Fast PGAS Implementation of Distributed Graph Algorithms

Guojing Cong, George Almasi, Vijay Saraswat
IBM TJ Watson Research Center
1101 Kitchawan Road, Route 134, Yorktown Heights, NY, 10598
{gcong,gheorghe,vsaraswa}@us.ibm.com

Abstract—**Due to the memory intensive workload and the erratic access pattern, irregular graph algorithms are notoriously hard to implement and optimize for high performance on distributed-memory systems. Although the PGAS paradigm proposed recently improves ease of programming, no high performance PGAS implementation of large-scale graph analysis is known.**

We present the first fast PGAS implementation of graph algorithms for the connected components and minimum spanning tree problems. By improving memory access locality, compared with the naive implementation, our implementation exhibits much better communication efficiency and cache performance on a cluster of SMPs. With additional algorithmic and PGAS-specific optimizations, our implementation achieves significant speedups over both the best sequential implementation and the best single-node SMP implementation for large, sparse graphs with more than a billion edges.

Keywords: PGAS, connected components, minimum spanning tree, parallel graph algorithms

I. Introduction

Large-scale graph analysis remains challenging on current architectures due to its memory intensive workload and irregular access pattern [4], [7], [20], [26]. Recent studies have proposed techniques to reduce the gap between algorithm and architecture for shared-memory platforms [14]. Implementing irregular algorithms with high performance is even harder on distributed-memory machines where the adverse impact of irregular accesses is magnified when memory requests are served by remote nodes. Yoo *et al.* implemented parallel breadth-first search (BFS) on BlueGene/L [32]. As far as we know, this is the only study that has demonstrated reasonable parallel performance on distributed-memory machines. Yet for the large collection of PRAM graph algorithms, the results do not constitute strong evidence for high performance implementation. As the individual CPU on BlueGene/L is relatively weak, the study did not establish a meaningful sequential baseline to compare the parallel performance against. Indeed only wall clock times and speedups with other reference architectures are reported. In addition, the parallel BFS implementation has a lower bound of $O(d)$ (d is the diameter of the input graph) for the running time regardless of the number of processors. Many poly-log time graph algorithms that scale to $O(n)$ processors exhibit different algorithmic behavior.

PGAS languages (e.g., [29], [11]) promise ease of programming for distributed-memory machines and leverage of tuning to the programmer. They present a shared-memory abstraction, and allow the programmer to control the data layout and work assignment. Mapping shared-memory graph algorithms, e.g., PRAM algorithms, onto distributed-memory machines is usually straightforward with the PGAS paradigm, yet it is unlikely that such naive implementation can achieve high performance for large, sparse inputs due to communication cost.

Considering communication through the network as a bottleneck, current literature proposes communication-efficient algorithms (e.g., [18], [22], [15], [21], [31]) that reduce communication rounds at the cost of increased local operations. The algorithms (e.g., [1], [18]) are bulk synchronous and drastically different from their shared-memory counterparts. Take the list ranking algorithm presented in [18] for example. The algorithm first contracts the distributed list to fit into the local memory on one node. It then invokes a sequential algorithm to rank the contracted list. Finally the contracted list is broadcast to all processors and the rank of each element in the original list is computed. The algorithm takes $O(\log p)$ (p is the number of processors) rounds of communication, regardless of the input size. While the communication rounds are reduced, all but one processor remain idle during the sequential processing step. As n/p can be large (n is the input size) when $n >> p$, the performance gain from reduced communication rounds may be offset by poor cache performance in the sequential processing step on architectures with deep memory hierarchy.

In this paper we present fast PGAS implementation of shared-memory algorithms for connected components (CC) and minimum spanning tree (MST) on a cluster of symmetric multiprocessors (SMPs). CC and MST are well-studied graph problems with a wide range of applications. The algorithmic pattern of CC and MST represent those of a large class of PRAM algorithms that scale to $O(n)$ processors, and the prior fast SMP implementations [2], [3] serve as good baselines for performance comparison. In our study we first evaluate the performance of the naive implementations on a cluster of SMPs. We then present techniques to coordinate the memory accesses for drastically improved communication efficiency (due to better spatial locality) between nodes and cache performance (due to better temporal locality) within a node. With additional algorithmic and PGAS specific optimizations, speedups up to 3 and 10.2 are achieved on 16 nodes for CC and MST, respectively, compared with the best single-node 16-processor SMP implementation. Compared with the best

©2010 IEEE Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

sequential implementation, speedups up to 10.1 and 10.2 are achieved on 16 nodes for CC and MST, respectively. Note that these experiments are done on inputs that fit in one node. On larger inputs that out-of-core techniques have to be applied for single-node implementation, we expect even better speedups. Speedups over 100 are achieved compared with the straightforward implementation. Considering the irregular nature of the algorithms, these numbers are remarkable and promising for PGAS implementation of graph algorithms on distributed-memory machines. One important consequence of our results is that instead of taking the approach of communication-efficient algorithms that have one processor work on the large contracted inputs to reduce communication rounds, it is faster to coordinate multiple processors to process the same input in parallel.

The rest of the paper is organized as follows. Section II surveys shared-memory algorithms and communication efficient algorithms for CC and MST. The shared-memory algorithms discussed in this paper include PRAM algorithms and SMP algorithms that are based on PRAM approaches but with additional features such as asynchronous steps and locks. Section III presents our PGAS implementation and analyzes the complexity. Section IV presents our optimizations through improving memory access locality. In section V we present algorithmic optimizations specific to the problem and optimizations specific to PGAS. Section VI is the performance results. In section VII we give our conclusion and future work.

II. Shared-memory algorithms and communication-efficient algorithms

For a sparse graph $G = (V, E)$ where $n = |V|$ and $m = |E|$, various techniques have been given for solving the connected components problem and the closely-related spanning tree problem on PRAM. Some algorithms are fairly complex.We refer interested readers to paper [2] for a brief survey. It is noted in [2] that few of these algorithms achieve good parallel performance with sparse, irregular inputs on SMPs when compared with the best sequential implementation.

Similarly, many theoretical results for solving MST in parallel are considered impractical because they are too complicated and have large constant factors hidden in the asymptotic complexity (see [3] for a survey). Bader and Cong gave the first parallel implementation that beats the best sequential implementation on SMPs [3].

Bader *et al.* [5] and Berry *et al.* [8] have shown that massively multi-threaded architectures may be suitable for graph algorithms with fine-grain parallelism.

