Importance of Runtime Considerations in Performance Engineering of Large-Scale Distributed Graph Algorithms

Jesun Sahariar Firoz(✉), Thejaka Amila Kanewala, Marcin Zalewski, Martina Barnas, and Andrew Lumsdaine

Center for Research in Extreme Scale Technologies (CREST), Indiana University, Bloomington, IN, USA
{jsfiroz,thejkane,zalewski,mbarnas,lums}@indiana.edu

Abstract. Due to the ever increasing complexity of the modern supercomputers, performance analysis of irregular applications became an experimental endeavor. We show that runtime considerations are inseparable from algorithmic concerns in performance engineering of large-scale distributed graph algorithms, and we argue that the whole system stack, starting with the algorithm at the top down to low-level communication libraries must be considered.

1 Introduction

Large graphs are ubiquitous in fields such as social network analytics, transportation optimization, artificial intelligence, and power grids. The largest of graph problems can only be solved using distributed graph algorithms (DGAs), a class of algorithmic approaches in which data is distributed over multiple computing nodes. A distributed computation on a modern supercomputer is built on a software/hardware stack that is more complex than ever before. This complexity and the resulting explosion of parameters, further exacerbated by the massive irregularity and data dependency of large-scale graph problems, made performance analysis of DGAs a predominantly experimental undertaking.

We distinguish algorithm-level contributions that are often prioritized in research results from runtime-level concerns that are harder to place in the context of DGAs and are often hidden from application developers. We argue that in order to obtain an accurate understanding of a DGA performance, algorithmic concerns need to be considered holistically with runtime properties. In this paper, we illustrate this inseparability of runtime and algorithmic considerations. Specifically, we concentrate on two graph traversal algorithms, breadth-first search (BFS) and single-source shortest paths (SSSP), which are essential building blocks for many other applications. For the sake of clarity, all experimental results presented here were obtained on Big Red 2 at Indiana University [2] with 3-D torus topology and Cray's Message Passing Toolkit (MPT) 6.2.2 MPI implementation. Note that differences in hardware between different machines constitute another layer of complexity. We show that, in the extreme,

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 553–564, 2015.
DOI: 10.1007/978-3-319-27308-2_45

a feature of the runtime can make an algorithmic approach ostensibly not viable. This implies that the runtime is such an integral part of DGAs that experimental results are difficult to interpret and extrapolate without understanding the properties of the runtime used.

This paper is organized as follows. First, we briefly describe the relevant features of the runtime systems we used in Sect. 2. Then, we provide a brief overview of the distributed control (DC) and Δ-stepping algorithms and explain why DC is particularly suitable to expose the importance of runtime in Sect. 3. Next, in Sect. 4, we show a dramatic change in scaling behavior comparing different SSSP algorithms when implemented with two different runtimes, AM++ and HPX-5. Finally, we show that the application performance is sensitive to even smaller level changes by varying features within the same runtime system (AM++). We provide our concluding remarks in Sect. 5.

2 Runtime Systems

In this paper we use two runtime systems, HPX-5 [1] and AM++ [7]. AM++ is our legacy system centered around active messaging of the Active Pebbles [8] model. HPX-5 is being developed at the Center for Research in Extreme Scale Technologies (CREST) at Indiana University to facilitate the transition to exascale computing.

HPX-5 is intended to enable dynamic adaptive resource management and task scheduling. It creates a global name and address space structured through a hierarchy of processes, each of which serve as execution contexts and may span multiple nodes. It employs a generalization of local ephemeral tasks that permit preemption and global mutable side-effects. It is event-driven, enabling the migration of continuations and the movement of work to data, when appropriate, based on sophisticated local control synchronization objects (e.g., futures, dataflow). HPX-5 is an evolving runtime system being employed to quantify effects of latency, overhead, contention, and parallelism. These performance parameters determine a tradeoff space within which dynamic control is performed for best performance. It is an area of active research driven by complex applications and advances in HPC architecture.

HPX-5 currently has two types of network transports: isend-irecv (ISIR), and put with completion (PWC) based on the Photon [3] network library. In the experiments presented here we used the ISIR transport option. ISIR transport is based on MPI two-sided communication paradigm with asynchronous sends and receives. Currently, HPX-5 thread support level in ISIR is *funneled*.

