

HipG: Parallel Processing of Large-Scale Graphs

Elzbieta Krepka, Thilo Kielmann, Wan Fokkink, Henri Bal
Department of Computer Science
VU University Amsterdam
1081 HV Amsterdam, Netherlands
{e.l.krepka,t.kielmann,w.j.fokkink,h.e.bal}@vu.nl

ABSTRACT

Distributed processing of real-world graphs is challenging due to their size and the inherent irregular structure of graph computations. We present HipG, a distributed framework that facilitates programming parallel graph algorithms by composing the parallel application automatically from the user-defined pieces of sequential work on graph nodes. To make the user code high-level, the framework provides a unified interface to executing methods on local and non-local graph nodes and an abstraction of exclusive execution. The graph computations are managed by logical objects called synchronizers, which we used, for example, to implement distributed divide-and-conquer decomposition into strongly connected components. The code written in HipG is independent of a particular graph representation, to the point that the graph can be created on-the-fly, i.e. by the algorithm that computes on this graph, which we used to implement a distributed model checker. HipG programs are in general short and elegant; they achieve good portability, memory utilization, and performance.

1. INTRODUCTION

The number of large real-world graphs is growing rapidly. For example, in 2010, Facebook, a popular social networking site, reached more than half a billion registered users [1]. In 2011, OpenStreetMap, a community-owned geographic data repository, reported more than a billion of nodes [2]. In 2008, in an official blog, Google reported indexing 1 trillion of unique web URLs [3]. And for decades now, the formal methods community has been verifying mission-critical protocols, with virtually unbounded state spaces [4–6]. With the increasing abundance of large graphs, there is a need for a parallel graph processing language that is easy-to-use, high-level, and both memory and computation efficient.

Because of their size, real-world graphs need to be partitioned between memories of multiple machines and processed in parallel in such a distributed environment. Real-world graphs tend to be sparse, as, for instance, the number

of links in a web page or the number of person’s friends are small compared to the size of the network. This allows for efficient storage of edges with their source nodes, *i.e.* as adjacency lists. Because of their size, partitioning graphs into chunks of balanced size and with a small number of edges spanning different chunks may be hard [7, 8].

Parallelizing graph algorithms is challenging. The computation is typically driven by a node-edge relation in an unstructured graph. Although the degree of parallelism is often considerable, the amount of computation per graph’s node is generally very small, and the communication overhead immense, especially when many edges span different graph chunks. Given the lack of structure of the computation, the computation is hard to partition and locality is affected [9]. In addition, on a distributed memory machine good load balancing is hard to obtain, because in general work cannot be migrated (part of the graph would have to be migrated and all workers informed).

While for sequential graph algorithms a few graph libraries exist, notably the Boost Graph Library [10], for parallel graph algorithms no standards have been established. The current state-of-the-art amongst users wanting to implement parallel graph algorithms is to either use the generic C++ Parallel Boost Graph Library (PBGL) [11, 12] or, most often, create ad-hoc implementations, which are usually structured around their communication scheme. Not only does the ad-hoc coding effort have to be repeated for each new algorithm, but it also results in obscuring the original elegant concept. A programmer spends considerable time tuning the communication, which is prone to errors. While it may result in a highly-optimized problem-tailored implementation, the code can only be maintained or modified with substantial effort.

In this paper we propose HipG¹, a distributed framework aimed at facilitating implementations of Hierarchical Parallel Graph algorithms that operate on large-scale graphs. Graphs can be read from disk, synthesized in memory or created on-the-fly during execution of the algorithm. They can be pre-partitioned by the user or partitioned automatically by the framework. Graphs are stored in memories of multiple machines that transparently communicate to execute graph algorithms. HipG delivers a unified interface to executing methods on local and non-local nodes, and thus allows implementing high-level fine-grained structure-driven graph

¹This paper is a modified (and further developed) version of an earlier conference paper [13].

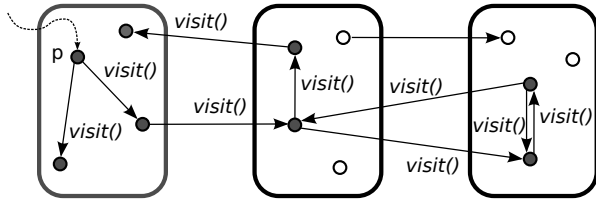


Figure 1: Reachability search from pivot p .

computations. Such computations are coordinated by logical objects called *synchronizers*. A HipG parallel program is composed automatically from the sequential-like components provided by the user: pieces of work on graph nodes and synchronizers that initiate such work. The basic model and interface of HipG are explained in Section 2. One of the examples used there implements the bulk synchronous parallel (BSP) [14] model, which thus can be easily expressed in HipG. The two sections that follow detail advanced uses of HipG. The HipG model supports, but is not limited to, creating divide-and-conquer graph algorithms, as synchronizers can spawn sub-synchronizers to solve graph sub-problems. In Section 3 we show how to implement a divide-and-conquer parallel decomposition of a graph into strongly-connected components. In this paper we extend the model of HipG [13] with the support for execution and implementation of algorithms operating on graphs that are generated on-the-fly. Section 4 presents how we used this new feature to create a distributed model checker: we implemented a distributed cycle detection algorithm [15].

Although the user must be aware that a HipG program runs in a distributed environment, the code is high-level: explicit communication is not exposed by the API, nor are the algorithms tied to graph representations. Parallel composition is done in a way that does not allow race conditions, so that no locks or thread synchronization code have to be implemented by the user. These facts, coupled with the use of an object-oriented language, makes for an easy-to-use, but expressive, language to code parallel graph algorithms.

We implemented HipG in Java. In Section 5 we discuss this choice as well as other implementation details, such as layout of the data structures used to store graphs (represented explicitly or generated on-the-fly) and organization of a worker. We evaluate performance of HipG in Section 6 on algorithms introduced in Sections 2-4. Using our newly-built cluster, DAS-4, we processed graphs of size of the order of 10^{10} (a magnitude larger than those used in [13]), and obtained good performance. The HipG code of the strongly connected components decomposition in HipG is an order of magnitude shorter than the hand-optimized C/MPI version of this program and three times shorter than the corresponding implementation in PBGL—See Section 7 for a discussion of the related work in the field of distributed graph processing. HipG’s current limitations and future work are discussed in the concluding Section 8.

2. BASIC HipG MODEL AND API

The input to a HipG program is a directed graph. HipG partitions the graph into equal-size *chunks*. A chunk is a set of graph nodes and their outgoing edges; in other words, edges are co-located with their source nodes. Undirected edges are modeled as two directed edges. Each node is an object con-

```

interface MyNode extends Node {
    public void visit();
}
class MyLocalNode implements MyNode
    extends LocalNode<MyNode> {
    boolean visited = false;
    public void visit() {
        if (!visited) {
            visited = true;
            for (int i=0; hasNeighbor(i); i++)
                neighbor(i).visit();
        }
    }
}

```

Figure 2: Reachability search in HipG.

taining arbitrary data and uniquely identified, for example by a pair (chunk, index). The target node of an edge is called a neighbor or a successor. Chunks are given to workers who are responsible for processing nodes that belong to them.

