Assessing the Performance Impact of an Active Global Address Space: A case for AGAS

1st Parsa Amini
Louisiana State University
Baton Rouge, USA
parsa@cct.lsu.edu

2nd Hartmut Kaiser
Louisiana State University
Baton Rouge, USA
hkaiser@cct.lsu.edu

***Abstract*—In this research, we describe the functionality of AGAS, a subsystem of the HPX runtime system that is designed to handle data, independent of the hardware and architecture configuration. AGAS enables runtime global data access and data migration, but incurs a an overhead cost at runtime. We present a method to assess the performance of AGAS and the amount of impact it has on execution of the OctoTiger application. With our assessment method we identify three problematic spots in the HPX version used in our experiments. We also demonstrate that the overhead caused by AGAS is low.**

***Index Terms*—HPX, AGAS, PGAS**

I. Introduction

Global address space systems attempt to boost productivity and simplify the application development cycle of distributed applications by providing a ubiquitous abstraction layer over memory spaces that are provided and managed by the operating system on each node in a large-scale system. SPMD-style (Single Program Multiple Data) Partitioned Global Address Space designs eliminate this layer during compilation to avoid the complexity of resolving global addresses at runtime at the cost of limiting productivity and imposing limitations on the code, while others like HPX [1], [2], HPX5 [3], Charm++ [4], UPC++ [5], and Chapel [6] provide a runtime component that maps global addresses to virtual addresses during application execution to provide true global addresses.

The demand for models that enable applications to process massive datasets within specific time, power, and budgetary constraints continues to pose challenges for the computer science community [7], [8]. One category of such complexities includes managing a large quantity of objects across several machines and memory partitions while maximizing data locality. Several Partitioned Global Address Space systems (PGAS) [9] try to address these needs by providing control over data distribution and facilitating data accesses across processes and machines. However, initial data placement alone is not sufficient to address more complex issues such as load balancing applications that run on heterogeneous clusters or applications that become scaling impaired over time due to increasing load imbalance like adaptive mesh refinement (AMR), dynamic graph applications, and partial differential equation (PDE) solvers [10]–[12]. Active Global Address Space (AGAS) is a system that tries to address performance impaired applications. AGAS was initially proposed for the ParalleX programming model [10] and implemented in HPX. It adds an abstraction layer on top of local objects on each compute node by mapping local virtual addresses to a global address and ensuring that global addresses are valid even if the object it refers to is migrated to a different physical location. AGAS enables applications to perform load balancing at runtime by using data migration. AGAS also reduces data movement by using active messages. Active messages significantly reduce the need for data movement by moving tasks to where the data is instead of moving the data to where work is. However, AGAS needs to execute code to resolve and maintain the references and takes a portion of the execution time.

Assessing the overheads caused by AGAS is the main objective of this paper. We address this objective by developing a system that uses measurement data from AGAS and apply it to identify AGAS functions that exhibit poor scaling behavior in HPX 0.99. In the rest of this work, we discuss other pertinent research in section II, present a general overview of AGAS functionality in section III, explain the criteria based on which the results can be evaluated in section IV, experiments that were run and examine their results in section V, and further analyze them and consider directions that are likely to produce more insights into improving AGAS considering our findings in section VI.

II. Related Work

The MPI programming model provides the user with complete control over data locality and performance. On the other hand, it does not have the programmability and data referencing simplicity of shared-memory systems. The global address space model combines the two models and enables processes to access shared memory locations with a global address while maintaining an explicit distinction between local and remote operations. This provides the means to implement distributed versions of commonly used data structures like arrays, sets, matrices, or any data structure that is based on pointers.

Some global address space implementations like Unified Parallel C (UPC) [13], Co-Array Fortran, Titanium [14], and SHMEM [15], [16], the global addresses are resolved to communication calls during compilation. Because global address resolution is performed at compile time, these implementations still do not provide the freedom programmers have in a shared memory application. Additionally, they still use global barriers as the synchronization mechanism, which does not trivially achieve shared memory applications performance.

Asynchronous PGAS implementations (e.g. Charm++, UPC++, Chapel, and X10) follow the asynchronous many task model to dynamically perform load balancing at runtime. They allow multiple tasks to run within each operating system thread and provide tools for controlling the memory layout and expressing multidimensional, sparse, associative, or unstructured data structures.

In recent years, there has been a significant increase in utilization of heterogeneous clusters that use GPUs and MICs in addition to CPUs [17]–[20]. One approach to manage such systems is using solutions [18], [21], [22] that separately use a library like MPI for explicit communication between nodes and a choice of shared-memory programming framework such as OpenMP [23], Kokkos [24], [25], or UPC. Other approaches include Asynchronous PGAS runtimes [26] and Charm++ that use dynamic multithreading to avoid fragmenting the application development process to separately manage communication and computation while maintaining portability between various cluster configurations and providing access to heterogeneous computing resources [27].

Most studies on distributed runtime systems do not include quantitative analysis of the performance of their global address space system but present the overall performance of applications using the respective model or implementation. However, amongst the runtime systems mentioned only HPX has a global address system that allows objects to be relocated at runtime. This unique property calls for a closer look into HPX's Active Global Address Space system behavior and performance and is the motivation behind this study.

III. Active Global Address Space

This section provides an overview of the implementation of the Active Global Address Space (AGAS) in HPX runtime system.

AGAS is a global memory addressing system that is designed to handle various memory configurations ranging from those implemented in single small machines to those typically found in a cluster composed of a large number of nodes with heterogeneous computing resources. AGAS is designed such that specific memory configuration characteristics are handled by the programmer through the API. Fig. 1 illustrates AGAS's role in HPX. Typically, each part of the distributed AGAS service is hosted on a separate node of a cluster.

