A High-Level Framework for Distributed Processing of Large-Scale Graphs

Elzbieta Krepska, Thilo Kielmann, Wan Fokkink, Henri Bal
{ekr,kielmann,wanf,bal}@cs.vu.nl

VU University Amsterdam

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 high-level programming of parallel graph algorithms by expressing them as a hierarchy of distributed computations executed independently and managed by the user. HIPG programs are in general short and elegant; they achieve good portability, memory utilization and performance.

1 Introduction

We live in a world of graphs. Some graphs exist physically, for example transportation networks or power grids. Many exist solely in electronic form, for instance a state space of a computer program, the network of Wikipedia entries, or social networks. Graphs such as protein interaction networks in bioinformatics or airplane triangulations in engineering are created by scientists to represent real-world objects and phenomena. 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 memory- and computation-efficient.

Real-world graphs reach billions of nodes and keep growing: the World Wide Web expands, new proteins are being discovered, and more complex programs need to be verified. Consequently, 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 is small compared to the size of the network. This allows for efficient storage of edges with the source nodes. Because of their size, partitioning graphs into balanced fragments with small a number of edges spanning different fragments is hard [1, 2].

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 spawn different graph chunks. Given the lack of structure of the computation, the computation is hard to partition and locality is affected [3]. 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 [4], 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 Graph Boost Library (PBGL) [5, 6] 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. The 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 HIPG, a distributed framework aimed at facilitating implementations of HIerarchical Parallel Graph algorithms that operate on large-scale graphs. HIPG offers an interface to perform structure-driven distributed graph computations. Distributed computations are organized into a hierarchy and coordinated by logical objects called *synchronizers*. The HIPG model supports, but is not limited to, creating divide-and-conquer graph algorithms. A HIPG parallel program is composed automatically from the sequential-like components provided by the user. The computational model of HIPG, and how it can be used to program graph algorithms, is explained in Section 2, where we present three graph algorithms in increasing order of complexity: reachability search, finding single-source shortest paths and strongly connected components decomposition. These are well-known algorithms explained for example in [7].

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. Parallel composition is done in a way that does not allow race conditions, so that no locks or thread synchronization code are necessary from the user's point of view. These facts, coupled with the use of an object-oriented language, makes for an easy-to-use, but expressive, language to code hierarchical parallel graph algorithms.

We have implemented HIPG in the Java language. We discuss this choice as well as details of the implementation in Section 3. Using HIPG we have implemented algorithms presented in Section 2 and we evaluate their performance in Section 4. We processed graphs of size of the order of 10^9 of nodes on our cluster and obtained good performance. The HIPG code of the most complex example discussed in this paper, the strongly connected components decomposition, is an order of magnitude shorter than the hand-written C/MPI version of this program and three times shorter than the corresponding implementation in PBGL—See Section 5 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 6.

2 The HIPG Model and Programming Interface

The input to a HIPG program is a directed graph. HIPG partitions the graph in a number of equal-size chunks and divides chunks between workers that are made responsible for processing nodes they own. A chunk consists of a number of nodes uniquely identified by pairs (chunk, index). HIPG uses the object-oriented paradigm of programming—namely, nodes are objects. Each node has arbitrary data and a number of outgoing edges associated and co-located with it. The target node of an edge is called a neighbor. In the current setup, the graph cannot be modified at runtime, but new graphs can be created.

```
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 (MyNode n : neighbors())
        n.visit();
    }
  }
}
```

Fig. 1. Reachability search in HIPG.

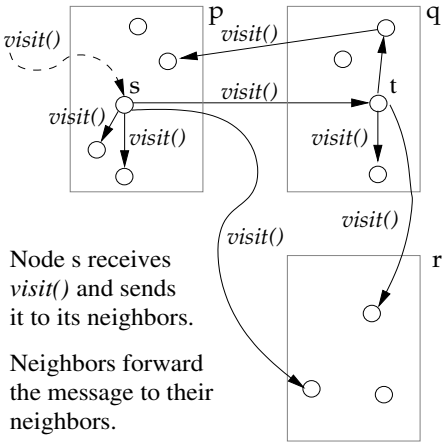

Fig. 2. Illustration of the reachability search.

Graphs are commonly processed starting at a certain graph node and by following the structure of the graph, *i.e.* the node-edge relationship, until all reached nodes are processed. HIPG allows to process graphs this way by offering a seamless interface to execute methods on local and remote nodes. When necessary these method calls are automatically translated by HIPG into messages. In Section 2.1 we show how the methods can be used to create a distributed graph computation in HIPG.

