The Thesis Committee for Manoj Dhanapal
Certifies that this is the approved version of the following thesis:

Design and Implementation of Distributed Galois

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

____
Keshav Pingali

____
Calvin Lin

Design and Implementation of Distributed Galois

By

Manoj Dhanapal, B.E.; M.S.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

May 2013

Acknowledgements

I would like to thank the members of the team working with Prof. Keshav Pingali for their help and support on this project. I am grateful to Andrew Lenharth for guiding me through the course of the thesis and helping me understand the Galois system. Special thanks to Rupesh Nasre for his help and constant encouragement while writing the thesis report. Finally, I thank Prof. Keshav Pingali for giving me this opportunity to work with him and constant support during the project.

MANOJ DHANAPAL

The University of Texas at Austin

May 2013

Design and Implementation of Distributed Galois

Manoj Dhanapal, MSCompSci

The University of Texas at Austin, 2013

Supervisor: Keshav Pingali

The Galois system provides a solution to the hard problem of parallelizing irregular algorithms using amorphous data-parallelism. The present system works on the shared-memory programming model. The programming model has limitations on the memory and processing power available to the application. A scalable distributed parallelization tool would give the application access to a very large amount of memory and processing power by interconnecting computers through a network.

This thesis presents the design for a distributed execution programming model for the Galois system. This distributed Galois system is capable of executing irregular graph based algorithms on a distributed environment. The API and programming model of the new distributed system has been designed to mirror that of the existing shared-memory Galois. This was done to enable existing applications on shared memory applications to run on distributed Galois with minimal porting effort. Finally, two existing test cases have been implemented on distributed Galois and shown to scale with increasing number of hosts and threads.

Table of Contents

Chapter 1 Introduction

Algorithms that use pointer based data structures such as trees and graphs are irregular [3], the work done at runtime by these algorithms are dependent on the input data and the data structure used. These algorithms are prevalent in domains such as social network analysis, natural language processing, data-mining and compilers. Most irregular algorithms work on large sparse graphs.

For instance, the page ranking algorithm used at Google calculates the ranking for every page on the internet [2]. The input to this algorithm could be a graph representation of the Internet, with webpages as nodes and links between them as edges. Since each page links to just a few other pages, this graph is sparse.

The memory available on a computer may not be enough to fit all the input data for large irregular programs, like those used for partitioning large graphs. A possible solution to this problem is to use the memory available on more machines that are interconnected by a network. This will require a distributed implementation of the algorithm.

Regular problems that use dense array and matrix operations are well understood and techniques for parallelization exist even on distributed systems. However, the execution for irregular algorithms is dependent on the data and the structure of the data structure making it much harder to parallelize. The information needed to process each node cannot be determined statically because each node's neighborhood is only known at runtime.

The Galois runtime system [3] introduces a possible solution to parallelize irregular algorithms using amorphous data-parallelism. The Galois system, however, works only on shared memory machines. In shared memory systems, all processors can access every other processor's local memory. Hence in the existing implementation of Galois the amount of parallel processing is limited by the number of processing units available on the machine.

This thesis discusses the design and implementation of a distributed version of Galois that solves the dual problem of providing the application with more memory and processing power. This is done by taking advantage of the memory and processors available on another machine connected through a network. The goal of the thesis is to design and implement scalable algorithms on the distributed Galois system. The existing programming model and API of Galois are not modified in the distributed implementation.

Section 2 of the thesis describes the features and the design constraints of the distributed Galois system. In particular, the user's code for the existing shared-memory Galois implementation shouldn't have to be modified to take advantage of the distributed Galois system. In section 3 the Distributed Galois design is discussed in detail. The various components of the system and the implementation are also given. The next section is on the directory design that is central in the implementation of distributed Galois. The applications that have been implemented on distributed Galois are explained in section 5. The results of the algorithms are discussed in section 6. And the report is concluded in section 7.

Chapter 2 Features and Design Constraints

2.1 PROGRAMMING MODEL

2.1.1 Irregular Algorithms

The Galois system is designed to implement opportunistic parallelism called amorphous data-parallelism for irregular algorithms [4]. The data dependencies and available parallelism are determined by the input data in the case of irregular algorithms. Parallel irregular algorithms use data structures like graphs and trees. Many real world problems such as social network analysis and single source shortest path which use sparse graphs are examples of irregular algorithms.

2.1.2 Operator Formulation

Galois uses the operator formulation technique to solve irregular problems. The algorithm is composed of individual operations performed on the active nodes of the graph. The nodes or edges of the graph that are required for the computation of the active node constitute the neighborhood of the active node.

For instance, Dijkstra's single source shortest path algorithm [7] can be viewed as the application the relaxation operation on the active node. The active node in this case is the unvisited node that has the smallest distance. In Dijkstra's SSSP algorithm the relaxation operation performed at each step, checks the distances computed so far on all the adjacent nodes, to update the distance of the active node. Thus the neighborhood for the active node is the set of all its adjacent nodes and the edges connecting the active node to the adjacent node.

2.1.3 Amorphous data-parallelism

Galois implements amorphous data-parallelism on the input algorithm using the specified operator for the active nodes. Amorphous data-parallelism is obtained by speculative processing of the active nodes in parallel considering the neighborhoods of the active nodes. Active nodes with overlapping neighborhoods cannot be processed in parallel. So when two nodes with overlapping neighborhood are scheduled in parallel, one of the iterations is aborted. This aborted iteration is rescheduled later.

The iterations can only be aborted before any node in the neighborhood is modified. This is implemented using exclusive locks on the nodes. All the nodes in the neighborhood of the active node being processed are locked before use. Conflicts are detected when an iteration tries to lock a node that has already been locked by another iteration. Thus there is a cautious point before which all the objects used in the iteration should be locked. This cautious point demarcates the read and write phases before which all the nodes required for the iteration are read. This is required for the speculative execution of the active nodes in parallel.

2.1.4 Topology-driven and data-driven algorithms

The order in which the nodes become active in an algorithm can be used to classify the Galois algorithms as either topology-driven or data-driven. If the activity on an active node causes another node to become active the algorithm is data-driven. In topology-driven algorithms the graph determines the nodes that become active.

For example, in Dijkstra's single source shortest path algorithm [7], the node with the least tentative distance so far is selected as the active node. Thus the operation in the previous step is used to determine the active node in this step, showing that Dijkstra's algorithm is a data-driven algorithm. In data-driven algorithms the active nodes cannot be determined in advance. Execution of active nodes results in discovering more active nodes.

The Bellman-Ford single source shortest path algorithm [5] however, executes the operator on all the nodes in the graph at each step. Thus the active nodes for each step are determined solely by the graph. Thus the Bellman-Ford algorithm is topology-driven. The algorithm is also unordered where the active nodes can be handled in any order.

Amorphous data-parallelism by definition works well on topology-driven unordered algorithms like Bellman-Ford.