Communication-efficient parallel algorithms were proposed to address the "bottleneck of processor-to-processor communication" (e.g., see [15], [18], [21], [31]). Goodrich [22] presented a communication-efficient sorting algorithm on weak-CREW BSP that runs in $\mathrm{O}(\log n/\log(h+1))$ communication rounds and $\mathrm{O}((n \log n)/p)$ local computation time, for $h = \Theta(n/p)$. Goodrich's sorting algorithm is frequently used in communication-efficient graph algorithms. Dehne *et al.* designed an efficient list ranking algorithm for coarse-grained multicomputers (CGM [19]) and BSP that takes $O(\log p)$ communication rounds with $O(n/p)$ local computation [18]. In the same study, a series of communication-efficient graph algorithms such as connected components, ear decomposition, and biconnected components are presented using the list ranking algorithm as a building block. On the BSP model, Adler *et. al.* [1] presented a communication-optimal MST algorithm. The list ranking algorithm and the MST algorithm take similar approaches to reduce the number of communication rounds. They both start by simulating several (e.g., $O(\log p)$ or $O(\log \log p)$) steps of the PRAM algorithms on the target model to reduce the input size so that it fits in the memory of a single node. A sequential algorithm is then invoked to process the reduced input of size $O(n/p)$, and finally the result is broadcast to all processors for computing the final solution.

Here we give a brief introduction of the shared memory algorithms for CC and MST used in our study. When in no danger of confusion, we use CC and MST to denote both the problems and the algorithms.

CC takes an edge list as input, and reports the results in array D of size n with $D[i]$ set as the component that vertex i belongs to. $D[i]$ is set to i initially. CC starts with n isolated vertices and m processors. Each processor P_i ($1 \leq i \leq m$) inspects edge $e_i = (u, v)$ and tries to graft vertex u to v under the constraint that $D[u] < D[v]$. Grafting creates $k \geq 1$ components, then short-cutting is applied on the components so that grafting may be possible again in the next iteration. Short-cutting is achieved by setting $D[i] \leftarrow D[D[i]]$ repeatedly for all $0 \leq i \leq n-1$ until $D[i] = D[D[i]]$. CC terminates when no more grafting is possible. The complexity is $O(\log^2 n)$ time with $m + n$ processors on arbitrary CRCW PRAM.

MST is a variant of the parallel Borůvka algorithm [14]. Each iteration of the Borůvka algorithm is comprised of three steps: 1. for each vertex v label the incident edge with the smallest weight to be in the MST; 2. identify connected components of the induced graph with edges found in Step 1; and 3. compact each connected component into a single supervertex. Compacting the graph is costly. To avoid graph compacting, the algorithm introduces for each vertex an associated label *supervertex* showing to which supervertex it belongs. In each iteration, the adjacent edge e with the smallest weight is found for each supervertex. The adjacent edges for a supervertex v are scattered among the adjacent edges of all vertices that belong to v, and different processors may work on these edges simultaneously. Fine-grained locks are used to guard against race conditions among these processors when they attempt to update the minimum-weight edge for v.

Both CC and MST are based on PRAM algorithms with optimizations for SMPs. In CC asynchronous executions are introduced, while in MST fine-grain locks are employed. In theory they both scale to $O(n)$ processors, and their execution time does not depend on the input topology. They have been shown to be the best performing algorithms on SMPs [2], [14].

III. PGAS IMPLEMENTATION OF CC AND MST

We implement the algorithms in UPC [29]. For CC we denote the SMP version and the UPC version as CC-SMP and CC-UPC, respectively. Figure 1 shows the two codes. In the figure, El is the input edge list with each edge represented as a pair of end points (u, v). CC-SMP and CC-UPC are almost identical except for the names of a few language constructs. The differences are shown in underscore. Figure 1 suggests that mapping existing shared memory algorithms to distributed memory machines using UPC is indeed straightforward. The two versions for MST are also almost identical. Due to limited space we do not show the UPC and SMP codes for MST.

Now we analyze the complexity of the UPC implementation. First we introduce some necessary notations. For the rest of the paper, we use p for the number of nodes in the cluster, t for the number of threads per node, and $s = p \times t$ for the total number of threads. The latency and the bandwidth of the network are denoted as L and B, respectively. The memory latency and bandwidth are denoted as L_M and B_M, respectively. We consider CC only and the readers can easily apply similar analysis to MST.

CC-UPC and CC-SMP are almost identical except in CC-UPC many of the irregular accesses result in communication through the network. The complexity of CC-SMP is analyzed under the SMP complexity model proposed in [2], [23]. The SMP complexity model uses two parameters: the input size n, and the number p of processors. Running time $T(n, p)$ is measured by the pair $\langle T_M(n,p)\ ;\ T_C(n,p)\rangle$ where $T_M(n,p)$ is the maximum number of non-contiguous main memory accesses required by any processor and $T_C(n,p)$ is an upper bound on the maximum local computational complexity of any of the processors. This model penalizes algorithms with non-contiguous memory accesses that often result in cache misses. CC-UPC has the same computational complexity as CC-SMP [2], that is,

$$T_C(n,p) = \mathrm{O}\left(\frac{n\log^2 n + m\log n}{p}\right) \quad (1)$$

The memory access complexity of CC under the SMP model is [2]

$$T_M(n,p) \le \frac{n\log^2 n}{p} + \left(4\frac{m}{p} + 2\right)\log n \quad (2)$$

The sequential and regular memory accesses in CC-UPC can be easily localized, while the irregular memory accesses incur large overhead of locating the target node and issuing remote accesses. For random graphs, we can assume even distribution of remote accesses to all nodes. The expected time spent on remote accesses on one thread is

$$T_{remote} \le \frac{p-1}{ps}\left(n\log n + 4m + 2s\right)\log n\left(L + \frac{1}{B}\right) \quad (3)$$

When blocking communication common in compiled code is used, the messages from the t threads on one node are serialized, and the communication time spent on one node is $\frac{p-1}{p^2}\left(n\log n + 4m + 2s\right)\log n(L + \frac{1}{B}) \approx \frac{1}{p}\left(n\log n + 4m + 2s\right)\log n(L + \frac{1}{B})$. According to equation 2, time spent on non-contiguous accesses in CC-SMP is $\frac{1}{p}\left(n\log n + 4m + 2\right)\log n(L_M + \frac{1}{B_M})$. To compare the two times, we look at the latency and bandwidth numbers on current systems. Infiniband [25] and PCI-EXPRESS [27] are current industry standards for interconnect and I/O bus technologies, respectively. Infiniband host channel adapter has a peak bandwidth of 4 GB/s, compatible with the bandwidth of PCI-EXPRESS. So we have $B \approx B_M$. As for latency, Infiniband latency is about 190 nanoseconds [25], while that of the DDR3 SDRAM is about 9 nanoseconds [17]. Plugging these numbers back to the complexity analysis, for data access, we estimate CC-UPC is over 20 times slower than CC-SMP. Other overhead in CC-UPC such as software handling of communication and network congestion incurred by numerous small messages can further degrade the performance.

