Agent and Spatial Based Parallelization of Biological Network Motif Search

Matthew Kipps, Wooyoung Kim, Munehiro Fukuda
Computing & Software Systems
University of Washington Bothell
18115 NE Campus Way, Bothell, WA 98011
{ matkips, kimw6, mfukuda }@uw.edu

***Abstract*— Most graph algorithms are challenging in parallelization, in particular executing fine-grain computation at each graph node in parallel from both programmability and performance viewpoints. To bridge the semantic gap between the original sequential algorithms and their corresponding parallelized programs, we have been developing MASS: a parallel library for multi-agent spatial simulation. The library allows software agents to crawl a distributed array, e.g., a graph mapped over a cluster system. To demonstrate the MASS library's fitness to graph parallelization, we have focused on biological network motif search. This paper compares MASS agent-based and the conventional MPI/OpenMP parallelizations, and discusses the MASS library's applicability to graph algorithms.**

I. Introduction

Graph algorithms, including information diffusion, subgraph and path finding, page ranking, and clustering can quickly exceed computing resources provided by a single computer and thus need parallelization for scaling up their problem size. It would be nice for developers to intuitively parallelize their graph programs using breadth-first search or exhaustive graph crawling, where computation at each graph node is executed in parallel by a different processor. However, such parallelization is not always intuitive, and does not always work effectively due to the nature of each node's fine-grain computation and hence with considerable overheads incurred by inter-node transitions of computation. Therefore, parallelization of graph algorithms in many cases ends up requiring drastic modification and tune-up of their sequential versions, using the conventional parallel-computing techniques such as multithreading and inter-process communication. This is a huge programming barrier to non-computing specialists.

Our research seeks to bridge this semantic gap in graph parallelization by mapping a graph to a cluster of computing nodes and walking a mega number of agents through the graph. We facilitated this agent-based graph algorithm with MASS, a parallel library for multi-agent spatial simulation. This paper intends to demonstrate the fitness of agent-based approach to graph parallelization, focusing on biological network motif search in particular finding all subgraphs in a target network.

II. Background

This section overviews network motif search, and differentiates our agent-based parallelization from related work.

A. Network Motifs

In the study of systems biology, there is interest in finding network motifs to examine how the molecular level of information relates to the system level. For example, biologists can model protein-protein interaction (PPI) networks as a set of interactions (edges) between proteins (vertices). This model can then be analyzed with motifs network, which might provide insight into connections between the molecular biology and systems biology.

A network motif is a significantly and uniquely recurring connected subgraph pattern within a target network [1]. Network motif finding is the process of determining these motifs through enumeration of subgraphs and statistical testing for the uniqueness of the subgraph pattern in random graph pools. In this paper we are focusing primarily on finding all subgraphs in the target network. We use the ESU (Enumerate Subgraph) algorithm [2] to efficiently traverse the target network to find all subgraphs of a given order. As shown in Fig. 1, to find all subgraphs with size k, ESU gets started with each vertex, v of a given graph (line 01), groups all vertices reachable from v as $V_{Extension}$ (line 03), chooses a vertex w from this group to create an ongoing subgraph $V_{Subgraph}$ (line E3), regroups a new extension, $V'_{Extension}$ reachable from $V_{Subgraph}$ (line E4), and recursively performs creating a subgraph and regrouping a new extension until the subgraph size reaches k (line E1).

```
Algorithm: EnumerateSubgraphs($G, k$) (esu)
Input: A graph $G = (V, E)$ and an integer $1 \leq k \leq |V|$.
Output: All size-$k$ subgraphs in $G$.
  01  for each vertex $v \in V$ do
  02      $V_{Extension} \leftarrow \{u \in N(\{v\}) : u > v\}$
  03      call ExtendSubgraph($\{v\}, V_{Extension}, v$)
  04  return
ExtendSubgraph($V_{Subgraph}, V_{Extension}, v$)
  E1  if $|V_{Subgraph}| = k$ then output $G[V_{Subgraph}]$ and return
  E2  while $V_{Extension} \neq \emptyset$ do
  E3      Remove an arbitrarily chosen vertex $w$ from $V_{Extension}$
  E4      $V'_{Extension} \leftarrow V_{Extension} \cup \{u \in N_{excl}(w, V_{Subgraph}) : u > v\}$
  E5      call ExtendSubgraph($V_{Subgraph} \cup \{w\}, V'_{Extension}, v$)
  E6  return
```

Fig. 1. The Enumerate Subgraph (ESU) algorithm [2].