AGAS consists of several subsystems that are depicted in Fig. 4 and are the following:

1) Primary Namespace: To provide uniform access to objects across the boundaries of physical partitions in a cluster, AGAS provides applications with 128-bit global identifiers (GIDs) to be used in place of virtual addresses that are local to specific nodes. Consequently, AGAS maintains mapping tables to be able to map GIDs to local virtual addresses.
2) Locality Namespace: Information about the nodes and computing resources allocated to each physical partition is held in the locality namespace. Each partition is called a "locality" and locality 0 is responsible for maintaining current information about all other localities.
3) Component Namespace: Types are registered in this namespace to facilitate resolving resource requirements during bulk memory allocations.
4) Symbolic Namespace: The symbolic namespace is a layer on top of the global address space that allows mapping symbolic names to global addresses for the purpose of resolving global addresses at runtime. This is useful in cases where data about specific events needs to be collected. For example, HPX performance counter system uses the symbolic namespace for collecting performance counter data.
5) AGAS Cache: AGAS cache stores mapping between the most recently used global addresses to localities where those object reside and local virtual addresses. If a task requires an object that does not live on the same locality then the task is sent to the locality where the object currently is. However, in order to do so, AGAS has to determine the current location of the queried object. If AGAS does not know the current location then it forwards the query to the locality where the object was originally created. The locality on which an object is created stays responsible for maintaining the current location of the object during its entire lifetime. If an object is likely to be accessed again then the locality that forwards a task will also request the original locality to report the object's current location. This information is stored in the AGAS cache. The AGAS cache is small since it is designed for speed and therefore, it does not know all local objects.
6) Garbage Collection: A distributed garbage collection system tracks objects during their lifetime and frees the consumed memory when an object goes out of scope and therefore can no longer be accessed in the program. HPX complies to the C++ standard specifications and hence, it provides structures similar to what the modern C++ standard requires. Additionally, GIDs in HPX can be managed or unmanaged and in case of the former AGAS tracks that GID until the reference is lost so that it can free up the memory space when possible. AGAS uses reference counting to determine if there are existing references to an object and a credit-based scheme for remote references. The local counter is updated when a new reference is created and when a reference goes out of scope on the same locality. As for remote references, the credit system works as demonstrated in Fig. 3

Garbage collection at runtime requires execution of code that is otherwise not present and consumes computing resources. Performing garbage collection requires executing code that is the not the user's application. AGAS tries to minimize garbage collection sweeps by performing it when

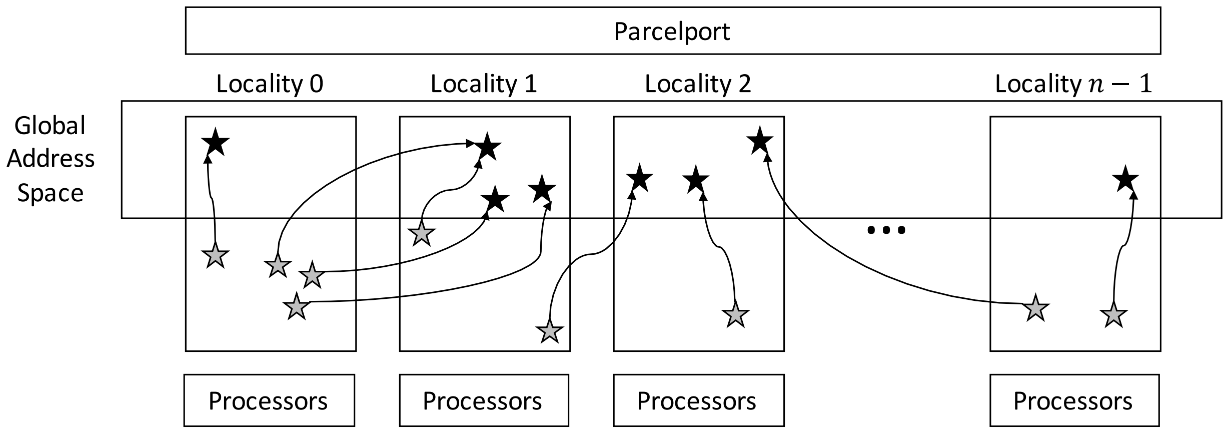


Fig. 1: AGAS provides an abstraction layer on top of virtual addresses local to each locality. Global objects are shown as black stars and gray stars indicate references to global objects. Each reference is connected to global objects by an arrow.

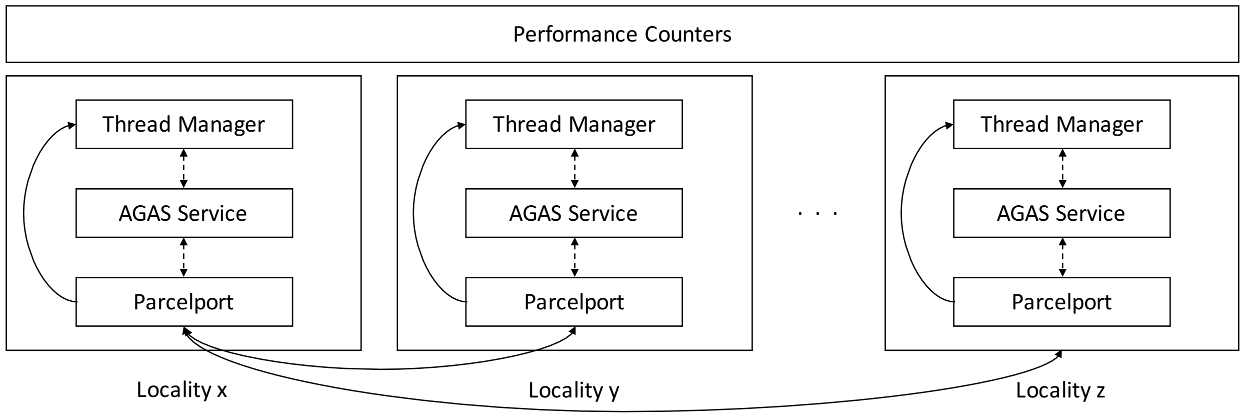


Fig. 2: When an HPX thread accesses a global object, AGAS determines if the object can be accessed locally. If the object is on a different locality the HPX thread is serialized and given to the parcelport. The parcelport unserializes the task and hands it to the thread manager for execution.

the volume of garbage reaches a certain threshold that can be specified by the users. It is also possible to manually initiate it inside applications by developers.