More complex algorithms require managing more than one such distributed computations. In particular, the objective of a divide-and-conquer graph algorithm is to divide computation on a graph into several sub-computations on sub-graphs. HIPG enables creation of sub-algorithms by introducing *synchronizers*—logical objects that manage distributed computations. The concept and API of a synchronizer are explained further in this section: in Section 2.2 we show how to use a single synchronizer, and in Section 2.3 an entire hierarchy of synchronizers is created to solve a divide-and-conquer graph problem.

2.1 Distributed computation

HIPG allows to implement graph computations with only regular methods executed on graph nodes. Typically, the user initializes the first method, which in turn executes methods on its neighbor nodes. In general, a node can execute methods on any node of which the unique identifier is known. To implement a graph computation, the user extends the provided LocalNode class with custom fields and methods. In a local node, neighbor nodes can be accessed with neighbors(), or inNeighbors() for incoming edges.

Under the hood, the methods executing on remote nodes are automatically translated by HIPG into asynchronous messages. On reception of such a message, an appropriate method is executed, which thus acts as a message handler. The order of received messages cannot be predicted. Method parameters are automatically serialized, and we strive to make the serialization efficient. Distributed computation terminates when there

```
interface MyNode extends Node {
  public void found(SSSP sp, int d);
}
class MyLocalNode
    extends LocalNode<MyNode>
    implements MyNode {
  int dist = −1;
  public void found(SSSP sp, int d) {
    if (dist < 0) {
      dist = d;
      sp.Q.add(this);
    }
  }
  public void found0(SSSP sp, int d) {
    for (MyNode n : neighbors())
      n.found(sp, d);
  }
}
```

```
class SSSP extends Synchronizer {
  Queue<MyLocalNode> Q = new Queue();
  int localQsize;
  public SSSP(MyLocalNode pivot) {
    if (pivot != null) Q.add(pivot);
    localQsize = Q.size();
  }
  @Reduce
  public int GlobalQSize(int s) {
    return s + Q.size();
  }
  public void run() {
    int depth = 0;
    do {
      for (int i = 0; i < localQsize; i++)
        Q.pop().found0(this, depth);
      barrier();
      depth++; localQsize = Q.size();
    } while (GlobalQSize(0) > 0);
  }
}
```

Fig. 3. Single-source shortest paths (breadth-first search) implemented in HIPG.

are no more messages present in the system, which is detected automatically. Since messages are asynchronous, returning a value of a method can be realized by sending a message back to the source. Typically, however, a dedicated mechanism, discussed later in this section, is used to compute the result of a distributed computation.

Example: Reachability search. In a directed graph, a node s is *reachable* from node t if a path from t to s exists. Reachability search computes the set of nodes reachable from a given pivot. A reachability search implemented in HIPG (Fig. 1) consists of an interface MyNode that represents any node and a local node implementation MyLocalNode. The visit() method visits a node and its neighbors (Fig. 2). The algorithm is initiated by pivot.visit(). We note that, if it was not for the unpredictable order of method executions, the code for visit() could be understood sequentially. In particular, no locks or synchronization code were needed.

2.2 Coordination of distributed computations

A dedicated layer of a HIPG algorithm coordinates the distributed computations. Its main building block is a *synchronizer*, which is a logical object that manages distributed computations. A synchronizer can initiate a distributed computation and wait for its termination. After a distributed computation has terminated, the synchronizer typically computes global results of the computation by invoking a global reduction operation. For example, the synchronizer may compute the global number of nodes reached by the

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

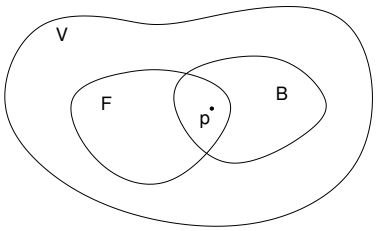

Fig. 4. FB: a divide-and-conquer algorithm to search for SCCs.

computation, or a globally elected pivot. Synchronizers can execute distributed computations in parallel or one after another.

To implement a synchronizer, the user subclasses Synchronizer and defines a run() method that, conceptually, will execute sequentially on all processors. Termination detection is provided by barrier(). The reduce methods, annotated @Reduce, must be commutative, as the order, in which they are executed, cannot be predicted.