B. Parallelization Approaches

Finding all subgraphs is an exponentially complex calculation with respect to the size of motif and size of the target network. First, the isomorphic testing is NP complete. Additionally, the number of non-isomorphic motifs grows exponentially with the size of the motif. For example, the number of size 10 undirected-non-isomorphic graphs reaches 11,716,571. [1]. Therefore, parallelization is a promising approach to finding subgraphs.

Most graph parallelization work has taken vertex-oriented approaches. MapReduce [3] performs computation at all vertices in parallel and thereafter combines the results from all the vertices, using the map() and reduce() functions respectively. Iterative MapReduce is required for some applications including page rank and shortest path search to repeat multiple steps, each forwarding the current computation at each vertex to its neighbors for the next step. Intermediate results must be sorted from map() to reduce() and fed back from reduce() to map(), which is considered a substantial overhead. Pregel [4] partitions a given graph into subgraphs, each allocated to a different worker machine that invokes the compute() function at all vertices within the given subgraph to update their states. Similarly to MapReduce, such updates must be diffused to neighboring vertices if applications consist of multiple computation steps, (i.e., called super-steps in Pregel), for whose purpose each vertex must exchange messages with their neighbors. GraphLab [5] is based on asynchronous distributed shared-memory paradigm that allows graph algorithms to access their local and adjacent vertices along their edges. No message send/receive or separate map/reduce functions are required. Therefore, algorithm designers can focus on their program development, isolated from data movement.

Contrary to these vertex-oriented approaches, flow-oriented approaches would work more intuitively for information diffusion or subgraph/path search, where algorithm designers write their programs from a car driver's viewpoint, in other words: as if computations drive from one to another vertex. Olden [6] facilitated thread migration over a graph distributed on top of a cluster system. Its threads can traverse a graph in parallel by cloning themselves upon encountering a vertex with multiple branches, using the futurecall() function. Since a new thread spawns along each edge, a large graph would results in considerable thread management overheads.

To remove these thread-incurred overheads, we modeled a graph as a distributed adjacency matrix and replaced migrating threads with mobile objects named ***agents*** that carry only their data members. Such agents can instantly jump to their next array element, (i.e, a vertex) for updating the vertex state, and migrate to a remote cluster node with other agents in an aggregated message. The next section describes this system.

III. MASS Library

The MASS library is a parallelization tool designed for **M**ulti-**A**gent **S**patial **S**imulation [7]. It uses a set of multi-threaded communicating processes that are forked across a cluster of computing nodes, using JSCH in Java and libssh2 in C++. MASS is designed around two key concepts: *Places* and *Agents*. Places is a multi-dimensional array of elements that are allocated over a cluster of multi-core computing nodes. Each element of a *Places* array is called a Place. Agents are a set of execution instances that can reside at a single place, access its public data/method members, migrate to other places, spawn child agents, and interact with other agents. A user designs a program by extending the Place and Agent base classes and specifying some behavior. Actual computation is performed between *MASS.init()* and *MASS.finish()*, using the following major methods, each performed in parallel.

Places Class

public Places(int handle, String className, int size...)   instantiates a shared array with *size* from *className*.
public Object[] callAll(int functionId, Object[] arguments)   calls the method specified with *functionId* of all elements as passing *arguments[i]* to element[i], and receives a return value into *Object[i]*.
public void exchangeAll(int handle, int functionId, Vector<int[]> destinations)   calls from each element to a given method of all neighbors, each indexed with a *Vector* element, and exchanges data among the elements.

Place Class

private size[]; private index[]   maintains the size of the shared array that each element belongs to and the index of each array element.

Agents Class

public Agents(int handle, String className, Places places)   populates agents from *className* onto a given places.
public Object callAll(int functionId, Object[] arguments)   is the same as *Places.callAll()*.
public void manageAll()   updates each agent's status, based on its latest calls of *migrate()*, *spawn()*, and *kill()*. These methods are invoked within *callAll()*.