Graphs are typically processed by following their structure, i.e. the node-edge relationship. For example, an algorithm may start processing at a pivot node, then process its neighbors, the neighbors’ neighbors, etc., until no nodes are left to be processed. Figure 1 illustrates such a computation of a set of nodes reachable from a pivot. This fine-grained structure-driven graph processing is the most basic “primitive” of HipG. It is realized by providing the user with a unified interface to executing methods on local and non-local nodes, and a seamless access to a node’s list of neighbors.

Reachability search. Figure 2 displays the reachability search in HipG in Java. First, a *node interface* is defined, `MyNode`, telling HipG which methods can be executed on remote nodes. In general, methods listed in the node interface can be executed on any graph node of which the unique identifier is known. The `MyLocalNode` the *node implementation*. Each node has a flag that denotes whether it has been visited. The `visit()` method visits an unvisited node and its neighbors. The parts underlined in the code are provided or required by HipG, the remaining parts were created by the user. There are several essential observations to be made about the code in Figure 2. First, no locks or other methods of synchronization were needed; the exclusive access to the node is assured by the framework. Lack of synchronization makes the code look sequential and therefore easy to program; nevertheless, the user must be aware that the code will execute in a parallel setting: the order in which methods execute cannot be predicted and relied upon in the algorithm. Even on a single processor, HipG might reorder node methods calls to prevent stack overflow. Second, the layout of the graph data structures is not exposed to the user; in fact, not only may the actual data structure vary in various graph implementations, but parts of it might not even be created yet (see Section 4). Finally, the user did not need to provide different handling of local and non-local neighbors: access to all graph’s nodes is unified. All these facts make HipG node methods easy to read and high-level: the code reflects the algorithm behind it.

The algorithm in Figure 2 is initiated at the pivot node and terminates when all reachable nodes have been processed. In HipG this is written as:

```

pivot.visit();
barrier();

```

This code is, in fact, the simplest example of a *synchronizer*, a logical object that manages distributed computations. The three basic operations of a synchronizer are:

- (i) Initiating multiple distributed computations that execute in parallel or in sequence. For example, the call to `pivot.visit()` starts a wave of `visit()` method calls illustrated in Figure 1.
- (ii) Waiting for issued computations to terminate by calling a `barrier()`. The barrier blocks the synchronizer until all computations initiated by this synchronizer have completed. For example, the barrier after `pivot.visit()` blocks until all reached nodes have been visited and there are no `visit()`'s in transfer.
- (iii) Computing global results of distributed computations e.g. a globally elected pivot, or a size of a set of nodes partitioned between workers (see the next example).

One can imagine a synchronizer as an "agent" (or an *instance* of the synchronizer) on each worker that manages distributed graph computations on behalf of the user. In HipG, the user writes a single-threaded program while automatic parallelization is provided by the library.

Breadth-first search. Figure 3 shows the breadth-first search implemented in HipG. Its major part is the BFS synchronizer, which executes on each worker. BFS maintains a queue `Q` of nodes in the current layer, partitioned between the workers. The constructor adds the pivot to the queue at the worker that owns the pivot. The `run` method loops over the nodes in the current layer, and appends their unvisited neighbors to `Q` in the method `found` (not shown), in this way building the new layer. Note that the new elements may be added to queues on other workers. The `barrier` blocks until the new layer is fully created. Afterwards, the `GlobalQsize` method computes the global size of the new layer, and BFS terminates when all layers have been processed. Without the `Reduce` annotation, the call to `GlobalQsize` would be a regular method call returning the size of the local queue. With the annotation, it is a *global reduce* operation, which blocks until the sum of the sizes of all queues is computed. A single call to a reduce operation combines a partial result, supplied as an argument, with local data in a synchronizer and returns the combined value. The final value is a result of a chain of applications of the reduce method by all workers: each worker applies the reduce operation to a value from another worker, while the initiator applies it to the value supplied by the user. We note that (i) each worker executes the reduce method exactly once, (ii) the execution blocks until the result of the reduction is obtained, (iii) the final result is consistent across all workers, and, most importantly, (iv) the order of execution of reduce operations cannot be predicted and relied upon in the user's code.

We note that synchronizers only use high-level communication routines such as barriers and reduce operations. No other synchronization mechanisms are needed, even if there are multiple synchronizers per worker. Conceptually, the framework executes each `run` method sequentially, with ex-

```

class BFS extends Synchronizer {
    Queue<MyLocalNode>Q = new Queue<MyLocalNode>();
    int localQsize;

    public BFS(MyLocalNode pivot) {
        if (pivot != null) Q.add(pivot);
        localQsize = Q.size();
    }

    public void run() {
        int depth = 0;
        do {
            for (int i = 0; i < localQsize; i++)
                Q.pop().found(this, depth);
            depth++;
            barrier();
            localQsize = Q.size();
        } while (GlobalQsize(0) > 0);
    }
    @Reduce public long GlobalQsize(long partialQsize) {
        return partialQsize + localQsize;
    }
}

```

Figure 3: Breadth-first search in HipG.

clusive access to the synchronizer's data structures, and independently of other synchronizers. We observe that BFS alternates computation with global synchronization. Such algorithms are called bulk synchronous parallel (BSP) [14].

Lifting to parallel applications. In this section we described and gave examples of the two components of a HipG program that are defined by the user: node methods representing graph computations, and synchronizers orchestrating these computations. The two components are lifted automatically by HipG into a parallel application for a distributed-memory machine. At compile-time HipG translates method calls on non-local graph nodes into asynchronous messages. Since messages are asynchronous, methods do not return values. Returning a value of a method can be realized by sending a message back to the source, although, typically, the mechanism of computing global results by reduction is more efficient. All executions of methods on nodes, and messages representing them, form *distributed graph computations* managed by synchronizers. Each synchronizer is represented at each worker: each instance stores local state of computations and can manage local computations by communicating with other instances (this is also illustrated later in this paper in Figure 8). Each synchronizer has a unique id, determined on spawn, and consistent across all its instances. A typical HipG graph application starts by obtaining a graph, creates a synchronizer and waits until it terminates. In the two following sections we discuss more advanced programming with HipG.

