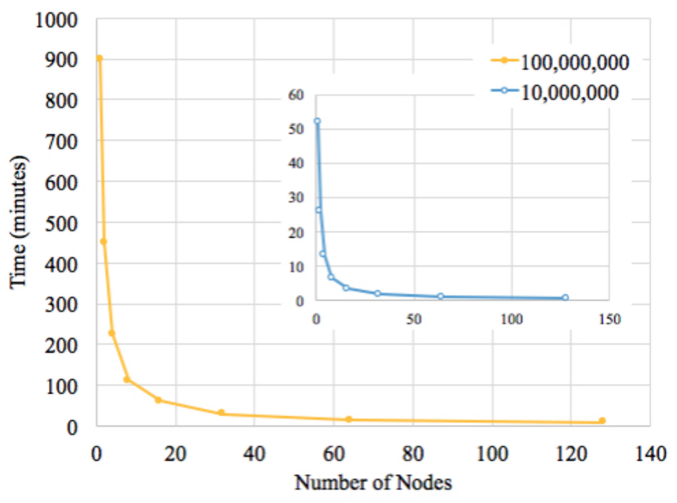


Figure 10: Execution times of the distributed N-Body application implemented with HPX on up to 128 nodes (20 cores per node) for 10^7 and 10^8 particles when executing 10^3 time steps.

number of nodes, it is usually hard to achieve a good scaling behavior. However, a close to perfect linear speedup of HPX is shown in Fig.11 for larger number of particles. It can be seen that the HPX application is able to scale almost perfectly from 1 node to 128 nodes with 20 cores on each, which has 128x speedup on 128 nodes for 10^9 particles. Fig.12 shows the perfect strong scaling (speedup) of approximately 100% for 10^8 and 10^9 particles calculated with Eq. 6,

$$\textit{Efficiency} = \frac{t_1}{(N \times t_N)} \times 100\% \qquad (6)$$

where t_1 is the execution time running on 1 node and t_N is the execution time running on N nodes. It is demonstrated that when the problem size is large enough, there will be enough work for all nodes, which allows for hiding the communication latencies behind useful work and which leads to better parallel efficiency.

In Fig.13, we compare the strong scaling between the applications written using HPX and the hybrid model (as described in [1] and [8]), where MPI is used for communication between different nodes and OpenMP is used for the parallel processing within each node. We get the comparable performance for both HPX and hybrid model when running on one node. Although non-blocking sends and receives are used in the hybrid N-Body code, the results illustrate a better performance by HPX for the larger number of the nodes, which is due to using a *future*-based request buffer between the remote nodes that allows the continuation of the process without waiting for the previous step to be completed.

Weak scaling experiments are performed to study the effects of the communication latencies, where the problem size is increased in proportion to the increase of the number of the nodes. So, the same amount of time for an N times larger problem is expected. Nearly ideal weak-scaling of the

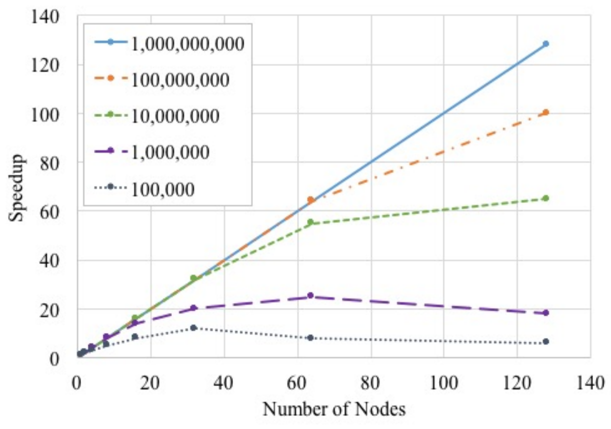


Figure 11: Strong scaling of the distributed N-Body application implemented with HPX on up to 128 nodes (20 cores per node). It shows a close to perfect HPX scalability for problem size of 10^9 particles.