2.2 Constraints

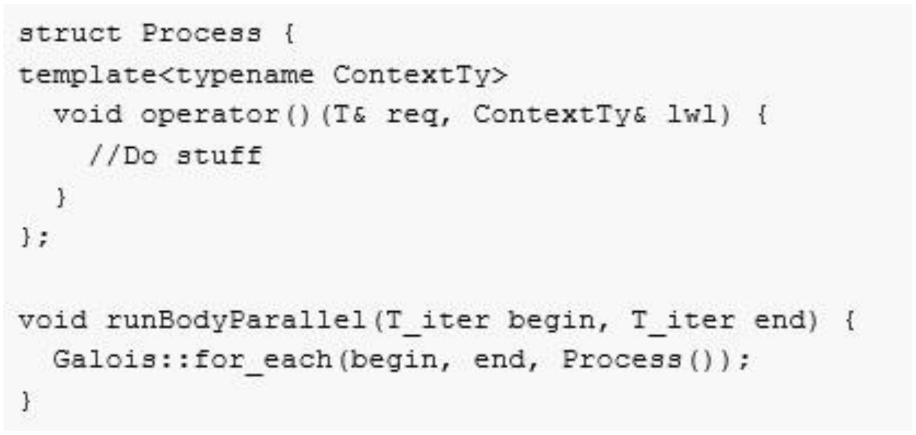
An important design choice made early during the development of distributed Galois was to continue with the existing programming model of the shared-memory Galois system. This means that distributed Galois should continue supporting operation formulation for the active nodes. The neighborhood constraints should still hold. Thus when an activity specified by the operator is performed the neighborhood of the active node should be locked exclusively.

Distributed Galois should allow amorphous data-parallelism among the active nodes. Thus speculative execution of the operator on the active nodes should not cause any problems. There should be a cautious point before which all the nodes in the neighborhood should be locked. The iterations should be aborted before the cautious point or proceed to completion, similar to the behavior in shared-memory Galois.

Chapter 3 Distributed Galois Design

3.1 Existing shared-memory Implementation

In shared-memory Galois, multiple threads work in parallel to execute the Galois algorithm. All the threads are involved in the execution of the Galois::for_each iterator. However, only the main thread executes all the other sections of the algorithm. So a Galois program is very similar to other sequential object-oriented programming languages like c++. The difference is mostly in the parallel for_each iterators.



```
struct Process {
template<typename ContextTy>
  void operator()(T& req, ContextTy& lwl) {
    //Do stuff
  }
};

void runBodyParallel(T_iter begin, T_iter end) {
  Galois::for_each(begin, end, Process());
}
```

Figure 3.1: Example for each operator

The above example shows an example of a for_each iterator. The for_each adds all the elements from begin to end in a worklist as the initial set of active nodes. The Process () operator is applied to each of these elements. The execution of these active nodes happens in parallel using the multiple threads. The user can specify the number of threads to be used for the execution.

The operator acts on a set of nodes. The nodes are accessed using a Galois provided data structure usually a graph. Every node accessed is locked exclusively in the threads context. If any node is already locked by another thread, this iteration is marked as a conflict and aborted. The aborted iterations are retried later.

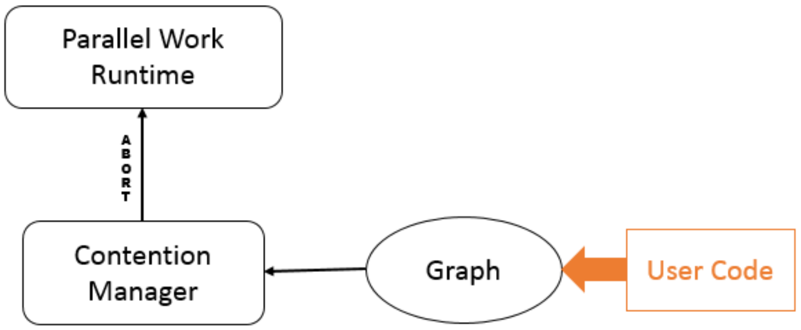


Figure 3.2: Interaction of the various components in shared memory Galois

For the algorithm to function correctly, the operator should have a cautious point before which all locks for the iteration should be acquired. Only after the cautious point, should the operator be allowed to modify the nodes in the graph. This is required as the operator should be allowed to abort only before the cautious point and should never abort after crossing the cautious point. This is required to enforce correctness in cases where the iteration has to be rescheduled.

3.2 Overview of the Distributed Design

The user interacts with the graph data structure for implementing the algorithms. This is the same as in the existing shared-memory Galois. An effort is made to hide the implementation details of distributed Galois completely from the user. So the distributed design strives to let already implemented applications on shared-memory Galois run on distributed Galois without any additional porting.

The user should be oblivious to the presence of local and remote data. This means that accessing both types of objects should be the same and resolved automatically whether local or remote. The user interacts with the underlying data structure (graph) to access the objects. This in turn speaks to the directory which resolves the object. This knowledge of whether the data is local or remote is encoded in the directory. However, accessing a remote object which has to be fetched by the directory, results in a conflict and aborts the iteration.

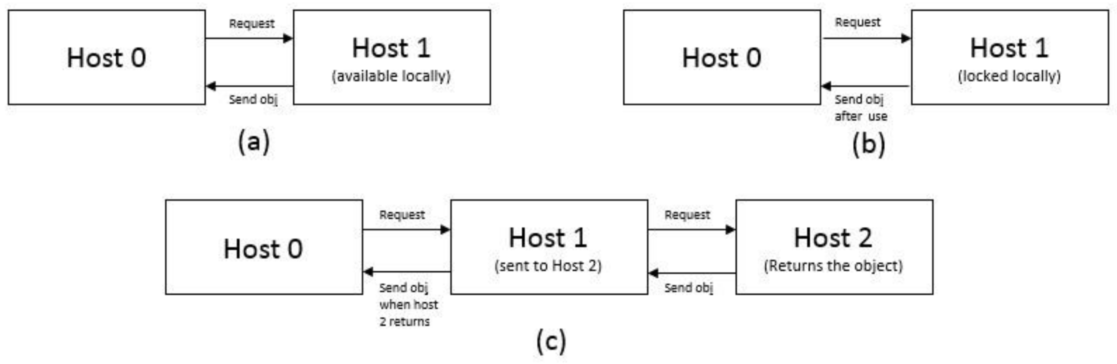


Figure 3.3: (a) the requested object is returned immediately, (b) the requested object is returned after local use, (c) the remote requested object is recalled from host 2

Accessing a remote object (say A) triggers a request to the owner of the object (say host 1) through the network layer. On the host 1 that owns the object, the network layer requests the local directory for the object. If the object is present, it is locked by the directory and sent to the requesting host 0. If the object however, is not available as it is used by a local iteration, the request is recorded by the directory and serviced at a later time. And in case the object is sent to another host 2, the request for the object A is recorded and the object is recalled from host 2.

The remote objects received by the directory are stored in the cache manager. The cache is implemented using a hash table.

