Improving Data Locality for Irregular Partitioned Global Address Space Parallel Programs

D. Brian Larkins
Department of Computer Science
Coastal Carolina University
Conway, SC 29528-6054
blarkins@coastal.edu

ABSTRACT
This paper describes a technique for improving the data reference locality of parallel programs using the Partitioned Global Address Space (PGAS) model of computation. One of the principal challenges in writing PGAS parallel applications is maximizing communication efficiency. This work describes an on-line technique based on run-time data reference profiling to organize fine-grained data elements into locality-aware blocks suitable for coarse-grained communication. This technique is applicable to parallel applications with large, irregular, pointer-based applications. The described system can perform automatic data relayout using the locality-aware mapping with either iterative (timestep) based applications or as a collective data relayout operation. An empirical evaluation of the approach shows that the technique is useful in increasing data reference locality and improves performance by 10-17% on the SPLASH-2 Barnes-Hut tree benchmark.

Categories and Subject Descriptors
D.1.3 [Concurrent Programming]: Parallel Programming

General Terms
Algorithms, Performance

Keywords
PGAS, data locality, profile-driven optimization, irregular data structures, distributed memory

1. INTRODUCTION
Achieving good performance from a parallel application that uses distributed irregular data structures is a challenging problem. Extracting the most performance from modern large-scale distributed memory clusters places a great burden on both application software and run-time systems. Irregular problems in particular, have unique challenges with respect to load balance and data locality. The nature of many of these applications and the systems on which they run makes statically anticipating and improving program performance difficult. Additionally, systems which are able to adapt and balance one component of the dynamic execution of a program, like workload, often end up being at odds with another, such as data locality.

The Global Chunks Library (GCL) provides a programming model and runtime system which allows developers to access a globally accessible pointer-based data structure, such as a tree or graph, using global pointers on a distributed memory system. The GCL framework is the basis for the Global Trees (GT) and Global Graphs (GG) data-structure specific parallel runtime systems. As described in [5, 6], the GCL framework permits fine-grained operations on global pointers, but performs coarse-grained communication in chunks to achieve communication efficiency. Chunks are collections of globally accessible shared data, similar to cache-lines, or pages in distributed shared memory systems. Programs can be written in a MIMD shared-memory style on distributed-memory clusters by using global pointers, however poor locality with highly non-uniform memory access costs can be devastating to performance.

Improving the data reference locality of a program can occur by addressing the inherent data reference locality, which is specific to a particular algorithm, as well as the realized data locality [1]. The realized data locality is concerned with the system-level mapping of program data to actual storage locations within the cluster. Common approaches which aim to improve data reference locality are algorithmic modification, static data mapping techniques, the use of heuristic allocation strategies, and using dynamic program behavior to optimize data placement. The GCL is a generalized framework which provides global pointers for distributed memory applications and very efficient support for fine-grained data access, commonly used in irregular linked data structures. This paper focuses on improving the realized data reference locality as provided by the GCL library.

Most of the communication in the GCL framework occurs at the chunk level, hence, element allocations (e.g. tree nodes) must be mapped to chunks. Both the mapping of data elements to chunks, as well mapping of chunks to specific processors will impact the amount of communication performed during program execution.

Static data mapping approaches, such as the default allocator in the GCL can perform very well when the data access pattern is able to be characterized statically. Heuristic allocation strategies employ a set of static rules along...
with the dynamic state of the program to make an educated guess about the best processor and chunk to allocate an element from. Since many tree and graph applications have a recursive structure during tree creation, static heuristics and call site information may not provide enough context to make a good allocation choice. For example, consider a tree-based application where the dominant access pattern is depth-first. If the tree is allocated in a depth-first fashion, then a static mapping is sufficient (e.g., elements map to chunks in allocation-order). If the tree is allocated in a different order, it is difficult to know whether the current allocation is for a leaf, or is the root of a long branch. This is important because chunks should be full in order to get the benefits of coarse-grained communication, but they should also contain related data, so as to take advantage of spatial locality within the tree. Without additional global knowledge of future allocations, it is very difficult to develop heuristics which result in improvements to realized data reference locality.

Profile-driven techniques measure various data access and communication patterns during a representative execution phase of the program and then use this information to guide future allocations. Since the profile is unique to each specific tree application, these approaches can be very accurate. Gathering profile data and performing analysis on it, however, can be expensive and may offset the benefits of better data placement. This paper makes the following contributions:

**Mapping Algorithm:** A profile-driven technique is described which maps nodes to chunks and processors in a GCL application. The profile collects the number of data references for each reference edge in the data structure and uses this information to guide the element/chunk mapping.

