Assessing the Performance Impact of using an Active Global Address Space in HPX: A Case for AGAS

Parsa Amini*
CCT, Louisiana State University
parsa@cct.lsu.edu

Hartmut Kaiser*
CCT, Louisiana State University
hkaiser@cct.lsu.edu

ABSTRACT

In this research, we describe the functionality of AGAS (Active Global Address Space), a subsystem of the HPX runtime system that is designed to handle data locality at runtime, independent of the hardware and architecture configuration. AGAS enables transparent runtime global data access and data migration, but incurs an overhead cost at runtime. We present a method to assess the performance of AGAS and the amount of impact it has on the execution time of the Octo-Tiger application. With our assessment method we empirically identify the most expensive AGAS operations in HPX and demonstrate that the overhead caused by AGAS is negligible.

KEYWORDS

HPX, AGAS, PGAS

ACM Reference Format:
Parsa Amini and Hartmut Kaiser. 2019. Assessing the Performance Impact of using an Active Global Address Space in HPX: A Case for AGAS. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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[2, 20], Charm++[21], UPC++[33], Chapel[1], and Regent[28] 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 [26, 29]. One category of these challenges is managing the large quantity of objects across several machines and memory partitions while maximizing data locality. Several Partitioned Global Address Space systems (PGAS) [5] 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 applications such as adaptive mesh refinement (AMR), dynamic graph applications, and partial differential equation (PDE) solvers[8, 12, 19] that become scaling impaired over time due to increasing load imbalances. Active Global Address Space (AGAS) is a system that tries to address these performance-impaired applications. AGAS was initially proposed for the ParalleX programming model[19] 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 located instead of moving the data to where work is executed. However, AGAS introduces overhead as it needs to execute code to resolve and maintain the references.

Assessing the overheads incurred by AGAS is the main objective of this paper. We address this objective by developing a method that uses measurement data from AGAS and apply it to quantify the performance of AGAS functions that demonstrate their scaling behavior. In the rest of this work, we discuss other pertinent research in section 2, present a general overview of AGAS functionality in section 3, explain the criteria based on which the results can be evaluated in section 4, experiments that were run and examine their results in section 5, and further analyze them and consider directions that are likely to produce more insights into improving AGAS considering our findings in section 6.

2 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.

*The STE||AR Group, stellar-group.org

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Some global address space implementations like Unified Parallel C (UPC)[7], Fortran (Coarrays), and SHMEM[9, 14], 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[22, 23, 27, 32]. One approach to manage such systems is using solutions[10, 24, 32] that separately use a library like MPI for explicit communication between nodes and a choice of shared-memory programming framework such as OpenMP[6], Kokkos[3, 13], or UPC. Other approaches include Asynchronous PGAS runtimes[25] 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[31].

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 for objects of arbitrary type 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.

3 THE ACTIVE GLOBAL ADDRESS SPACE

This section provides an overview of the implementation of the Active Global Address Space (AGAS) in the HPX runtime system as well as a description of key features of the utility and the HPX tools we used to to gather the performance information presented in this work.

AGAS is a global memory addressing system that is designed to handle various memory configurations ranging from ones implemented in single small machines to what is typically found in a cluster composed of a large number of nodes with heterogeneous computing resources. AGAS is designed to enable programmers to control the memory layout of a distributed application. In Fig. 1 we illustrate the AGAS abstraction layer. HPX implements AGAS as a distributed service where every locality hosts an AGAS instance that is in charge of global objects on that node. Typically, each part of the distributed AGAS service is hosted on a separate node of a cluster.

AGAS consists of several subsystems:

(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. 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.
(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. This information is replicated across all localities to improve performance.
(3) Component Namespace: Types are registered in this namespace to facilitate resolving resource requirements during bulk memory allocations. It is primarily used for type safety checking and debugging purposes.
(4) Symbolic Namespace: The symbolic namespace is a layer on top of the global address space that allows users to map symbolic names to global addresses, often to resolve global addresses at runtime. This is useful in cases where data about specific events needs to be collected. For example, the HPX performance counters use the symbolic namespace for collecting performance counter data.
(5) AGAS Cache: The AGAS cache stores mapping between the most recently used global addresses to localities where those object reside and their local virtual addresses. If a task should be executed on data stored in an object that does not live on the same locality then the task is sent to the locality where the object is currently located. However, in order to do so, AGAS has to determine the current location of the queried object. If the local AGAS instance 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 requesting locality will also solicit the object's current location from the locality of the object's origin. This information is stored in the AGAS cache. The AGAS cache is small since it is designed for speed.
(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. 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

