Active Global Address Space (AGAS): Global Virtual Memory for Dynamic Asynchronous Many-Tasking (AMT) Runtimes

Abhishek Kulkarni, Andrew Lumsdaine (advisor)

{adkulkar, lums}@cs.indiana.edu
Center for Research In Extreme Scale Technologies (CREST)
Indiana University

***Abstract*—While the communicating sequential processes (CSP) model, as realized by the Message Passing Interface (MPI), is presently the dominant scalable computing paradigm, it neither excels at the irregular computation patterns present in big-data and adaptive execution, nor obviously scales to exascale. Dynamic asynchronous many-tasking (AMT) execution models are suggested as alternatives to MPI in these realms. In such models, programs are described in terms of lightweight tasks concurrently operating on data residing in a global address space. Partitioned Global Address Space (PGAS) are less well-suited for event-driven and active-message execution, where ephemeral computation is performed on data in the global address space at the location at which the data resides. The static distribution encoded in the PGAS addresses restricts the system's ability to load balance computation and communication.**

Maintaining a scalable high-performance virtual global address space using distributed memory hardware has proven to be challenging. Distributed Shared Memory (DSM) models operated on a page-level granularity and incurred high overheads due to consistency. We present High Performance ParalleX (HPX-5), a dynamic adaptive many-tasking runtime system for extreme-scale and exascale applications. We describe the design and implementation of *active* global address space (AGAS) in HPX-5 and demonstrate its benefit for global load balancing. Finally, we evaluate a new approach for implementing an active global address space that leverages the capabilities of the network fabric to manage addressing, rather than software at the endpoint hosts.

I. Introduction

Abstraction of resources is a fundamental capability of modern operating systems. A particularly elegant example of resource abstraction is virtualization of memory which separates the logical address space unique to each process from the physical address space provided by the computer's memory hardware. This virtualization enables remarkably powerful: process isolation, swapping, shared memory, to name a few. Similarly, abstraction of location in a distributed system—through the separation of logical and physical global memory addresses—enables flexibility of execution at runtime. Particularly, it allows the runtime system to manage both load and locality concerns by moving the *work* to *data* or vice-versa. We introduce global virtual memory for a dynamic asynchronous many-tasking (AMT) runtime system referred to as the *active global address space*. Our active global address space is "active" in two senses. It is virtualized and dynamically relocatable (unlike existing static PGAS models such as UPC and SHMEM). In addition—and as a bit of a pun—its usage is primarily through the use of active messages [1] and put/get operations (often implemented with active messages), as opposed to the local read/write model in current distributed shared memory models. We present HPX-5, an implementation of the ParalleX model [2] in C, that extends a many-tasking execution paradigm suited for extreme-scale applications. We also present a novel approach to implementing active global address spaces that uses the interconnect fabric rather than endpoints to maintain global to local mappings. This approach greatly simplifies the process of providing an active global address space while also avoiding many of the associated overheads.