Agent Class

migrate(int[] index...); spawn(int nChildren); kill( )   moves a calling *Agent* to a place specified with *index*, spawns children, and terminates the agent respectively.

IV. Parallelization

Our parallelization work focuses on finding all subgraphs. In the following, we examine three different parallelization approaches: (1) MASS agent-based, (2) MPI-based, and (3) MASS place-based parallelization. All parallelization work has been done in Java to maintain the compatibility with other graph-computing tools that we have used.

A. MASS Agent-based Parallelization

To parallelize subgraph finding with MASS agents, we define a *Place* subclass, *GraphNode*, as well as an *Agent* subclass, *GraphCrawler*. Each *GraphNode* represents a node (vertex) of the target network and maintains its emanating edges as an adjacency list. It also stores any subgraphs deposited by crawlers. On the other hand, each *GraphCrawler* represents an agent that will find a single subgraph enumeration within the network. This *GraphCrawler* behavior is described with pseudocode in Algorithm 1. Whenever a crawler has more than one choice to make, it spawns as many clones as additional choices, and each clone follows a branch. When a crawler completes its subgraph, the subgraph is collected at the *GraphNode*, and the crawler is terminated.

Fig. 2 shows the main program that initializes a single *GraphCrawler* at each *GraphNode* (lines 6-9), and updates the crawlers using *callAll()* (line 12) followed by *manageAll()* (line 13) until there are no more remaining *GraphCrawler*s. As described in section III, *Agents.manageAll()* updates *GraphCrawler* objects based on any calls to *migrate()*, *spawn()*, or *kill()*. When all crawlers are terminated, the subgraphs can be collected from the *GraphNode* objects (line 16), grouped into a single collection, and sent to *labelg* that groups subgraphs (line 18).

Algorithm 1: The pseudocode for the *GraphCrawler crawl()* method, which finds a unique subgraph within the target network.

```
node = current GraphNode;
add node to subgraph;
if subgraph is complete then
    send subgraph to node;
    // deposit it to node
    kill(); // terminate crawler
else
    update extension set per ESU algorithm;
    size = size of extension set;
    if size is 0 then
        // no more nodes to migrate to
        kill();
    else if size is 1 then
        migrate to extension[0];
    else
        for $i = 1; i < size; i++$ do
            spawn clone at extension[i];
        end
        migrate to extension[0];
    end
end
```

```
import MASS.*;
public void main(String[] args) {
   Graph input = ...; create graph object
   MASS.init(args); // start MASS
   // initialize simulation collections
   Places graph = new Places("GraphNode",
                              input.size());
   Agents crawlers = new Agents("GraphCrawler", graph,
                                 input.size());
   // enumerate subgraphs
   while (crawlers.nAgents() > 0) {
      crawlers.callAll(crawl_);
      crawlers.manageAll();
   }
   // collect the results
   Object[] results = graph.callAll(getSubgraphs_);
   MASS.finish(); // finish MASS
   ...; // use labelg to get isomorphs
}
```

Fig. 2. MASS agent-based parallelization

B. MPI-based Parallelization

The MPI-based implementation uses mpiJava [8] and Java threads to parallelize subgraph finding by partitioning the sequential approach to the ESU algorithm at the root subgraph level. For p processes across a cluster, given a target network of order n, each process is given a partition of n/p root vertices. Essentially, we parallelize the outer for-loop in Fig. 1.