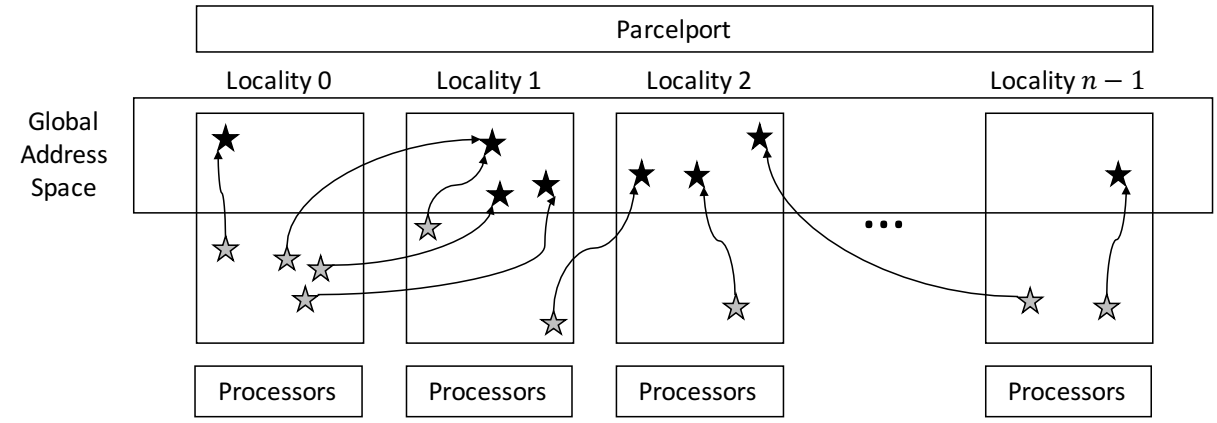


Figure 1: AGAS provides an abstraction layer on top of virtual addresses local to each locality. Global objects are shown as black stars and gray stars indicate references to global objects. Each reference is connected to global objects by an arrow.

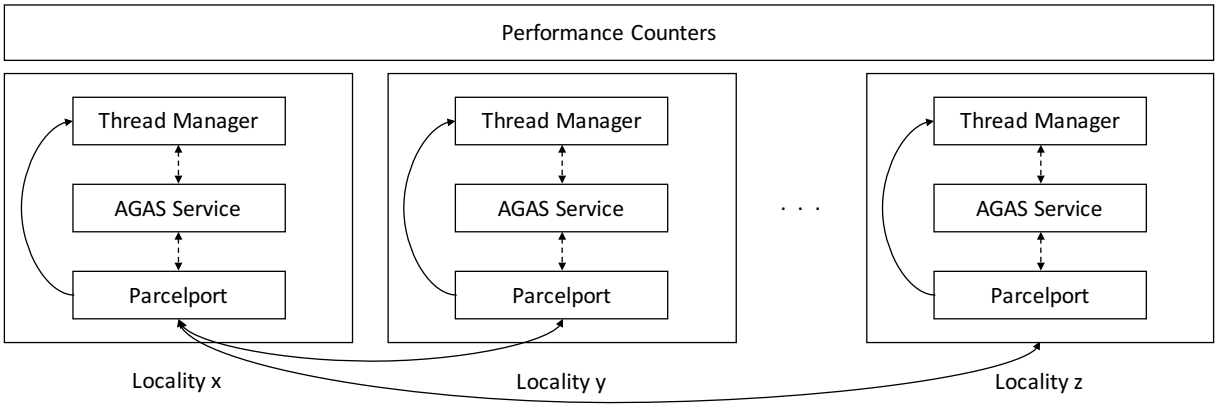


Figure 2: When an HPX thread accesses a global object, AGAS determines if the object can be accessed locally. If the object is on a different locality the HPX task is serialized and given to the parcelport. The parcelport unserializes the task, creates an HPX thread, and hands it to the thread manager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

the same locality. As for remote references, the credit system works as demonstrated in Fig. 3