II. Motivation

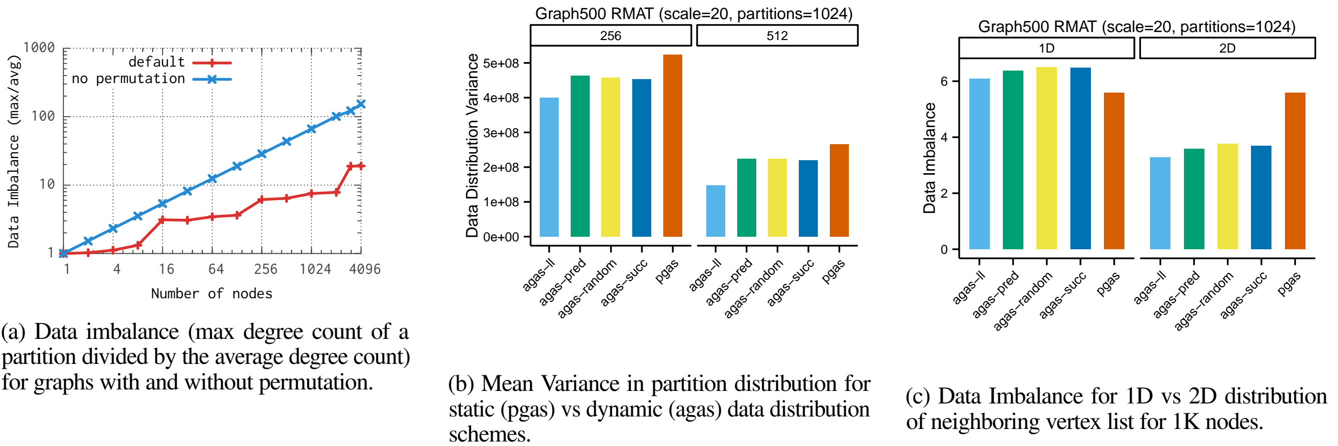
For dynamic irregular applications such as graph algorithms, it is difficult to determine a favorable initial data distribution. In some cases, it is even impossible as the data being operated on changes during the course of execution. One prominent example is the 2D adaptive mesh refinement (AMR) technique which selectively discretizes a grid to finer grids when a higher resolution is required. The key data structure involved is a level of grid hierarchies linked together representing varying discretizations of the grid. As the algorithm proceeds, it leads to a high degree of computational and data imbalance among the nodes. Favorably, we want the runtime to be able to dynamically relocate data and work to avoid scalability bottlenecks due to imbalance.

Global virtual memory introduces an extra indirection between the memory's address and its physical location. This allows flexibility in terms of relocating data and also being able to represent complex distributed data structures and data distributions in the global address space. The capability to relocate data is crucial for load balancing irregular algorithms with dynamically evolving data. Data redistribution can be achieved through explicit prorammer-specified annotations, or dynamic data migration in the runtime system can be carried out in response to introspective capabilities.

Load and Data Imbalance in Graph Algorithms: We consider load imbalance in a distributed 1D Breadth-First Search (BFS) algorithm operating on a Graph500 graph generated using the Recursive MATrix (R-MAT) algorithm (with parameters A=0.57, B=0.19, C=0.19, D=0.05). The graph is represented as an index array of vertices. Each vertex maintains a list of its neighbors that represent its outgoing edges. The amount of work done by each node roughly corresponds to the sum of outgoing edges of all of its partitions. The data imbalance of a graph is given by the maximum degree count of a partition divided by the average degree count. As seen in (Figure 1a), the data imbalance increases linearly as the number of nodes increase. Note that We consider a tiny graph in this example but observed similar trends for larger graphs. In Figure 1, the graph is divided into 1K partitions of 1K vertices each (16 bytes per vertex). The vertex with the minimum degree count is redistributed to another node if a vertex that meets the following condition is found: $degree(v) >= 1.4 \times degree_{avg}$. Different policies to choose the destination node were evaluated: least-loaded (agas-ll), predecessor (agas-pred), successor (agas-succ) or random (agas-random). The distribution imbalance increases as the number of nodes are increased (fewer partitions per node). At 512 nodes (2 partitions/node), we see 29 moves with an average of 568 KB and the imbalance factor reduces from 3.24 to 2.79. At 256 nodes (4 partitions/node), there are 7 moves (avg of 428 KB) and the imbalance factor reduces from 1.98 to 1.81. With fewer partitions per node, a 2D distribution of the neighboring vertices is much more profitable as shown in Figure 1c.

A. High Performance ParalleX (HPX-5)

HPX-5 is a library-based implementation of the ParalleX model [2] in C. HPX-5 provides programming interfaces for global address



(a) Data imbalance (max degree count of a partition divided by the average degree count) for graphs with and without permutation.

(b) Mean Variance in partition distribution for static (pgas) vs dynamic (agas) data distribution schemes.