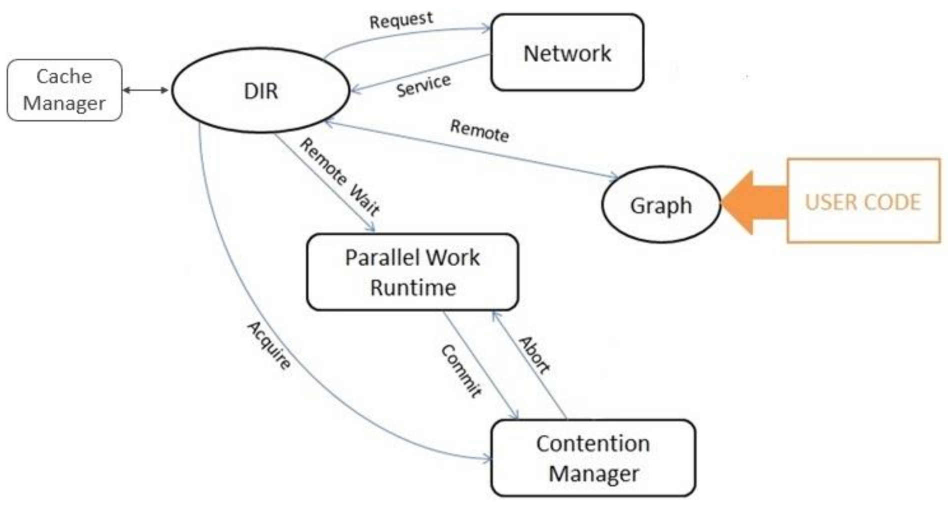


Figure 3.4: Various layers of Distributed Galois

3.3 IMPLEMENTATION

3.3.1 Granularity

The granularity of data communication in distributed Galois is an object (node of the graph). Whenever an iteration for an active node involves a remote object, it aborts the iteration after making a request for the object. These aborts due to remote objects required by the transaction are handled by a separate abort queue. Unlike local conflicts a remote conflict is rescheduled only when the remote object is received. Thus the Galois system makes sure that remote abort queue is not accessed until the requested remote object is received.

3.3.2 Remote Procedure Calls

The messages communicated between the hosts are for objects however, the data is communicated as a stream of bytes. This is a problem as the type of the object being passed has to be communicated to the receiver. To solve this issue a landing pad for the data is sent with every message. This landing pad is type aware and used by the receiving host to unpack the data.

Thus every message communicated executes a Remote Procedure on the receiving host. These RPCs are used to communicate the type of the object being sent to the host.

3.4 Types of Objects

The objects in distributed Galois can be classified as regular objects and persistent objects. The persistent objects are copied from the owner during their first access and then never modified or deleted by the remote host. These persistent objects are used for the implementation of distributed data structures. For instance, one implementation of a graph would be to store the graph as a persistent object. The persistent graph object could then point to the list of nodes on each host. This was exactly how the first version of the distributed graph was implemented.

The regular objects are local on the host that they are created and remote on the other hosts. In the case of algorithms which iterate through all the nodes at each step of the algorithm, a good optimization would be to partition the graph and store the different parts on different hosts. Each host can then iterate over their respective local nodes. The Bellman-Ford algorithm for single source shortest path [5] is an example for such an algorithm which iterates through all the nodes at each step. The neighborhood for each node is the set of all nodes connected by an edge. Hence in this example all the neighbors of the active node are locked locally to execute the iteration. If a node is locked exclusively by a remote host the iteration is aborted and a request is made for the remote object.

Remote objects can also be acquired in the shared state. In this case the object is acquired through the shared directory without acquiring an explicit lock in the local directory of the remote host. All the shared objects can then be released by the user. The shared state is useful in algorithms where there is contention on a few objects between the iterations. For example, the calculation of the forces using the Barnes-Hut n-body algorithm requires the traversal of an octree. Thus for every iteration the root of the tree is accessed, acquiring an exclusive lock will result in the serialization of the algorithm as all the iterations will have to get a lock on the root node. This issue is resolved using a shared lock on the root of the octree.

3.5 Network Layer

The network layer uses MPI for communication. This follows a single program multiple data execution model [6]. Thus an execution of distributed Galois consists of a number of processes (hosts) each executing the same executable. These hosts can communicate with each other and there is a unique rank associated with each host. This rank is used along with the object's address to form a global address space (Partitioned Global Address Space) to uniquely access any memory location in any host.

Any host can request for any object in the global address space. The request is forwarded to the corresponding owner (host) of the requested object.

Distributed Galois also supports multiple threads on every host. Most available MPI libraries however, are not thread-safe. Thus early in the design phase a choice was made to limit all the MPI calls to the main thread. The network layer has a queue for pending requests that are populated by all the threads. The first thread periodically checks the queue and places the corresponding MPI requests.

The Galois programming model dictates that only one thread should execute code that is not present in the for_each loop. Hence, except the main thread on the first host (host 0) all other threads loop over the communication layer monitoring for any for_each execution requests. Once the for_each is broadcast to all the hosts, the data for each host is communicated. The remote hosts then execute the cautious operator over all the given data.

3.6. Distributed Objects

In distributed Galois each running process has a virtual address space that is uniquely addressable by that process. Each location in the virtual address space maps to a particular location in the physical address space of the machine. These virtual addresses are used by the running process (host) to access an object in the memory. However the virtual address available on one host is also available on every other host.

Remote hosts can access a use a combination of MPI host id and the virtual address on the object in the process to uniquely identify any memory location on any running host. This leads to the implementation of distributed objects in the system using a Partitioned Global Address Space (PGAS) where each host has a local memory that can also be accessed by the remote hosts.

Take the simple example of finding the sum of numbers stored in an array distributed among all the hosts of the application. One way to solve this problem is to accumulate the sum of the elements in each host and eventually sum the accumulated values across all the hosts. This algorithm requires that the variables used to calculate the local sum in each host, be finally accessible by one host which calculates the final total. However these variables used to store the local sum are in the local address space of each host. Hence in this example, the virtual address space of each host should be accessible from other hosts. Distributed Galois requires a globally unique address space across all the hosts of the application. This ensures that any object created by a particular host can be accessed by all other hosts.

The distributed objects in Galois are implemented using a distributed pointer. Distributed pointers are a combination of the host id and the local address of the object on that host. The host id corresponds to the MPI rank of the process. The combination of host rank along with the object's address within the process gives a globally unique address space.

The virtual address available to a particular process however is accessible only by that owner process. To resolve this problem the dereference operator is overloaded for the distributed pointer. Every time an object is dereferenced the directory is checked to make sure that the object is locally available. If the object is locked exclusively by another host, a network request for the object is sent to the remote host. The iteration requesting this object is then aborted and rescheduled later when the object is available.