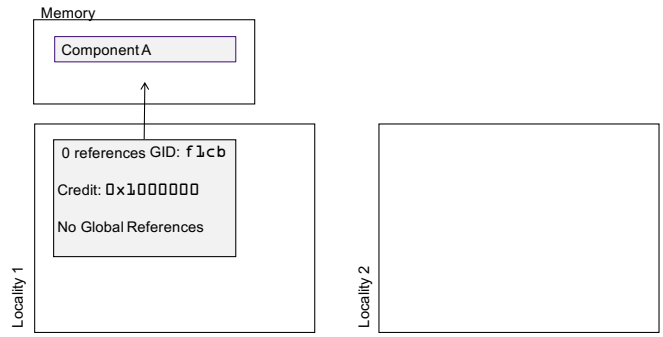
Garbage collection at runtime requires execution of code that is otherwise not present and consumes computing resources. Performing garbage collection requires executing code that is the not the user's application. AGAS tries to minimize garbage collection sweeps by performing it when the volume of garbage reaches a certain threshold that can be specified by the users. It is also possible to manually initiate it inside applications by developers.

3.1 AGAS Performance Counters

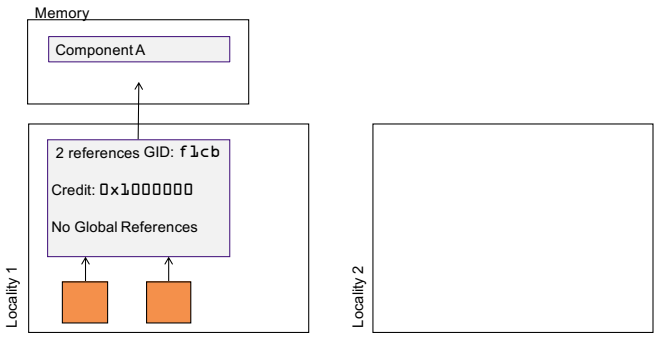
The HPX runtime system includes novel abstractions on top of ordinary operating systems and hardware that are more difficult to benchmark using traditional performance measuring tools such as hardware performance counters included in Intel or NVIDIA processors. HPX introduces a set of performance counters 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[17] uses runtime data provided by the performance counter system to make autotuning policy decisions. It is also possible to ask HPX to print the performance counter data so that it can be consumed by a user or other tools in post-processing.

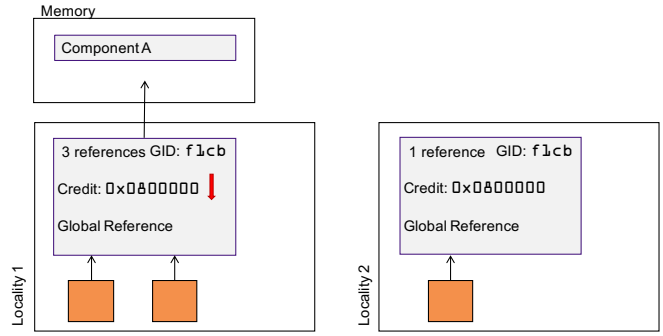
Similar to hardware performance counters, HPX performance counters[15] are designed to expose performance data on the underlying function calls. HPX has performance counters that measure performance of AGAS subsystems. Each AGAS performance counter either reports the number of invocations or the total execution time of the selected operation.



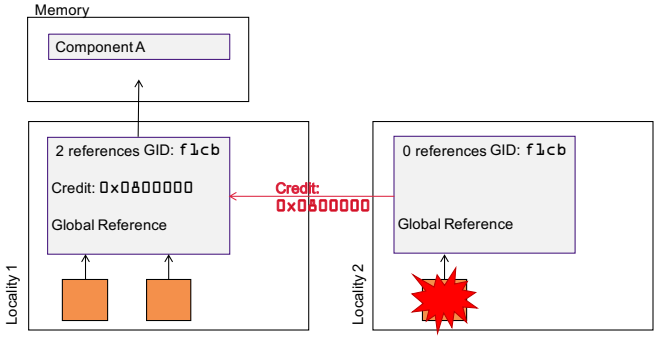
(a) An object is created. AGAS sets the object's credit to a large number.



(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.

Figure 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.

3.2 Migration

Objects registered in AGAS can be physically moved to a different locality while retaining the same (global) address. Moreover, this operation does not need the application execution to be suspended and is completely transparent to the application. 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 (AGAS caches are updated, if needed).

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.