When the code accesses the object referred to by a GID, AGAS looks up the GID in the primary namespace and returns the local virtual address for the object if the object lives on the same locality. As for remote objects, AGAS interacts with the parcelport service to resolve the remote reference access as shown in Fig. 2. This design hides the communication latencies by resolving the remote reference accesses asynchronously.

A. *AGAS Performance Counters*

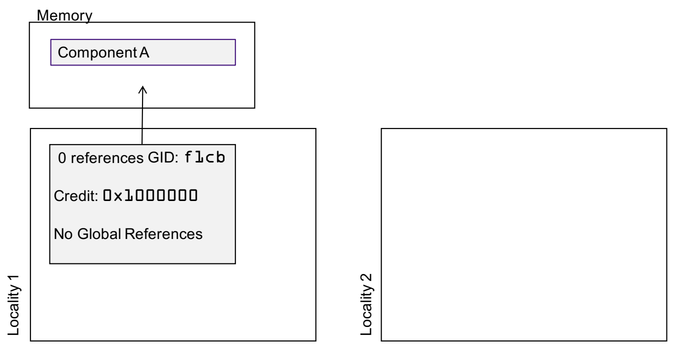
HPX is a runtime system that includes novel abstractions on top of ordinary operating systems and hardware that are more difficult to benchmark using traditional performance measuring tools such as hardware performance counters included in Intel processors. HPX introduces a set of performance counters to let developers monitor the performance of its subsystems, including AGAS, during execution. This allows the users to use performance counters to debug their code and locate performance bottlenecks. Users can also develop their own performance counters to retrieve arbitrary information during execution.

Performance counters can be queried at runtime. For example, APEX [28] uses data provided by the performance counter system at runtime to make decisions in its policy engine to perform autotuning on the running HPX application and improve execution time. It is also possible to ask HPX to print the performance counter data.

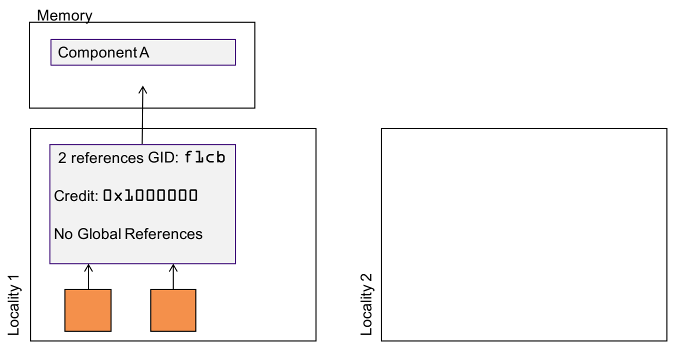
Similar to hardware performance counters, HPX performance counters [29] are designed to expose performance data on the underlying function calls. HPX has performance counters that measure performance of AGAS subsystems. Each AGAS performance counter either reports the number of invocations or the total execution time of the selected operation.

B. *Migration*

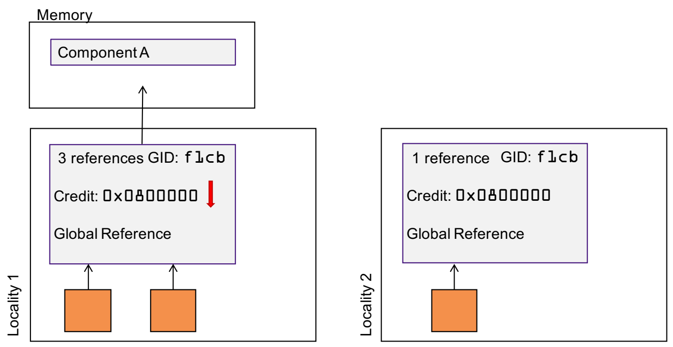
Objects registered in AGAS can be physically moved to a different locality while retaining the same address. Moreover,



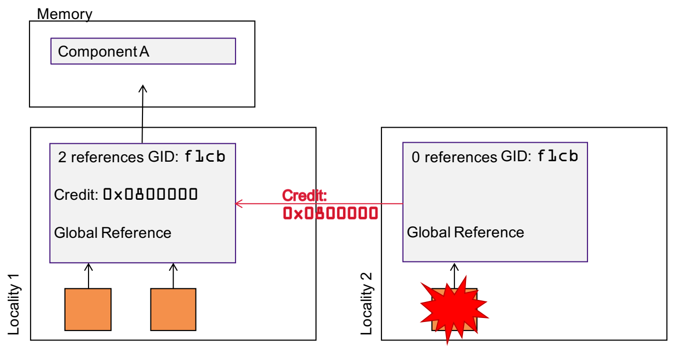
(a) An object is created. AGAS sets the object's credit to a large number.



(b) A global object referenced by two local copies of the GID. Credit is not affected by local copies.



(c) When an object is referenced by another locality the object's credit is split in half and a copy of the reference is kept at both localities, and a flag is set on the original reference to indicate that the object is referenced globally. When a copy runs out of credit, it will ask the lender AGAS instance (the locality where the reference is located) for more. If the original reference itself does not have enough credit, the AGAS instance responsible for that reference (where the object resides) will grant it more credit.



(d) When a reference goes out of scope AGAS returns all borrowed credits to the original reference. If there are no local references and all credits have been returned then AGAS can remove the object from memory during garbage collection.

Fig. 3: AGAS credit system tracks global references. When a global reference goes out of scope all of its credit is returned to the lender. When there are no local or global references the memory allocated by the object can be freed by the garbage collector.

this operation does not need the application execution to be suspended. After a global object is relocated to the new locality, all reference accesses that try to access the object are forwarded to the new locality and their localities are notified of the move.

Migration is an especially useful feature for applications that suffer from poor data locality and/or are balance impaired and it can be used to adaptively improve data locality when scaling is being hurt.

C. HPX bootstrap and teardown

Before an HPX application can start executing, HPX has to initialize. This process is called bootstrap and includes registering runtime services, data types, performance counters, and symbols. Similarly during teardown, after applications built with HPX complete, HPX removes all objects, frees all allocated hardware resources and may perform additional tasks such as collecting performance counter data when applicable.