(c) Data Imbalance for 1D vs 2D distribution of neighboring vertex list for 1K nodes.

Fig. 1: Effect of varying vertex and edge-list distributions on imbalance.

manipulation, translation, and allocation; parcels, lightweight threads, and LCOs. Internally, HPX-5 is organized around a cooperative work-stealing thread scheduler, a parcel transport with a flexible networking interface, and unified access to the global address space. HPX-5 provides networking based on Photon [3] as well as MPI. In addition to active message parcel operations, HPX-5 provides direct access to the global address space through asynchronous put/get operations. Generic support for both parcel transport and put/get operations is designed around a novel networking abstraction, put-with-remote-notification. HPX-5 provides two implementations of this abstraction. The PWC network uses Photon's put-with-remote-completion support directly and implements a parcel emulation layer for parcel transport. The ISIR network provides parcel transport on top of the traditional MPI Isend/Irecv point-to-point messaging, and emulates put-with-remote-notification with parcels. The internal architecture of HPX-5 is shown in Figure 2.

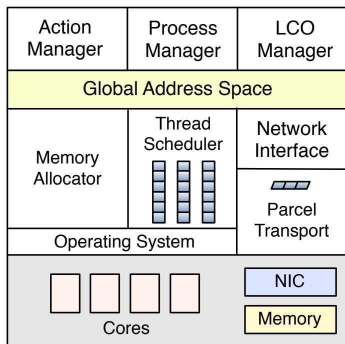


Fig. 2: HPX-5 Architecture. At the core of HPX's architecture is a thread/parcel scheduler and the global address space. LCOs and processes are allocated in the Global Address Space.

In HPX-5, parcels target global addresses, carry payload data, identify message handlers known as *actions* for execution, and specify continuation addresses. Parcel transport is reliable but unordered. Parcel delivery spawns a cooperative lightweight thread executing the parcel's action. Lightweight threads have the ability to acquire resources, send additional parcels, and wait for asynchronous events. A thread automatically sends its result to the global address as specified in its progenitor parcel.

Programming with parcels is inherently asynchronous. Data and control dependencies are expressed through the use of local control objects (LCOs) that enable threads to wait for data and events without consuming execution resources. HPX-5 provides a number of built-in LCO types, including futures, simple reductions, and semaphores, and provides an interface for user-defined types as well. LCOs are allocated in the global address space and are accessed through a uniform interface, irrespective of their actual affinity. LCOs serve as parcel continuation addresses, and the thread continuation operation automatically sends the continued data to the LCO and updates its state if necessary. This enables distributed sequential computation in addition to the parallel computation provided with parcels.

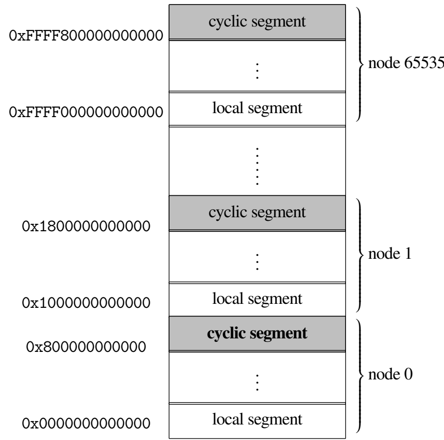
III. Active Global Address Space

The HPX-5 programming model defines a linear, byte-addressable, active global address space (AGAS), organized into distinct logical *blocks* consisting of relocatable units (known as *chunks*). The first block of each distinct cyclic or blocked allocation begins with affinity to the root locality. AGAS in HPX-5 supports the following allocation schemes:

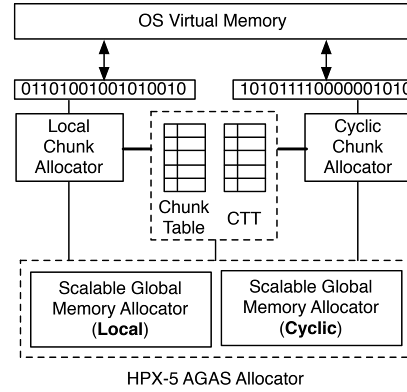
- **Local**: This returns memory from a "local chunk" with affinity to the calling locality. If the local segment runs out of memory, this allocation request may be satisfied by another locality. All of the allocated memory is treated as one chunk.
- **Cyclic**[1]: This allocates memory from "cyclic chunks" that are cyclically distributed among the available localities. The chunk type is a `char[]` and the chunk size is configurable on a per allocation basis. The cyclic address space is backed by a cyclic symmetric virtual heap managed by the root locality (id 0).
- **Blocked**: Blocked allocations (also known as super-block cyclic allocations) group the chunks up to a user-specified number of blocks, and divide the block of chunks among available localities. For instance, Locality 0 has blocks $0 \ldots \frac{m}{localities} - 1$, Locality 1 has blocks $\frac{m}{localities} \cdots \frac{2m}{localities} - 1$, so on and so forth.
- **User-defined**: User-defined distributions in HPX-5 are parameterized by a function that maps (the index of) a chunk to a locality.

Virtual Address Representation: The `hpx_addr_t` representation of a global virtual address in HPX-5 is simply a 64-bit integer. It encodes the locality affinity of the chunk (home), a bit representing if the address is part of a cyclic allocation or not (c), a size class (size), as well as an absolute byte offset within the allocation (offset). While the offset can be fairly large (2^{42} bits), the size class limits chunk sizes to 4TB (2^{32} bits).

[1] Note that this differs from the traditional cyclic allocation in PGAS models as the minimum relocatable unit of allocation is a chunk, and not necessarily a byte.



(a) Global Virtual Address Space in AGAS.



(b) Scalable AGAS allocator in HPX-5. The heaps are divided for local and cyclic memory, which itself is managed by a scalable, concurrent memory allocator.

Fig. 3: Software AGAS implementation in HPX-5

Global Virtual Address Space: Due to the global address representation scheme chosen, the global virtual address space is segmented as shown in Figure 3a. With 16-bits for node identifiers, a maximum of 64K nodes are supported. The local and cyclic segments are divided into separate regions for allocations of different sizes.

IV. Network-managed Global Virtual Memory

There are significant overheads involved in the software implementation of a global virtual memory system. We describe a hardware-assisted implementation that performs address translation using the interconnect fabric. This was prototyped using a GASNet conduit, called *Photon*, that uses IB UD multicast. Messages are forwarded to their target global address by looking up an address-to-port mapping in an OpenFlow-enabled switch, essentially performing address resolution in the network.

Figure 4 outlines this approach. During block initialization, the AGAS layer registers local blocks as multicast groups via Photon (1) and pushes flow entries to the discovered switch(es) in the network (2,3). Step (1) allows the receiver to accept messages destined to that block from any sender, while steps (2,3) allow the switch to direct a given AGAS block, identified by its mapped multicast Ethernet MAC address, to the appropriate port. In our example, node A owns block 1, and when nodes B and C send a message to destination block 1, the destination node A is unknown at the time of the send operation. Only when the switch matches on the flow entry for block 1 is the destination node determined, steps (5,7) and (6,7), and the message delivered to the current owner of the block. When a block moves, the AGAS remapping operation updates the flow entry(ies) in the switch along with the receiver multicast groups. In practice, the out-of-band signaling latency from our embedded controller to the switch, and the synchronization required during dynamic remapping, creates a bottleneck for latency-sensitive applications (Section V-A).

We note that our hardware-based AGAS could also be deployed on a native IB network where a SM that implements "send only" multicast joins is available. The multicast join incurs noticeable latency that significantly impacts performance and scalability of our network-assisted approach. The IB specification defines the joining of a multicast GID as a "send only" member, which would resolve the issue of senders also receiving packets from peers addressing the same GID. However, this feature is not supported by OpenSM and we were unable to provide results using the native IB fabric.