We compare the performance of CC-UPC and CC-SMP on our target platform. The machine is a cluster of 16 IBM P575+ nodes connected with a due-plane 2GB/s High-performance Switch (HIPS). Each node is configured as 16 CPUs running at 1.9 GHz with 64GB DDR2 memory. Unless noted otherwise, the UPC implementation uses all 16 nodes. We use the largest possible configuration so that we can study the scaling behavior of the distributed-memory algorithms. The number of threads per node used in each experiment may vary. CC-SMP is run on one node with 16 processors. Our code is compiled with the Berkeley UPC compiler [30] at optimization level O3 (with IBM XL compiler for the sequential code). CC-SMP is compiled with the IBM XL compiler at optimization level O3.

The inputs are random graphs with different edge densities. A random graph of n vertices and m edges is created by randomly adding m unique edges to the vertex set. Scale-free graphs [7] have been shown to represent the structures of many real-world graphs. Generating large scale-free graphs is very time consuming. Moreover, the scale-free graphs generated by the popular RMAT model [10] contain artificial locality, and random permutation on the vertices needs to be performed. For our purpose of studying the scalability of the algorithms, we also require the permutations generated with different number of threads be identical. No known fast permutation algorithms have this property (see [13] for a summary of high performance random permutation algorithms). In our study we take a hybrid approach that generates graphs with features of both random graphs and scale-free graphs. We first select $2\sqrt{n}$ vertices randomly to generate a scale-free graph on them. We then randomly add edges to the n vertices until we have the desired number of edges. Such graphs do not contain obvious locality pattern, and at the same time, they contain vertices of very large degrees ($O(\sqrt{n})$) that could pose a problem for load-balancing.

Figure 2 compares the performance of CC-UPC and CC-SMP for four different random graphs. The X axis shows the number of vertices and average edge densities (m/n) of the graphs, and the Y axis shows the execution time. CC-UPC uses 16 threads per node. The UPC implementation is much slower than the SMP implementation. In fact, if we normalize

```
gr = 0;
upc_forall(i=0; i<m; i++; &El[i])
{
   u = El[i].u; v = El[i].v;
   if(D[u] < D[v]){
         D[D[v]] = D[u];
         gr = 1;
   }
}
gr = all_reduce_i(gr, UPC_ADD);
if(gr == 0) break;
upc_forall(i=0; i<n; i++; &D[i])
{
      while(D[i] != D[D[i]])
          D[i] = D[D[i]];
}
```

```
gr = 0;
pardo(i,0,m, 1)
{
   u = El[i].u;v = El[i].v;
   if(D[u] < D[v]){
           D[D[v]] = D[u];
           gr = 1;
   }
}
gr = node_Reduce_i(gr,SUM);
if(gr == 0) break;
pardo(i,0,n,1)
{
    while(D[i] != D[D[i]])
      D[i] = D[D[i]];
}
```

Fig. 1. Comparison of CC-UPC and CC-SMP. CC-SMP is implemented using SIMPLE [6]

the performance to one processor (time divided by the number of processors), CC-UPC is 3 orders of magnitude slower than CC-SMP. Figure 2 suggests that drastic performance improvement is necessary for the UPC implementation to compete with prior implementations.

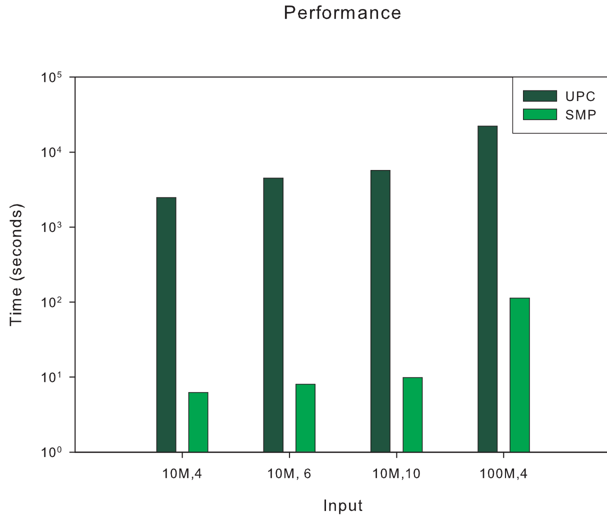

Fig. 2. Comparison of CC-UPC and CC-SMP. The scale of the Y-axis is logarithmic

The UPC implementation of MST performs poorly on our target platform. We had to abort most of the runs after hours passed without termination.

IV. Optimization through Improved Access Locality

Literal translation of shared memory algorithms with UPC performs poorly due to the many small messages to random nodes and the erratic accesses within a node. On a cluster of SMPs with deep memory hierarchy, communication efficiency and cache performance are critical to fast implementation. We optimize both communication efficiency and cache performance through improving access locality.

Consider the case of parallel accessing $D[R[i]]$ in one step where $|D| = n$, $|R| = m$, and $0 \leq i \leq m-1$. We propose a recursive access scheduling scheme shown in Algorithm 1 that can be mapped onto a cluster of SMPs. W.l.o.g., we assume W divides n and m. In Algorithm 1, $\sqcup$ is an operator that concatenates two sequences.

Algorithm 1 has four phases: *partition*, *group*, *access*, and *permute*. In *partition*, R and D are divided evenly into W blocks, R_k and D_k, respectively, $0 \leq k \leq W-1$. In *group*, the requests from R_k to block D_j, $0 \leq k, j \leq W-1$, are gathered into consecutive locations in R_k^j. Requests to D_j are served in the *access* phase, after which the data requested by R_k^j are also in consecutive locations in S_k^j. All access requests to the kth block D_k, $0 \leq k \leq W-1$, are served together. That is, once it starts to access D_k, the algorithm does not touch other blocks until all requests to D_k are served. The algorithm enforces an ordering on the otherwise random accesses to D and reduces the working set size from n to $Blk = n/W$. After they are retrieved, the D values are then permuted back to locations that match the original request order in the *permute* phase. Recursive calls are made for accessing the kth block, $0 \leq k \leq W-1$, where the block is further partitioned into smaller blocks until the block size becomes 1. Algorithm 1 does not specify the routines for sorting and permutation. Their choices are dependent on the choice of W. In fact, each recursion level can choose a different W, and the base block size can be larger than 1. By adjusting W and the recursion depth, we balance the cost of additional work and the gain from ordered memory accesses. Parallel writes in a parallel step can be scheduled similarly.