3. DIVIDE-AND-CONQUER IN HipG

Divide-and-conquer graph algorithms divide computations on a graph into several sub-computations on sub-graphs. HipG enables creation of sub-algorithms by allowing synchronizers to spawn any number of *sub-synchronizers*. Therefore, a HipG algorithm is, in fact, a tree of executing synchronizers, and thus a hierarchy of distributed algorithms. Synchronizers can manage child synchronizers, for example wait for child termination. Unless explicitly synchro-

```

FB(V):
  p = pick a pivot from V
  F = FWD(p)
  B = BWD(p)
  Report  $(F \cap B)$  as SCC
  In parallel:
    FB( $F \setminus B$ )
    FB( $B \setminus F$ )
    FB( $V \setminus (F \cup B)$ )

```

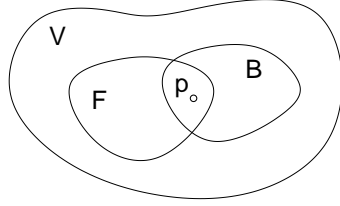


Figure 4: Divide-and-conquer SCC-decomposition.

nized, all synchronizers execute *independently* and *in parallel*. The user starts a graph algorithm by explicitly creating and spawning the root synchronizer. The system terminates when all synchronizers terminate. We illustrate divide-and-conquer graph algorithms in HipG with an example of decomposition into strongly-connected components.

Strongly-connected components. A strongly connected component (SCC) of a directed graph is a maximal set of nodes S such that there exists a path in S between any pair of nodes in S . We briefly describe FB [16], a divide-and-conquer graph algorithm for computing SCCs, and sketch its implementation in HipG. The concept is explained in Figure 4. FB partitions the problem of finding SCCs of a set of nodes V into three sub-problems on three disjoint subsets of V . First an arbitrary pivot node is selected from V . Two sets F and B are computed as the sets of nodes that are, respectively, forward reachable and backward reachable (i.e. reachable in the transposed graph) from the pivot. The set $F \cap B$ is an SCC. All SCCs remaining in V must be entirely contained either within $F \setminus B$ or within $B \setminus F$ or within the complement set $V \setminus (F \cup B)$.

The crucial part of a synchronizer FB is displayed in Figure 5. First, a global pivot is selected from V with the `SelectPivot` reduce operation. The pivot owner initializes forward and backward reachability searches that create sets F and B in V by flagging the reached nodes and storing them in separate queues. After F and B are fully computed, three sub-synchronizers are spawned to solve three sub-problems on $F \setminus B$, $B \setminus F$ and $V \setminus (F \cup B)$. To label sets of nodes uniquely and consistently across all workers, the synchronizer’s unique identifier was utilized. We note that this algorithm uses a transposed graph; the transpose has to either be provided by the user or can be created by HipG. The HipG interface contains counterpart routines that work on a transpose, prefixed with `in`, for example `inNeighbor`. Most importantly, we observe that Figure 5 elegantly reflects the algorithm in Figure 4. A corresponding C/MPI application (see Section 6) has over 1700 lines of code that entirely obscures the algorithm, the PBGL implementation has 341 lines, while the entire FB in HipG is only 113 lines.

4. ON-THE-FLY GRAPH ALGORITHMS

Encapsulating graph data structures and exposing only a high-level graph interface to the user, makes HipG highly malleable. Not only are algorithms not tied to particular graph representations, but also graphs can be created *on-the-fly*, i.e. during execution: a node is created on first access to it. This allows overlapping graph creation with computation for speed, and is essential in cases when the algorithm

```

class FB extends Synchronizer {
  ...
  public void run() {
    MyNode pivot = SelectPivot(null);
    if (pivot == null) return;
    if (pivot.isLocal()) {
      pivot.fwd(this, 2*getId());
      pivot.bwd(this, 2*getId()+1);
    }
    barrier();
    spawn(new FB(F \ B));
    spawn(new FB(B \ F));
    spawn(new FB(V \ (F \cup B)));
  }
}

```

Figure 5: FB algorithm in HipG.

only requires a part of the graph to execute, while the entire graph would not fit the memory. To execute an on-the-fly algorithm the user only provides a definition of a *next neighbor* function prior to execution. Using this feature we implemented a distributed model checker, which otherwise might have taken months to develop from scratch.

Distributed model checking. Model checking [4] is a widely-used technique that allows automatic verification of properties of computer programs (models) by systematically enumerating and examining their state spaces. The major problem this technique is facing is the *state explosion*: the state space grows exponentially with the number of variables or processes in the program to be checked. One way of alleviating this problem is to use distributed-memory model checkers, which make use of memories of multiple machines, but also render model checking algorithms more challenging. Such programs exist (see Section 7) and are typically large projects; the high-level API of HipG allows to vastly speed up development and try new algorithms with little effort.

We implemented *SpinJedi*, a distributed model checker based on SpinJa² [17], a recently developed clean Java reimplementation of Spin [5], the state-of-the-art sequential model checker. The input to the model checker is a Promela [5] file, which represents a multithreaded program augmented with assertions and a property to be checked. In general, model checking algorithms generate the state space of the model, while checking certain properties of generated states. In this section, a state of a program corresponds to a graph node, a state space corresponds to a graph, and checking properties of the state space to a graph algorithm.

Two algorithms play a major role in distributed enumerative LTL³ model checking; SpinJedi invokes one of them, depending on the options supplied by the user. The *on-the-fly visitor* is implemented similarly to the code in Figure 2, but augmented with safety checks (assertions, deadlocks) on visited states. The states are generated and checked until an error is found or the state-space is exhausted. The second algorithm checks properties of infinite executions of the model (for example a property that a certain “stable” state is

²SpinJa rhymes with Ninja; SpinJedi rhymes with Jedi

³LTL stands for Linear Temporal Logic – properties in Spin(Ja)(di) are expressed with the natural notion of linear time, i.e. words such as *next*, *until*, *always*, *eventually*

```

interface MapNode extends Node {
    public void map(MAP algo, long propag);
}
class LocalMapNode extends LocalNode<MapNode>
    implements MapNode {
    long map = BOTTOM;
    public void map(MAP algo, long propag) {
        if (algo.accCycle) return;
        if (id() == propag) {
            algo.accCycle();
        } else if (propag > map) {
            map = propag;
            if (accepting)
                propag = max(map, id());
            for (int i = 0; hasNeighbor(i) && !algo.accCycle; i++)
                neighbor(i).map(algo, propag);
        }
    }
}

```

Figure 6: Node implementation in the MAP.

eventually reached from any other state), which is more challenging. Brim *et al.* [15] show that this can be accomplished by searching for an *accepting cycle*. Namely, the Promela input file annotates certain states (graph nodes) as "bad", or *accepting*. Existence of an accepting cycle, i.e. a cycle that contains an accepting state, proves that the property under consideration does not hold. Therefore, the second algorithm is the distributed on-the-fly Maximal Accepting Predecessors (MAP) algorithm [15]. MAP assumes that the graph nodes have unique totally-ordered identifiers. It relies upon the observation that an accepting cycle exists if and only if there exists a node with itself as its maximal (with respect to id) accepting predecessor. Therefore, in each iteration, MAP computes the maximal accepting predecessor for each node. If one of the accepting nodes is its own maximal accepting predecessor, an accepting cycle is reported. Otherwise, nodes which cannot be on an accepting cycle are discarded. The algorithm is described in detail and its correctness proved in [15]. Next, we briefly discuss how it was implemented in HipG.