Example: Single-source shortest paths. Fig. 3 shows an implementation of a parallel single-source shortest paths algorithm. For simplicity, each edge has equal weight, so that the algorithm is in fact a breadth-first search [7]. We define an SSSP synchronizer, which owns a queue Q that represents the current layer of graph nodes. The run() method loops over all nodes in the current layer to create the next layer. The barrier blocks until the current layer is entirely processed. GlobalQSize computes the global size of Q by summing the sizes of queues Q on all processors. The algorithm terminates when all layers have been processed.

2.3 Hierarchical coordination of distributed computations

The key idea of the HIPG coordination layer is that synchronizers can spawn any number of sub-synchronizers to solve graph sub-problems. Therefore, the coordination layer is, in fact, a *tree* of executing synchronizers, and thus a hierarchy of distributed algorithms. All synchronizers execute *independently* and *in parallel*. The order, in which synchronizers progress cannot be predicted, unless they are causally related or explicitly synchronized. The user starts a graph algorithm by spawning root synchronizers. The system terminates when all synchronizers terminate.

The HIPG parallel program is composed automatically from the two components provided by the user, namely node methods (message handlers) and the synchronizer code (coordination layer). Parallel composition is done in a way which does not allow race conditions. No explicit communication or thread synchronization is needed.

Example: 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. In Fig. 4 we describe FB [8], a divide-and-conquer graph algorithm for computing SCCs. 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

```
interface MyNode extends Node {
  public void fwd(FB fb, int f, int b);
  public void bwd(FB fb, int f, int b);
}
class MyLocalNode implements MyNode
    extends LocalNode<MyNode> {
  int labelF = −1, labelB = −1;
  public void fwd(FB fb, int f, int b) {
    if (labelF == fb.ff && (labelB == b||labelB == fb.bb)){
      labelF = f; fb.F.add(this);
      for (MyNode n : neighbors())
        n.fwd();
    }
  }
  public void bwd(FB fb, int f, int b) {
    if (labelB == fb.bb && (labelF == f||labelF == fb.ff)){
      labelB = b; fb.B.add(this);
      for (MyNode n : inNeighbors())
        n.bwd();
    }
  }
}
```

```
class FB extends Synchronizer {
  Queue<MyLocalNode> V, F, B; int ff, bb;
  FB(int f, int b, Queue<MyLocalNode> V0) {
    V = V0; F,B = new Queue(); ff = f; bb = b;
  }
  @Reduce MyNode SelectPivot(MyNode p) {
    return (p==null && !V.isEmpty())? V.pop():null;
  }
  public void run() {
    MyNode pivot = SelectPivot(null);
    if (pivot == null) return;
    int f = 2∗getId(), b = f+1;
    if (pivot.isLocal()) {
      pivot.fwd(this, f, b);
      pivot.bwd(this, f, b);
    }
    barrier();
    spawn(f, bb, new FB(F.filterB(b));
    spawn(ff, b, new FB(B.filterF(f));
    spawn(f, b, new FB(V.filterFuB(f, b));
  }
}
```

Fig. 5. Implementation of the FB algorithm in HIPG.

node is selected from V. Two sets F and B are computed as the sets of nodes that are, respectively, forward and backward reachable 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 HIPG implementation of the FB algorithm is displayed in Fig. 5. The FB creates subsets F and B of V by executing forward and backward reachability searches from a global pivot. Each set is labeled with a unique pair of integers (f,b). FB spawns three sub-synchronizers to solve sub-problems on $F \setminus B$, $B \setminus F$ and $V \setminus (F \cup B)$.

We note that the algorithm in Fig. 5 reflects the original elegant algorithm in Fig. 4. The entire HIPG program is 113 lines of code, while a corresponding C/MPI application (see Section 4) has over 1700 lines, and the PBGL implementation has 341 lines.

3 Implementation

HIPG is designed to execute in a distributed-memory environment. We chose to implement it in Java because of the portability and performance (due to just-in-time compilation) as well as excellent software support of the language, although Java required us to carefully ensure that the memory is utilized efficiently. We used the Ibis [9] message-passing communication library and the Java 6 virtual machine implemented by Sun [10].