**Iterative Data Layout:** A technique for applying the new mapping to algorithms which have an iterative structure, such as the Barnes-Hut algorithm. One or a small number of iterations are used to generate the data profile, then future iterations use the locality-optimized mapping for placing data elements.

**Collective Data Layout:** A technique for applying the mapping layout to algorithms using a collective data relay-out operation. The program is allowed to operate for a period of time in the profile gathering phase. After the profile has been obtained, the program can collectively call a routine which computes a mapping and moves the global data to new locations according to the mapping.

**Empirical Evaluation:** An empirical evaluation of these techniques is presented which demonstrates the effectiveness of this approach. Performance is evaluated using a breadth-first allocation / depth-first access benchmark, as well as the Barnes-Hut application from the SPLASH-2 benchmark suite [9].

### 2. ANALYSIS

Parallel applications are burdened with additional overhead not present in the corresponding sequential code. This is primarily due to communication overhead that the sequential code can ignore as well as the overhead due to the parallel runtime system which manages the global data and communication. Traditionally, the definition of $t_{comm}$ shown in Eq. 1 accounts for the time spent in non-overlapping communication, and is the sum of startup cost and transfer time for $i$ communication events in the execution:

$$t_{comm} = \sum_{i} (t_s + B_it_b)$$  \hspace{1cm} (1)

In the PGAS programming model, global data is distributed amongst the participating processors. The computation on a given processor will need a portion of this globally shared data to compute, which can be broken into two groups, local and non-local data. For a given computation, presume that $\lambda$ data references are local and $\rho$ data references are non-local. The total number of data references ($\tau$) can be expressed as: $\tau = \lambda + \rho$. Since local accesses involve no communication, we focus on reducing remote accesses.

The GCL runtime uses the concept of chunks to perform bulk data transfers and improve locality. Non-local data may be cached by the runtime, which may be subsequently accessed with only local access cost. If we define $\lambda_p$ as the number of non-local data accesses which are served from cache and $\rho'$ as the non-local accesses requiring communication, we get the following relation:

$$\tau = \lambda + \lambda_p + \rho'$$  \hspace{1cm} (2)

Similar to unreferenced elements within a cache-line, unnecessary data may also be communicated in a chunk communication due to an imperfect element-to-chunk mapping. The number of data elements which are communicated, but never referenced is defined as $\sigma$.

Now, if we let $|\rho|$ and $|\sigma|$ represent the respective data volume of these elements in bytes, we get the following expression for $t_{comm}$:

$$t_{comm} = \sum_{i} \rho't_s + (|\rho| + |\sigma|)t_b$$  \hspace{1cm} (3)

Thus, to improve communication time, the following factors must be considered, assuming $t_s$ and $t_b$ are inherent properties of the system:

- **$\rho'$** - Non-local data accesses requiring communication. Reducing the number of communication events will reduce the contribution of $t_b$ to overall runtime.
- **$\rho$** - All non-local data accesses. Reducing the amount of remote data referenced will reduce the overall communication volume.
- **$\sigma$** - Unnecessary data transferred. Any unreferenced data which is communicated without computational overlap is wasted time spent in transfer.

In general, the problem of finding an optimal algorithm for data placement and code arrangement that minimizes the number of cache misses (i.e. $\rho'$), is NP-Hard. It has also been shown that efficient approximate algorithms produce sub-optimal mappings relative to an optimal solution [7].

Instead of attempting to make good placement choices at allocation time, we opt for a profile-driven approach to data layout. This approach takes a data reference trace for a Global Chunks parallel program and uses this information to determine an element/chunk mapping for a subsequent phase of computation. The element mapping can be used to move nodes as a collective operation, or used to inform future element allocations. This approach extends the GCL to support dynamic data distributions.

One of the key techniques to reduce overhead when handling global pointers is to take advantage of an optimization for handling intra-chunk references with the *relative pointer*
representation whenever possible. In the GCL, dereferencing an absolute global pointer involves function call overhead, several conditionals, a directory lookup, and some pointer arithmetic, whereas a relative pointer is limited to a single conditional ("is this a relative pointer?") and two pointer additions[6]. When the computational workload relative to each global pointer dereference is low, global pointer overhead can be an impediment to good performance. Any system-level support for increasing the use of relative pointers (by co-locating related data in the same chunk) will reduce runtime overhead and improve performance.

3. PROFILE-GUIDED DATA LAYOUT

Many applications have a phase for data structure creation and then another for computation. Frequently, the data access pattern used in the computation differs from the specific order used to create the structure. The GCL must make a decision to place an element in a chunk at allocation time, with imperfect knowledge about subsequent allocations or the dominant data access pattern. As a result, profile data may improve data placement after the application has run in a steady state for awhile.