In this research, we exclude data from the bootstrap and teardown phases since they do not directly provide useful data on how AGAS might impact the performance or scaling behavior during runtime.

IV. Quantifying AGAS Performance

The main challenge of any HPC global addressing system, including AGAS, is to efficiently support the massive amounts of data that applications use during execution. Our aim here, therefore, is to understand the efficiency of AGAS through measuring its performance of associated overheads. To study AGAS, we use OctoTiger [30]–[32] as our application and run it on the cluster provided by the Center of Computation and Technology at Louisiana State University. It has to be noted that any other application benchmark can be used in place of OctoTiger. However, developing an HPX application takes work and at this point, OctoTiger is the only available real world HPX application. In the rest of this section we present

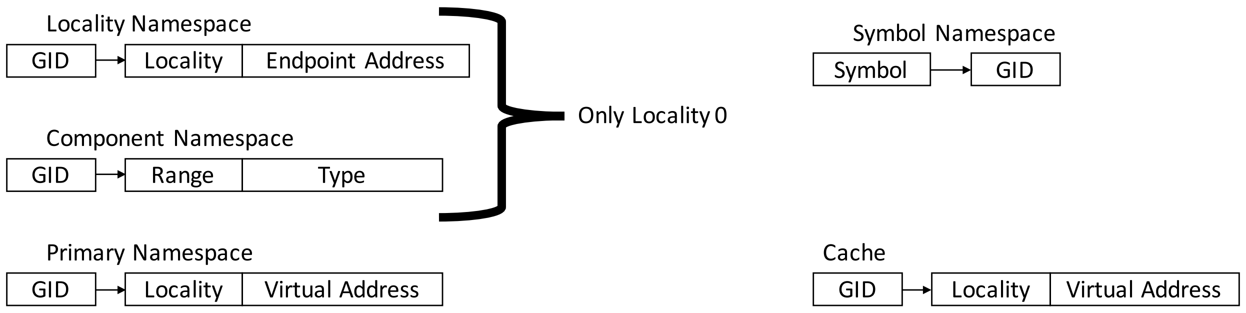


Fig. 4: The four namespaces inside AGAS. Primary namespace on each AGAS instance contains GID to local address mappings. Locality namespace holds information about all AGAS instances. Component namespace tracks bulk memory allocations dedicated to types. Symbolic namespace contains mappings between GIDs and special strings that can be used for various purposes such as facilitating the collection of data about application execution.

OctoTiger and the performance metrics we use to measure the overheads of AGAS.

A. *OctoTiger*

LSU OctoTiger is a state of the art multi-physics AMR HPX application that simulates the merger of double white dwarf binaries. Its computations include hydrodynamics, gravitational, and radiation transportation solvers. It fully utilizes the capabilities of AGAS in HPX since it dynamically changes the computational resolution based on the actual needs of the simulation and thus, it is an inherently balance impaired application. It is also a memory intensive application compared to most exascale challenge problems. For example, Quicksilver and Pennant from CORAL2 benchmarks have a high water mark of 16% and 8%, respectively, whereas for OctoTiger this number is about 60%.

We use OctoTiger to study the behavior of AGAS when used by applications running on large machines. We ran our experiments on SuperMIC at Louisiana State University, a hybrid cluster composed of Xeon, Xeon Phi Knights Corner, and NVIDIA Tesla processors. Due to the complexity of the computations performed by OctoTiger, adjusting the size of the problem with reasonable accuracy to demonstrate weak scaling is not practical and therefore we presenting the impacts of strong scaling on AGAS behavior.

B. *System and Environment Setup*

We run our experiments on SuperMIC, a hybrid Xeon/Xeon Phi cluster that comprises 360 compute nodes located at Louisiana State University. More details about the configuration of SuperMIC is included in Table I. We run OctoTiger on SuperMIC from two nodes to 194 nodes only using the Xeon processors on each machine using HPX 0.99 [33]. The software configuration is listed in Table II

- Performance Metrics: Every globally accessible object in HPX has a global ID that AGAS manages and resolves to local virtual addresses at runtime. However, address resolution at runtime is an overhead that consumes computational resources that application developers expect to be used by the applications rather than the runtime system. To quantify and study the overhead of AGAS, we look at the following fundamental operations, the number of calls, and the amount of time that is spent performing these operations.

TABLE I: SuperMIC Configuration

Nodes available	360
Architecture	Ivy Bridge, Knights Corner
Cores/Node	2×10 cores
Processor	Intel Xeon E5-2680 @ 2.8 GHz, Intel Xeon Phi 7120P @1.238 GHz
Memory	64 GB DDR3 @ 1,866 MHz
Connection	56 Gbps Infinitband
Operating System	Red Hat Enterprise Linux 6.8

TABLE II: Software used in the experiment

Compiler	GCC 4.9.0
MPI	MVAPICH 2.0
Boost	1.61.0
hwloc	1.10
HPX	0.99

- Bind, Unbind, Object Lookup: Bind and Unbind operations occur when a global object is created and deleted, respectively. An object lookup operation takes place each time AGAS attempts to resolve a global ID.
- Locality Lookup: Each AGAS instance only knows about itself and locality 0. When an AGAS instance needs to communicate with another locality and does not have information about the appropriate communication endpoint to do so, it needs to query that information from locality 0.
- Parcel Routing: HPX implements active messages in the form of parcels [34]. A parcel is packaged information that triggers an operation upon reception by an AGAS instance. For example, when an HPX task needs to operate on information that resides on another locality, AGAS serializes the task along with its arguments and state information and forwards it to the locality on which