3.3 Lifetime of AGAS

During the initialization of the HPX runtime, HPX needs to initialize all AGAS instances before the application can start executing. This process is called bootstrap and includes registering runtime services, data types, performance counters, and symbols with AGAS. Similarly during teardown, after an HPX application ends. HPX instructs AGAS to remove all objects and free all allocated hardware resources. When applicable, HPX performs additional tasks such as collecting performance counter data from AGAS.

4 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 handle during execution. Our aim in this work, therefore, is to understand the efficiency of AGAS by measuring its

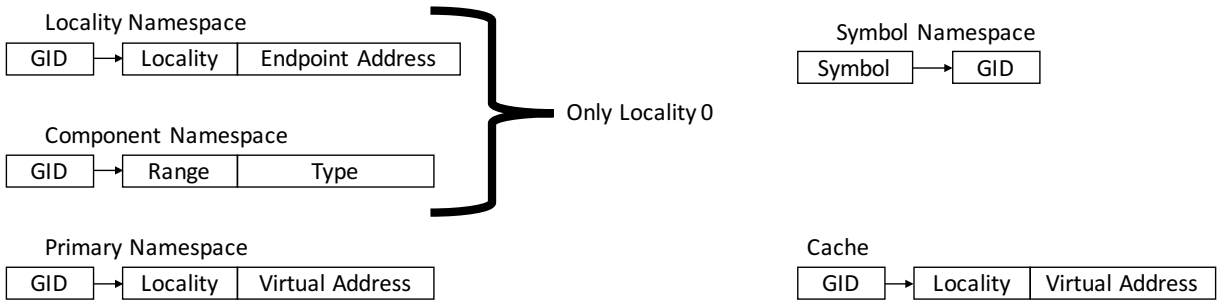


Figure 4: AGAS instances maintain address resolution information in four namespaces and the AGAS cache. Arrows show mappings. *Primary namespace* on each AGAS instance contains mappings from global identifiers (GID) to local address mappings. *Locality namespace* holds information about all AGAS instances. *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. The AGAS cache

overheads. This section identifies metrics that measure the amount of CPU time an HPX application spends in AGAS during execution. Subsequently, we introduce a benchmark application and explain how the performance measurements from this application extend to other applications. Finally, we describe the machine we run our experiments on.

4.1 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 creates some overhead. To quantify and study the overhead introduced by AGAS, we look at the following fundamental operations, the number of calls, and the amount of CPU time that is spent performing these operations.

- 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[30]. A parcel is an active message that triggers an operation upon its receipt by a locality. 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 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. In order to improve performance, the AGAS cache is designed to hold a limited number of entries. The exact number of these entries can be defined by user the user.
- Garbage Collection: HPX provides managed objects whose lifetime is controlled by the garbage collection mechanism. HPX uses reference counting to track local references to global objects 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. As an optimization, garbage collection may not occur instantly once an object goes out of scope, but when the HPX runtime decides the time is right to perform it or when the user explicitly triggers it. Registration and removal of global references is done by calling *increment_credit* and *decrement_credit* functions, respectively.

4.2 Benchmark Application

To study AGAS, we use Octo-Tiger [4, 18] as our application and run it on Piz Daint, a supercomputer at the Swiss National Supercomputing Centre. It should to be noted that other applications could have been used instead of Octo-Tiger. However, we decided to use this application as Octo-Tiger is an open source, actively developed HPX application that is being tuned for scalable performance [11, 16]. The scalability of Octo-Tiger has been previously demonstrated on Cori and Piz Daint.

AGAS is agnostic of the type of application that is using it. The results from studying the performance of AGAS in Octo-Tiger can be applied to other applications because all HPX applications rely on the same functionality and therefore, we can expect to observe similar scaling behavior and performance impact.

Octo-Tiger is a state of the art multi-physics, AMR code that simulates the merger binaries. It uses the HPX runtime system to implement a three-dimensional finite volume octree. It uses several solvers to make predictions including hydrodynamics, Newtonian gravity, and radiation transportation.

Octo-Tiger relies on the capabilities of AGAS in HPX to dynamically changes the computational resolution based on the actual needs of the simulation. It is also a memory intensive application compared to most exascale challenge problems. For example, Quicksilver and Pennant from CORAL2 benchmarks have a memory high water mark of 16% and 8%, respectively, whereas for Octo-Tiger this

Table 1: Configuration of Piz Daint.