Partitioning an input graph into equal-size chunks means that each chunk contains similar number of nodes and edges (currently, minimization of the number of edges spawning different chunks is not taken into account). Each worker stores one chunk in the form of an array of nodes. Outgoing edges are *not* stored within the node object. This would be impractical due to memory overhead (in 64-bit HotSpot this overhead is

16 B per object). As a compromise, nodes are objects but edges are not—rather, they are all stored in a single large integer array. We note that, although this structure is not elegant, it is transparent to the user, unless explicitly requested, *e.g.* when the program needs to be highly optimized. In addition, 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).

After reading the graph, a HIPG program typically initiates root synchronizers, waits for completion, and handles the computed results. A part of the system that executes synchronizers we refer to as a *worker*. A worker consists of one main thread that emulates the abstraction of independent executions of synchronizers by looping over an array of active synchronizers and making progress with them in turn. When all synchronizers have terminated, the worker returns control to the user's main program.

We describe the implementation from the synchronizer's point of view. A synchronizer is given a unique identifier, determined on spawn. Each synchronizer can take one of the three actions: either it communicates while waiting for a distributed routine to finish; or it proceeds when the distributed routine is finished; or it terminates. The bulk of synchronizer's communication consists of messages that correspond to methods executed on graph nodes. Such messages contain identifiers of the synchronizer, the graph, the node and the executed method, followed by serialized method parameters. The messages are combined in non-blocking buffers and flushed repeatedly. Besides communicating, synchronizers perform distributed routines. Barriers are implemented with the distributed termination detection algorithm by Safra [11]. When a barrier returns, it means that no messages that belong to the synchronizer are present in the system. The reduce operation is also implemented by token traversal [12] and the result announced to all workers.

Before a HIPG program can be executed, its Java bytecode has to be instrumented. Besides optimizing object serialization by Ibis [9], the graph program is modified: methods are translated into messages, neighbor access is optimized, and synchronizers are rewritten so that no separate thread is needed for each synchronizer instance. The latter is done by translating the blocking routines into a checkpoint followed by a return. This way a worker can execute a synchronizer's run() method step-by-step. The instrumentation is part of the provided HIPG library, and needs to be called before execution. No special Java compiler is necessary.

Release. More implementation details and a GPL release of HIPG can be found at `http://www.cs.vu.nl/~ekr/hipg`.

4 Memory utilization and performance 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-3 system [13]. The cluster consists of 85 dual-core, dual-CPU 2.4 GHz Opteron compute nodes, each equipped with 4 GB of memory. The processors are interconnected with Myri-10G (MX) and 1G Ethernet links. 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 re-

Appl.	Workers	Input	Time[s]	Mem[GB]
Visitor	8	Bin-27	19.1	2.8
Visitor	16	Bin-28	21.4	2.9
Visitor	32	Bin-29	24.5	3.1
Visitor	64	Bin-29	16.9	2.1
SSSP	8	Bin-27	31.5	2.8
SSSP	16	Bin-28	38.0	3.0
SSSP	32	Bin-29	42.5	3.2
SSSP	64	Bin-29	29.8	2.4
SSSP	8	LN-80	30.8	1.3
SSSP	16	LN-160	33.7	1.5
SSSP	32	LN-320	34.6	1.7
SSSP	64	LN-640	38.5	2.0

Tab. 1. Performance of VISITOR and SSSP.

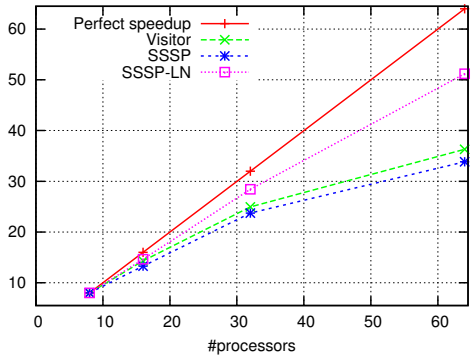

Fig. 6. Speedup of VISITOR and SSSP.

mote edges is thus $\frac{p-1}{p}$, which is very high (87-99% in used graphs) and realistic when modeling an unfavorable partitioning (many edges spawning different chunks).