Fig. 3 shows the simplified main program. After parsing the input file at rank 0 (line 3), *MPI.COMM_WORLD.Bcast()* distributes the network graph in its entirety throughout the cluster (line 6), such that there exists one copy of the network graph in memory at each cluster node. Each node determines the local partition using *MPI.COMM_WORLD.Rank()* and the size of the network graph. The partition for each cluster node is then dynamically distributed to the local computational threads at that cluster node. Each thread picks up an available root vertex from the local cluster node partition, and then extends all subgraphs from that root using a standard sequential ESU approach (lines 7-8). Whenever a subgraph is completed, it is saved in a synchronized collection. After enumerating all subgraphs for its partition, each cluster node uses *labelg* locally to get the isomorphism classes of the subgraphs, and then transfers the set of isomorphism classes to rank 0, using *MPI.COMM_WORLD.Gather()* (line 12). At rank 0, the isomorphism classes are aggregated.

```
1  import mpi.*;
2  public void main(String[] args) {
3     Graph input = ...; // create graph object
4     MPI.Init(args);
5     // distribute graph across the MPI cluster
6     MPI.COMM_WORLD.Bcast(input, masterRank);
7     ...; // start threads
8     ...; // join on threads
9     Map<String, Integer> isomorphs = ...; // from labelg
10    Object[] allIsomorphs =
11       new Object[MPI.COMM_WORLD.Size()];
12    MPI.COMM_WORLD.Gather(isomorphs, allIsomorphs,
13                          masterRank);
14    MPI.Finalize();
15 }
```

Fig. 3. MPI-based parallelization

C. MASS Place-based Parallelization

We implemented a third version of parallelization of subgraph finding, which used only MASS Places to mimic the concepts of the MPI-based implementation. Because the MASS library partitions Place objects evenly across the threads of a computing cluster, we can use this characteristic to distribute the work similar to MPI. We defined a simple Place object that represents a single iteration of the ESU for-loop (the outer loop of the algorithm in Fig. 1). The MASS place-based implementation of the ESU algorithm is almost identical to that of the MPI-based program. However, at the user level, the program does not implement any multi-threading or synchronization management that are supported by MASS.

V. Evaluation

In the following, we will compare MASS and MPI in terms of programmability and execution performance.

A. Programmability

We discuss the MASS programmability cover the following three measures: (1) **Separation between algorithm and parallelization logics:** The MASS library provide an easier object-oriented conceptual model for graph applications with MASS *Place* and *Agent* classes. On the other hand, MPI provides only communication utilities. The lack of abstraction also makes it harder to maintain clear separation of parallelization logic and algorithm logic. Particularly in the Java environment, the MPI-based version cannot use OpenMP for automatic loop parallelization. (2) **Intuitive flow-oriented approach to implement graph algorithms:** MASS agents facilitate the best intuitive programming model to implement graph algorithms such as information diffusion or subgraph/path search. Algorithm designers write their programs from a car driver's viewpoint, in other words: as if computations flows from one to another vertex. On the other hand, the MPI- and MASS place-based parallelizations are based on vertex-oriented approaches, where they perform examining vertices one after another from $V_{Extension}$. (3) **Flexibility in optimizing graph algorithms:** MPI provides a high degree of flexibility that allows algorithms to make specific optimizations. In MASS, we sacrifice the ability to implement behavior such as local and global dynamic network partition. Furthermore, it is difficult to control the rate of crawler creation, which leads to the memory issues that impact performance.

Overall, we feel that MASS eases parallelization of graph algorithms with an intuitive programming framework.

B. Execution Performance

For performance analysis, we used a cluster of 16 computing nodes, each with 4-core 3.40GHz CPU (*Intel i7-3770*) and 16 GB memory. Figs. 4, 5, and 6 shows that MASS agent-based implementation struggles to parallelize within the cluster environment, whereas both MPI-based and MASS place-based implementations demonstrate desirable parallelization characteristics. The primary problem is an explosion of agents, which results in poor memory usage. For the 2365-node network, with motif size 4, there were over 400,000 subgraphs, and with motif size 5, there were 5.5 million subgraphs. A significant amount of time is spent allocating and freeing memory to create or destroy these crawlers.

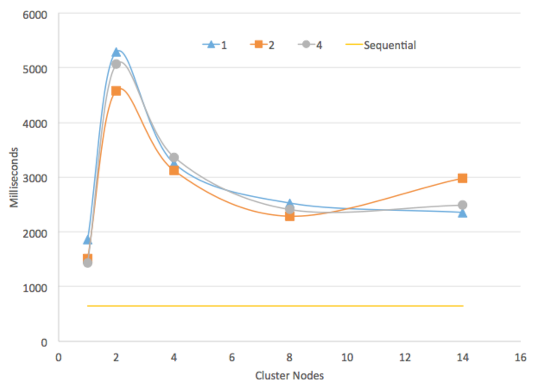

Fig. 4. MASS agents' performance. Network size is 2365; motif size is 4.