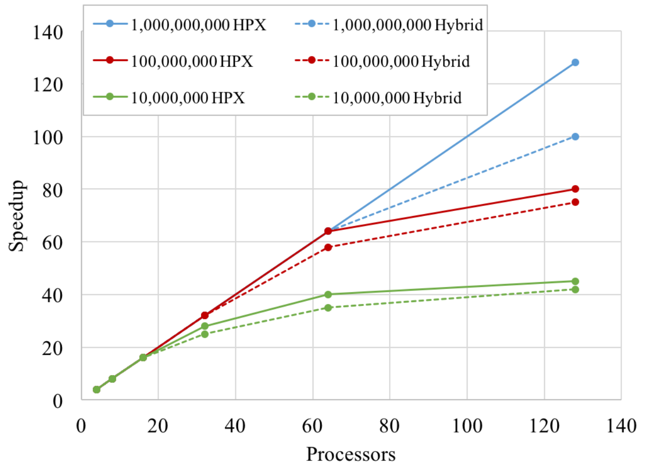


Figure 13: Comparison results of the strong scaling between HPX and a hybrid model (OpenMP/MPI), with up to 128 nodes (20 cores per node). The results illustrate a better performance for the HPX application for larger number the nodes, which is due to using a *future*-based request buffer between the remote nodes.

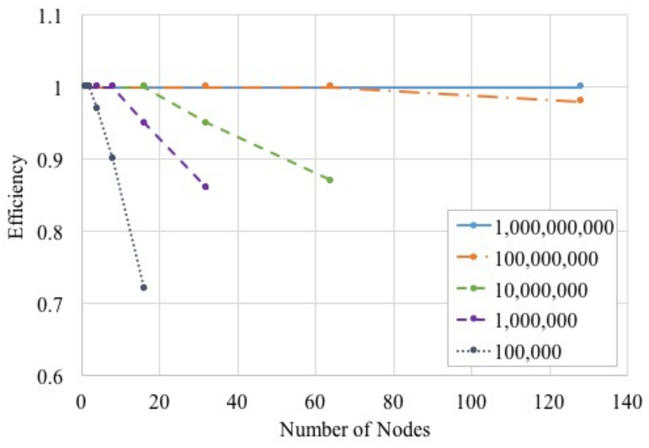


Figure 12: Weak scaling efficiency for Fig.11 using Eq.6. It shows that when the problem size is large enough, there will be enough work for all nodes, which hides the communication latencies behind useful work.

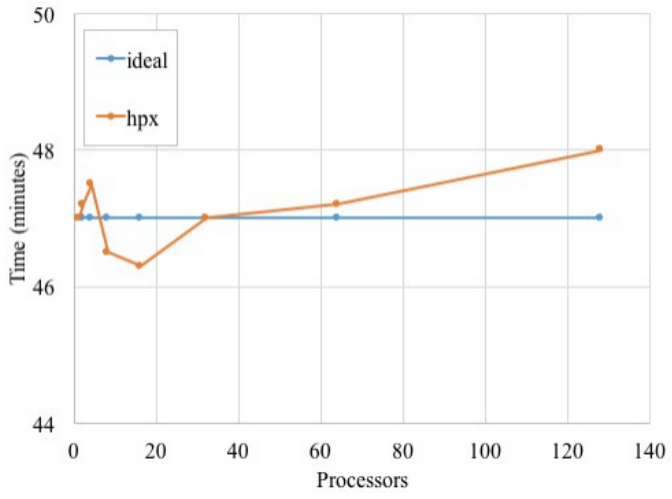


Figure 14: Weak scaling for the distributed N-Body application implemented with HPX on up to 128 nodes (20 cores per node), which shows the perfect overlap of computation with communication enabled by HPX.

distributed N-Body problem with HPX for up to 128 nodes is shown in Fig.14. The results illustrate that HPX avoids unnecessary communication overheads and enables seamless overlap of communication with communication.