We start with the evaluation of performance of applications that almost solely communicate (only one synchronizer spawned). Visitor, the reachability search (see Section 2.1) was started at the root node of a large binary tree directed towards the leaves. SSSP, the single-source shortest paths (breadth first search) (see Section 2.2), was started at the root node of the binary tree, and at a random node of a synthetic social network. The results are presented in Tab. 1 and Fig. 6. We tested both applications on 8–64 processors on Myrinet. To obtain more fair results, rather than keeping the problem size constant and dividing the input into more chunks, we doubled the problem size with doubling the number of processors (Tab. 1, with the exception of Bin–30 that should have been run on 64 processors but did not fit the memory). 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 (Fig. 6). We used binary trees, Bin–n, of height $n = 27..29$ that have $0.27–1.0 \cdot 10^9$ nodes and edges. The LN–n graphs are random directed graphs with degrees of nodes sampled from the log-normal distribution $\ln \mathcal{N}(4, 1.3)$, aimed to resemble real-world social networks [14, 15]. An LN–n graph has $n \cdot 10^5$ nodes and $n \cdot 1.27 \cdot 10^6$ edges. We used LN–n graphs of size $n = 80..640$ and thus up to $64 \cdot 10^6$ nodes and $8 \cdot 10^9$ edges. In each experiment, all edges of the input graphs were visited. Both applications achieved about 60% efficiency on a binary tree graph on 64 processors, which is satisfactory for an application with little computation, $\mathcal{O}(n)$, compared to $\mathcal{O}(n)$ communication. The efficiency achieved by SSSP 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.

To evaluate the performance of hierarchical graph algorithms written in HIPG, we ran the OBFR-MP algorithm that decomposes a graph into SCCs [16]. OBFR-MP is a divide-and-conquer algorithm like FB [8] (see Section 2.3), 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

Tab. 2. Performance comparison of the OBFR-MP SCC-decomposition algorithm tested on three LnLnTm graphs. OM (OpenMPI) and P4 are socket-based MPI implementations, while the MX MPI implementation directly uses the Myrinet interface. Time is given in seconds.

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

in [16] and kindly provided to us by the authors. The HIPG version was implemented to maximally 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 MPI implementation of OBFR-MP, on which we don't have any influence. Rather, we want to see the difference in performance between an optimized C/MPI version and HIPG version of *the same* application. In general, we found that the HIPG version was substantially faster when compared with MPI implementations that used sockets. The detailed results are presented in Tab. 2. We used two different implementations of MPI over Myrinet: the MPICH-MX implementation provided by Myricom that directly accesses the interface, and OpenMPI that goes through TCP sockets. On Ethernet we used the standard MPI implementation (P4). 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 strongly depends on the SCC-structure of the input graph. 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. The OpenMPI and P4 MPI implementations offer a more realistic comparison as they use a deeper software stack (sockets) like HIPG: HIPG ran on average 2.2 times faster than the C/MPI in this case. 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 with further scaling of the application.

The communication pattern of many graph algorithms is an intensive all-to-all communication. Generally, message sizes decrease with the increase of the number of processors. Good performance results from balancing the size of flushed messages and the frequency of flushing: too many flushes decrease performance, while too few flushes cause other processors to stall. Throughput on 32 processors over MX for the VISITOR application on Bin-29 is constant (not shown): the application sends 16 GB in 24 s.

A worker's memory is divided between the graph, the communication buffers and the memory allocated by the user's code in synchronizers. On a 64-bit machine, a graph node uses 80 B in VISITOR and on average 1 KB in SSSP, including the edges and

all overhead. Tab. 1 presents the maximum heap size used by a VISITOR/SPPP worker. Expectedly, it remains almost constant. SSSP uses more memory than Visitor, because it stores a queue of nodes (see Section 2.2).

The results in this section do not aim to prove that we obtained the most efficient implementations of the VISITOR, SSSP 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 hierarchical graph algorithms.

5 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.* [3] that, in fact, advocates using massively multithreaded shared-memory machines for this purpose. However, such machines are very expensive and software support is lacking [3]. The library in [17] realizes this concept on a Cray machine. Another interesting alternative would be to use the partitioned global address space languages like UPC [18], X10 [19] or ZPL [20], but we are not aware of support for graph algorithms in these languages, except for a shared memory solution [21] based on X10 and Cilk.

The Bulk Synchronous Parallel (BSP) model of computation [22] alternates work and communication phases. We know of two BSP-based libraries that support the development of distributed graph algorithms: CGM*graph* and Pregel. CGM*graph* [23] uses the unified communication API and parallel routines offered by CGM*lib*, which is conceptually close to MPI [24]. In Google's Pregel [15] 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 the Bulk Synchronous Parallel model, 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 SSSP application (Section 2.2).