MAP. Figure 6 shows MAP's "primitive": computation of the maximal accepting predecessor for each graph node. The identifier of the current maximal accepting predecessor is stored in the variable `map`. If a node receives its own identifier, an accepting cycle is reported. Otherwise, only `map` values greater than the current value are accepted for propagation. An accepting node propagates the maximum of its `map` and its identifier. The computation terminates when all `map` values stabilize. The *global* MAP algorithm is displayed in Figure 7. In each iteration, it computes all `map` values (lines 9–11). If an accepting cycle was reported, MAP terminates (line 12). Otherwise, it discards nodes that cannot be on an accepting cycle (`map < id`) and restarts the `map` computation (lines 14–21). When no accepting cycle is found and the state space is exhausted and all accepting states discarded, no accepting cycle exists. We note that a node that has found an accepting cycle conveniently reports it to all workers with a `Notification` method. Without the annotation, the method would only execute locally. With the annotation, it executes on all workers, effectively notifying all instances of the synchronizer. Notifications do not block and are in general useful in many parallel search algorithms.

```

1 class MAP extends Synchronizer {
    Graph<MapNode> g; MapNode pivot;

    int accNodes = 0;
    boolean accCycle = false;

6     public void run() {
        do {
            if (pivot != null)
                g.node(pivot).map(this, 0, NIL);
11         barrier();
            if (accCycle) break;
            accNodes = 0;
            for (MapNode node : g) {
                node.map = BOTTOM;
16                 if (node.accepting) {
                    if (node.map < node.id())
                        node.accepting = false;
                    else accNodes++;
                }
            }
21         barrier();
        } while (GlobalAccNodes(0) > 0);
    }
    @Notification public void accCycle() {
26         accCycle = true;
    }
    @Reduce public long GlobalAccNodes(long s) {
        return s + accNodes;
    }
31 }

```

Figure 7: MAP algorithm in HipG.

5. IMPLEMENTATION

HipG is designed to execute in a distributed-memory (message-passing) environment. We chose to implement HipG in Java because of its portability and performance (due to the just-in-time compilation) as well as excellent software support of the language, although Java required us to carefully ensure that memory is utilized efficiently. We used the Ibis [18] message-passing communication library and the Java 6 virtual machine implemented by Sun [19].

A HipG program is executed by a number of workers. Each worker stores a single chunk of the graph. Logically, HipG executes a set of synchronizers in parallel. In this section we describe the implementation of workers and synchronizers, and briefly mention the compile-time instrumentation, that provides the syntactic sugar of a seamless graph interface without any language extensions.

Graph storage on a worker. The input to a HipG program is a directed graph (or graphs). If the graph is not pre-partitioned by the user, HipG partitions the graph into equal-size chunks by uniformly hashing each node to its owner. Currently the number of edges spanning different chunks is not minimized. Each worker stores the entire chunk in memory. A chunk is a collection of graph nodes and their outgoing edges. Two chunk layouts are currently implemented: *explicit* and *map*. In the *explicit* layout all nodes are stored in an array and uniquely identified by a pair of integers (worker, index). Edges on a worker are stored in two big global arrays, one for references to local nodes and one for identifiers of remote nodes. Although this structure is not elegant, it is transparent to the user and memory-

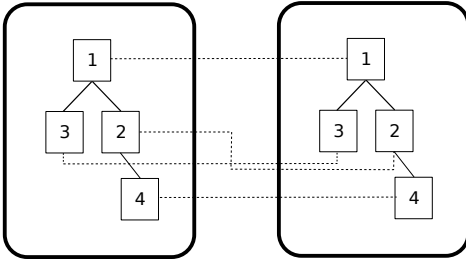


Figure 8: Tree of 4 synchronizers on 2 workers.

efficient, as it minimizes the prohibitive per-object memory overhead of garbage-collected languages (16 bytes per-object in 64-bit HotSpot). The explicit layout is efficient but difficult to modify at run-time, which is in contrast with the *map* implementation based on a hash table. The user defines a key of this table and a hashing method. The map representation is used in on-the-fly graph algorithms. In the distributed model checker that we implemented, the key used was a byte array that represents a state in the checked program. In the on-the-fly algorithms the edges are not stored, but generated by the user’s successor function. Last but not least, as most of the worker’s memory is used to store the graph, we tuned the garbage collector to use a relatively small young generation size (5–10% of the heap size).

Communication. The HipG workers communicate intensively all-to-all. Messages representing execution of methods on remote nodes account for the bulk of the traffic between the workers. Workers only execute methods on graph nodes that they own (“owner-computes”). Each method call belongs to some synchronizer, which is the first argument of the method. A message consists of an identifier of a synchronizer it belongs to, an identifier of a graph, an identifier of the target node, and serialized arguments. Arguments are serialized automatically, and we strive to make the serialization efficient. On reception of a message, the target node is retrieved; if it belongs to a graph generated on-the-fly, the node might not exist yet, in which case it is created and stored. Processing of a message consists in determining the method to execute and de-serializing its parameters. During the execution of the method, new node methods can be spawned. If such a method is called on a local node, it is executed right-away (until some depth), or stored on the synchronizer’s queue (when that depth is exceeded) to be processed later. If the method is called on a remote node, it is buffered for sending. The messages are combined in non-blocking buffers and flushed repeatedly by the worker’s sender thread. Asynchronous receiving is performed by a pool of threads provided by the Ibis communication library.

Synchronizer implementation. Each synchronizer has a unique identifier, determined on spawn by the worker with rank 0 and communicated to all workers. A synchronizer can spawn any number of sub-synchronizers, so it also maintains information about its father and children. Each of the synchronizers in the tree is represented on each worker. An example of this structure is shown in Figure 8, where numbers indicate synchronizer’s identifier and dotted lines represent instances of the same synchronizer. The execution of a synchronizer, i.e. its run method, can be understood as alternating communication phases, when methods on nodes are executed, and synchronization phases, i.e. blocking calls. The

blocking calls of synchronizers include barriers and reduce methods. Barriers are implemented with the token-based distributed termination detection algorithm by Safra [20]. When a barrier returns, it means that method calls that belong to the synchronizer have been processed. The reduce operation is also implemented by token traversal [21] and the result announced to all workers by worker with rank 0. Notifications are implemented as acknowledged asynchronous messages.