3.7. Serialization and deserialization

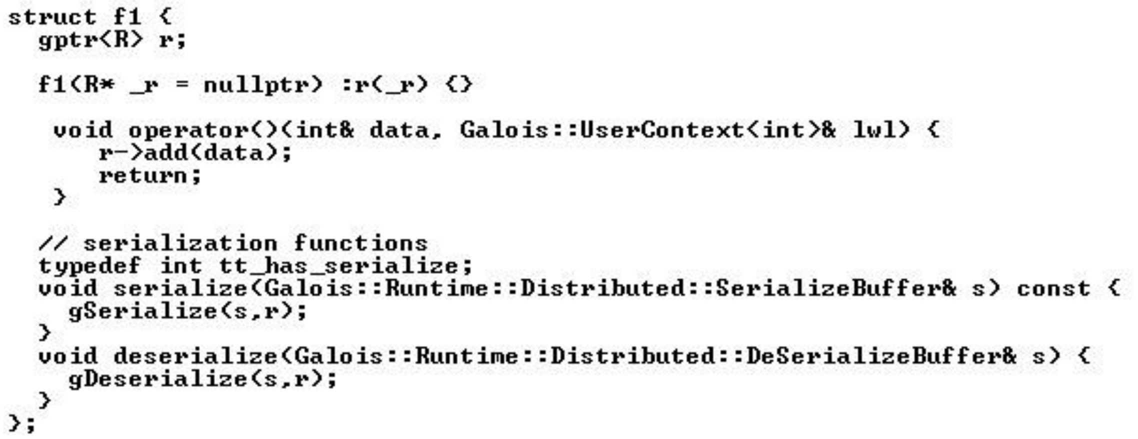
Distributed objects being transferred between the hosts have to be serialized by the sender and deserialized by the receiving host. Another requirement is that the object's type information has to be communicated to the receiving host. As discussed in section 3.3.2, type aware remote procedures are used to communicate the type information.

The need for serialization is unique to distributed Galois and not required on shared-memory Galois. Thus every object that is communicated between the hosts should also have the implementation of their corresponding serialization and deserialization functions. This is one change that the user should make for porting an existing shared-memory algorithm to distributed Galois.

The serialization process itself requires the encoding of individual fields of the object into a stream of characters. The character stream is sent to the remote host using an asynchronous MPI call. On the remote host the deserialization function for the object is called. The deserialization involves interpreting the received character stream to the corresponding fields.

The communication of plain old data types (pod) do not require the user to specify the serialization and deserialization functions, as they are interpreted as a stream of characters of the type. The user has to just specify the fields of the object that have to be serialized. These fields are recursively serialized until a type that is a plain old data type. Then the default implementation of the serialization function for the pod types is called for each of these fields. A similar recursive implementation is used for deserialization of the object at the host receiving the data.

Serialization of objects is used when a node required by an iteration is remote. The object is then serialized by the sender and deserialized on the host that receives the object. Cautious operators executed by the Galois::for_each iterators may also have local data that should be communicated by the main thread to the other hosts before the iterations are executed on the remote hosts.



```
struct f1 {
  gptr<R> r;

  f1(R* _r = nullptr) :r(_r) {}

   void operator()(int& data, Galois::UserContext<int>& lwl) {
      r->add(data);
      return;
   }

  // serialization functions
  typedef int tt_has_serialize;
  void serialize(Galois::Runtime::Distributed::SerializeBuffer& s) const {
    gSerialize(s,r);
  }
  void deserialize(Galois::Runtime::Distributed::DeSerializeBuffer& s) {
    gDeserialize(s,r);
  }
};
```

Figure 3.5: Example of a distributed operator (with the methods for serialization/deserialization)

In the above example the operator has a distributed object as local data that it updates while executing the operator on the active nodes. The distributed pointer (gptr) has to be communicated to all the hosts before they begin execution of the operator for their respective local active nodes.

Failure to communicate the distributed pointer will result in errors in the algorithm, as the different hosts will not use the same distributed object while executing the operator. To accomplish this task the distributed pointer is communicated to all the remote hosts as a first step. Thus the distributed pointer has to be serialized and deserialized.

Chapter 4 Directory

The distributed objects are mapped to local objects using a directory lookup. The directory is further subdivided into the following types, based on the type of the object being looked up.

4.1 Local Directory

The local directory is used to track objects that are owned by the host. Every time a local object is passed to a remote host an entry is made in the local directory. In shared memory Galois whenever an object is accessed by an iteration the object is first locked in the thread's context. If any other thread executing another iteration tries accessing the same object a conflict will be signaled and the second thread's iteration is aborted. This ensures that no other thread can access the object when being used by one iteration.

In distributed Galois, the object is locked by the local directory before being sent to another host, this prevents the remote object from being used in any local iterations. Local iterations trying to access this object locked by the local directory are aborted and scheduled later. This model of exclusive locking by the local directory in distributed Galois conforms to the programming model of shared memory Galois.

When an object sent to a remote host A is requested for another iteration in remote host B, the owner host C of the object sends a request to the host A for the object to be returned. Host B's request is also recorded and serviced when the remote host A returns the object. Since the requests for an object are remembered, no request for an object has to be resent to a host. This removes the need for extra communication between the hosts.

4.2 Remote Directory

The remote directory tracks the distributed objects not owned by the host but required for an iteration running in this host. The object is locked in the local directory of the owner and tracked using the remote directory on the remote host. This object is serialized by the owner and sent by the network layer to the requesting remote host. On the remote host a new object of the same type is allocated, initialized and the sent data is deserialized in the new object. The cache manager stores the mapping from the object's unique global address to this new local copy of the object.

When the owner requests for the remote object. A check is done to make sure the object is not locked locally by another iteration and sent back to the owner. If in case the object is locked locally the request is stored and serviced later when the object is released by the local iteration.

To ensure that forward progress occurs a remote object is locked in a special context by the remote directory of the requesting host. When the iteration which previously aborted on this lock is rescheduled this object's lock is transferred to the iteration's context. This is done to make sure that the application doesn't enter into a live lock when multiple iterations request for the same object.

4.3 Persistent Directory

There are objects that should be persistent in the host once it has been initialized. They are never modified or lost once they are copied from the owner until the application terminates. These objects can be used in the implementation of data structures. They are useful to store pointers to the location of other data that are not moved.

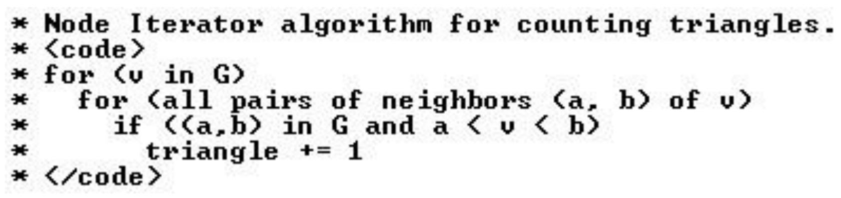
These persistent objects are stored using a persistent directory. When a remote persistent object is first accessed the object is copied locally and a mapping to the local object is stored in the persistent directory. Any subsequent access to the object will refer this mapping and return the local object.

Chapter 5 Applications

5.1 Triangles

The node iterator algorithm [12] used to find triangles in a graph was ported to Distributed Galois. The triangles in a graph are cycles in the graph formed by three nodes. Finding triangles in a graph has uses in various applications. The clustering coefficient of a node calculated on social networks shows how closely knit the community around a node. As described in [13] this computation can be reduced to counting the number of triangles incident on that node. Counting triangles also has various uses in the analysis of large networks. For instance, local clustering coefficient of a node gives the likelihood that two of its neighbors are connected.