V. Conclusions

In this paper, we presented the efficient performance of a parallel distributed N-Body application, in which HPX was used to implement both, the communication between different nodes and the parallel processing within each node. A *future*-based request buffer was used to decouple overall application progress between different nodes and along each spatial direction in order to send and receive data to/from a remote node. This enables progress without waiting for all nodes to complete the previous step, which effectively resulted in removing the global barrier after each time step. Moreover, particles were distributed across the nodes by considering the number of the nodes to determine the destination node of the particle, which also reduced the node starvation significantly. Furthermore, it was shown that the fine-grained workload used in HPX helped in having a better load balancing and lower overheads when compared to the hybrid model. We showed the excellent scalability of an N-Body with 128x speedup on 128 distributed nodes for 10^9 particles using HPX that indicates it has the potential to continue to scale on more even cores.

Acknowledgements

We would like to thank Bryce Adelstein-Lelbach from Lawrence Berkeley National Laboratory for the invaluable and helpful comments and suggestions to improve the quality of the paper. This works was supported by NSF awards 1447831 and CCF-111798.

References

[1] Robert Speck, Lukas Arnold, and Paul Gibbon. Towards a petascale tree code: Scaling and efficiency of the PEPC library. *Journal of Computational Science*, 2(2):138–143, 2011.

[2] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Scientific and Statistical Database Management*, pages 132–150. Springer, 2010.

[3] Kei Simon Pedersen and Brian Vinter. Java PastSet: A Structured Distributed Shared Memory System. In *Software, IEE Proceedings-*, volume 150, pages 147–153. IET, 2003.

[4] Robert Speck, Daniel Ruprecht, Rolf Krause, Matthew Emmett, Michael Minion, Mathias Winkel, and Paul Gibbon. A massively space-time parallel N-body solver. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 92. IEEE Computer Society Press, 2012.

[5] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, 1995.

[6] Rocco Aversa, Beniamino Di Martino, Nicola Mazzocca, and Salvatore Venticinque. Performance analysis of hybrid OpenMP/MPI n-body application. In *Shared Memory Parallel Programming with Open MP*, pages 12–18. Springer, 2004.

[7] Chirag Dekate, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, Bryce Adelstein-Lelbach, and Thomas Sterling. Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model. *International Journal of High Performance Computing Applications*, 2012.

[8] Mathias Winkel, Robert Speck, Helge Hübner, Lukas Arnold, Rolf Krause, and Paul Gibbon. A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale N-body simulations. *Computer physics communications*, 183(4):880–889, 2012.

[9] Alexander Shirokov and Edmund Bertschinger. GRACOS: Scalable and load balanced P3M cosmological N-body code. *arXiv preprint astro-ph/0505087*, 2005.

[10] Rolf Rabenseifner, Georg Hager, Gabriele Jost, and Rainer Keller. Hybrid MPI and OpenMP parallel programming. In *PVM/MPI*, page 11, 2006.

[11] Ashay Rane and Dan 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.

[12] LA Smith. Mixed mode MPI/OpenMP programming. *UK High-End Computing Technology Report*, pages 1–25, 2000.

[13] Patricia Grubel, Hartmut Kaiser, Jeanine Cook, and Adrian Serio. The Performance Implication of Task Size for Applications on the HPX Runtime System. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 682–689. IEEE, 2015.

[14] 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.

[15] J Mark Bull. Measuring synchronization and scheduling overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP*, volume 8, page 49, 1999.

[16] Olga Pearce, Todd Gamblin, Bronis R de Supinski, Tom Arsenlis, and Nancy M Amato. Load balancing n-body simulations with highly non-uniform density. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 113–122. ACM, 2014.

[17] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[18] Lorna Smith and Mark Bull. Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9(2-3):83–98, 2001.

[19] Hari Sundar, Rahul S Sampath, and George Biros. Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.

[20] Tiankai Tu, David R O'Hallaron, and Omar Ghattas. Scalable parallel octree meshing for terascale applications. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 4–4. IEEE, 2005.

[21] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing*, 36(1):C66–C93, 2014.

[22] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based fmm for heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, 2015.

[23] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.

[24] Michael S Warren and John K Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21. ACM, 1993.

[25] Hartmut Kaiser, Thomas Heller, Agustin Berge, and Bryce 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.