Model	Cray XC50
Nodes available	5704
Architecture	Intel Haswell, NVIDIA Pascal
Cores/Node	2×12 cores
Processor	Intel Xeon E5-2960 v3 @ 2.60 GHz, NVIDIA Tesla P100 for PICe-Based Servers
Memory	64 GB; 16 GB CoWoS HBM2
Connection	Cray Aries routing and communications ASIC
Operating System	Cray Linux Environment (UNICOS)

Table 2: Software used in the experiment

HPX	45f3d80	CUDA	9.2
Compiler	GCC 7.3.0	Vc	1.4.1
MPI	Cray MPICH 7.7.2	tcmalloc	2.7
HDF5	1.10.4	Boost	1.68.0
Silo	4.10.2	hwloc	2.0.3

number is about 60%. In our work Octo-Tiger handles 1.5 million HPX objects, which have a memory footprint of 2.305 TB.

We use Octo-Tiger to study the behavior of AGAS when used by applications running on large machines. We ran our experiments on Piz Daint, a cluster composed of Intel Xeon and and NVIDIA Tesla processors. Due to the complexity of the solvers used by Octo-Tiger, we are not able to demonstrate weak scaling as we are not able to adjust the size of the problem with reasonable accuracy. Therefore, we will present the impacts of strong scaling on AGAS behavior. It is worth mentioning that while Octo-Tiger uses GPUs, this has no impact on AGAS as AGAS operations are performed on the CPUs and the number of objects AGAS handles and their access patterns does not change.

4.3 System and Environment Setup

We run our experiments on Piz Daint, an Intel x86/NVIDIA Tesla cluster that comprises 5704 compute nodes. More details about the configuration of Piz Daint is included in Table 1. We run Octo-Tiger on Piz Daint from 2 nodes to 1024 nodes using all processors and HPX commit 45f3d80. The software configuration is listed in Table 2

5 PERFORMANCE RESULTS

As mentioned before, an objective of this study is to quantify the overheads of AGAS. In this section, we present the results of this study. We show the effects of strong scaling on AGAS and consider the overhead AGAS imposes on application execution time. We also show the individual AGAS operations that are the primary drivers of AGAS overheads.

Fig 5 shows the percentage of the total amount of CPU time that is spent on AGAS operations application-wide. While we cannot currently explain the rise between 16 and 32 nodes, we observe that the overhead exponentially decreases past 32 nodes and that

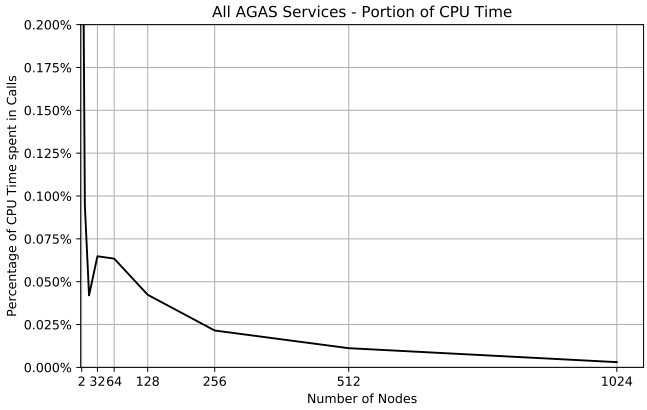


Figure 5: AGAS overhead in total as percentage of overall processor time while running Octo-Tiger on 2 to 1024 nodes of Piz Daint with a fixed problem size (Strong Scaling). The values for the 2 and 4 node runs, cut off from the graph, are 1.6% and 0.4%, respectively.

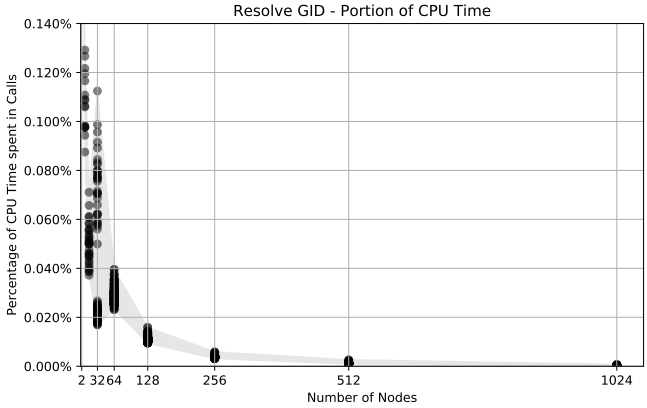


Figure 6: Total CPU time spent in resolve GID calls while running Octo-Tiger on 2 to 1024 nodes of Piz Daint with fixed problem size (Strong Scaling). Each (black) circle refers to the measured amount of time a locality has spent executing resolve GID calls. Higher intensities indicate overlapping measurements. The values for the 2 node run, cut off from the graph, are 1.1% and 0.4%, respectively.