Worker implementation. After reading the graph, the user’s main program typically initiates root synchronizers, waits for all synchronizers to terminate, and handles the computed results. That part of the runtime that executes synchronizers we refer to as a *worker*. A worker is a single thread that stores all synchronizer instances and emulates execution of multiple independent synchronizers by looping over a queue of active synchronizers. If a synchronizer is ready to progress (a blocking routine has just terminated), the worker executes the next “step” of the run methods, or terminates the synchronizer, when no such step exists. A worker terminates when all synchronizers terminate.

Program instrumentation. Before executing, HipG programs have to be instrumented. The Ibis rewriter [18] optimizes object serialization. The HipG rewriter translates remote method calls into messages, and breaks down the run methods into “steps” at each blocking call. Just before finishing a step, the run method is automatically checkpointed; at the beginning of the next step, execution is restored. The user perceives a synchronizer’s run method as a thread. Thanks for instrumentation, the single worker thread executes all synchronizers, without the need for many context switches. Instrumentation is part of the provided HipG library, and needs to be called before execution. No special Java compiler is necessary.

HipG is released under GPL at www.cs.vu.nl/~ekr/HipG.

6. EVALUATION

In this section we report on the results of experiments conducted with HipG. The evaluation was carried out on the VU-cluster of the DAS-4 system [22]. The cluster consists of 74 dual quad-core 2.4 GHz Intel Xeon CPUs, each equipped with 12 GB of memory, thus 24 GB of memory per compute node. The processors are interconnected with 32Gbps-capable 4x QDR InfiniBand. The time to initialize workers and input graphs was not included in the measurements.

All graphs were partitioned randomly—meaning that if a graph is partitioned in p chunks, a graph node is assigned to a chunk with probability $\frac{1}{p}$. The portion of remote edges is thus $\frac{p-1}{p}$, which is very high (75-99% in used graphs) and realistic to model an unfavorable partitioning (many edges spanning different chunks). An advantage of this scheme is load-balancing: the numbers of edges stored at the workers are likely to be similar. We also note that in this setting, when computing a graph problem with twice more workers, $2 \cdot p$, the amount of computation stays constant, but the volume of communication increases by a factor $(1 + \frac{1}{2(p-1)})$. For $p \geq 4$, used in this evaluation, this factor is below 17%.

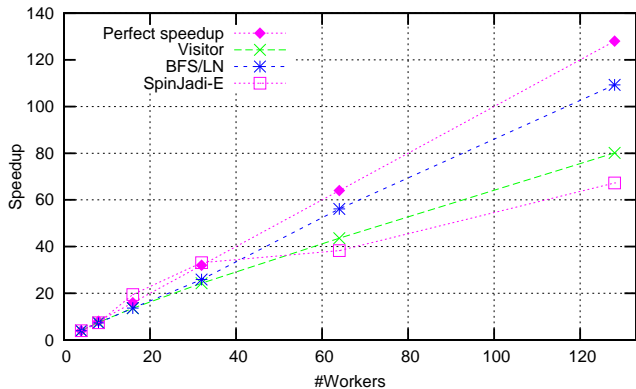


Figure 9: Speedup of Visitor, BFS and SpinJadi.

Message-based applications. We start with the evaluation of performance of applications that almost solely communicate (only one synchronizer spawned). VISITOR, the reachability search (see Figure 2) was started at the root node of a large binary tree directed towards the leaves. BFS, a breadth-first search (see Figure 3), was started at a random node of a synthetic social network. Both graphs are stored explicitly in memory prior to execution. The results are presented in Table 1 and Figure 9. We tested both applications on 2–64 processors, running two workers per compute code. To obtain more fair results, rather than keeping the problem size constant, we double the problem size when we double the number of workers. We note that this can only be done for very regular graphs and computation structure, and in this case we expect constant numbers in Table 1. Thanks to this we avoid spurious improvement due to better cache behavior, keep the heap filled, but also avoid too many small messages that occur if the stored portion of a graph is small. We normalized the results for the speedup computation (Figure 9) as follows. Let $T_p(s)$ denote execution time on p workers on problem of size s . The graph problem is “regular”, if solving a doubled problem with the same number of workers takes twice as long, i.e. $T_p(2s) = 2T_p(s)$, for any p, s , which is the case for VISITOR/Bin- n and BFS (statistically) on LN- n for large n . The speedup plotted is given by the formula $p \cdot T_{\min}(s_{\min}) / T_p(p \cdot s_{\min})$, equal thus to the “traditional” speedup formula.

For VISITOR we used binary trees, Bin- n , of height $n = 27..32$, i.e. up to $8.6 \cdot 10^9$ nodes and edges. The LN- n graphs used for BFS are random directed graphs with degrees of nodes sampled from the log-normal distribution $\text{Ln}\mathcal{N}(4.0, 1.3)$, aimed to resemble real-world social networks [23, 24]. An LN- n graph has $n \cdot 10^7$ nodes and expected $n \cdot 1.3 \cdot 10^9$ edges. We used LN- n graphs for n up to 64 and thus up to $6.4 \cdot 10^8$ nodes and $8.1 \cdot 10^9$ edges. In both experiments, all edges of the input graphs were visited. Both applications achieved at least 60% efficiency on 128 workers, which is satisfactory for applications with little computation, $\mathcal{O}(n)$, compared to $\mathcal{O}(n)$ communication. The efficiency achieved by BFS on LN- n graphs reaches almost 80%, as the input is more randomized, and has a small diameter compared to a binary tree, which reduces the number of barriers performed.

On-the-fly applications. We performed two kinds of evaluations of the distributed model checker (see Section 4): safety checking SPINJADI-E, which enumerates the entire

| Appl. | Workers | Input | Time[s] | Mem[GB] |
|----------|---------|------------|---------|---------|
| VISITOR | 4 | Bin-27 | 28.57 | 7.0 |
| VISITOR | 8 | Bin-28 | 30.36 | 7.1 |
| VISITOR | 16 | Bin-29 | 33.71 | 7.5 |
| VISITOR | 32 | Bin-30 | 37.68 | 7.7 |
| VISITOR | 64 | Bin-31 | 41.93 | 8.0 |
| VISITOR | 128 | Bin-32 | 45.64 | 8.1 |
| BFS | 4 | LN-2 | 118.62 | 8.4 |
| BFS | 8 | LN-4 | 129.39 | 8.7 |
| BFS | 16 | LN-8 | 138.83 | 8.8 |
| BFS | 32 | LN-16 | 146.72 | 9.2 |
| BFS | 64 | LN-32 | 135.25 | 9.5 |
| BFS | 128 | LN-64 | 138.95 | 9.6 |
| S.JADI-E | 4 | peterson.7 | 414.40 | 8.0 |
| S.JADI-E | 8 | peterson.7 | 224.64 | 6.2 |
| S.JADI-E | 16 | peterson.7 | 85.31 | 5.4 |
| S.JADI-E | 32 | peterson.7 | 70.03 | 5.2 |
| S.JADI-E | 64 | peterson.7 | 43.34 | 5.1 |
| S.JADI-E | 128 | peterson.7 | 24.65 | 5.2 |
| S.JADI-A | 4 | anderson.6 | 853.21 | 8.3 |
| S.JADI-A | 8 | anderson.6 | 534.11 | 6.6 |
| S.JADI-A | 16 | anderson.6 | 93.92 | 5.2 |
| S.JADI-A | 32 | anderson.6 | 47.33 | 4.9 |
| S.JADI-A | 64 | anderson.6 | 22.8 | 4.9 |
| S.JADI-A | 128 | anderson.6 | 7.95 | 5.0 |