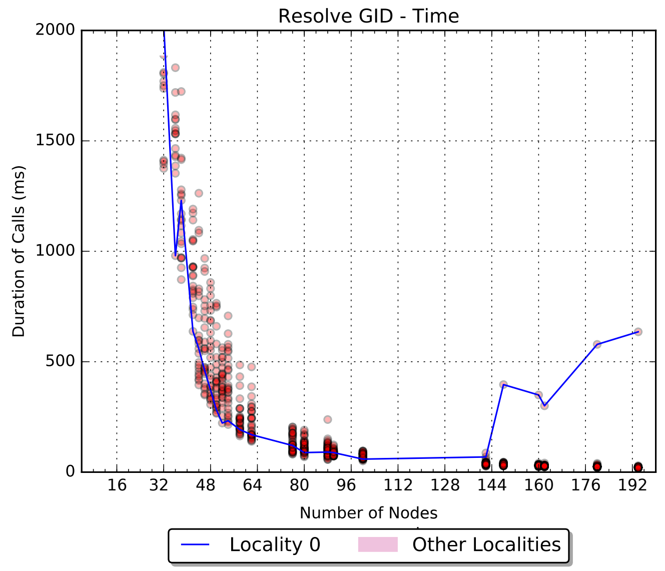


Fig. 5: Total time spent to resolve GID calls while running OctoTiger on 32 to 194 nodes of SuperMIC with fixed problem size (Strong Scaling). Each (red) circle refers to the measured amount of time a locality has spent executing resolve GID calls. Higher intensities indicate overlapping measurements. The line highlights locality 0's behavior. X axis starts from 2 to keep consistency with the X axis on other plots.

data resides.

- Cache: Whenever AGAS decides that an address is likely to be queried again, it stores that information in the local AGAS cache of that locality. To serve its purpose of actually improving performance the AGAS cache is designed to hold a limited number of entries that is defined by user configuration.
- Garbage Collection: HPX provides managed objects whose lifetime is controlled by the garbage collection mechanism instead of the programmer having to free the memory consumed by each object. HPX uses reference counting to track references local to each AGAS instance and a credit scheme to manage global references. Once an object no longer has any local references and/or all its global reference credits are returned, it is deemed garbage and is collected when the garbage collection operation is triggered by reaching a certain threshold or by the application developer in their code. Registration and removal of global references is done by calling *increment_credit* and *decrement_credit* functions, respectively.

V. Performance Results

As mentioned before, the main objective of this study is to identify the AGAS operations that need performance improvement and to study the effects of strong scaling on AGAS. In this section, we present the results of this study.

Appropriate distribution of work is of particular interest to computer scientists. In case of AGAS, ideally we expect the number of specific operations and the amount of time spent to perform them to be similar. Address resolution is one of the main functions of AGAS that translates a HPX GID to the actual virtual address on a machine. Its operation is complicated if the object is not currently located on the locality that address resolution is taking place and in such case AGAS tries to determine which locality the object currently lives on, serialize the task that asked to access the foreign GID in an HPX parcel, and send it to the locality the object is living in. If an AGAS instance has no knowledge of the locality that is holding an object then it takes advantage of the fact that a locality on which an object is created stays responsible for it during the object's entire lifetime and uses the metadata in the GID to determine the locality on which the object was originally created and sends the query there. Fig. 5 shows the amount of time GID resolution takes while running OctoTiger. By looking at the graph we notice that while the amount of work performed by each locality is relatively similar up to 146 nodes, locality 0 starts to perform more work than other localities. This may indicate that most objects globally referenced were created on locality 0 and it requires further investigation to determine if this effect is caused by application behavior, non-optimal data distribution, or HPX itself.

Parcels are the basic communication block in HPX. Parcels are active messages that trigger an operation on the target AGAS instance that opens them and usually contains a serialized task and its arguments. Parcel routing is the operation in which a parcel is sent to a different locality. We expect that localities exchange a similar amount of parcels and thus, spend a similar amount of time. Fig. 6 shows that locality 0 ends up handling a disproportionate and increasing number of routing requests regardless of the change in data access pattern and size due to scaling. Therefore, the routing operation is a candidate for optimization.

For a garbage collection system to function the HPX runtime system must be able to determine if an object is currently being used. HPX uses reference counting for local references and a credit-based system to keep track of global references. The operations that lend and return credits are called decrement credit and increment credit, respectively.

Fig 7 depicts the behavior of *decrement_credit* calls that take place while running OctoTiger with the same number of objects as the number of nodes are increased. This figure shows that a linear increase in number of objects results in an exponential growth in the amount of time spent on managing global references. Although this behavior is likely to be different for applications that do not have as many global references per each object and therefore *decrement_credit* is a good candidate for further optimizations.

Table III shows the percentage of execution time that is spent on maintaining AGAS and the percentage of execution time that is spent on the three most expensive AGAS operations. It is worth mentioning that we did not plan to collect execution times for all runs and incidentally collected these values, but we nevertheless expect the values to be similar for other node configurations that are not in the table. Based on these values we observe that AGAS takes only a small amount

TABLE III: AGAS overhead in total and for the most expensive AGAS operations as percentages of execution time for 4 node configurations.

Number of localities	All AGAS operations in total	Decrement credit operations	Resolve GID operations	Parcel routing operations
2	0.0031	0.0016	0.0009	0.0001
16	0.0031	0.0015	0.0011	0.0001
88	0.0032	0.0015	0.0010	0.0003
194	0.0051	0.0017	0.0011	0.0020

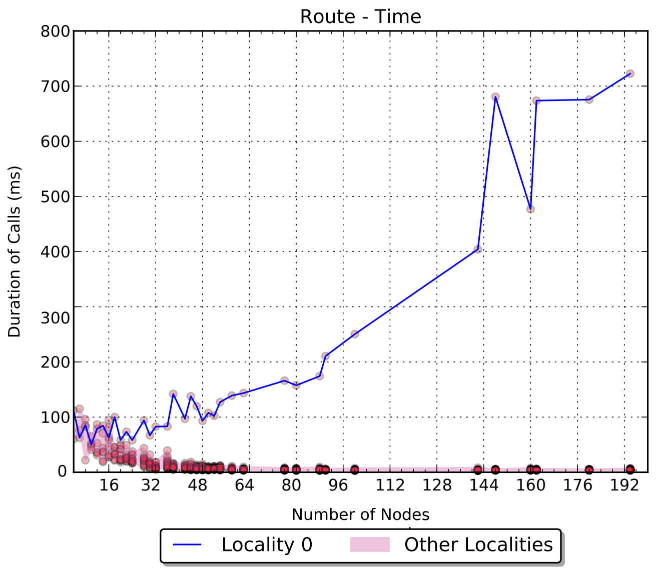


Fig. 6: Total time spent to route calls while running OctoTiger on 2 to 194 nodes of SuperMIC with fixed problem size (Strong Scaling). Each (red) circle refers to the measured amount of time a locality has spent executing parcel route calls. Higher intensities indicate overlapping measurements. The line highlights locality 0's behavior.