for 8 nodes and above it is below 0.2%. These values are small and show that only a small amount of execution time is spent in AGAS.

Next, we determine which AGAS operations are the most expensive and drive the total cost of AGAS operations. We measured the CPU time of individual AGAS operations and found that four AGAS operations primarily drive AGAS overhead.

Object Lookup is the AGAS function that translates an HPX GID to the local 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,

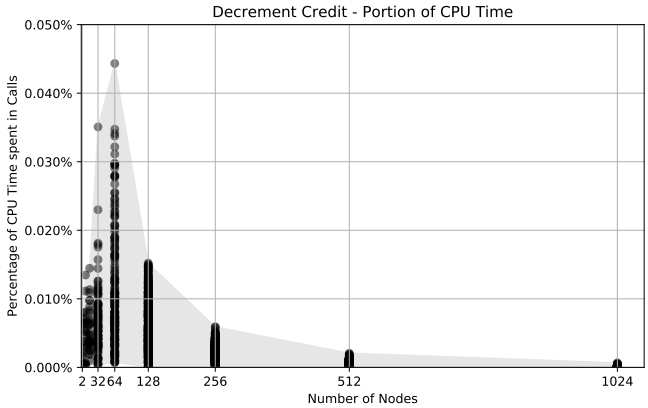


Figure 7: Total CPU time spent in decrement credit calls while running Octo-Tiger on 2 to 1024 nodes of Piz Daint with fixed problem size (Strong Scaling). Each (black) circle refers to the measured amount of time a locality has spent executing resolve GID calls. Higher intensities indicate overlapping measurements.

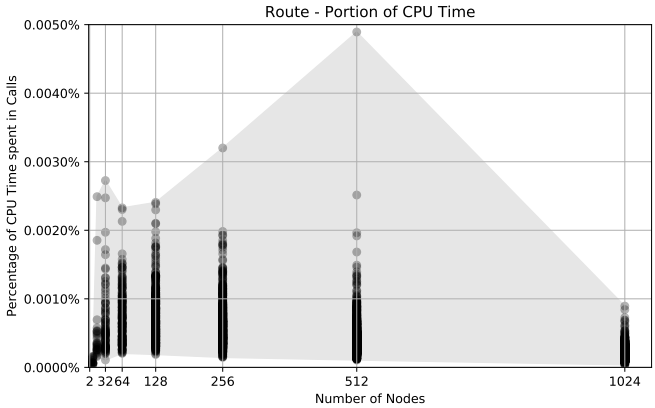


Figure 8: Total CPU time spent in route calls while running Octo-Tiger on 2 to 1024 nodes of Piz Daint with fixed problem size (Strong Scaling). Each (black) circle refers to the measured amount of time a locality has spent executing parcel route calls. Higher intensities indicate overlapping measurements. Values for locality 0 are excluded from this figure because of errors in measurement.

and send it to the locality the object is living in. If a local 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, uses the metadata in the GID to determine the locality on which the object was originally created, and sends the query there. Fig. 6 shows the amount of CPU time *resolve_gid* takes while running Octo-Tiger. Note that the amount of work performed by each locality is relatively similar.

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 operation that lends credit is called *decrement_credit*. Fig 7 shows the overhead that decrement credit calls cause on CPU time.

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 AGAS function that send a parcel to a different locality. We expect that localities exchange a similar amount of parcels and thus, spend a similar amount of time. Fig. 8 shows this behavior.

Finally, Fig 9 depicts the behavior of Bind calls that take place while running Octo-Tiger with the same number of objects as the number of nodes are increased. This figure shows similar scaling behavior as Fig 6, but about 70 times smaller.

The four mentioned AGAS operations account for between 85% and 99% of the CPU time spent in AGAS. Therefore, we do not show the overheads of the rest of AGAS operations in this work.