Setting $W = n$ yields one level of recursion. The access scheduling employs one pass of sorting and one pass of permutation, and is similar to the classic technique used in I/O-efficient algorithms [12]. Setting $W = n$ on a distributed-memory platform requires efficient distributed sorting and permutation. Enforcing total ordering among m elements is an overkill for cache performance once the blocks fit into cache. Assuming the same amount of requests from R to D_k, $0 \leq k \leq W-1$, setting $W = 2$ at each recursion level yields $O(\log n)$ recursions with $O((m+n)\log n)$ work. Compared with $O(n)$ work of the original scheduling, the overhead associated with the extra $\log n$ factor may offset

Data : Array D of size n, array R of size m, and integer W, $1 \leq W \leq n$
Result : Array C of size m with $C[i]$=$D[R[i]]$
if n=1 **then** $C[i] \leftarrow D[0]$, $0 \leq i \leq m-1$;
else
    Let Blk $\leftarrow \frac{n}{W}$;
    //**partition**;
    Partition D into W blocks, such that $D_k \leftarrow \{\ D[\frac{kn}{W}], \cdots, D[\frac{(k+1)n}{W}-1]\}$, $0 \leq k \leq W-1$;
    Partition R into W blocks, such that $R_k \leftarrow \{\ R[\frac{km}{W}], \cdots, R[\frac{(k+1)m}{W}-1]\}$, $0 \leq k \leq W-1$;
    //**group**;
    for $0 \leq k \leq W-1$ *in parallel* **do**
        Sort R_k using $\frac{R_k[j]}{Blk}$ as key, and store the original location of the jth element in $P_k[j]$, $0 \leq j \leq m/W$;
        Partition R_k into W blocks R_k^j, $0 \leq j \leq W-1$, such that $\forall r \in R_k^j$, $\frac{r}{Blk} = j$;
    end
    Let $R'_k \leftarrow \sqcup_{j=0}^{W-1} R_j^k$, $0 \leq k \leq W-1$;
    //**access**;
    for $0 \leq k \leq W-1$ *in parallel* **do**
        Call Algorithm 1 with D_k, R'_k, and W returning S_k;
    end
    Partition S_k into W consecutive blocks S_k^j, $0 \leq j \leq W-1$, such that $|S_k^j| = |R_k^j|$;
    Let $S'_k \leftarrow \sqcup_{j=0}^{W-1} S_j^k$, $0 \leq k \leq W-1$;
    //**permute**;
    for $0 \leq k \leq W-1$ *in parallel* **do**
        Permute S'_k into C_k such that $C_k[P_k[j]] = S'_k[j]$, $0 \leq j \leq Blk-1$;
    end
    return $C \leftarrow \sqcup_{k=0}^{W-1} C_k$;
end

Algorithm 1: Scheduling for memory access in one parallel step

gains from improved cache performance. Moreover, deep recursion demands efficient scheduling of distributed, dynamic activities on a cluster of SMPs. CILK ([9]) provides dynamic activity scheduling on a single node. Most PGAS runtimes do not provide dynamic scheduling for multiple nodes. For practical implementation, Ws should be chosen according to the number of nodes in the cluster, the number of threads on one node, and the cache size. To reduce overhead we limit the recursion depth in our implementation to no more than three levels. We study the impact of our scheduling on communication efficiency and cache performance separately in the next two subsections.

A. *Improving communication efficiency*

Setting $W = p$ at the top level recursion divides R and D evenly among the nodes. Accesses to remote D blocks are consecutive and can be coalesced into a single message to reduce the impact of network latency. *Communication coalescing* is used in earlier studies (e.g.,[16]) for optimizing compiler and runtime support for irregular scientific applications. Applying communication coalescing in effect simulates a shared-memory algorithm on CGM. As we are analyzing communication cost, we assume one thread per node, that is $p = s$. Each communication round in CGM consists of routing a single h-relation with $h = O(n/p)$. CGM requires that all information sent from a given processor to another processor in one communication round be packed into one long message, thereby minimizing the message overhead. Assuming even distribution of the input, when simulating a PRAM algorithm on p processors, in one step each processor accesses $O(n/p)$ data. After communication coalescing, these data can be accessed in one communication round where one processor sends at most one message to another processor. The expected time spent on remote access is $O\left(\log^2 n \left(pL + (\frac{p-1}{p^2})(\frac{m}{B})\right)\right)$. Compared with equation 3, when $m >> p$, the communication time is drastically reduced.

In applying the scheduling described in Algorithm 1, we define two collective operations *GetD* and *SetD* in UPC that coordinate the remote memory accesses in a parallel step for MST and CC. These collectives are called by all participating threads. Algorithm 2 essentially applies one level of recursion of Algorithm 1 on a cluster of p nodes, and describes the *GetD* collective in UPC for parallel memory reads for processor thr_i, $0 \leq i \leq p-1$.

On thr_i *GetD* first sorts the indices according to their target thread *id*s. It then informs all threads of its requests by writing to the shared data structures. Array *needFrom* is **private**, and records the number of elements thr_i needs from all other threads. *SMatrix* and *PMatrix* are **shared** two dimensional arrays. *SMatrix*$[i][j]$ is the number of elements thr_i sends to thr_j, and *PMatrix*$[i][j]$ is the position in thr_j's buffer where thr_i should deposit the elements. Computing *needFrom*, *SMatrix*, and *PMatrix* prepares for the actual communication

Input: **shared** array D, **private** array $indices$ with global indices to D, size of $indices$ is k
Output: **private** output array $outD$
begin
1. Sort *indices* using the target thread *id* of *D*[*indices*[*l*]] $(0 \leq l \leq k-1)$ as key
2. Compute number of elements to request from each thread *needFrom*[*j*] $(0 \leq j \leq p-1)$
3. Inform all threads of number of elements and their target locations to sent to thr_i
$s \leftarrow 0$
for $j \leftarrow 0$ *to* $p-1$ **do**
3.1 *SMatrix*$[j][i] \leftarrow$ *needFrom*$[j]$
3.2 *PMatrix*$[j][i] \leftarrow s$
3.3 $s \leftarrow s$ + *needFrom*$[j]$
end
4. All threads enter barrier
UPC_barrier;
5. Inspect received requests and transfer data elements in batches
for $j \leftarrow 0$ *to* $p-1$ **do**
5.1 *nSend* $\leftarrow$ *SMatrix*$[i][j]$
5.2 *targetPosition* $\leftarrow$ *PMatrix*$[i][j]$
5.3 Collect data indices requested by thread thr_j
5.4 Gather the data elements
5.5 Put data to thr_j at position *targetPosition*
end
6. Permute received data to match the request order
end

Algorithm 2: *GetD* on thread thr_i, for $(0 \leq i \leq p-1)$.

of elements. After issuing its requests, thr_i serves requests from other threads. It locates the elements in its local portion of the shared array, then sends them back to the requesting threads. Algorithm 2 can be divided into two phases. Steps 1, 2, and 3 compute the number of elements expected from other threads, and prepare the auxiliary data structures. Step 5 sends the data to all threads that have made access requests, and step 6 permutes the data to the right location to match the requests. The barrier in step 4 separates the two phases. The *SetD* collective works similarly.