The profile-guided layout process works in two steps: First, the program runs in a profile gathering mode, where data is collected about data references during an applicationspecified phase of the program. Next, the runtime performs a partitioned layout which assigns elements to chunks, and chunks to processes based on the information gathered from the profile data.

3.1 Data Structures

An edge is the relationship between a global pointer and the referenced data object. In linked structures, global pointers will largely be contained within other globally shared data elements (e.g. a linked list). An edge traversal is the process of dereferencing a global pointer contained within a global data element (e.g. dereferencing a pointer to a child from the parent node in a tree).

The partitioning step relies on two key data structures. The first data structure, edgerefs, tracks the number of times a specific edge has been traversed throughout a phase of computation. This data is maintained during the profiling step and is provided as input to the partitioning step. The edge is defined to be a two-tuple containing the unique identifiers of each element on the edge, and the edgerefs structure is a mapping from edges to reference counts:

\[
\text{edgerefs} = \text{global_id} \times \text{global_id} \rightarrow \text{deref\_count}
\]

The ptrs structure, is a collection keyed by unique node identifiers and contains the global pointer mapping from the profile run to the new node location in the partitioned layout. This structure also keeps a boolean flag assigned, which is initially false, and set once the node has been placed into a chunk within the layout algorithm.

\[
\text{ptrs} = \text{global_id} \rightarrow \text{old\_ptr} \times \text{new\_ptr} \times \text{assigned}
\]

3.2 Profile Gathering

The edgerefs structure is updated throughout the profile phase of the computation. When the application dereferences a pointer contained within a global data element, a counter corresponding to the edge in edgerefs is incremented.

A key premise of this approach is the ability to identify a specific location in the tree which is stable between runs and over different phases of computation. We desire an approach which is stable in the presence of minor differences in the tree structures, which eliminates using global pointers for this purpose, a technique used by Chilimbi in his work [2].

For spatial tree applications, each tree node represents a specific domain in space, at least in a relative sense. (In Barnes-Hut, for example, the bounding box represented by the octree may change each iteration). Still, because there is a fixed out-degree of node, each node can be identified by either a level and index pair, or by enumerating all possible locations in the tree and using this index to identify a node.

A more general approach is to uniquely identify a node by its path from the root of the tree to itself. The path consists of the indices of each child pointer that would need to be followed to find the given node. This has the advantage that it can be used to determine the unique identifiers of parents and children. This identifier may be very long, however. Additionally, applications may also have specific techniques for uniquely identifying nodes, which could also be used as a key into the edgerefs mapping.

3.3 Element Mapping Algorithm

During a profile run, global pointers are dereferenced using the GCL API and the data reference counts are stored in the edgerefs structure.

The element partitioning algorithm has the following steps:

1. Sort edgerefs in descending order as edgerefs'.
2. Initialize the ptrs structure from edgerefs.
3. Partition the nodes into logical chunks.
4. Assign the chunks to specific processors.
5. Merge small chunks together.
6. Assign all nodes physical <chunk, offset> addresses.
7. Emit the allocation map to all processes.

First, the list of edges is sorted in descending order of the reference counts for each edge access and stored in edgerefs'. Each vertex of the edge is stored in the ptrs structure, initially unallocated. Next, the vertices of the edges in edgerefs' are co-located in the same chunk, in descending reference count order. This algorithm is detailed in Algorithm 1. The chunks that are assigned are logical chunks, which may be partially filled and have not been assigned to a specific processor. Once nodes have been placed in chunks, partially filled chunks are assigned to specific processors and merged together. Finally, the mapping of unique identifiers to partitioned chunk/offsets is output and is used for the data relayout operation or for allocations in the next iteration.

The first step is \(O(n \log n)\) in the number of data elements, which dominates performance of the algorithm. The other steps are either \(O(n)\) in the number of elements, or \(O(c \log c)\) in the number of chunks if sorting is used in step 5.