AM++ supports fine-grained parallelism of active messages with communication optimization techniques such as object-based addressing, active routing, message coalescing, message reduction, and termination detection. While less feature-rich than HPX-5, active messages share the fine parallelism approach with HPX-5. In addition, AM++ is a relatively well-optimized implementation. For these reasons, it was our choice during the development of DC approach. Of particular relevance is its suitability to balance the competing needs of quick delivery of work vs. minimal communication overhead.

While AM++ and HPX-5 share some features and goals, there are important differences between them. AM++ is designed for bulk processing of distributed messages, while HPX-5 is a complete system providing inter and intra-node parallelism. HPX-5 provides global address space while AM++ provides only a lightweight object-based addressing layer. In HPX-5 work is divided into first-class tasks with stacks, while AM++ only executes message handler functions on the incoming message data. These features result in significant differences in scheduling.

2.1 More Details About AM++

One of the results of this paper is that performance of an application can be sensitive to the utilized nuances of runtime features. Hence, more detailed description of the runtime that we used to show this is in order.

AM++ is based on the Active Pebbles (AP) model [8]. At the core of the AP model are *pebbles*, lightweight active messages that are sent explicitly but received implicitly. The implicit receive mechanism is based on *handlers*, which are user-defined functions that are executed in response to the received pebbles. AM++ is a library interface that can be executed by many workers (threads). Each worker can execute independently, and when it calls AM++ interfaces it may execute tasks from the AM++ *task queue*, which schedules tasks such as network polling, buffer flushing, and pending handlers. At the lowest level, pebbles are sent and received using *transports* that encapsulate all low level AM++ functionality such as network communication and termination detection. Currently, the low-level network transport of AM++ is built atop of MPI, but none of the MPI interfaces are exposed to the programmer, and AM++ has supported other transports before. In order to send and handle active pebbles, individual *message types* must be registered with the transport. A transport, given the type of data being sent and the type of the message handler, can create a complete message type object. To increase bandwidth utilization, AM++ performs *message coalescing*, combining multiple pebbles sent to the same destination into a single, larger message. In the current implementation, a buffer of messages is kept by each node for each message type that uses coalescing and for each possible destination. The size of coalescing buffers is determined by the maximum number of pebbles to be coalesced and the pebble size. Messages are appended to the buffer, and the entire buffer is sent when it becomes full or it is *flushed* when there is no more activity. Message coalescing increases the rate and decreases the overhead at which small messages can be sent over a network. The transport layer costs (bookkeeping, message injection) are amortized over many messages at some cost to latency. However, this cost is expected to be offset by large problem scales that depend on throughput more than on latency.

The AM++ programming model is based on *epochs*, which are periods in which messages can be sent and during which all the resulting handlers are executed (*termination detection*). All workers must enter and exit the epoch collectively, with the exit possible only after all the handlers in the epoch are executed. AM++ only guarantees that the handlers for all the messages sent

within a given epoch will have completed by the end of that epoch, but it also guarantees that calling end of an epoch test interface will progress AM++ execution. Because AM++ allows handlers to send arbitrary new messages, it relies on a *termination detection algorithm* to discover when no more handlers are left to execute and no more pebbles are in flight.

In general, AM++ workers perform two kinds of work: the worker's "private" work, and AM++ progress that can occur any time an AM++ interface is called. A thread's private work includes tasks such as local bookkeeping or preparing for an epoch. AM++ progress consists of handler execution, crucial to algorithm progress, and bookkeeping and maintenance tasks such as network polling, buffer flushing, and termination detection. In general, an AM++ program consists of general setup, including creating a transport and registering message types with the transport along with required properties such as coalescing and object-based addressing.

After all the necessary machinery is created, an AM++ program executes one or more epochs. Epochs can be executed in two significantly different ways. In *scoped epoch* execution, application work is executed first, and when the epoch ends, AM++ executes final progress including member handlers. The application can still send messages, and progress can be executed when messages are sent, but, in general, progress is only guaranteed to occur at the end of the scoped epoch resulting in a two-part work pattern. In *end-epoch test* model, application executes some work in a loop, testing for the end of the epoch. This model allows an application to interleave its own work with AM++ progress, resulting in a pattern of potentially unequal periods of time spent in each portion of the work. The end-epoch test model allows for interaction between the application and AM++, where the application generates new tasks, and the execution of handlers in AM++ progress generates more work for the application, repeating the loop until all work is exhausted.

3 Algorithms

In this section we describe the distributed graph algorithmic approaches pertinent to understanding of our results.