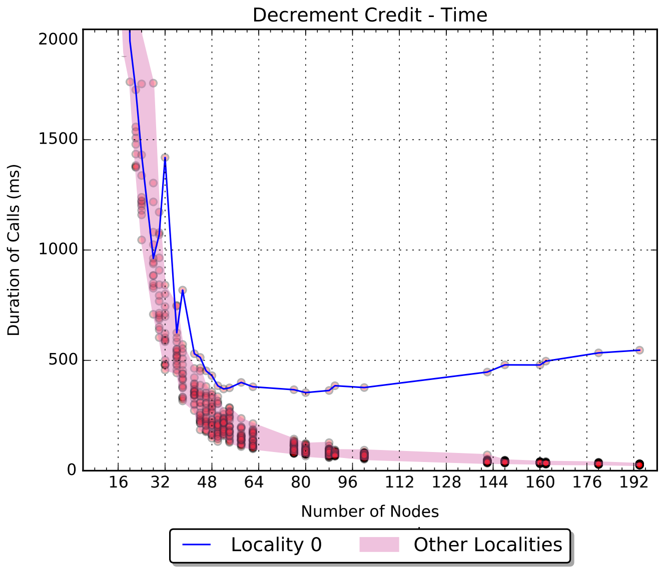


Fig. 7: Total time spent to decrement credit calls while running OctoTiger on 2 to 194 nodes of SuperMIC with fixed problem size (Strong Scaling). Each (red) circle refers to the measured amount of time a locality has spent executing resolve GID calls. Higher intensities indicate overlapping measurements. The line highlights locality 0's behavior.

of execution time.

VI. Conclusions and Implications for Future Work

In this work we introduce AGAS, its subsystems, and a method to study how the amount of time that is spent executing AGAS code. This is an overhead that does not exist in PGAS systems that statically resolve global references during compilation. We observe how AGAS's most expensive operations are affected as the number of nodes increase.

To study AGAS's behavior we chose a multiphysics AMR application called OctoTiger that generates and works with a significant number of objects. We identify the performance metrics that expose AGAS's performance and use the corresponding counters in HPX's Performance Counter framework to collect performance data from our strong scaling experiments.

Our observations show that in the cases of the three most expensive AGAS operations, resolve GID, route, and decrement credit, the AGAS instance on locality 0 performs an increasing amount work as the problem is strongly scaled. Among the three, parcel routing, the operation that is affected the most, takes 0.0016% of execution time that is spent on routing parcels by AGAS-invoked tasks on locality 0 on 194 nodes.

Using our methods we were able to identify the issues that pose possible performance bottlenecks that will be investigated and reported in future work. Inclusion of a copy of the locality namespace to each locality's AGAS cache in the current version of HPX is one such optimization that reduces traffic forwarding to locality 0 due to not knowing the endpoint address for a locality.

The most important feature of AGAS is that it allows applications to dynamically perform data load balancing at runtime without stopping the execution and it also works with other components of HPX to deliver a distributed asynchronous multithreaded system. We observe that in total AGAS operations cause an average of 0.0036% of overhead. This is hardly a performance issue and indicates that using AGAS is a relatively inexpensive choice for the boost in productivity it provides.

ACKNOWLEDGMENT

This work was partly funded by the STAR project under the NSF award 1240655 Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

APPENDIX A
ARTIFACT DESCRIPTION: ASSESSING THE PERFORMANCE IMPACT OF AN ACTIVE GLOBAL ADDRESS SPACE: A CASE FOR AGAS

A. Abstract

In this section we describe the configuration and environment needed to run our experiments. The experiments are run on SuperMIC [35] from 2 to 194 nodes. We present the steps to build the software on SuperMIC and repeat our experiments and how to gather the performance counter data.

B. Description

1) Check-list (artifact meta information):

- **Program: OctoTiger**
- **Compilation: GCC 6.0**
- **Hardware: SuperMIC**
- **Run-time state:**
- **Output: Performance counter results in text**
- **Publicly available?: Yes**

2) How software can be obtained (if available): HPX can be obtained from https://github.com/STEllAR-GROUP/hpx. OctoTiger can be downloaded from https://github.com/STEllAR-GROUP/octotiger.

3) Hardware dependencies: Access to SuperMIC can be requested through XSEDE. See https://portal.xsede.org/allocations/announcements for more information.

4) Software dependencies: The following modules need to be loaded to build HPX and OctoTiger:

- gcc/4.9.0
- mvapich2/2.0/INTEL-14.0.2
- hwloc/1.10.0/INTEL-14.0.2

The following software need to be built from their source since the versions available on SuperMIC are too old:

- CMake. The latest version of CMake can be downloaded from https://cmake.org/download/.
- Boost. It is available on https://sourceforge.net/projects/boost/files/boost/1.61.0/.
- Silo

The restart file X.1260.chk needs to be downloaded from http://www.nic.uoregon.edu/~khuck/octotiger-restart-files/X.1260.chk

C. Installation

- HPX

```
cmake
  –H<PATH_TO_HPX_CODE>
  –B<PATH_TO_HPX_BUILD>
  –DCMAKE_BUILD_TYPE=Release
  –DCMAKE_CXX_FLAGS="–std=c++1y"
  –DHPX_WITH_CXX11=True
  –DCMAKE_C_COMPILER=gcc
  –DCMAKE_CXX_COMPILER=g++
  –DBOOST_ROOT=<PATH_TO_BOOST>
  –DBoost_NO_SYSTEM_PATHS=True
  –DHPX_WITH_MALLOC="custom"
  –DHPX_WITH_PARCELPORT_MPI=True
  –DHPX_WITH_PARCELPORT_TCP=False
  –DHPX_WITH_PARCELPORT_IBVERBS=False
  –DHPX_WITH_EXAMPLES=False
cmake ––build <PATH_TO_HPX_BUILD>
```