Table 1: Performance of Visitor, BFS and SpinJadi.

state space (ignores errors), and SPINJADI-A, which searches for accepting cycles. We used some of the larger examples from the BEEM repository [25] of model checking benchmarks: Peterson’s mutual exclusion protocol for 7 processes and Anderson’s mutual exclusion protocol for 6 processes. The liveness property tested for the latter model was: if a process waits for entering a critical section, it will eventually get access to it. We tested both application on 4–128 workers, as presented in Table 1 and Figure 9. As expected, the performance of the SPINJADI-E scales similarly to VISITOR; the major difference between the VISITOR and SPINJADI-E is that the VISITOR allocates memory prior to execution (not timed), while SPINJADI-E allocates almost all memory during execution. Verification of the Peterson’s algorithm generates altogether 142 mln states, and 615 mln of transitions. The SPINJADI-A is a parallel search algorithm and it shows superlinear speedup. On the Anderson.7 example, the workers generate altogether about 50–120 mln of states, every time finding a bug in the model.

Synchronizer-based applications. To evaluate the performance of hierarchical graph algorithms written in HipG, we ran the OBFR-MP algorithm [26] that decomposes a graph into strongly connected components (SCCs). OBFR-MP is a divide-and-conquer algorithm like FB [16] (see Section 3), but processes the graph in layers. We compared the performance of the OBFR-MP implemented in HipG against a highly-optimized C/MPI version of this program used for performance evaluation in [26] and kindly provided to us by the authors. The HipG version was implemented to resemble the C/MPI version – the data structures used and messages sent are the same. Here we are *not* interested in the speedup of the decomposition algorithm, which may vary depending on the input [26]; rather, we want to see the difference in performance between an optimized C/MPI version and HipG version of the same application. We performed

| p | Myri | | | Eth | |
|-------------------|------|-------|------|-------|-------|
| | MX | OM | HipG | P4 | HipG |
| L487L487T5 | | | | | |
| 4 | 36.6 | 141.4 | 41.1 | 94.8 | 45.7 |
| 8 | 26.6 | 81.6 | 22.1 | 82.5 | 30.0 |
| 16 | 96.5 | 60.5 | 48.4 | 179.0 | 37.0 |
| 32 | 40.0 | 57.3 | 39.1 | 163.4 | 41.0 |
| 64 | 24.1 | 46.7 | 24.4 | 234.6 | 41.8 |
| L10L10T16 | | | | | |
| 4 | 69 | 255 | 148 | 302 | 225 |
| 8 | 73 | 280 | 226 | 462 | 330 |
| 16 | 89 | 376 | 315 | 804 | 506 |
| 32 | 136 | 661 | 485 | 1794 | 851 |
| 64 | 128 | 646 | 277 | 1659 | 461 |
| L60L60T11 | | | | | |
| 4 | 45.1 | 152.9 | 47.3 | 110.8 | 98.8 |
| 8 | 34.5 | 99.8 | 46.8 | 111.5 | 116.0 |
| 16 | 37.1 | 128.6 | 60.4 | 216.2 | 125.9 |
| 32 | 30.1 | 82.0 | 57.4 | 214.7 | 171.8 |
| 64 | 32.0 | 108.8 | 66.1 | 311.4 | 141.2 |

Table 2: Performance of OBFR-MP.

the experiments on the DAS-3 [27] cluster, which has less memory than DAS-4, but allows to perform a richer performance analysis. DAS-3 consists of 74 dual dual-core 2.4 GHz AMD Opteron with 4 GB of memory per compute node. The compute nodes are interconnected with 10G-Myrinet and 1G-Ethernet. We compare HipG against (i) MX, an MPI implementation from Myrinet tied to the interface, low-latency, almost unbeatable; (ii) OM, meaning OpenMPI, a newer implementation of MPI, socket-based, running over Myrinet; and (iii) P4, the standard implementation of MPI, running on Ethernet. We used up to 64 compute nodes and a single worker per compute node. We tested OBFR-MP on synthetic graphs called $LmLmTn$, which are in essence trees of height n of SCCs, such that each SCC is a lattice $(m+1) \times (m+1)$. An $LmLmTn$ graph has thus $(2^{n+1} - 1)$ SCCs, each of size $(m+1)^2$. The performance of the OBFR-MP algorithm inherently depends on the SCC-structure of the input graph, which is clearly visible in the MX, OM and P4 columns of Table 2. We used three graphs: one with a small number of large SCCs, L487L487T5; one with a large number of small SCCs, L10L10T16; and one that balances the number of SCCs and their size, L60L60T11. Each graph contains a little over $15 \cdot 10^6$ nodes and $45 \cdot 10^6$ edges. The performance of the C/MPI application running over MX is the fastest, as it has the smallest software stack. HipG performs, on average, 1.8 times slower than MX, but the most “fair” opponents for HipG are OM and P4, which have similar (deeper) socket-based software stack. Table 2 is summarized in Figure 10, where execution times are scaled against MX and P4. On average, HipG is 2.0 times faster than OM on Myrinet, and 2.5 times faster on Ethernet. Most importantly, the speedup or slowdown of HipG follows the speedup or slowdown of the C/MPI application run over MX, which suggests that the overhead of HipG will not explode for larger problem sizes.

Memory utilization. More important than speedup is in graph algorithms memory efficiency. The memory of a HipG worker is divided between the graph, the communi-

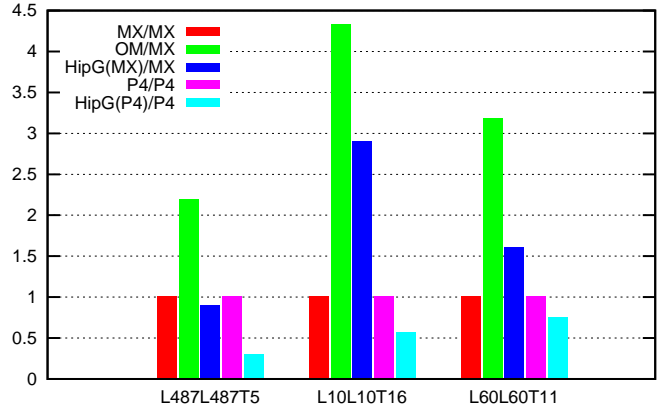


Figure 10: Scaled OBFR-MP execution time.

cation buffers and the memory allocated explicitly by the user. On a 64-bit machine, a graph node uses 80 bytes in VISITOR and on average 1 KB in BFS, including the edges and all overhead. Table 1 presents the maximum heap size used per-worker. As expected, it remains almost constant. BFS uses in general more memory than VISITOR, because it stores a queue of nodes (see Figure 3). We note that the sizes of graphs tested with HipG are of the order of the largest existing real-life graphs mentioned in Section 1.