The node iterator algorithm used, is specified in [12] as the "node-iterator" algorithm. The algorithm iterates over all the nodes in the graph, while checking if any pair of neighbors of the node are connected by an edge.



```
* Node Iterator algorithm for counting triangles.
* <code>
* for (v in G)
*   for (all pairs of neighbors (a, b) of v)
*     if ((a,b) in G and a < v < b)
*       triangle += 1
* </code>
```

Figure 5.1: Node Iterator Algorithm

The above figure gives the node iterator algorithm implemented in this test case. This is implemented using the edge iterators available from the graph implementation. All the nodes connected by edges to the active node constitute the neighborhood of the active node.

The algorithm is highly parallel on shared memory Galois as none of the nodes are modified during execution. However on distributed Galois some of the neighbors of the node may be remote. These remote neighbors will cause the iteration to abort while the remote object is fetched by the host.

5.2 Delaunay Mesh Refinement

The Delaunay Mesh is a mesh of triangles which satisfies the condition that no circumcircle of any triangle contains any other point from the mesh [4]. The algorithm generates a mesh which satisfies an additional quality constraint that no triangle has an angle less than 30 degrees. Triangles which do not satisfy this quality constraint are referred to as bad triangles. The algorithm works to remove all the bad triangles from the input mesh creating new triangles in the process. If any of the new triangles do not satisfy the quality constraint they are processed by the algorithm again. The mesh is modified till there are no bad triangles in the mesh.

The following steps are performed by the algorithm:

1. A bad triangle is selected to be worked on from the worklist.
2. The neighborhood (cavity) around the bad triangle that would be affected by the operation is discovered.
3. The cavity is worked on by the algorithm.
4. Any new bad triangles created by the operation are added to the workist.

The order in which the triangles are operated leads to different outputs. However any output generated by the algorithm that has no bad triangle is a valid solution.

```
Mesh m = /* read input mesh */;
Workset ws = new Workset(m.getBad());
foreach (Triangle t : ws) {
  Cavity c = new Cavity(t);
  c.expand();
  c.retriangulate();
  m.updateMesh(c);
  ws.add(c.getBad());
}
```

Figure 5.2: Psuedocode for Delaunay mesh refinement [8]

The first line reads the input mesh. In the second line all the bad triangles from the mesh are added to the worklist. Each bad triangle is then selected and worked on by the algorithm. The cavity for the bad triangle is created in the fourth line. This is then expanded to select the neighborhood in line five. New triangles are created to replace the selected cavity and the mesh is updated with these new triangles in the lines 6 and 7. Any new bad triangles created are added to the worklist the eighth line.

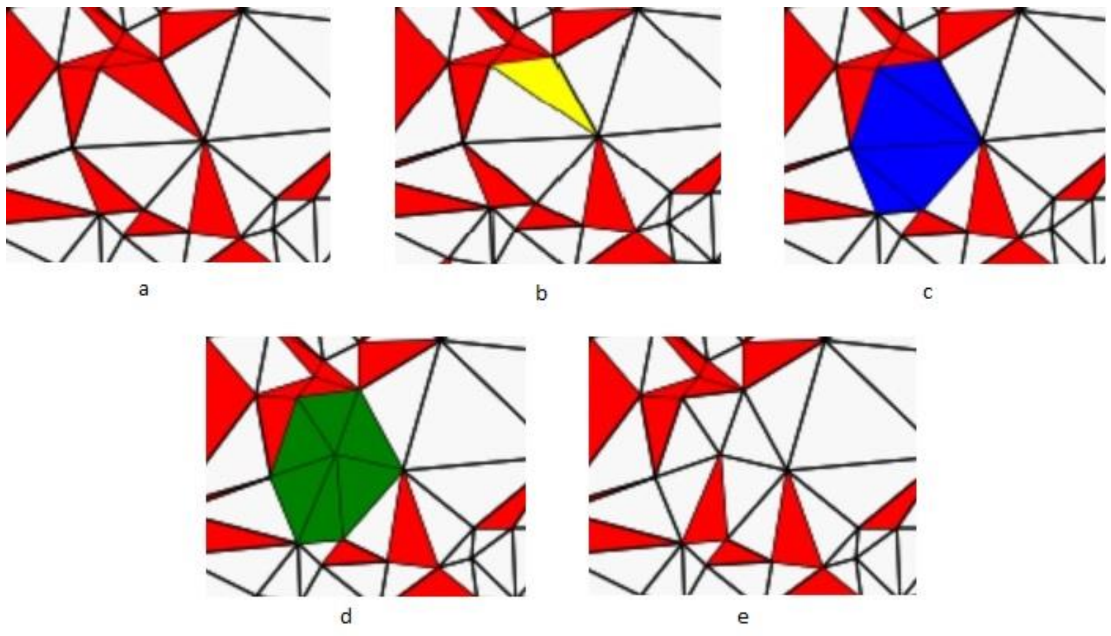


Figure 5.3: The above diagram shows the steps of dmr: (a) Shows the initial mesh, (b) Bad Triangle, (c) Cavity (d) Retriangulation and (e) new triangles

5.2.1 Selection of the Cavity

The neighborhood is selected by placing a new point in the circumcenter of the bad triangle being worked on [4]. As stated earlier the constraint for a Delaunay mesh is that no point of the mesh should lie in the circumcircle of the triangles in the mesh. All the triangles whose circumcircle contain this new point do not satisfy the Delaunay mesh constraint. All such triangles are selected in the cavity of the bad triangle.

5.2.2 Parallelism

The algorithm exhibits plenty of opportunities for exploiting parallelism. Any two bad triangles can be operated on in parallel as long as their respective neighborhoods do not overlap. This works well with the amorphous data parallelism used by Galois. The discovery of all the triangles that constitute the cavity corresponds to the cautious point before which all the elements (triangles) that are modified should be locked. After the cautious point is reached no more triangles that wasn't already locked can be modified. This condition holds for the algorithm and can be worked on in parallel using the operator formulation by Galois.

5.2.3 Partitioning the mesh

The inputted mesh is a 2-dimensional Delaunay mesh. As stated in the 5.1.2 section, this mesh can be worked on in parallel. However to minimize data movement between the hosts (processors) as much as possible the mesh is further divided into 2-dimentional grids. All triangles in a grid are stored together by the thread on the memory closest to the processor where the thread is bound. This reduces the access time for the thread to access the cavity which it is working on. Storing the spatially close triangles, together, reduces the chance of having to perform remote fetches on the triangles that are part of a cavity. This reduction in data movement improves the overall performance of the algorithm.

5.3 Optimizations for distributed Delaunay Mesh refinement

The performance of the resulting application depends on various optimizations. These optimizations also help improve the scaling of the algorithm across threads and hosts.

5.3.1 Prefetching the nodes

The first phase of graph creation involves dividing the input mesh into partitions with each partition storing the triangles spatially close to each other as explained in the previous section. Next the graph nodes (triangles) in the various partitions are created on the local memories of corresponding processors. The graph nodes are sent to the hosts running on the cluster if required, so that the nodes can be stored on the local memories of the remote processors. This is followed by the creation of the edges between the created nodes (triangle) of the graph. The nodes are moved around for the creation of edges. As edges, connecting nodes across partitions, require at least one node to be moved to a remote processor for the creation of the edge.