Now we rewrite CC and MST using *GetD* and *SetD*. In the grafting step, data to be accessed are fetched into a local buffer using *GetD*. After comparison, the results are put back to the shared array using *SetD*. The short-cutting step of CC (refer to the second *upc_forall* loop in Figure 1) is asynchronous, and can not be rewritten directly. We insert artificial synchronizations into pointer-jumping. That is, the algorithm applies pointer-jumping to all vertices in lock step until all trees become rooted stars. The extra synchronization could be disastrous for performance on SMPs as it introduces unnecessary barriers and memory access overhead. In a distributed-memory environment, however, the modification makes communication coalescing possible. After rewriting all remote accesses in CC occur in $O(\log^2 n)$ collectives. On each

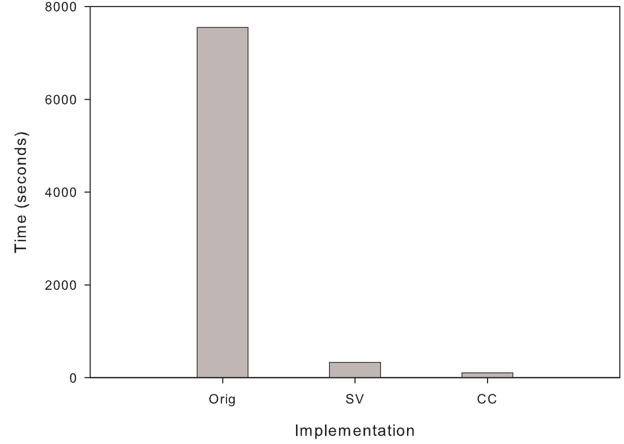

Fig. 3. Impact of communication coalescing

thread, each collective incurs $O(p)$ messages.

We also rewrite the classic Shiloach-Vishkin connected components algorithm (SV) [28]. Prior studies (e.g., see [2]) show that SV is slower than CC on SMPs. Yet the synchronous nature of SV makes it easy for rewriting. The major difference between SV and CC is in the short-cutting step. Only one level of pointer-jumping is applied in SV, hence the components are not reduced to rooted stars. To achieve the complexity of $O(\log n)$ time with $m+n$ processors on arbitrary CRCW PRAM, SV allows grafting rooted stars to other components when the normal grafting condition does not occur.

Figure 3 shows the performance impact of communication coalescing on CC and SV. The input is a random graph with 10M vertices, 40M edges. One thread is used on each node. Note here CC and SV are both rewritten with collectives. Orig shows the performance of the naive implementation. Here *GetD* and *SetD* are not optimized (optimizations are discussed later in section V), and we use quick sort that is more than 50 times slower than count sort on the same data. Still, the rewritten CC is about 70 times faster than the naive implementation. SV is slower than CC due to more collective calls in one iteration. For the rest of the paper we focus on CC rewritten with collectives only. Similar results are observed for hybrid graphs.

To rewrite MST for efficient execution, we propose a new collective *SetDMin* that obviates the need of locking. *SetDMin* works similarly as *SetD* except that when multiple threads compete to write to the same location the request with the smallest value wins. We can view *GetD* as a mechanism for current reads while *SetD* and *SetDMin* for current writes. *SetD* implements arbitrary concurrent writes, while *SetDMin* implements priority concurrent writes. In the new implementation all threads first collectively retrieve the D values for all vertices appearing in their local edge lists. For each edge $e = (u, v)$, when u and v belong to different components $(D[u] \neq D[v])$, all threads collectively assign

$D[D[u]] \leftarrow D[v]$ and $D[D[v]] \leftarrow D[u]$ using the weight w of e to arbitrate the race. Implementing priority concurrent writes is trivial with *SetDMin* as writes to the same memory location are always done by a single thread. Computing connected components for the induced graph is similar to CC.

We defer detailed performance discussion to section VI where we compare the UPC implementation with the SMP implementation, as the naive implementation of MST takes too long to finish.

B. Improving single node cache performance

With $W = p$ for the top level recursion, step 5.4 of Algorithm 2 serves random access requests to D in local memory. We analyze the impact of one more level of recursion on cache performance. To avoid carrying subscripts, we still use R, D, m, and n to denote for one thread the request array, data array, request array size, and data array size, respectively. Our analysis assumes a one level cache with cache size $z > 2\sqrt{n}$. In case of $z \leq 2\sqrt{n}$, more recursions may be applied to reduce n. Sequentially accessing k elements is charged $L_M + k/B_M$ time considering the prefetch or bulk transfer optimization in current architectures. We divide R and D into W blocks, and select W such that $n/Z < W < \sqrt{n}$. The memory access time T_M for the original code is

$$T_M = m\left(L_M + \frac{1}{B_M}\right) \quad (4)$$

We now analyze the memory access time for different stages of Algorithm 1. The *partition* phase does not incur any memory transfers. In the *group* phase, we use linear time count sort to group requests. The counter histogram size is W and fits in cache. Count sort takes $2(L_M + m/B_M) + 2W(L_M + 1/B_M)$ time. Routing requests to match the blocks takes $WL_M + m/B_M$ time with W block transfers. Suppose each block gets m_i requests, $0 \leq i \leq W - 1$. The *access* phase takes $\sum_{i=0}^{W-1} min(m_i, \frac{n}{W})L_M + \frac{m_i}{B_M} \leq nL_M + \frac{m}{B_M}$ time to access m elements. Note that once the elements in a block are brought into cache, the later accesses will be cache hits. So we pay the cost of n misses instead of m misses. The reduced cache misses are due to the smaller working set size with improved temporal locality. Collecting data back take $(WL_M + m/B_M)$ time with W block transfers. Similar to the *access* phase, the *permutation* phase takes $n(L_M + \frac{m}{B_M})$ time. Adding them all up, the memory access time after scheduling is

$$T_M = (2n + 2W + 2)L_M + \frac{4m + 2W}{B_M} \quad (5)$$

From equations 4 and 5, we see that for most graphs with $m > 3n$ and most networks with $L_M B_M > 9$, our scheduling improves cache performance.

According to our input size and the cache configuration on our target platform, we apply two more levels of recursions in our implementation. First the local D block is partitioned among t processors on the node, and then if necessary, each block is further partitioned into t' blocks. In practice, we avoid the costly recursion calls by simulating $t \times t'$ virtual processors using t physical processors. Now each thread owns t' blocks, and the size of t' is chosen such that the block fits into a certain level cache hierarchy (e.g. L2).

We evaluate the impact of our access scheduling on cache performance by studying the performance of CC on a single SMP node. In addition to rewriting CC using the shared-memory versions of *GetD* and *SetD*, we have each physical thread simulate t' virtual threads. Figure 4 shows for three inputs (n=100M and m=400M, n=100M and m=1G, and $n = 200M$ and $m = 800M$) the relative performance of CC with different t' against the prior SMP implementation. Both implementations are run with 16 processors. With $t' = 1$, CC with collectives already runs faster than the SMP implementation. As t' increases, the execution time of CC with collectives first decreases and then increases. For the smallest input (n=100M, m=400M) the best performance is achieved at $t' = 12$, while for the two larger input the best t' values are at 18. The fastest CC with collectives is nearly twice as fast as the SMP implementation. Similar behavior is observed on hybrid graphs.