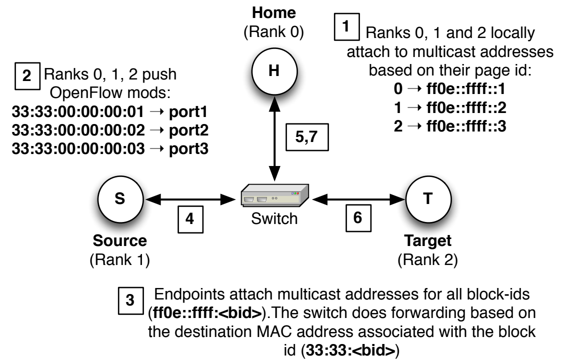


Fig. 4: Use of multicast addressing in RDMA over Ethernet.

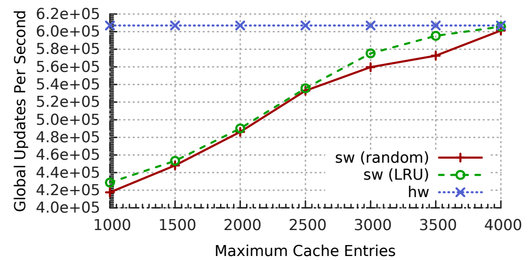


Fig. 5: Effect of bounded cache and varying cache replacement policies on the overall performance of the GUPS random access microbenchmark at 192 cores.

V. Benefits of Hardware-Assisted AGAS over Software AGAS

At larger scales, an efficient cache directory implementation needs to be bounded in size as memory on a node is shared by the runtime with the application. We extended the sequential direct-mapped cache with two standard cache replacement policies, *random* and *LRU (least recently used)*. Bounding the cache size adds extra cache eviction logic and incurs capacity cache misses, which negatively impacts cache operations on the critical path. Figure 5 demonstrates that enforcing an upper bound on the Software cached global directory size degrades performance of puts and gets in the global address space. In our test, each cache entry

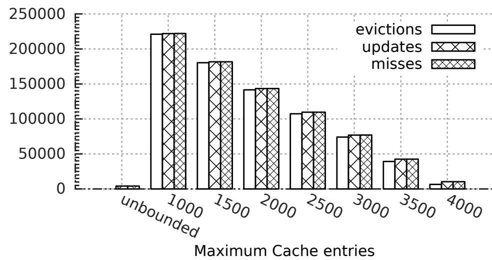


Fig. 6: LRU Cache statistics at Rank 0 for 2^{26} random updates of a 2^{21} words table.

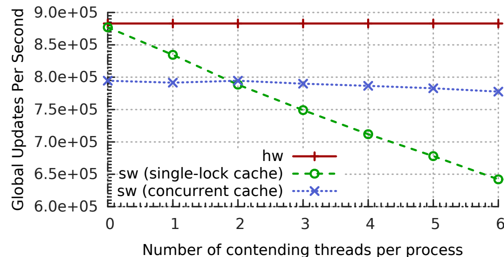
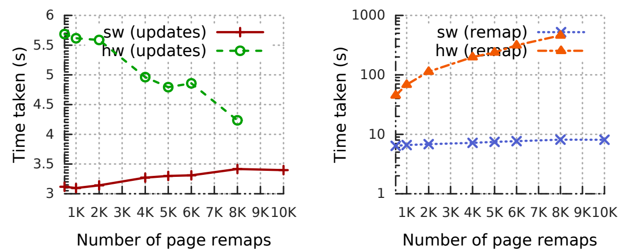


Fig. 7: Performance degradation of global updates due to thread contention in concurrently accessing the cache.