The subsequent process is to compute the list of bad triangles in the graph. Since this step just requires a computation on the triangles itself and not on the neighbors, they can be processed locally on the processor that is closest to the memory where the triangle is stored. This would reduce the communication of the graph nodes (triangles) in the subsequent steps thereby improving the performance.

5.3.2 Avoiding Livelocks

A problem noticed while executing Delaunay mesh refinement on distributed Galois was the possibility of livelocks. These live locks mostly occurred due to sending back the received data without any forward progress. This would result in the iteration going back to the previous state when the specified was requested to complete the iteration. The following approaches to avoid the live lock was applied on the system.

5.3.2.1Usage of a special context

Received objects are placed in a special context. They are then transferred to the context of the iteration which aborted as a result of requesting for this remote object. This ensures that the local iteration that aborted at least one chance to use the object before releasing the object. This however fails in the following case where two iterations on different hosts require the same 2 objects A and B but in different order.

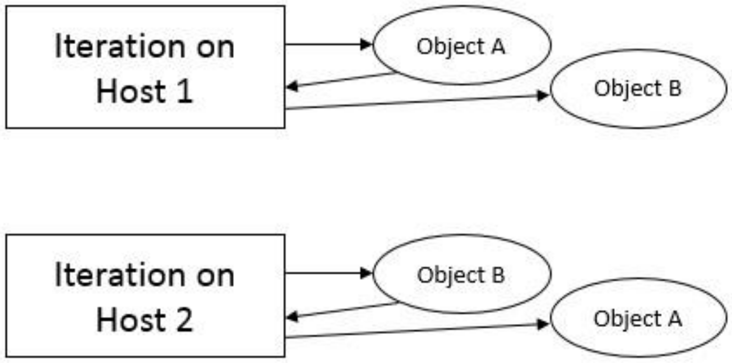


Figure 5.4: Iterations on host 1 and 2 request same Objects A and B but in different order

In the above figure the iteration on host 1 requests for the object A first and then after exclusively locking the object it requests object B. Similarly the iteration on host 2 requests object B first and then object A. There is a possibility of a livelock in this case where the first time the iterations execute they acquire objects A and B respectively and abort requesting for B and A objects. The next time these iterations are executed they have now the objects B and A locked in their respective contexts but fail again requesting for the other object. This cycle of can theoretically continue forever and the system is busy transferring the objects between the 2 hosts.

To solve this problem the iteration should not unlock the objects that it already held the last time the iteration executed. However this solution was not possible as the existing programming model in Galois requires that the thread's context be used for locks acquired while executing the iteration.

5.3.2.2 Global Lock

To avoid the livelocks in distributed Galois the use of a global lock was explored. This required detecting iterations that resulted in livelocks and executing them serially. The iterations are serialized using the global lock. Where the iterations that result in a livelock can be executed only if they hold the global lock. Failure to acquire the global lock will result in the iteration aborting and getting rescheduled.

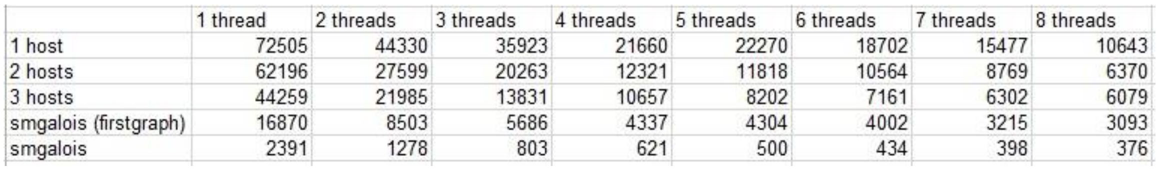
The lock count during aborts is used to determine if the iteration is involved in a livelock. For instance, in the above example when the iteration in host 1 aborts as object B is remote it has a lock count of 1 as it has object A locked in its context. However, when it runs again the object A is remote and this results in a lock count of zero. Thus whenever an iteration's lock count decreases in successive runs it indicates a possible livelock. This iteration will now require the global lock before it can proceed.

Chapter 6 Results

The performance and the correctness of the algorithms were checked on three very similar machine in the ices department. They have four Intel Xeon E7-4860 (2.27 GHz) processors which have 10 cores each. The machines have 128GB RAM each and they are interconnected by gigabit Ethernet. Gcc version 4.7.2 was used to compile the distributed Galois system. OpenMPI version 1.6 was used as the message passing interface.

6.1 Triangles

The input grid used for this test case was an ordered graph with 2e26 nodes. Each node approximately has 2 edges. The graphs are also ordered with neighboring nodes stored spatially closer to each other. This helps distribute the nodes across the various hosts and threads with very few edges that cross the different hosts.



	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
1 host	72505	44330	35923	21660	22270	18702	15477	10643
2 hosts	62196	27599	20263	12321	11818	10564	8769	6370
3 hosts	44259	21985	13831	10657	8202	7161	6302	6079
smgalois (firstgraph)	16870	8503	5686	4337	4304	4002	3215	3093
smgalois	2391	1278	803	621	500	434	398	376

Table 6.1: Runtimes of the Triangles application on an input with 2e26 nodes (in milliseconds). This is compared varying the threads on the specified number of hosts.

The above table shows the runtimes of the Triangles application on 1, 2 and 3 hosts compared with the performance with the implementation on shared-memory Galois. The last row gives the implementation on shared-memory Galois with the highly optimized NUMA graph. To compare the relative performance on distributed Galois the implementation using the similar FirstGraph on shared-memory Galois is also given.

The scaling in the runtimes as the number of threads/hosts are increased is given in the below figure. The scaling is given comparing the runtime for the specified number of threads to the single thread.