Graph traversal constitutes the main kernel for solving many graph theoretic problems and is a good representative of irregular applications. In this paper, we employ two graph traversal problems, namely SSSP and BFS. Let us denote an undirected graph with $n$vertices and m edges by $G(V, E)$. Here $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$ represent vertex set and edge set respectively. Each edge e_i is a triple (v_j, v_k, w_{jk}) consisting of two endpoint vertices and the edge weight. We assume that each edge has a nonzero cost (weight) for traversal. In *single source shortest path* (*SSSP*) problem, given a graph G and a source vertex s, we are interested in finding the shortest distance between s and all other vertices in the graph. *Breadth First Search* (*BFS*) can be regarded as a special case of SSSP when the edge weights w_{ij} are set to one. In this section, we discuss two distributed graph algorithms for SSSP: Distributed Control and Δ-stepping.

3.1 Distributed Control

Unordered algorithms [6] are invariant under ordering of tasks and are therefore easier to parallelize than those algorithms in which task ordering impacts correctness of results. Nevertheless, while not necessary for correctness, task ordering can help improve performance of unordered algorithms. Our *distributed control* (DC) [9] is a work scheduling method that removes overhead of synchronization and global data structures while providing partial ordering of tasks according to a priority measure. DC uses only local knowledge to select the best work, thus obtaining an approximation of global ordering.

Specifically, global data is divided into domains. Within each domain, workers are assigned a shared memory. Processing tasks in one domain may generate work that depends on other domains. Workers put the work they generate into an unordered global task bag, and continuously try to retrieve work from there to put it into their private working sets, which are ordered according to task priority metric. For example, in the case of SSSP, the metric is the distance. The more tasks in the bag, rather than in the private working sets, the further the approximated ordering is from the ideal global ordering (Dijkstra's priority queue). DC does not use global data structures and synchronization. Ideally, the underlying runtime system delivers a task to the appropriate worker as soon as it becomes available. On the other hand, quick delivery is costly. These competing trends need to be balanced for optimal performance. For this reason, DC is particularly sensitive to the properties of runtime. In what follows, we use Δ-stepping algorithm for reference, which can be expected to be less dependent on properties of the underlying runtime system.

DC algorithm for SSSP is described in Algorithm 1. The algorithm consists of 3 parts that are executed by all threads on a distributed node: the main loop that processes tasks from the local priority queue and performs progress, the message handler that receives tasks from other workers, and the relax function that updates distances and generates new work. The main loop is preceded by initialization of the distance map, starting an epoch (Lines 1–2), and by relaxing the source vertex (Lines 3–5). In the while loop, the work on the graph is performed by removing a task from the thread-local priority queue in every iteration and then relaxing the vertex targeted by the task (Lines 8–15). Vertex relaxation checks whether the distance sent to a vertex v is better than the distance already in the distance map, and it sends a relax message (task) to all the neighbors of v with the new distance computed from v's distance d_v and the weight of the edge between v and v's neighbors v_n. Relax handler receives the messages sent from the relax function, and its only purpose is to insert the incoming tasks into the thread-local priority queue. When a handler finishes executing, it is counted as finished in *termination detection*. Because our handlers actually only postpone work, every time we insert a task into an empty queue we increment activity counter. Then, in the main loop, when the queue becomes empty, we can decrease the activity counter when all the postponed tasks are handled and the queue is empty. Note that there is no synchronization barrier in the algorithm.

Algorithm 1. Distributed Control Algorithm

Main loop

Input: Graph $G(V, E)$, source s, distances D

```
1:  Init(D)                                              {set distances to ∞}
2:  Epoch e                                              {start epoch}
3:  if thread_id = 0 and  owner(s) then
4:     relax(s, 0)                                       {relax source}
5:  end if
6:  while not  e.end() do
7:                                  {q_t is the priority queue of the current thread}
8:     if not q_t.empty() then
9:        (v, d) ← q_t.pop()                             {next task to process}
10:       d_v ← D(v)                                     {current distance for v}
11:       if d_v < d then
12:          continue
13:       else
14:          relax(v, d)
15:       end if
16:       if q_t.empty() then
17:          activity_count−−
18:       end if
19:       if iterations mod frequency = 0 and e.end() then
20:          return
21:       end if
22:    end if
23: end while
```

Relax handler

Input: Task (v, d)

```
1:  if q_t.empty() then
2:     activity_count++
3:  end if
4:  q_t.push(v, d)                                       {insert task into priority queue}
```