- OctoTiger

```
cmake
  –H<PATH_TO_HPX_CODE>
  –B<PATH_TO_HPX_BUILD>
  –DHPX_DIR=<PATH_TO_HPX_BUILD/lib/cmake>
cmake ––build <PATH_TO_OCTOTIGER_BUILD>
```

D. Experiment workflow

- Get a job from Torque for the appropriate number of nodes and ensure all required modules listed in section A-B4 are loaded.
- Run the OctoTiger binary with mpirun the following commands:

```
cat $PBS_NODEFILE |
  awk 'NR % 20 == 0' > node.list
export HPX_NODEFILE=node.list
export NPROCS=$(wc -l
  $HPX_NODEFILE |gawk '//{print $1}')
mpirun
  -np $NPROCS
  --machinefile $HPX_NODEFILE
  ./octotiger
  -t20
  -Ngrids=100000
  -Xscale=8.031372549
  -Problem=dwd
  -Max_level=11
  -VariableOmega=0
  -Restart=restart.chk
  10
  --hpx:threads 20
  -Datadir=<DATA_LOCATION>
  -ParallelSilo
  --hpx:print-counter=
    /agas{locality#*/total}/count/
      allocate
  --hpx:print-counter=
    /agas{locality#*/total}/count/
      bind
  --hpx:print-counter=
    /agas{locality#*/total}/count/
      bind_gid
```

```
--hpx:print-counter=
  /agas{locality#*/total}/count/
    cache-evictions
--hpx:print-counter=
  /agas{locality#*/total}/count/
    cache-hits
--hpx:print-counter=
  /agas{locality#*/total}/count/
    cache-insertions
--hpx:print-counter=
  /agas{locality#*/total}/count/
    cache-misses
--hpx:print-counter=
  /agas{locality#*/total}/count/
    cache_erase_entry
--hpx:print-counter=
  /agas{locality#*/total}/count/
    cache_get_entry
--hpx:print-counter=
  /agas{locality#*/total}/count/
    cache_insert_entry
--hpx:print-counter=
  /agas{locality#*/total}/count/
    cache_update_entry
--hpx:print-counter=
  /agas{locality#*/total}/count/
    decrement_credit
--hpx:print-counter=
  /agas{locality#*/total}/count/
    increment_credit
--hpx:print-counter=
  /agas{locality#*/total}/count/
    resolve
--hpx:print-counter=
  /agas{locality#*/total}/count/
    resolve_gid
--hpx:print-counter=
  /agas{locality#*/total}/count/
    route
--hpx:print-counter=
  /agas{locality#*/total}/count/
    unbind
--hpx:print-counter=
  /agas{locality#*/total}/count/
    unbind_gid
--hpx:print-counter=
  /agas{locality#*/total}/primary/
    count
--hpx:print-counter=
  /agas{locality#*/total}/primary/
    time
--hpx:print-counter=
  /agas{locality#*/total}/symbol/
    count
--hpx:print-counter=
  /agas{locality#*/total}/symbol/
    time
--hpx:print-counter=
  /agas{locality#*/total}/time/
    allocate
--hpx:print-counter=
  /agas{locality#*/total}/time/
    bind
--hpx:print-counter=
  /agas{locality#*/total}/time/
    bind_gid
--hpx:print-counter=
  /agas{locality#*/total}/time/
    cache_erase_entry
--hpx:print-counter=
  /agas{locality#*/total}/time/
    cache_get_entry
--hpx:print-counter=
  /agas{locality#*/total}/time/
    cache_insert_entry
--hpx:print-counter=
  /agas{locality#*/total}/time/
    cache_update_entry
--hpx:print-counter=
  /agas{locality#*/total}/time/
    decrement_credit
--hpx:print-counter=
  /agas{locality#*/total}/time/
    increment_credit
--hpx:print-counter=
  /agas{locality#*/total}/time/
    resolve
--hpx:print-counter=
  /agas{locality#*/total}/time/
    resolve_gid
--hpx:print-counter=
  /agas{locality#*/total}/time/
    route
--hpx:print-counter=
  /agas{locality#*/total}/time/
    unbind
--hpx:print-counter=
  /agas{locality#*/total}/time/
    unbind_gid
--hpx:print-counter=
  /agas{locality#0/total}/count/
    bind_name
--hpx:print-counter=
  /agas{locality#0/total}/count/
    bind_prefix
--hpx:print-counter=
  /agas{locality#0/total}/component/
    count
--hpx:print-counter=
  /agas{locality#0/total}/component/
    time
--hpx:print-counter=
    /agas{locality#0/total}/count/
      free
  --hpx:print-counter=
    /agas{locality#0/total}/count/
      localities
  --hpx:print-counter=
    /agas{locality#0/total}/count/
      num_localities
  --hpx:print-counter=
    /agas{locality#0/total}/count/
      num_localities_type
  --hpx:print-counter=
    /agas{locality#0/total}/count/
      num_threads
  --hpx:print-counter=
    /agas{locality#0/total}/count/
      resolve_id
  --hpx:print-counter=
    /agas{locality#0/total}/count/
      resolve_locality
  --hpx:print-counter=
    /agas{locality#0/total}/count/
      resolved_localities
  --hpx:print-counter=
    /agas{locality#0/total}/count/
      unbind_name
```

- Repeat the same experiment for 2 to 194 nodes.

E. Evaluation and expected result

The performance counters that contain "/count/" indicate the number of times an AGAS operation is invoked or the number of indicated object. The performance counters that contain "/time/" on the other hand indicate the total amount of time spent executing that AGAS operation. More information about performance counters can be viewed at https://stellar-group.github.io/hpx/docs/html/hpx/manual/performance_counters.html.

References

[1] 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*. Eugene, OR, USA: ACM, 2014, p. 6. [Online]. Available: http://stellar.cct.lsu.edu/pubs/pgas14.pdfhttp://doi.acm.org/10.1145/2676870.2676883

[2] "HPX on Github," https://github.com/STEllAR-GROUP/hpx.

[3] "HPX-5," http://hpx.crest.iu.edu/, CREST at Indiana University.