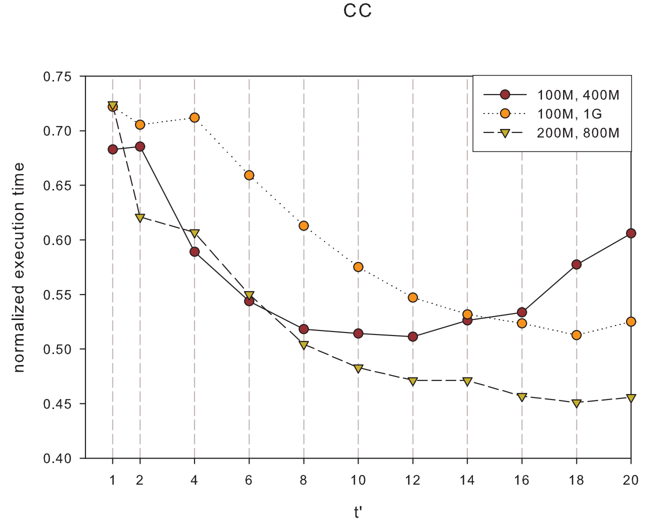

Fig. 4. Performance of CC vs. parameter t'

We do not conduct similar comparisons for MST as the gain of cache performance is overshadowed by reduced locking costs for large inputs.

V. FURTHER OPTIMIZATIONS

In addition to optimizing memory access behavior, we apply algorithmic optimizations as well as algorithm engineering techniques specific to UPC applications.

For graph algorithms that take an edge list as input, usually the size of the list determines the number of elements to sort and to request from remote nodes. As CC and MST progress, more and more edges fall within components thus no longer contribute to the merging of connected components or finding new tree edges. These edges can be filtered out. Filtering reduces both local work and communication time. We denote this optimization as *compact*.

The behavior of some graph algorithms makes certain threads potential communication hotspots. The problem is more obvious in the pointer-jumping step of the connected components algorithm and the closely related spanning tree algorithm. Grafting in CC shoots a pointer from a vertex with larger numbering to one with smaller numbering. Thread thr_0 owns vertex 0, and may quickly become a communication hotspot as increasingly more queries are made from other threads about $D[0]$. This is not a problem on SMPs as the value is cached. In a distributed-memory environment, thread thr_0 is easily overwhelmed by requests from other nodes. Fortunately, $D[0]$ remains constant for CC. A possible optimization is to avoid requesting $D[0]$ but to replace it with value 0. For each thread issuing a *GetD* operation, it first checks whether the index is 0. If it is, it knows the value already and drops this element from the request list. We denote this optimization as *offload*.

Careful orchestration of the communication is necessary to fully utilizing the network capacity. In *GetD*, *SetD*, and *SetDMin*, each thread receives elements from (or sends elements to) other threads. If all threads initiate communication between themselves and others in the order of $0, 2, \cdots, s-1$, at step i, thread thr_i has to service $O(s)$ requests. This can overwhelm thr_i when s is large. We orchestrate the communication pattern so that each thread thr_i starts with itself and wraps around using modulo arithmetic in the order of $i, i+1, \cdots, (i+s)\ mod\ s$. In this manner, in each loop step a thread is only serving one request. We denote this optimization as *circular*.

Some operations in the collectives only access the local portion of the shared arrays, e.g., copying data elements to local $outD$ buffers and setting D with values from the inD buffers. The compiler is not able to recognize these as local operations as they are highly irregular. We use UPC private pointer arithmetics for these data accesses to avoid the runtime overhead generated by the compiler. We denote this optimization as *localcpy*.

Another costly operation is to invoke compiler intrinsics to determine the target thread id of an access for sorting. Computing target thread ids is done for every iteration. We implement several optimizations. First, we compute these ids directly instead of invoking the intrinsics. These computations can also be vectorized. Noticing that the target ids do not change across iteration, we compute them once and store them in a global buffer. We denote these optimizations collectively as *id*.

Finally, as communication coalescing makes the messages larger, it is possible to use *remote direct memory access* (RDMA) of HIPS to improve the communication time. RDMA [24] allows processes to directly access the memory of processes running on a remote node through an RDMA-capable network such as HIPS. RDMA improves the communication efficiency with large messages.

The current UPC standard presents a flat organization of threads that does not facilitate hierarchical algorithm design. The limitation of the flat organization is that messages from threads on the same node can not be easily aggregated. The impact of such restrictions is discussed further in section VI. As the UPC language evolves, we expect more flexible support for hierarchical designs. For example, the Berkeley UPC compiler already provides some measures towards this direction.

Figure 5 shows the performance impact of different optimizations on CC. The input is a random graph of 100M vertices and 400M edges. We use 8 threads on each node. The execution time is broken down into six categories. *Comm* measures the communication time spent on *upc_memget* and *upc_memput*. *Sort* is the sorting time. *Copy* measures the time spent on reading data from and writing data to the local portion of the shared arrays. *Irregular* measures the time spent on reordering the retrieved elements in the buffer to the right order. *Setup* measures the time spent on setting up the matrices in Algorithm 2. *Work* measures the time spent on allocating local and shared arrays, initialization, and computing the target thread ids. The *base* case applies two levels of recursions described in Algorithm 1. From left to right, the bars accumulate the performance improvement of optimizations. For example, the *circular* bar combines all optimizations of *base*, *compact*, *offload*, and *circular*. This is meaningful as the effects of the optimizations are accumulative.

The *compact* optimization improves the execution time of almost all categories. Communication time is reduced by a factor of 2 with *circular*, and *localcpy* improves the time spent on the copying data to and from the local portion of the shared arrays by about a factor of 2. The *id* optimization greatly improve the time spent on local work. Similar impact is also observed for the hybrid graph shown in Figure 6. The highly connected hubs in the hybrid graph do not create load-balancing problems for our implementation. As we partition work by dividing the edges evenly instead of the vertices, work associated with hubs is distributed to multiple threads. Also with the optimized implementation all reads and writes to a shared location are conducted by a single thread and there is at most one message exchange between any two threads in one collective call, accessing hubs in the hybrid graph does not create communication hotspots. Our implementation in this experiment runs faster on the hybrid graph than on the random graph. This is due to fewer iterations determined by the graph topology and the vertex numbering. Such observation does not generalize to other instances. As hubs do not become hurdles to high performance for our implementation, due to limited space, in later sections we report in detail only results achieved on random graphs. We present summaries for results achieved on hybrid graphs.

We show the overall performance impact on MST in section VI.