The prominent sequential Boost Graph Library (BGL) [4] gave rise to a parallelization that adopts a different approach to graph algorithms. Parallel BGL [5,6] 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 [25] 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 divide-and-conquer graph algorithms.

To store graphs we used the SVC-II distributed graph format advocated in [26]. 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 [26] 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 Wikipedia [27]. RDF [28] is used to represent semantic graphs in the form of triples (source, edge, target). Contrastingly, in bioinformatics, graphs are stored in many databases and integrating them is ongoing research [29].

6 Conclusions and future work

In this paper we described HIPG, a model and a distributed framework that allows users to code, with little effort, hierarchical parallel graph algorithms. The parallel program is automatically composed of sequential-like components provided by the user: node methods and synchronizers, which coordinate distributed computations. We realized the model in Java and obtained short and elegant implementations of several published graph algorithms, good memory utilization and performance, as well as out-of-the box portability.

Fault-tolerance has not been implemented in the current implementation of HIPG, as the programs that we executed so far 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 [15]. 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 snapshot algorithm in the background such as the one by Lai-Yang [12]. Distributed snapshot poses overhead on messages, which however can be minimized when using message combining, which is the case in HIPG.

HIPG is work in progress. We would like to improve speedup by using better graph partitioning methods, *e.g.* [1]. If needed, we could implement graph modification during runtime, although in all cases that we looked at, this could be solved by creating new graphs during execution, which is possible in HIPG. We are currently working on providing tailored support for multicore processors and extending the framework to execute on a grid. Currently the size of the graph that can be handled is limited to the amount of memory available. Therefore, we are interested if a portion of a graph could be temporarily stored on disk without completely sacrificing efficiency [30].

Acknowledgments. We thank Jaco van de Pol who initiated this work and provided C code, and Ceriel Jacobs for helping with the implementation.

References

1. G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. of Par. and Distr. Computing*, 48(1):71 – 95, 1998.
2. U. Feige and R. Krauthgamer. A polylog approximation of the minimum bisection. *SIAM Review*, 48(1):99–130, 2006.
3. A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *PPL*, 17(1):5–20, 2007.
4. J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.
5. D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing*, 2005.
6. D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *OOPSLA*, 40(10):423–437, 2005.
7. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press, 1990.
8. L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In J. Rolim, editor, *Irregular'00*, volume 1800 of *LNCS*, pp 505–511, 2000.
9. H.E., 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. *IEEE*, 43(8):54–62, 2010.
10. The Java SE HotSpot virtual machine. java.sun.com/products/hotspot.
11. E. Dijkstra. Shmuel Safra's version of termination detection. Circulated privately, Jan 1987.
12. G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2000.
13. Distributed ASCI Supercomputer DAS-3. www.cs.vu.nl/das3.
14. 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.
15. 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 *SIGMOD*, pp 135–146, 2010.
16. J. Barnat, J. Chaloupka, and J. van de Pol. Improved distributed algorithms for SCC decomposition. *PDMC'07*, ENTCS 198(1):63–77, 2008.
17. J. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Graph software development and performance on the MTA-2 and Eldorado. At 48-th Cray Users Group meeting, 2006.
18. C. Coarfa et al. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *PPoPP'05*, pp 36–47. ACM, 2005.
19. P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pp 519–538. ACM, 2005.
20. 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.
21. G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pp 536–545. IEEE, 2008.
22. L. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, 1990.
23. A. Chan, F. Dehne, and R. Taylor. CGMgraph/CGMlib: Implementing and testing CGM-graph alg. on PC clusters and shared memory machines. *J of HPC App*, 19(1):81–97, 2005.
24. MPI Forum. MPI: A message passing interface. *J of Supercomp Appl*, 8(3/4):169–416, 1994.
25. F. Hielscher and P. Gottschling. ParGraph library. pargraph.sourceforge.net, 2004.
26. S. Blom, I. van Langevelde, and B. Lisser. Compressed and distributed file formats for labeled transition systems. *PDMC'03*, ENTCS 89:68–83, 2003.
27. L. Denoyer and P. Gallinari. The Wikipedia XML corpus. *SIGIR Forum*, 40(1):64–69, 2006.
28. Resource description framework. www.w3.org/RDF.
29. 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.
30. M. Hammer and M. Weber. To store or not to store reloaded: Reclaiming memory on demand. In *FMICS'06*, LNCS 4346, pp 51–66, 2006.