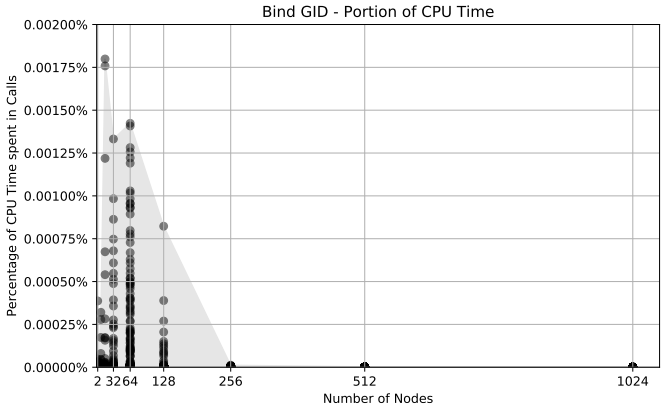


Figure 9: Total CPU time spent in bind GID calls while running Octo-Tiger on 2 to 1024 nodes of Piz Daint with fixed problem size (Strong Scaling). Each (black) circle refers to the measured amount of time a locality has spent executing bind GID calls. Higher intensities indicate overlapping measurements.

6 CONCLUSIONS AND IMPLICATIONS FOR FUTURE WORK

In this work we introduce AGAS, its subsystems, and a method to study the overheads introduced by AGAS. These overheads do 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 multi-physics, AMR application called Octo-Tiger that generates and works on 1.5 million HPX objects, with a total memory footprint of 2.305 TB. 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.

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 multi-threaded system. Future work has to study how the migration feature of AGAS impacts HPX applications. Additionally, devising weak scaling experiments will reveal more interesting information about the overheads of AGAS.

Our observations show that distributed HPX applications running on large machines processors will spend less than 0.5% of their time performing AGAS operations. They also show that the four most expensive AGAS operations, listed in order, are resolve GID, decrement credit, route, and bind GID. Among the four operations, resolve GID is the operation driving at least 80% of the overhead. These numbers indicate that AGAS is hardly a performance bottleneck and AGAS is a relatively inexpensive choice in return for the boost in productivity it provides.