VI. Experimental evaluation

We evaluate the performance of optimized CC on two large random graphs, one with 100M vertices and 400M edges, and the other with 100M vertices and 1G edges. The results are shown in Figures 7 and 8, respectively. We use all 16 nodes in all the runs, and vary the number of threads on

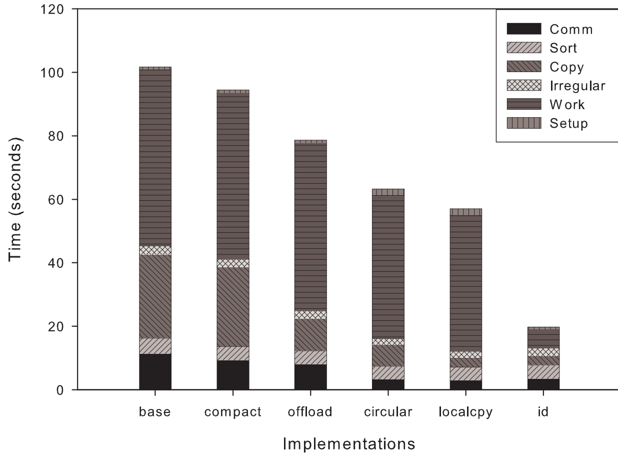

Fig. 5. Execution time breakdown (random graph)

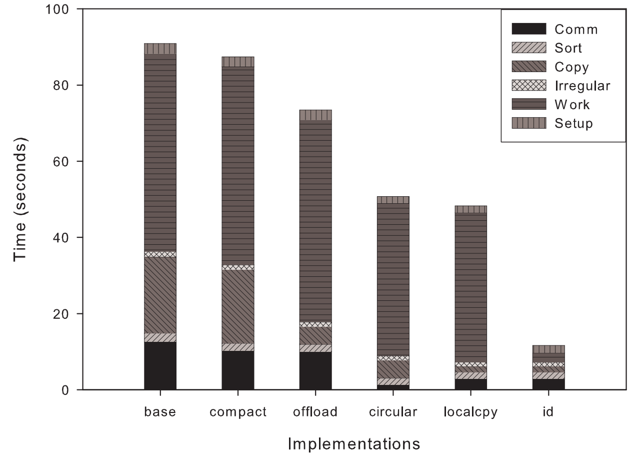

Fig. 6. Execution time breakdown (hybrid graph)

each node. Conceptually we apply three levels of recursions of Algorithm 1 on the scheduling of memory accesses, with $W = p$, $W = t$, and $W = t' = 16/t$, respectively. In the actual implementation, we simulate the three levels of recursions using p nodes with each of the t threads on one node simulating t' virtual threads. The execution time of the optimized implementation in UPC is presented. The horizontal solid line in each plot shows the execution time of the SMP version of CC with 16 threads on 1 node, while the horizontal dotted line shows the execution time of BFS on a single thread.

In both plots, *Optimized* beats the SMP implementation. The best speedups, 2.2 and 3, respectively, are achieved at 8 threads per node. This result is remarkable considering the memory intensive workload and the highly irregular nature of the memory access pattern. The best speedups against the best sequential implementation are about 9 and 11, respectively. On hybrid graphs with the same number of vertices and edges, speedups of 2.5 and 2.8, respectively, are achieved at 8 threads per node. Against the best sequential implementation, the speedups are about 9 and 10, respectively.

For *Optimized*, the performance degrades at 16 threads. Profiling the codes shows that the majority of the degradation comes from line 3 in Algorithm 2. In setting up the *SMatrix* and *PMatrix*, the algorithm is essentially doing an AlltoAll communication within the 256 threads in the cluster. The burst of the short messages overwhelms the cluster and the nodes. The performance degradation is about 10 times as the number of threads from each node goes from 8 to 16. Using asynchronous primitives does not help. This problem is due to the flat organization of threads in UPC. We do not address this problem in our study as we believe the solution lies either in better runtime support or language support. The thread-process hierarchy is exposed to the runtime, and the AlltoAll collective does not have to involve $s = p \times t$ threads in communication across the network. Instead, it may involve only p processes.

The performance of MST is also evaluated on the same two graphs with edge weights randomly chosen between 0 and the maximum integer number. The performance results are shown in Figures 9 and 10, respectively. The MST implementation is the final optimized version. Again we use all 16 nodes in all the runs, and vary the number of threads on each node. The horizontal solid line in each plot shows the execution time of the SMP version of MST with 16 threads on 1 node, while the horizontal dotted line shows the execution time of the best sequential algorithm (in this case Kruskal's algorithm beats both the Prim's and Borůvka's algorithms) on a single thread. We use the cache-friendly merge sort in implementing Kruskal's algorithm. In both plots, MST beats the SMP implementation. The best speedups (5.5 and 10.2, respectively) are again achieved at 8 threads. In both plots, the SMP implementation is either slower or only slightly faster than the sequential Kruskal implementation. This is largely due to the locking overhead with using $100M$ locks. With smaller inputs, e.g., $n = 10M$ and $m = 40M$, the SMP implementation is much faster than Kruskal's algorithm. On hybrid graphs, speedups of 5.1 and 6.7 are achieved over the sequential implementation for $n = 100M$ and $m = 400M$, and $n = 100M$ and $m = 1G$, respectively.

As the input size increases, the UPC implementation is expected to achieve better speedups compared with the SMP implementation. Eventually the problem will not fit in the memory of a single node, and either out-of-core processing or distributed-memory processing has to be employed. Indeed it is an important advantage of efficient distributed-memory graph algorithms to be able to handle large problem sizes.

The fact that significant speedups are achieved using a cluster of SMPs over a single SMP node for inputs that fit in the memory of one node impacts practical design of PGAS graph algorithms. Instead of adopting the approach of many communication efficient algorithms to reduce the number of communication rounds, our optimization techniques help efficiently mapping existing shared-memory algorithms to