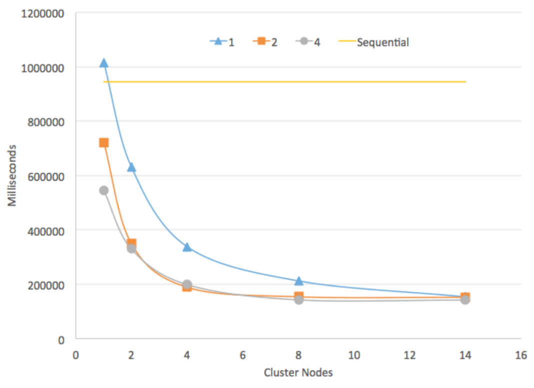

Fig. 5. MPI's performance. Network size is 5134; motif size is 5.

We are currently working on the MASS library's performance tune-up, particularly for its agent management: (1) **Asynchronous agent migration:** MASS agents need a synchronous *Agents.manageAll()* invocation for each cycle of their actual duplication, termination, and migration. This implementation increases communication from the main program to all remote MASS processes. Asynchronous migration without invoking *Agents.manageAll()* mitigates this communication overheads. Our latest experiment confirmed 2.5-time faster

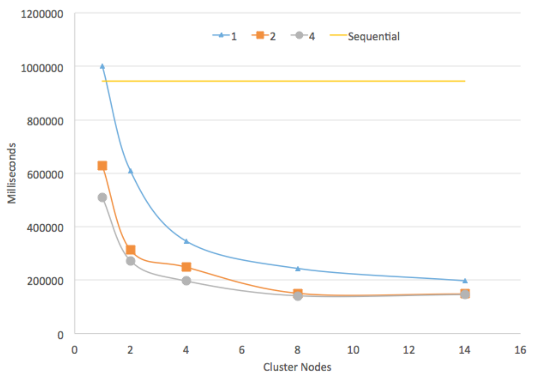

Fig. 6. MASS places' performance. Network size is 5134; motif size is 5.

performance than synchronous migration. (2) **Pool of idle agents:** A numerous repetition of memory allocation and deallocation kills multithreaded parallelization. Pooling idle agents is expected to mitigate this repetition so as to use heap space more effectively. (3) **Restriction of agent explosion:** The current implementation does not terminate an agent even when arriving at a place, (i.e., a vertex) that another agent has already visited, which tends to explode the number of alive agents. We will introduce an agent footprint to each place to terminate agents that found another agent's footprint.

VI. Conclusions

The MASS library can intuitively parallelize biological network motif search, using an agent-oriented approach, however it still needs to improve its execution performance. With the three techniques described in Section V-B, we feel that the MASS library will serve as a promising tool to parallelize graph algorithms intuitively with small performance penalty. At present, MASS is internally available at `http://depts.washington.edu/dslab/MASS/` but will be made available soon to the public.

References

[1] W. Kim, M. Diko, and K. Rawson, "Network motif detection: Algorithms, parallel and cloud computing, and related tools," *Tsinghua Science and Technology Journal*, vol. 18, no. 5, pp. 469–489, June 2011.

[2] S. Wernicke, "Efficient detection of network motifs," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 3, no. 4, pp. 347–359, Oct. 2006.

[3] Apache Hadoop, "http://hadoop.apache.org/."

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proc. of SIGMOD'10*. Indianapolis, IN: ACM, June 2010, pp. 135–145.

[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A Framework fro Machine Learning and Data Mining in the Cloud," in *Proc. of the 38th International Conference on Very Large Data Bases, Vol. 5, No. 8*. Istanbul, Turkey: VLDB Endowment, August 2012, pp. 716–727.

[6] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting Dyanmic Data Structures on Distributed-Memory Machines," *TOPLAS*, vol. Vol.17, no. No.2, pp. 233–263, March 1995.

[7] T. Chuang and M. Fukuda, "A parallel multi-agent spatial simulation environment for cluster systems," in *Proc. 16th IEEE International Conference on Computational Science and Engineering - CSE2013*. Sydney, Australia: IEEE CS, December 2013, pp. 140–153.

[8] mpiJava, "http://www.hpjava.org/mpijava.html."