Relax

Input: Task (v, d), distances D

```
1:  if d < D(v) then
2:     D(v) ← d
3:     ∀v_n ∈ neighbors(G, v) : send(v_n, d_v + weight(v, v_n))
4:  end if
```

3.2 Δ-Stepping

Δ-stepping [4] approximates the ideal priority ordering by arranging tasks into distance ranges (buckets) of size Δ and executing buckets in order. Within a bucket, tasks are not ordered, and can be executed in parallel. Processing a bucket may produce extra work for the same bucket or for the successive buckets. After processing each bucket, all processes must synchronize before processing

the next bucket to maintain task ordering approximation. The more buckets (the smaller the Δ value), the more time spent on synchronization. Similarly, the fewer buckets (the larger the Δ value), the more suboptimal work the algorithm generates because larger buckets provide less ordering.

Even though Δ-stepping ordering approximates Dijkstra's global ordering, the overhead of synchronization after each bucket is significant. All workers need to wait for the last straggler to proceed. The more buckets Δ-stepping needs to process, the more computing power is wasted due to straggler effect.

Δ-stepping SSSP algorithm is described in Algorithm 2. In each epoch, vertices within the range $i\Delta - (i+1)\Delta$ contained in a bucket B_i are processed asynchronously by worker threads(Lines 11–27). Edges are classified into two categories: light edges and heavy edges. Edges with weight less than or equal to Δ are called *light* edges and are processsed first at the beginning of the epoch (Line 13). Once processing of all light edges contained in the current bucket is done, then *heavy* edges (weight greater than Δ) are processed (Line 21). Processing vertices in the current bucket can generate new pairs for the current bucket. Worker threads cannot proceed to the next bucket unless all workers on each of the distributed node has finished processing vertices contained in the current bucket. This requires global synchronization barrier (which is implicit in the epoch). The relax function is responsible for putting a vertex in the appropriate bucket if the updated distance is better (Line 5).

DC is not equivalent to Δ-stepping with Δ set to ∞, even though in this limit, there is one unordered global task bag similar to *DC*. However, Δ-stepping algorithm does not impose any ordering on tasks. In contrast, *DC* orders the tasks retrieved by each worker (thread) from the global task bag in a thread-local priority queue. This in effect reduces the amount of redundant work to be executed. By eliminating the need for global synchronization and maintaining partial ordering of tasks based on local view, *DC* can achieve better runtime.

4 Application Performance Sensitivity to Runtime Features

In this section we present results which tell a cautionary tale of how changes in runtime system affect performance.

Figure 1 shows size scaling for DC and Δ-stepping with HPX-5 (Fig. 1a) and AM++ (Fig. 1b). For size scaling we keep the number of cores constant at 1024, and we increase the size of the problem, taking an average of 10 runs per point. The experiments were run on Graph 500 [5] inputs. In HPX-5 Δ-stepping clearly outperforms DC, which performs no better than chaotic execution. In AM++, DC outperforms two different versions of Δ-stepping. The Δ-stepping version, in which we sort the messages in the AM++ receive buffers as they arrive at a particular destination, is termed *delta-sort*. While at smaller scales the averages follow the same trend, they are indistinguishable within errors. For this reason, we show results at larger scales.