[4] L. V. Kale and S. Krishnan, "CHARM++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications - OOPSLA '93*, ser. OOPSLA '93, vol. 28, no. 10. New York, New York, USA: ACM Press, oct 1993, pp. 91–108. [Online]. Available: http://portal.acm.org/citation.cfm?doid=165854.165874

[5] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS Extension for C++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS 14. Phoenix, AZ, USA: IEEE, may 2014, pp. 1105–1114. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6877339

[6] "Cascade High Productivity Language - Cray," http://chapel.cray.com/.

[7] V. Sarkar, W. Harrod, and A. E. Snavely, "Software challenges in extreme scale systems," *Journal of Physics: Conference Series*, vol. 180, p. 012045, jul 2009. [Online]. Available: http://stacks.iop.org/1742-6596/180/i=1/a=012045?key=crossref.f67cd168995105550454fe5e8f7d8cf2

[8] T. Sterling and D. Stark, "A High-Performance Computing Forecast: Partly Cloudy," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 42–49, jul 2009. [Online]. Available: http://ieeexplore.ieee.org/document/5076318/

[9] "PGAS - Partitioned Global Address Space Languages," http://www.pgas.org.

[10] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications," in *2009 International Conference on Parallel Processing Workshops*, ser. ICPPW, L. F. Barolli and W.-c. V. T. Feng, Eds. Vienna, Austria: IEEE, sep 2009, pp. 394–401. [Online]. Available: http://stellar.cct.lsu.edu/pubs/icpp09.pdfhttp://ieeexplore.ieee.org/document/5364511/

[11] M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. Sterling, "Adaptive Mesh Refinement for Astrophysics Applications with ParalleX," oct 2011. [Online]. Available: http://arxiv.org/abs/1110.1131

[12] C. Dekate, "Extreme Scale Parallel N-Body Algorithm with Event-Driven Constraint-Based Execution Model," Ph.D. dissertation, Louisiana State University, Baton Rouge, LA, USA, 2011.

[13] "Unified Parallel C," https://upc-lang.org.

[14] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella *et al.*, "Titanium: a high-performance java dialect," *Concurrency and Computation: Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[15] K. Feind, "Shared memory access (shmem) routines," *Cray Research*, 1995.

[16] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, p. 2.

[17] O. Lena, "GPI2 for GPUs: A PGAS framework for efficient communication in hybrid clusters," *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, vol. 25, p. 461, 2014.

[18] C.-T. Yang, C.-L. Huang, and C.-F. Lin, "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters," *Computer Physics Communications*, vol. 182, no. 1, pp. 266–269, jan 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0010465510002262

[19] S. Potluri, "Enabling Efficient Use of MPI and PGAS Programming Models ion Heterogeneous Clusters with High Performance Interconnects," Ph.D. dissertation, Ohio State University, 2014. [Online]. Available: https://etd.ohiolink.edu/pg{_}10?0::NO:10:P10{_}ACCESSION{_}NUM:osu1397797221

[20] A. Sidelnik, B. L. Chamberlain, M. J. Garzaran, and D. Padua, "Using the High Productivity Language Chapel to Target GPGPU Architectures," Tech. Rep., 2011. [Online]. Available: http://hdl.handle.net/2142/18874

[21] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, ser. PDND '09. Weimar, Germany: IEEE, feb 2009, pp. 427–436. [Online]. Available: http://ieeexplore.ieee.org/document/4912964/

[22] M. J. Chorley and D. W. Walker, "Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters," *Journal of Computational Science*, vol. 1, no. 3, pp. 168–174, aug 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1877750310000396

[23] "The OpenMP API specification for parallel programming," http://www.openmp.org/specifications/.

[24] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[25] "Kokkos on Github," https://github.com/kokkos/kokkos.

[26] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu, "The Asynchronous Partitioned Global Address Space Model," in *The First Workshop on Advances in Message Passing*, 2010, pp. 1–8.

[27] M. Wong, H. Kaiser, and T. Heller, "Towards Massive Parallelism (aka Heterogeneous Devices/Accelerator/GPGPU) support in C++ with

HPX," Tech. Rep., 2015. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0234r0.pdf

[28] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An Autonomic Performance Environment for Exascale," *Supercomputing Frontiers and Innovations*, vol. 2, no. 3, pp. 49–66, jul 2015. [Online]. Available: http://superfri.org/superfri/article/view/64http://dx.doi.org/10.14529/jsfi150305

[29] 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.

[30] K. Kadam, G. C. Clayton, P. M. Motl, D. Marcello, and J. Frank, "Numerical simulations of close and contact binary systems having bipolytropic equation of state," in *American Astronomical Society Meeting Abstracts*, vol. 229, 2017.

[31] T. Heller, B. Lelbach, K. Huck, J. Biddiscombe, P. Grubel, A. Koniges, M. Kretz, D. Marcello, D. Pfander, A. Serio *et al.*, "Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars," Technical Report, Tech. Rep., 2017.

[32] "OctoTiger on Github," https://github.com/STEllAR-GROUP/octotiger.

[33] H. Kaiser, B. Adelstein-Lelbach, T. Heller, A. Berg, A. Bikineev, G. Mercer, J. Habraken, M. Anderson, A. Serio, J. Biddiscombe, M. Stumpf, A. Schfer, S. R. Brandt, D. Bourgeois, P. Grubel, V. Amatya, K. Huck, D. Bacharwar, L. Viklund, S. Yang, E. Schnetter, Bcorde5, M. Brodowicz, Z. Byerly, bghimire, P. Jahkola, C. Bross, andreasbuhr, C. Guo, and atrantan, "hpx: HPX V0.9.11: A general purpose C++ runtime system for parallel and distributed applications of any scale," Nov. 2015. [Online]. Available: https://doi.org/10.5281/zenodo.33656

[34] B. Wagle, S. Kellar, A. Serio, and H. Kaiser, "Methodology for adaptive active message coalescing in task based runtime systems," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 1133–1140.

[35] "SuperMIC," http://www.hpc.lsu.edu/docs/guides.php?system=SuperMIC.