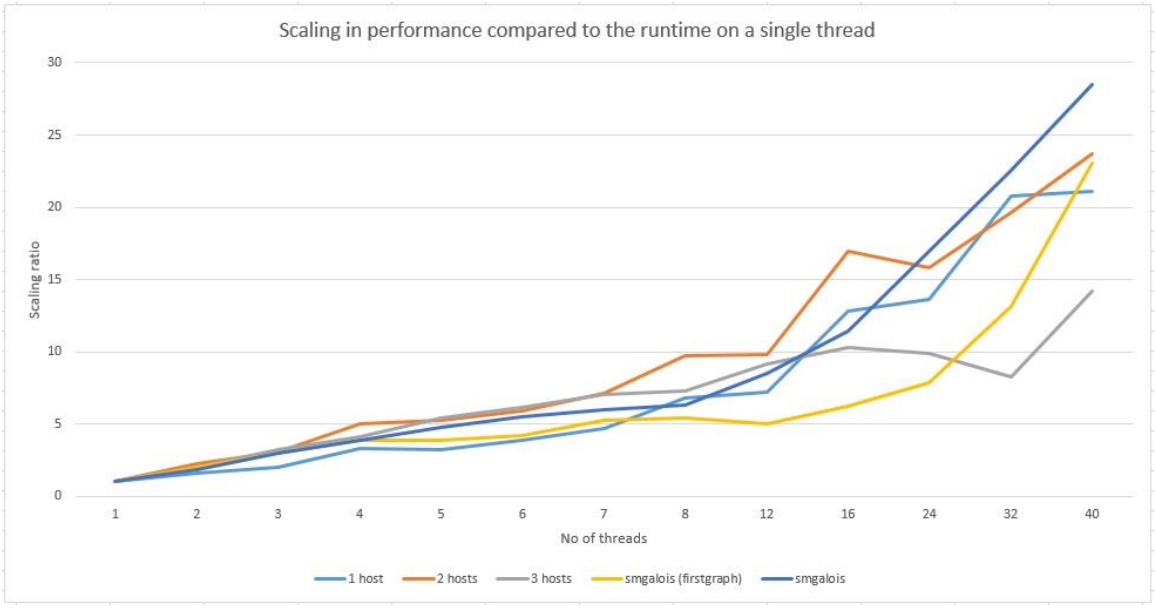


Figure 6.1: Scaling of performance as the number of threads are increased

As can be seen from the above chart the scaling is very similar to that seen on 1, 2 and 3 hosts compared to shared-memory Galois up to 16 threads on the hosts.

6.2 Delaunay Mesh Refinement

The input graph used for these tests contains 5 million triangles. The graph was generated by random triangularization of a square. This was done by the addition of a random point in the grid at each step.

The below chart gives the performance of distributed Galois comparing it with the performance of existing shared memory Galois. The plots are for the runs on 1 host, 2 hosts and 3 hosts for distributed Galois compared with the runs on shared memory Galois for the same input of 5 million nodes. The change in performance as threads are added closely follows the pattern of shared memory Galois.

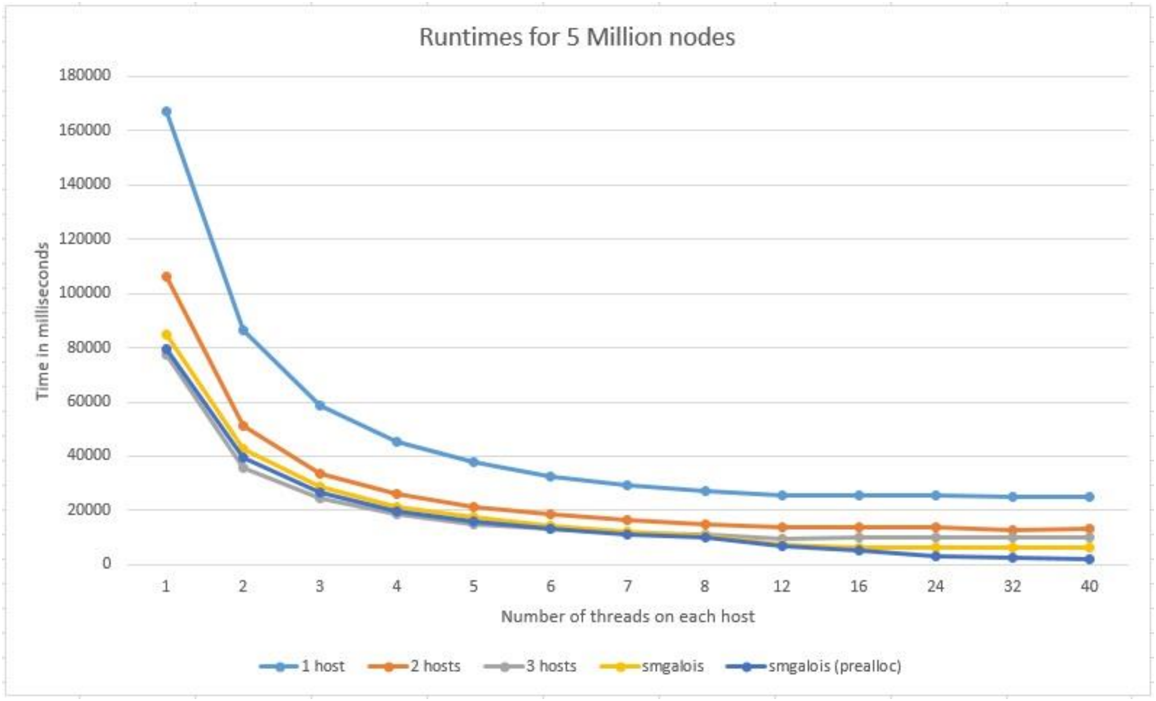


Figure 6.2: Runtimes for a graph with 5 million nodes

The below table lists the runtimes for the various hosts varying the number of threads executed per host. The performance of distributed Galois on one host is around half when compared to that on shared memory Galois for the same number of threads.

	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads	7 threads	8 threads
1 host	167013	86376	58672	45441	37624	32570	29189	27180
2 hosts	106086	51089	33816	26022	21246	18529	16455	14900
3 hosts	77595	35858	24686	18593	14967	13127	11418	10925
smgalois	84660	42554	28765	21274	17280	14229	12222	10750
(prealloc)	79451	39633	26470	19909	15859	13224	11349	9946

Table 6.2: Runtimes in milliseconds for the 5 million node grid

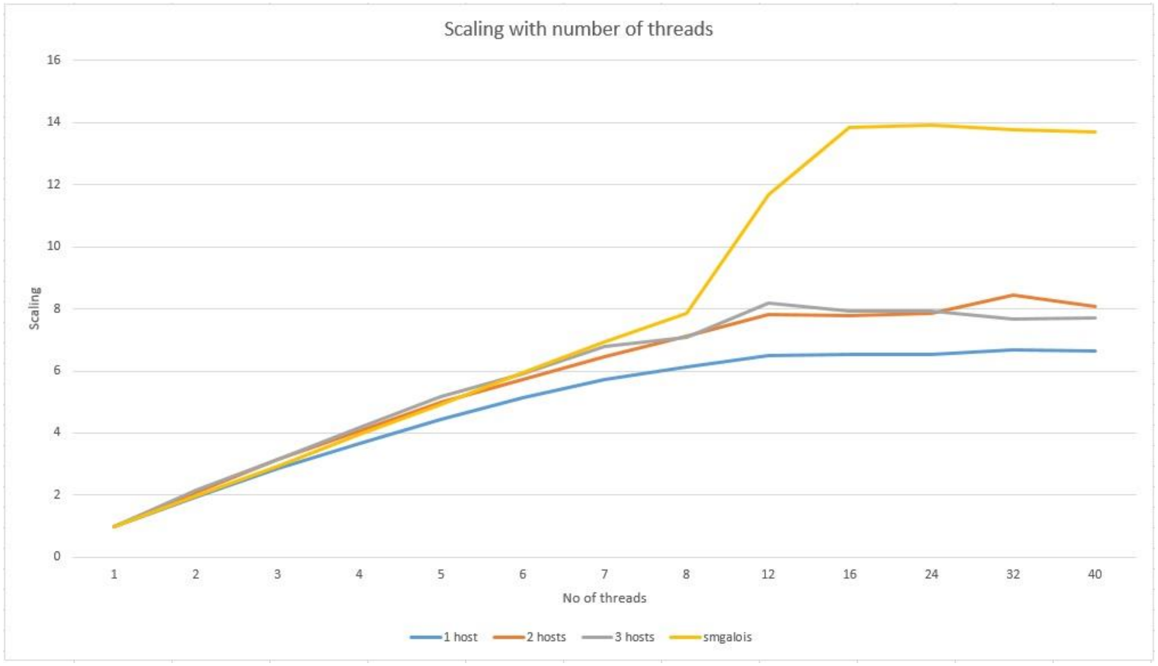


Figure 6.3: Scaling relative to runtime with one thread

The scaling in performance as the number of threads are increased follows the scaling exhibited by shared memory Galois. However, shared memory Galois continues to scale as the number of threads are increased to 16 threads. Distributed Galois stops scaling when the number of threads are increased to around 8 threads. This can be attributed to the failure to pre allocate memory in distributed Galois.