Algorithm 2. Δ-stepping Algorithm

Main loop

Input: Graph $G(V, E)$, source s, distances D, Δ

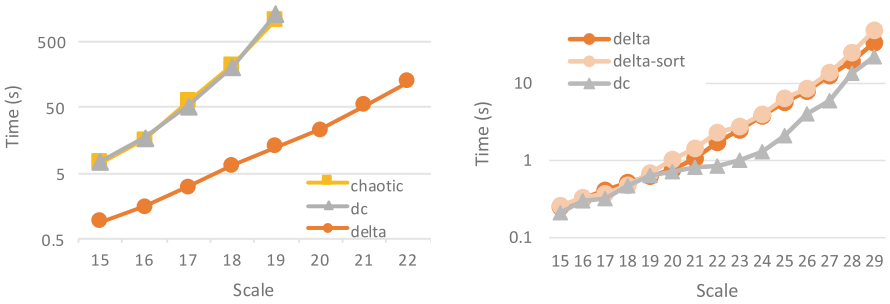
```
1: Init(D)                                                  {set distances to ∞}
2: for ∀v ∈ V do
3:     {group heavy and light edges}
4:     heavy(v) ← {(v, w) ∈ E | weight(v, w) > Δ}
5:     light(v) ← {(v, w) ∈ E | weight(v, w) ≤ Δ}
6: end for
7: i ← 0; B_0.put(s)                                        {put source in Bucket 0}
8: while not B.empty() do
9:                 {B is the array of buckets containing vertices within each Δ range}
10:    Epoch e.start()                                      {start epoch}
11:    while not B_i.empty() do
12:        M ← ∅
13:        R ← {(w, d) | ∀ v ∈ B_i ∧ weight(v, w) ∈ light ∧ d = D(v)+weight(v, w)}
14:        {process the neighbours in light edge set}
15:        activity_count − −
16:        M ← M ∪ B_i; B_i = ∅
17:        for ∀(v, d) ∈ R do
18:            relax(v, d)
19:        end for
20:    end while
21:    R ← {(w, d) | ∀ v ∈ M ∧ weight(v, w) ∈ heavy ∧ d = D(v)+weight(v, w)}
22:    {process the neighbours in heavy edge set}
23:    for ∀(v, d) ∈ R do
24:        relax (v, d)
25:        activity_count − −
26:    end for
27:    e.end()
28:    i ← i + 1
29: end while
```

Relax

Input: Task (v, d), distances D

```
1: if d < D(v) then
2:     oldindex ← D(v)/Δ
3:     B_oldindex ← B_oldindex \ v
4:     newindex ← d/Δ
5:     B_newindex ← B_newindex ∪ v
6: end if
```

This is a dramatic, qualitative change that shows how intertwined the algorithm and runtime system are. While there is an obvious difference between the two runtimes, performance can change drastically with just small changes within the same runtime system. Such changes are often outside of the application



(a) Size scaling in HPX-5. (b) Size scaling in AM++.

Fig. 1. Size scaling comparison of Δ-stepping and DC on Graph 500 input. DC is the better algorithm in AM++ but performs no better than simple chaotic version in HPX-5. Each data point is an average of 10 runs (Color figure online).

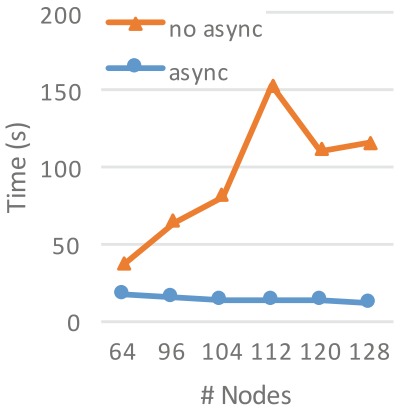


Fig. 2. Impact of asynchronous MPI progress thread on performance of distributed control in AM++.

developer's control, and can lead to misguided conclusions about algorithm performance. For example, at first, when we experimented with DC on Big Red 2, we found that DC was performing poorly (see the top line in Fig. 2), which raised concerns that DC might not be viable. The performance was decreasing with increasing number of nodes in strong scaling. Suspecting that message latencies were to blame, we experimented with transport progress despite Cray's warning at the time that thread-multiple progress required for asynchronous progress "is not considered a high-performance implementation." Fig. 2 presents strong-scaling results on Graph500 scale 31. With *asynchronous progress* (the bottom line in Fig. 2), the performance of DC has improved more than tenfold with growing node counts, entirely changing the viability of the approach.

Another example of a runtime feature that is unknown at application development time is the size of coalescing buffers. To decrease overhead at the cost

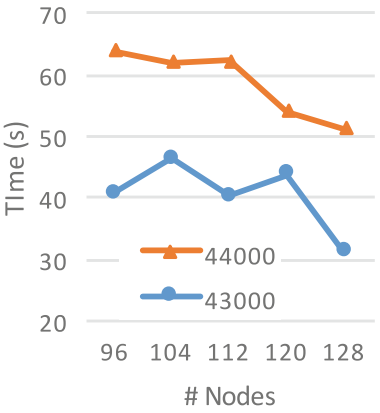


Fig. 3. Impact of coalescing buffer sizes on performance of distributed control in AM++.