was only 4 bytes resulting in a 1:1000 ratio of metadata overhead per addressable memory page. In practice, cache entries often encode other attributes pertaining to the page, and an overhead of mere 0.1% can potentially limit the scalability of the runtime to 1000 processors. In Figure 5, 64 million global updates were performed on a 2 million word table from 192 processors. The unbounded direct-mapped software cache overhead (not shown) is nearly identical with the hardware directory overhead, as cache replacement does not play any role in the operation of either of those. At 1K cache entries, both replacement policies perform 30% fewer global updates per second as a result of higher capacity cache misses and evictions. The LRU cache statistics for rank 0 are shown in Figure 6. In the unbounded case, each rank addresses at most 4096 pages without incurring any misses. More evictions to increased cache misses. For a random workload, no cache replacement policy is the best. The key takeaway is that none of the cache replacement policies scale in comparison to the hardware directory implementation.

Modern active-message based runtime systems have multi-threaded domains where multiple threads are running active message handlers at once. In addition to the storage overheads, a major drawback for a software cache implementation is its scaling limitation. The effect of thread contention on the software cache performance is shown in Figure 7. First, we compared a simple global single-lock cache against a sequential no-lock cache. Multi-threaded domains were simulated by spawning threads that access the cache at the same frequency as the main thread with a random bias (by sleeping between $2-10\mu$secs between accesses). In this test, 16 million random global updates were performed on a 1 million word table by 32 processes on 16 nodes. We spawned up to 6 contending threads per process. The single-lock cache provides a comparison baseline for the concurrent cuckoo hashtable. While the concurrent cache scales well with respect to the number of contending threads, there is an opportunity cost of concurrency that leads to much lower single-thread performance. The concurrent cache is 25% slower than the hardware directory (cache-bypass) solution.



(a) The update time measures the time it required to perform global updates ignoring the overhead of active remap operations.

(b) The remap phase measures the time to move the data associated with the GVA, and update its global mappings.

Fig. 8: Effect of active dynamic remapping to the performance of random access of global data for Software and Hardware AGAS.

A. Dynamic Remapping of Global Memory

We used the GUPS microbenchmark to evaluate the impact of dynamically moving pages (i.e., "remapping" blocks) within our AGAS runtime by initializing an experiment with a small table size that resulted in a total of four logical pages. Distributed across 192 cores, remapping these pages ensures that at least 25% of all subsequent global updates will target a different process. In Figure 8a we show only the total global random update time (discounting the remap operation itself) between software and hardware cases over increasing numbers of remap operations. The key takeaway is that as page movement frequency increase, the software approach takes increasingly longer as more global updates require two or more hops to resolve their new page owners. In contrast, the direct lookups afforded by the hardware case ensure a constant, or improved, run time as remaps increase. In our small-scale evaluation, the overall time trend even decreases for hardware as pages frequently become local to a single physical node that would otherwise require a network hop. On the other hand, the limitations of our OpenFlow-based implementation become apparent in Figure 8b. For each remap operation, the time needed to update the switch table via out-of-band signaling, as opposed to updating the software owner cache with an in-band active message, is considerably higher. The switch table update mechanism is an area that requires a tighter coupling between our AGAS runtime and the supporting hardware, and we are investigating solutions that provide an in-band, "bump-in-the-wire" remap mechanism.

VI. Conclusions

The HPX-5 runtime system is backed by an active global address space (AGAS) which provides the dynamic features necessary for adaptive execution of large-scale irregular applications. The results of our network-assisted design with the current generation of hardware are mixed. We have demonstrated the benefit of the hardware implementation over software, however remapping remains an expensive operation due to the cost of programming flows.

References

[1] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *International Symposium on Computer Architecture*. New York, NY, USA: ACM, 1992, pp. 256–266.

[2] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "ParalleX: A study of a new parallel computation model," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–6.

[3] A. Danalis, A. Brown, L. Pollock, M. Swany, and J. Cavazos, "Gravel: a communication library to fast path MPI," in *In Proceedings of EuroPVM/MPI*, Sep. 2008.