Chapter 7 Related Work

The prevalence of shared-memory multiprocessors, where each processor can access the local memory of every other processor, has introduced the need for a parallel shared-memory programming model. This has been implemented by systems like OpenMP [9] which provide a set of compiler directives and library functions that provide parallel runtime behavior. OpenMP can be used for multithreading the iterations of a loop inside an application. So existing sequential code can be easily modified to execute sections of it in parallel. However OpenMP works on predetermined iterations of the loop and new iterations cannot be added during execution. OpenMP also assumes that there are no dependencies between the iterations. This works well for simple loops however complex irregular algorithms cannot make use of these constructs due to conflicting neighborhoods between the iterations. Another limitation of the OpenMP model is that it works only on the shared memory model which places a restriction on the number of processors the program can scale to.

To scale to the distributed memory model, there are a group of parallel programming languages that make use of the Partitioned Global Address Space [10]. These languages provide a global view of the available memory using the partitioned local memory of the individual processes or threads. This is usually implemented with the SPMD (Single Program, Multiple Data) model where multiple tasks (processes or threads) are spawned on the available processors. The user is oblivious to the implicit communication which is triggered by accessing the non-local data [6]. Thus they provide a higher-level mechanism for specifying communication. This class of languages provide shared memory semantics on distributed memory architectures that traditionally use explicit message passing to communicate.

Cilk is also a system for multithreaded parallelization [15]. The programmer marks the code that can be executed in parallel using a set of Cilk keywords. Removing these keywords, results in a valid sequential C program called the serial elison of the Cilk program. The compiler and the runtime are responsible for scheduling the threads. It uses work sharing and stealing techniques to balance the computational load on the system.

Unified Parallel C is a PGAS language which lets the users have control over data placement and load balancing. UPC also allows remote accesses using language-level one-sided communication [11]. UPC provides the user with shared and private memories. All threads have access to read and write objects in the shared address space. However, accessing a shared object is significantly slower than the private objects. The user can control the placement of data using shared arrays where the array is distributed as blocks over all the threads in a cyclic manner. UPC provides calls for explicitly transferring of blocks of data from one thread to another. This lets the user retain control over the communication of the data.

The programmer decomposes the computation into objects which are then scheduled on virtual processors by Charm++ [14]. These parallel tasks called chares are scheduled by the runtime system (RTS) on a physical processor based on the workload. The communication between the chares is also mediated by the RTS. Hence the communication and the load balancing are optimized and handled automatically by the runtime system (RTS). Messages during communication result in function calls on the destination chare. Thus the execution model depends on the messages being sent by the chares in the system. These messages trigger execution of an entry method.

Chapel stands for Cascade High Productivity Language [10]. Chapel diverges from most other parallel languages as it doesn’t build on an existing language like C or Fortran. Locales are the computational units in Chapel which provide the high level abstractions for parallelism. Every program is provided with an array of locales which present the underlying machine. Domains are a language construct used to distribute arrays on the various locales. Locales and domain distributions help the developer control where the data is stored. Domains are derived from the regions concept developed in ZPL. Parallel tasks are coordinated using synchronization variables.

Chapter 8 Future Work and Conclusion

The present system requires a distributed worklist to handle load balancing across different hosts. This was not required for the applications discussed in this report. The present system uses a global lock for avoiding livelocks across the hosts. This solution becomes a severe overhead as the number of hosts increase. A distributed worklist may help resolve this problem of livelock by moving the conflicting iterations onto a single host and executing the iterations sequentially on that host.

The aim of the thesis was to design and implement a basic distributed execution model for the Galois system for irregular graph based algorithms. A limitation placed on the design was that it should mirror the existing programming model for shared-memory Galois. The final design as described in this report was modified extensively while implementing the Delaunay Mesh Refinement algorithm. The system's performance scales with increase in the number of threads and hosts on the applications analyzed so far. This implementation of distributed Galois can serve as a platform for the future development of other distributed applications.

References

[1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplied Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150

[2] Page L., Brin S., Motwani R. and Winograd T. The PageRank Citation Ranking: Bringing Order to the Web. Tech. Rep. 1999-66, Stanford University. Available on the Internet at http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

[3] Keshav Pingali, Donald Nguyen, Milind Kulkarni et al. The tao of parallelism in algorithms. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, pages 12-25, New York, NY, USA, 2011. ACM.

[4] Milind Kulkarni, Keshav Pingali, Bruce Walter et al. Optimistic Parallelism Requires Abstractions. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation Pages 211-222 ACM New York, NY, USA, 2007.

[5] A. V. Goldberg and T. Radzik. A Heuristic Improvement of the Bellman-Form Algorithm. Applied Math. Let., 6:3-6, 1993.

[6] Junchao Zhang, Babak Behzad, Marc Snir. Optimizing the Barnes-Hut Algorithm in UPC. In proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. Article No. 75 ACM New York, NY, USA, 2011.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms, MIT Press, Cambridge, MA, 2001.

[8] Galois benchmarks page. URL: http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/delaunay_mesh_refinement

[9] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. IEEE Computational Science and Engineering, Volume 5 (Issue 1):46–55, 1998.

[10] B. L. Chamberllain, D. Callahan and H. P. Zima. Parallel Programmability and the Chapel Language. The International Journal of High Performance Computing Applications, Volume 21, No. 3, Fall 2007, pp. 291-312.

[11] Wei-Yu Chen, Constin Lancu and Katherine Yelick. Communication Optimizations for UPC Applications. PACT '05 Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques Pages 267-278, IEEE Computer Society Washington, DC, USA 2005.

[12] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In Proceedings on the 4th International Workshop on Experimental and Efficient Algorithms (WEA '05), volume 3503 of Lecture Notes in Computer Science. Springer-Verlag, 2005.

[13] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In Proceedings of the 20th International Conference on World Wide Web. Pages 607-614, ACM New York, NY, USA 2011.

[14] L. V. Kale. Performance and productivity in parallel programming via processor virtualization. In Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10), Madrid, Spain, 2004.

[15] Frigo M, Leiserson C. E, Randall K. H. The implementation of the Cilk-5 multithreaded language. In Proceedings 1998 SIGPLAN Conf. Prog. Lang. Design Impl. (PLDI '98), pp. 212-223 (1998).