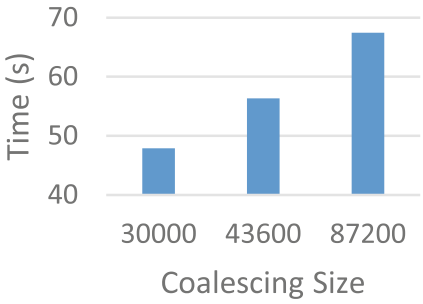


Fig. 4. Effect of coalescing size on DC BFS algorithm on a scale 31 graph in AM++.

of increased latency, AM++ performs *message coalescing*, combining multiple messages sent to the same destination into a single, larger message. Messages are appended to per-destination buffers. To handle partially filled buffers, a periodic check is performed to check for activity. In the case of DC SSSP, a single message consists of a tuple of a destination vertex and distance, 12 bytes in total. With such small messages, coalescing has great impact on the performance, but finding the optimal size is difficult.

We investigated the impact of coalescing in Graph500 scale 31 graphs when running DC SSSP with max edge weight of 100 (Figs. 3 and 4). Figure 3 shows the large impact of a small change in the coalescing size, measured by the number of SSSP messages per coalescing buffer. Changing the coalescing size by less than 2 % causes over 50 % increase in the run time. This unexpected effect is caused by the specifics of Cray MPI protocols. At the smaller coalescing size, full message buffers fit into rendezvous R0 protocol that sends messages of up to 512 K using one RDMA GET, while the larger buffers hit R1 protocol that sends chunks of 512 K using RDMA PUT operations. At the size of 44000, the bulk

of the message fits into the first 512 K buffer, and the small remainder requires another RDMA PUT, causing overheads. The sizes 43000 and 86000 fill out 1 and 2 buffers, respectively, achieving similar performance. The larger size, 86000, results in better scaling properties.

Figure 4 shows the effects of coalescing on a DC BFS, which is SSSP with maximum weight of 1. Surprisingly, in this case increasing the coalescing size impacts performance negatively. We suspect that with smaller weights the probability of reward from optimistic parallelism in DC decreases, and the added latency of coalescing has a much larger effect than with larger weights. Also note that we have not actually discovered the optimal coalescing size, which would require more experiments and more resources. This shows that adjusting the coalescing size is important, and that the optimal value is not static. Rather, it depends on algorithmic concerns such as reward from optimistic parallelism.

5 Conclusions

In this paper we demonstrated that performance of DGAs strongly depends on characteristics and features of the underlying runtime. Within a particular runtime, low-level components such as bit transport can drastically impact the performance. We demonstrated this using our DC work scheduling algorithm and two examples of low-level features, namely asynchronous progress and coalescing, but the implications are of general validity. This means that for performance engineering, application developers need to design their algorithms so that they are in sync with the runtime features. Unfortunately, the interplay between different components of the stack, while acknowledged, is not well understood yet. More work is needed to explore how features such as communication paradigm, network topology, message routing and task scheduling interact with algorithmic concerns in order to determine guidelines for optimal performance.

Acknowledgments. This research used Big Red2 (Funded by Lilly Endowment, Inc. and Indiana METACyt Initiative). Support by NSF grant 1111888 gratefully acknowledged.

References

1. http://hpx.crest.iu.edu/. Accessed 25 May 2015
2. Big Red II at Indiana University. http://rt.uits.iu.edu/ci/systems/BRII.php#info. Accessed 17 Apr 2015
3. Kissel, E., Swany, M.: Session layer burst switching for high performance data movement. In: Proceedings of the 8th International Workshop on Protocols for Future, Large-Scale & Diverse Network Transports (2010)
4. Meyer, U., Sanders, P.: Δ-stepping: a parallelizable shortest path algorithm. J. Algorithms **49**(1), 114–152 (2003)
5. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the graph 500 benchmark. Cray User's Group (CUG) (2010)

6. Pingali, K., et al.: The Tao of Parallelism in Algorithms. ACM SIGPLAN Not. **46**(6), 12–25 (2011)
7. Willcock, J.J., Hoefler, T., Edmonds, N.G., Lumsdaine, A.: AM++: a generalized active message framework. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, pp. 401–410. ACM (2010)
8. Willcock, J.J., Hoefler, T., Edmonds, N.G., Lumsdaine, A.: Active pebbles: a programming model for highly parallel fine-grained data-driven computations. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, pp. 305–306. ACM (2011)
9. Zalewski, M., Kanewala, T.A., Firoz, J.S., Lumsdaine, A.: Distributed control: priority scheduling for single source shortest paths without synchronization. In: Proceedings of the Fourth Workshop on Irregular Applications: Architectures and Algorithms, pp. 17–24. IEEE (2014)