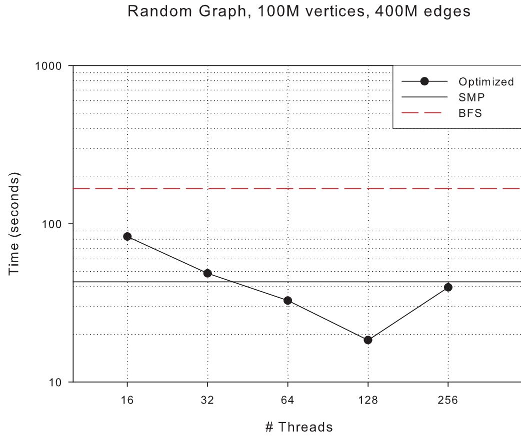

Fig. 7. CC-UPC performance on a graph with 400M edges

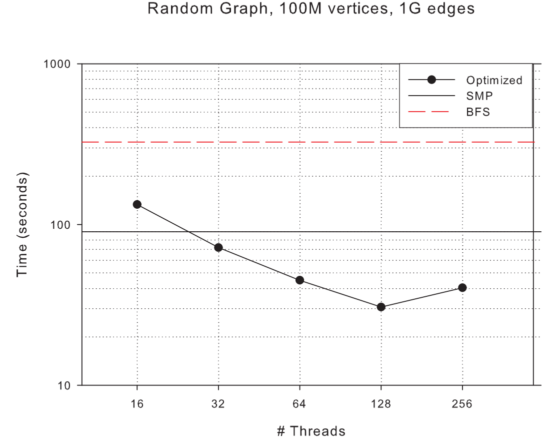

Fig. 8. CC-UPC performance on a graph with 1G edges

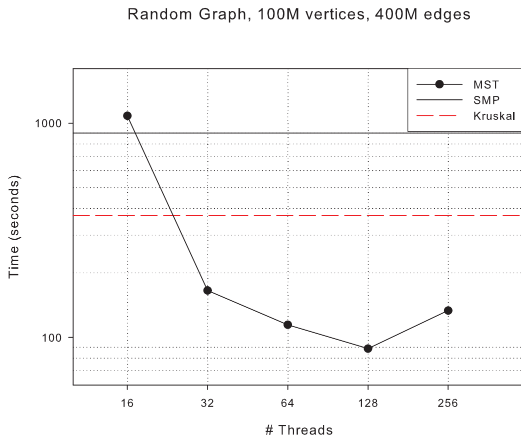

Fig. 9. Performance of MST on a graph with 400M edges

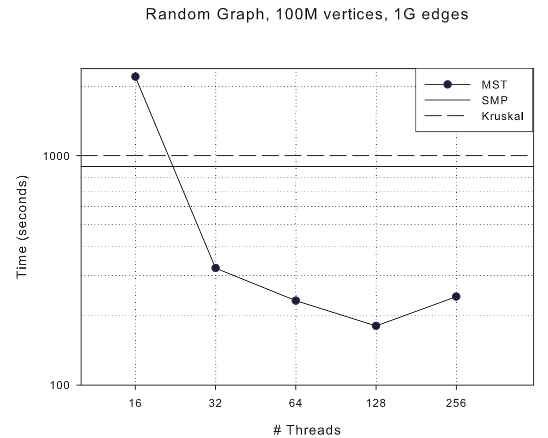

Fig. 10. Performance of MST on a graph with 1G edges

the PGAS environment.

VII. Conclusion and Future Work

We present our study of parallel graph algorithms in UPC on a cluster of SMPs. We show that with appropriate optimizations, shared-memory graph algorithms can be mapped to the PGAS environment with high performance. On inputs that fit in the main memory on one node, our implementation achieves good speedups over the best SMP implementation and the best sequential implementation. We propose collectives that can be used to rewrite shared-memory algorithms for both better communication efficiency and cache performance. For experimental algorithm study, our results suggest that effective use of processors and caches can bring better performance than simply reducing the communication rounds.

The code structure of our implementation suggests opportunities for improvement in compiler, language design, and runtime support. The scheduling of memory accesses may be automated (in part) by the compiler and the runtime. Among the optimizations presented in section V, *compact* and *offload* are algorithm specific, and should be left to the programmer. The *circular* and *localcpy* optimizations occur inside the collectives, and can be implemented by the runtime. Some optimizations packaged in *id*, for example, fast computation of the sorting keys, can be done by the compiler with efficient runtime support.

In our future work we will continue to investigate the performance of shared-memory algorithms implemented with PGAS runtime. We also plan to study the performance of these algorithms on machines with a very large number of processors. It is also interesting to compare the performance of algorithms based on different algorithmic models.

References

[1] M. Adler, W. Dittrich, B. Juurlink, M. Kutyłowski, and I. Rieping. Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 27–36, New York, NY, USA, 1998. ACM.

[2] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, Apr 2004.

[3] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, New Mexico, Apr 2004.

[4] D. A. Bader and G. Cong. Efficient parallel algorithms for multi-core and multiprocessors. In S. Rajasekaran and J. Reif, editors, *Handbook of Parallel Computing: Models, Algorithms, and Applications*. CRC press, 2007.

[5] D.A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proceeding of the 2005 International Conference on Parallel Processing*, pages 547–556, 2005.

[6] D.A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). Technical Report CS-TR-3798 and UMIACS-TR-97-48, Institute for Advanced Computer Studies (UMIACS), University of Maryland, College Park, MD, May 1997. www.umiacs.umd.edu/research/EXPAR.

[7] A-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[8] Jonathan W. Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Proc. International Parallel and Distributed Processing Symposium*, volume 0, page 495, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel & Distributed Comput.*, 37(1):55–69, 1996.

[10] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining*, April 2004.

[11] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. Van Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 519–538, San Diego, CA, 2005.

[12] Y-J Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *Proceedings of the 1995 Symposium on Discrete Algorithms*, pages 139–149, 1995.

[13] G. Cong and D.A. Bader. An empirical analysis of parallel random permutation algorithms on smps. In *the 18th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, 2005.

[14] G. Cong and D.A. Bader. Designing irregular parallel algorithms with mutual exclusion and lock-free protocols. *Journal of Parallel and Distributed Computing*, 66:854–866, June 2006.

[15] D. E. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser. Fast parallel sorting under LogP: From theory to practice. In *Portability and Performance for Parallel Processing*, chapter 4, pages 71–98. John Wiley & Sons, 1993.

[16] R. Das, M. Uysal, J. Saltz, and Y-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22:462–479, 1993.

[17] *DDR3 SDRAM*, URL:http://en.wikipedia.org/wiki/DDR3_SDRAM.

[18] F. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, 33:183–200, 2002.

[19] Frank Dehne, Andreas Fabri, and Andrew Rau-chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6:298–307, 1994.

[20] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *Proc. ACM SIGCOMM*, pages 251–262, 1999.

[21] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel & Distributed Comput.*, 22(2):251–267, 1994.

[22] M.T. Goodrich. Communication-efficient parallel sorting. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 247–256, New York, NY, USA, 1996. ACM.

[23] D.R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, January 1999. Springer-Verlag.

[24] IBM Corporation. *Remote Direct Memory Access Overview*. http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.sni/doc/sni/Remote.htm.

[25] *Infiniband*, URL:http://en.wikipedia.org/wiki/Infiniband.

[26] B.M.E. Moret, D.A. Bader, T. Warnow, S.K. Wyman, and M. Yan. GRAPPA: a high-performance computational tool for phylogeny reconstruction from gene-order data. In *Proc. Botany*, Albuquerque, NM, August 2001.

[27] *PCI Express*, URL:http://en.wikipedia.org/wiki/PCI_Express.

[28] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.

[29] *Unified Parallel C*, URL:http://en.wikipedia.org/wiki/Unified_Parallel_C.

[30] K. Yelick *et al.*, *The Berkeley UPC Compiler*, URL:http://upc.lbl.gov/.

[31] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[32] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and Ü. V. Çatalyürek. A scalable distributed parallel breadth-first search algorithm on Bluegene/L. In *Proc. Supercomputing (SC 2005)*, Seattle, WA, November 2005.