The partitioning algorithm takes a greedy approach, working through the list of edges with the highest reference counts first. Each vertex of the edge is checked to see if it has already been assigned to a chunk. If neither node has been assigned, a new chunk is created and both nodes are placed within it. If only one node has been placed in a new chunk, then we check to see if there is enough room in the same chunk, and the nodes are co-allocated. If there is not, the unassigned node is placed into a new chunk. If both nodes have been assigned, we attempt to merge the two chunks together if they are less than the max chunksize.
Algorithm 1 Element Mapping Algorithm

for edge((n1,n2) ∈ edgerefs) do
  if ¬ ptrs[n1].assigned ∧ ptrs[n2].assigned then
    // neither vertex has been assigned
    c = new-chunk()
    np1 = add-node(c,n1)
    np2 = add-node(c,n2)
    ptrs[n1].new_ptr = np1
    ptrs[n2].new_ptr = np2
  else if ¬ ptrs[n1].assigned then
    // only n2 has been assigned
    if size(chunk-of(n2)) < chunksize then
      np1 = add-node(chunk-of(n2),n1)
      ptrs[n1].new_ptr = np1
    else
      np1 = add-node(new-chunk(),n1)
      ptrs[n1].new_ptr = np1
    end if
  else if ¬ ptrs[n2].assigned then
    // only n1 has been assigned
    if size(chunk-of(n1)) < chunksize then
      np2 = add-node(chunk-of(n1),n2)
      ptrs[n2].new_ptr = np2
    else
      np2 = add-node(new-chunk(),n2)
      ptrs[n2].new_ptr = np2
    end if
  else if size(chunk-of(n1)) + size(chunk-of(n2)) < chunksize then
    merge-chunks(chunk-of(n1), chunk-of(n2))
  end if
end for

3.4 Chunk Mapping

The procedure listed in Algorithm 1 results in a set of chunks which have logical identifiers, but have not been assigned to a specific processor. Ideally, each chunk will be assigned to an owner which will minimize $\rho$, the number of non-local data accesses requiring communication. In practice, imperfect processor mappings can be tolerated as long as the benefits from reducing unnecessary data communication, using more relative pointers, and dynamic load balancing outweigh inaccuracies in the chunk/processor mapping.

There are several approaches to assigning chunks to processes, each with various advantages and disadvantages. The simplest technique is to round-robin each chunk to a different processor. This partitioning is very fast and balanced, but ignores processor-level data locality. Another approach is to use the inter-chunk edges between data elements to guide chunk placement. This technique is appropriate for use with locality-aware dynamic load balancing systems, but may suffer with static task mapping since chunks are placed without regard to processor locality.

An alternative to using inter-chunk edge references for chunk placement is to use the number of references made by each processor to govern placement. In this approach, an additional vector of counters is kept for each chunk which allows the layout engine to determine which process consumed the most data for a given chunk. This distribution can be implemented to provide a uniform data distribution, which may require limiting the number of chunks assigned to processor. Often, better performance is observed with a non-uniform data distribution due to the reduced communication. If this is acceptable, the partitioner iterates over each chunk and maps the chunk to the index corresponding to the maximum value (most accesses) of the chunk counter.

A related technique is to use a similar counter to track the allocating processor of each element in the chunk from the profile run. The chunk is then mapped to the processor with the most nodes allocated during the profile run. This approach can be highly effective in reducing remote node allocation when performing iterative profile-driven data layout.

3.5 Allocation in Iterative Applications

Once the element mapping process is completed, we are left with a $\text{ptrs}$ structure which maps global node identifier to a new global pointer which specifies the location that the node should be moved to. Many irregular scientific applications have an iterative nature, where the data structure is used to model a physical system over time. In the case of a dynamic system, for example, a tree is constructed every timestep representing the physical space with leaf nodes corresponding to particles (planets, stars, electrons, etc.). Since the simulation timestep is usually small, the tree structure remains similar from step to step.

We can use the profile gathered from one timestep to build a mapping which is used to specify locations to allocate specific nodes from during the next timestep. In this case, the mapping only is used to specify future allocations according to the data layout, so the $\text{ptrs}$ structure does not need to maintain the old $\text{ptr}$ information containing the previous location in the mapping.

This information is used with a specialized allocator which gathers the profile using the default allocation strategy. Once a sufficient profile has been collected, the layout engine creates the new element mapping to be utilized by the specialized allocator. When new elements are created, the allocator determines the unique identifier of the requested allocation and returns the location specified by the mapping to the application program. Data may be allocated remotely, adding communication overhead during subsequent creation phases.

3.6 Collective Data Relayout

To handle applications which do not follow this iterative structure, the GCL provides a collective data layout operation which re-organizes the data in the PGAS memory in-place. As with the iterative approach, the program operates in a profiling phase for a period of time, with new elements placed using the default allocator. The profiling phase is specified explicitly by the application programmer. When a sufficient data reference profile has been generated, the layout engine develops a new element mapping, resulting in a $\text{ptrs}$ structure which specifies both old and new locations.