The results in this section do not aim to prove that we obtained the most efficient implementations of the VISITOR, BFS, MAP or OBFR-MP algorithms. When processing large-scale graphs, the speedup is of secondary importance; it is of primary importance to be able to store the graph in memory and process it in acceptable time. We aimed to show that large-scale graphs *can* be handled by HipG and satisfactory performance can be obtained with little coding effort, even for complex on-the-fly or hierarchical graph algorithms.

7. RELATED WORK

HipG is a distributed framework aimed at providing users with a way to code, with little effort, parallel algorithms that operate on partitioned graphs. An analysis of other platforms suitable for the execution of graph algorithms is provided in an inspiring paper by Lumsdaine *et al.* [9] that, in fact, advocates using massively multithreaded shared-memory machines for this purpose. However, such machines are very expensive and software support is lacking [9]. The library in [28] realizes this concept on a Cray machine. Yet another interesting alternative would be to use partitioned global address space languages like UPC [29], X10 [30] or ZPL [31], but we are not aware of support for graph algorithms in these languages, except for the shared memory solution [32] based on X10 and Cilk.

A graph programming framework that most closely resembles HipG is the Signal/Collect [33] framework targeted at the Semantic Web community. In Signal/Collect graph computations are expressed in terms of signals sent along edges, which correspond to HipG’s execution of methods on graph nodes. An advantage of the Signal/Collect model is that the scheduling of signals allows for malleability: the model provides synchronized, asynchronous, and prioritized executions. However, similarly to Pregel, the user controls signals but not the global execution; HipG allows the global execu-

tion to be defined by the user (via synchronizers). Signal/Collect is only implemented for shared memory systems.

The Bulk Synchronous Parallel (BSP) model of computation [14] alternates work and communication phases. We know of two BSP-based libraries that support the development of distributed graph algorithms: *CGMgraph* and Pregel. *CGMgraph* [34] uses the unified communication API and parallel routines offered by *CGMlib*, which is conceptually close to MPI [35]. In Google’s Pregel [24] the graph program is a series of supersteps. In each superstep the `Compute(messages)` method, implemented by the user, is executed in parallel on all vertices. The system supports fault-tolerance consisting of heartbeats and checkpointing. Impressively, Pregel is reported to be able to handle billions of nodes and use hundreds of workers. Unfortunately, it is not available for download. Pregel is similar to HipG in two aspects: the vertex-centered programming and composing the parallel program automatically from user-provided simple sequential-like components. However, the repeated global synchronization phase in BSP, although suitable for many applications, is not always desirable. HipG is fundamentally different from BSP in this respect, as it uses asynchronous messages with computation synchronized on the user’s request. Notably, HipG can simulate the BSP model as we did in the BFS application (Section 2).

The prominent sequential Boost Graph Library (BGL) [10] gave rise to a parallelization that adopts a different approach to graph algorithms. Parallel BGL [11,12] is a generic C++ library that implements distributed graph data structures and graph algorithms. The main focus is to reuse existing sequential algorithms, only applying them to distributed data structures, to obtain parallel algorithms. PBGL supports a rich set of parallel graph implementations and property maps. The system keeps information about ghost (remote) vertices, although that works well only if the number of edges spanning different processors is small. Parallel BGL offers a very general model, while both Pregel and HipG trade expressiveness (for example neither offers any form of remote read) for more predictable performance. ParGraph [36] is another parallelization of BGL, similar to PBGL, but less developed; it does not seem to be maintained. We are not aware of any work directly supporting the development of divide-and-conquer graph algorithms.

What HipG does not currently support is combining the memory with external storage. In [37] some nodes created during enumerative model checking are stored on disk and accessed through Bloom filters to reduce the number of I/O operations. In [38] parts of the graph are stored on solid-state memory devices that are significantly faster than disks. Both solutions were designed for shared memory systems.

Besides general graph programming frameworks, tailored solutions to some parallel graph problems exist. In the formal methods community a number of distributed model checkers were developed to cope with the state explosion problem. The DiVinE LTL model checker [6,15,26,39] can utilize both multi-cores and distributed memory. DiVinE has been optimized for performance [40]. Another notable model checking tool, LTSmin [41], introduces a new high-level layer in which new algorithms and new interface languages can be

plugged in. Both tools are implemented in MPI/C++ and they cannot be used for general graph processing.

To store graphs we used the SVC-II distributed graph format advocated in [42]. Graph formats are standardized only within selected communities. In case of large graphs, binary formats are typically preferable to text-based formats, as compression is not needed. See [42] for a comparison of a number of formats used in the formal methods community. A popular text format is XML, which is used for example to store OpenStreetMap [43]. RDF [44] is used to represent semantic graphs in the form of triples (source, edge, target). Najork [45] describes how the web graph can be compactly stored in memory. By contrast, in bioinformatics, graphs are stored in many databases and integrating them is ongoing research [46].

8. CONCLUSIONS AND FUTURE WORK

This paper described HipG, a model and a distributed framework that allows users to code, with little effort, parallel graph algorithms. The parallel program is automatically composed of user-defined sequential components: pieces of sequential work on graph nodes and synchronizers to coordinate this work. Advanced features of HipG allow to implement divide-and-conquer graph algorithms and algorithms that generate the graph on-the-fly. We realized the model in Java and obtained elegant and short implementations of several published graph algorithms, good memory utilization and performance, as well as out-of-the-box portability.

Fault-tolerance has not been currently implemented in HipG, as the programs that we executed so far had short execution time, run on a cluster and were not mission-critical. A solution using checkpointing could be implemented, in which when a machine fails, a new machine is requested and the entire computation restarted from the last checkpoint. Such a solution is standard and similar to the one used in [24]. Creating a checkpoint takes somewhat more effort, because of the lack of global synchronization phases in HipG. Creating a consistent image of the state space could be done either by freezing the entire computation or with a distributed background snapshot algorithm such as the one by Lai-Yang [21]. Distributed snapshot poses overhead on messages that can be minimized when using message combining, which is the case in HipG.