ACKNOWLEDGMENT

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

REFERENCES

[1] [n.d.]. Cascade High Productivity Language - Cray. http://chapel.cray.com/.
[2] [n.d.]. HPX on Github. https://github.com/STEllAR-GROUP/hpx.
[3] [n.d.]. Kokkos on Github. https://github.com/kokkos/kokkos.
[4] [n.d.]. OctoTiger on Github. https://github.com/STEllAR-GROUP/octotiger.
[5] [n.d.]. PGAS - Partitioned Global Address Space Languages. http://www.pgas.org.
[6] [n.d.]. The OpenMP API specification for parallel programming. http://www.openmp.org/specifications/.
[7] [n.d.]. Unified Parallel C. https://upc-lang.org.
[8] Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, Bryce Adelstein-Lelbach, and Thomas Sterling. 2011. Adaptive Mesh Refinement for Astrophysics Applications with ParalleX. (oct 2011). arXiv:1110.1131 http://arxiv.org/abs/1110.1131
[9] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2.
[10] Martin J Chorley and David W Walker. 2010. Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. *Journal of Computational Science* 1, 3 (aug 2010), 168–174. https://doi.org/10.1016/j.jocs.2010.05.001
[11] Gregor Daiß, Parsa Amini, John Biddiscombe, Patrick Diehl, Juhan Frank, Kevin Huck, Hartmut Kaiser, Dominic Marcello, David Pfander, and Dirk Pflüger. 2019. From Piz Daint to the Stars: Simulation of Stellar Mergers using High-Level Abstractions. *arXiv e-prints*, Article arXiv:1908.03121 (Aug 2019), arXiv:1908.03121 pages. arXiv:cs.DC/1908.03121
[12] Chirag Dekate. 2011. *Extreme Scale Parallel N-Body Algorithm with Event-Driven Constraint-Based Execution Model*. Ph.D. Dissertation. Louisiana State University, Baton Rouge, LA, USA.
[13] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.
[14] Karl Feind. 1995. Shared memory access (SHMEM) routines. *Cray Research* (1995).
[15] Patricia Grubel, Hartmut Kaiser, Jeanine Cook, and Adrian Serio. 2015. The performance implication of task size for applications on the hpx runtime system. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 682–689.
[16] Thomas Heller, Bryce Adelstein Lelbach, Kevin A Huck, John Biddiscombe, Patricia Grubel, Alice E Koniges, Matthias Kretz, Dominic Marcello, David Pfander, Adrian Serio, Juhan Frank, Geoffrey C Clayton, Dirk Pflüger, David Eder, and Hartmut Kaiser. 2019. Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars. *The International Journal of High Performance Computing Applications* 33, 4 (2019), 699–715. https://doi.org/10.1177/1094342018819744 arXiv:https://doi.org/10.1177/1094342018819744
[17] Kevin A. Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen D. Malony, Thomas Sterling, and Rob Fowler. 2015. An Autonomic Performance Environment for Exascale. *Supercomputing Frontiers and Innovations* 2, 3 (jul 2015), 49–66. https://doi.org/10.14529/jsfi150305
[18] Kundan Kadam, Geoffrey C Clayton, Patrick M Motl, Dominic Marcello, and Juhan Frank. 2017. Numerical Simulations of Close and Contact Binary Systems Having Bipolytropic Equation of State. In *American Astronomical Society Meeting Abstracts*, Vol. 229.
[19] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. 2009. ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *2009 International Conference on Parallel Processing Workshops (ICPPW)*, Leonard (FIT) Barolli and Wu-chun (Virginia Tech.) Feng (Eds.). IEEE, Vienna, Austria, 394–401. https://doi.org/10.1109/ICPPW.2009.14
[20] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX - A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, Eugene, OR, USA, 6. https://doi.org/10.1145/2676870.2676883
[21] Laxmikant V Kale and Sanjeev Krishnan. 1993. CHARM++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications - OOPSLA '93 (OOPSLA '93)*, Vol. 28. ACM Press, New York, New York, USA, 91–108. https://doi.org/10.1145/165854.165874
[22] ODEN Lena. 2014. GPI2 for GPUs: A PGAS framework for efficient communication in hybrid clusters. *Parallel Computing: Accelerating Computational Science and Engineering (CSE)* 25 (2014), 461.
[23] Sreeram Potluri. 2014. *Enabling Efficient Use of MPI and PGAS Programming Models ion Heterogeneous Clusters with High Performance Interconnects*. Ph.D. Dissertation. Ohio State University. https://etd.ohiolink.edu/pg{_}10?0::NO:10:P10{_}ACCESSION{_}NUM:osu1397797221
[24] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. 2009. 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 (PDND '09)*. IEEE, Weimar, Germany, 427–436. https://doi.org/10.1109/PDP.2009.43
[25] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. 2010. The Asynchronous Partitioned Global Address Space Model. In *The First Workshop on Advances in Message Passing*. 1–8.
[26] Vivek Sarkar, William Harrod, and Allan E Snavely. 2009. Software challenges in extreme scale systems. *Journal of Physics: Conference Series* 180 (jul 2009), 012045. https://doi.org/10.1088/1742-6596/180/1/012045
[27] Albert Sidelnik, Bradford L Chamberlain, Maria J Garzaran, and David Padua. 2011. *Using the High Productivity Language Chapel to Target GPGPU Architectures*. Technical Report. http://hdl.handle.net/2142/18874
[28] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 81.
[29] Thomas Sterling and Dylan Stark. 2009. A High-Performance Computing Forecast: Partly Cloudy. *Computing in Science & Engineering* 11, 4 (jul 2009), 42–49. https://doi.org/10.1109/MCSE.2009.111
[30] Bibek Wagle, Samuel Kellar, Adrian Serio, and Hartmut Kaiser. 2018. Methodology for Adaptive Active Message Coalescing in Task Based Runtime Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1133–1140.
[31] Michael Wong, Hartmut Kaiser, and Thomas Heller. 2015. *Towards Massive Parallelism (aka Heterogeneous Devices/Accelerator/GPGPU) support in C++ with HPX*. Technical Report. 16 pages. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0234r0.pdf
[32] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. 2011. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications* 182, 1 (jan 2011), 266–269. https://doi.org/10.1016/j.cpc.2010.06.035
[33] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS 14)*. IEEE, Phoenix, AZ, USA, 1105–1114. https://doi.org/10.1109/IPDPS.2014.115