The collective data layout step is effectively a copy operation, with data being copied from old locations to the new ones in the element mapping. To avoid synchronization problems, this step requires enough free memory in the partitioned global address space to make a full copy of the tree. Since the layout engine knows the size of the global data structure, a sufficient number of free chunks are reserved by the mapping engine and used as targets to copy into during the data movement phase. Each element containing references to global data is updated by rewriting all
4. EVALUATION

The evaluation of the profile-driven layout technique comprises two sets of experiments. The first experiments use a benchmark which does breadth-first tree construction with a depth-first dominant access pattern. The second experiment uses a GT/GCL variant of the Barnes-Hut SPLASH-2 benchmark [9] and is compared with the allocation-order chunk mapping (local-open) described in [5, 6].

These experiments were conducted on the Glenn cluster at the Ohio Supercomputing Center. This cluster consists of 2.6 GHz quad-core AMD Opteron processors configured with 8GB of memory. The cluster is connected via a 20Gbps Infiniband interconnect.

4.1 Breadth-first Creation / Depth-first Access

The benchmark used for these experiments simulates an application which creates a tree incrementally, in a breadth-first style. As nodes are added to the tree, the top of the tree fills out first, expanding level-by-level. The dominant access pattern uses depth-first traversal, which is repeated many times, in a phase distinct from the tree creation. While this benchmark is designed to exhibit an extreme difference between creation and access patterns, large differences between creation and access patterns are not that uncommon in parallel tree-based algorithms such as R-trees or certain operators in MADNESS [3, 4]. The first traversal runs in the profile-gathering mode, then the program performs the collective re-layout operation, and runs the remaining traversals using the new data layout. These experiments are based on a tree with approximately 1.4 million nodes and a local chunk cache size of about 100MB.

Figure 1 shows that the performance of the profile-driven benchmark on a small number of processors. With a only a few processors, the profile-driven layout shows a higher number of chunk cache misses. As more processors are added, the profile-driven run stabilizes and the number of cache misses slightly increases, but much slower than the default mapping. These results show that the layout mapping is successful in placing data on the correct processor, especially at higher process counts.

4.2 Barnes-Hut

The Barnes-Hut algorithm solves the n-body problem to compute force interactions between n bodies in a k-dimensional body of space. For the SPLASH-2 benchmark, this is 3-dimensional space which is decomposed and represented using a shared octree structure. Initially, bodies in the system are assigned a position and mass from a known model, distributed among the processors. Each process owns several regions of space and is responsible for inserting bodies into the subtrees that correspond to these regions.

When computing the interaction, the particles are partitioned by a space filling curve, which places spatially close bodies on the same processor. Each process iterates over the bodies and performs a partial traversal of the octree, evaluating the force contributions from other bodies in the system. The experiments below were run on a Barnes-Hut simulation with 512k bodies, simulated for 10 timesteps.

Since the purpose of the simulation is to understand the behavior of the particles in the system, rapid changes to the bodies in space are undesirable. As such, the tree does not change drastically between each timestep. As shown in Figure 3, the tree retains over 80% of its structure over several timesteps. From experience, tree similarity does drop off with each subsequent timestep, but the decline is stable, keeping 75% tree similarity at 20 timesteps.

The tree represents a spatial domain, so two trees are similar when they have the nodes in the same locations. Similarity is computed by the counting the nodes that are the same in two trees, divided by the number of nodes in both trees. Alternate techniques exist for computing tree similarity for trees which do not have a fixed spatial representation [8].

The relative improvement in performance in Figure 4 compares both the default mapping against the profile-driven mapping. The profile-driven allocator shows between a 10-17% improvement in overall run-time.
Figure 3: Similarity in Tree Structure over Simulation Iterations

Figure 4: Barnes-Hut Relative Performance

Analysis of the program performance data shows that the number of cache misses between the local-open and profile-driven runs is roughly the same. This indicates that the distribution of data to processes done within the implementation of the Barnes-Hut algorithm between tree-creation and force calculation is quite good by default.

Since the allocation pattern differs from the access pattern, the performance improvement is realized through the more efficient use of relative global pointer accesses. This is demonstrated in Figure 5. As the processor count increases, relative pointer usage improves for both cases. Since the total tree data is fixed, each processor requires a smaller portion of the overall tree as the number of processors increases. Because these portions are smaller, there are fewer chunks (and inter-chunk links) to traverse and the effective impact of relative pointers increases.

5. CONCLUSION

Reducing communication in parallel programs is an important part of achieving high performance. In PGAS parallel programs that rely on irregular, unbalanced data structures, locality aware data distribution is key to good performance. This paper has described an algorithm for improving the mapping of fine-grained data elements to coarse-grained chunks for efficient communication. This technique has been shown to be effective as a tool for collective data relayout and also to improve data allocation in iterative applications.

6. REFERENCES