HipG is work in progress. We already support multiple graphs in a single application. The upcoming release of HipG will contain seamless support for associating values with edges with no object overhead. We would like to improve speedup by using better graph partitioning methods, *e.g.* [7]. Graphs stored explicitly cannot be modified at runtime, however, in all cases that we looked at, this could be solved by creating new graphs during execution—which is possible in HipG—or by using a map-based graph. We are currently working on providing tailored support for multicore processors. The next phase of HipG development may be extending the framework to execute on a grid/cloud, where we can use more workers. Currently the size of the graph that can be handled is limited to the amount of memory available. Therefore, we would be interested in temporarily storing a portion of a graph on disk, without completely sacrificing efficiency [37].

9. ACKNOWLEDGMENTS

We thank Jaco van de Pol who initiated this work and provided C code, Ciel Jacobs for helping with the implementation, and Stefan Vijzelaar for supporting development of the distributed Spinja.

10. REFERENCES

- [1] M. Sweney. Mark Zuckerberg: Facebook 'almost guaranteed' to reach 1 billion users. *The Guardian*, Jun 2010.
- [2] OpenStreetMap. wiki.openstreetmap.org/stats.
- [3] J. Alpert and N. Hajaj. We knew the web was big... *The Official Google Blog*, Jul 2008.
- [4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [5] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [6] J. Barnat, L. Brim, M. Češka, and P. Ročkal. DiViNE: Parallel distributed model checker (tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC'10)*, pp 4–7. IEEE, 2010.
- [7] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71 – 95, 1998.
- [8] U. Feige and R. Krauthgamer. A polylog approximation of the minimum bisection. *SIAM Review*, 48(1):99–130, 2006.
- [9] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.
- [10] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.
- [11] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC'05)*, 2005.
- [12] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pp 423–437, 2005.
- [13] E. Krepška, T. Kielmann, W. Fokkink, and H. Bal. A high-level framework for distributed processing of large-scale graphs. In M. Aguilera, H. Yu, N. Vaidya, V. Srinivasan, and R. Choudhury, editors, *International Conference on Distributed Computing and Networking (ICDCN'11)*, LNCS vol. 6522, pp 155–166, 2011.
- [14] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
- [15] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed ltl model-checking. In A. Hu and A. Martin, editors, *Formal Methods in Computer Aided Design (FMCAD'04)*, volume 3312 of LNCS, pp 352–366, 2004.
- [16] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. *Parallel and Distributed Processing (Irregular'00)*, 1586:505–511, 2000.
- [17] M. de Jonge and T. C. Ruys. The spinJa model checker. In *International SPIN Workshop on Model Checking of Software (SPIN'10)*, pp 124–128, 2010.
- [18] H. Bal, J. Maassen, R. van Nieuwpoort, N. Drost, R. Kemp, T. van Kessel, N. Palmer, G. Wrzesińska, T. Kielmann, K. van Reeuwijk, F. Seinstra, C. Jacobs, and K. Verstoep. Real-world distributed computing with Ibis. *Computer*, 43(8):54–62, 2010.
- [19] The Java SE HotSpot virtual machine. java.sun.com/products/hotspot.
- [20] E. W. Dijkstra. Shmuel Safra's version of termination detection. Circulated privately (EWD 998), Jan 1987.
- [21] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2000.
- [22] Distributed ASCI Supercomputer DAS-4. www.cs.vu.nl/das4.
- [23] D. M. Pennock, G. W. Flake, S. Lawrence, E. J. Glover, and C. L. Giles. Winners don't take all: Characterizing the competition for links on the web. *PNAS*, 99(8):5207–5211, 2002.
- [24] 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 *Special Interest Group on Management of Data (SIGMOD'10)*, pp 135–146, 2010.
- [25] R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *Proceedings of the 14th International SPIN Conference on Model Checking Software (SPIN'07)*, pp 263–267, 2007.
- [26] J. Barnat, J. Chaloupka, and J. van de Pol. Improved distributed algorithms for SCC decomposition. In *Parallel and Distributed Methods in Verification (PDMC'08)*, volume 198 of ENTCS, pp 63–77. Elsevier, 2008.
- [27] Distributed ASCI Supercomputer DAS-3. www.cs.vu.nl/das3.
- [28] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Graph software development and performance on the MTA-2 and Eldorado. Presented at 48-th *Cray User Group* meeting, 2006.
- [29] C. Coarfa et al. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *Principles and Practice of Parallel Programming (PPoPP'05)*, pp 36–47. ACM, 2005.
- [30] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pp 519–538. ACM, 2005.
- [31] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, L. Snyder, W. D. Weathersby, and C. Lin. The case for high-level parallel programming in ZPL. *IEEE Comput. Sci. Eng.*, 5(3):76–86, 1998.
- [32] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *International Conference on Parallel Processing (ICPP'08)*, pp 536–545. IEEE, 2008.
- [33] P. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika,

- L. Zhang, J. Pan, I. Horrocks, and B. Glimm, editors. *Signal/Collect: Graph Algorithms for the (Semantic) Web*, volume 6496 of *LNCS*, 2010.
- [34] A. Chan, F. Dehne, and R. Taylor. CGMgraph/CGMlib: Implementing and testing CGMgraph alg. on PC clusters and shared memory machines. *Journal of HPC Applications*, 19(1):81–97, 2005.
- [35] MPI Forum. MPI: A message passing interface. *J of Supercomp Appl*, 8(3/4):169–416, 1994.
- [36] F. Hielscher and P. Gottschling. ParGraph library. pagraph.sourceforge.net, 2004.
- [37] M. Hammer and M. Weber. To store or not to store reloaded: Reclaiming memory on demand. In *Formal Methods in Industrial Critical Systems (FMICS'06)*, LNCS 4346, pp 51–66, 2006.
- [38] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pp 1–11, 2010.
- [39] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkait, and P. Šimeček. DiVinE – A tool for distributed verification. In T. Ball and R. B. Jones, editors, *Computer-Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pp 278–281, 2006.
- [40] K. Verstoep, H. E. Bal, J. Barnat, and L. Brim. Efficient large-scale model checking. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, pp 1–12, 2009.
- [41] S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer-Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pp 354–359, 2010.
- [42] S. Blom, I. van Langevelde, and B. Lissner. Compressed and distributed file formats for labeled transition systems. In *Parallel and Distributed Methods in Verification (PDMC'03)*, volume 89 of *ENTCS*, pp 68–83. Elsevier, 2003.
- [43] L. Denoyer and P. Gallinari. The Wikipedia XML corpus. *SIGIR Forum*, 40(1):64–69, 2006.
- [44] Resource description framework. www.w3.org/RDF.
- [45] M. Najork. The scalable hyperlink store. In *ACM Conference on Hypertext and Hypermedia (HT'09)*, pp 89–98. ACM, 2009.
- [46] A. R. Joyce and B. O. Palsson. The model organism as a system: Integrating 'omics' data sets. *Nat Rev Mol Cell Biol*, 7(3):198–210, 2006.